SPECIAL ISSUE PAPER

Tool support for refinement of non-functional specifications

Simone Röttger · Steffen Zschaler

Abstract Model driven architecture (MDA) views application development as a continuous transformation of models of the target system. We propose a methodology which extends this view to non-functional properties. In previous publications we have shown how we can use so-called context models to make the specification of non-functional measurements independent of their application in concrete system specifications. We have also shown how this allows us to distinguish two roles in the development process: the measurement designer and the application designer.

In this paper we use the notion of context models to allow the measurement designer to provide measurement definitions at different levels of abstraction. A measurement in our terminology is a non-functional dimension that can be constrained to describe a non-functional property. Requiring the measurement designer to define transformations between context models, and applying them to measurement definitions, enables us to provide tool support for refinement of non-functional constraints to the application designer. The paper presents the concepts for such tool support as well as a prototype implementation.

Keywords Non-functional properties · Model transformation · Refinement · CASE tool support

Communicated by Ana Moreira and Thomas Baar.

S. Röttger (✉) · S. Zschaler
Technische Universität Dresden, Dresden, Germany
e-mail: Simone.Roettger@inf.tu-dresden.de

S. Zschaler
e-mail: Steffen.Zschaler@inf.tu-dresden.de

1 Introduction

Non-functional properties of a system – for example, Quality of Service (QoS) or security aspects – need to be considered as early as possible in the development cycle to analyse the non-functional behaviour of the system. This is especially true for component-based systems because all context dependencies need to be made explicit. In the context of the COMQUAD project[1] we develop a methodology supporting the modelling of component-based systems with particular emphasis on non-functional aspects. In this paper we focus on the models required by the methodology. Although they are directly applicable to QoS properties only (such as response time, delay, memory usage), we believe that they can be extended to cover other non-functional product properties–such as security–as well. For the purpose of this paper we will consider the terms non-functional and QoS to be synonyms.

The core concept of QoS specifications is the measurement–or characteristic [15]. A measurement is a mapping from states, objects, or events of a physical system (e.g., an implemented and running application) to a formal system (for example, the set of real numbers). Examples for measurements are: response time (a mapping from an operation call in a running system to a real number representing the time taken from invocation to return), or confidentiality (a mapping from a channel used to transfer information to a value indicating the level of confidentiality achieved by this channel).

Measurements and constraints over them must be modelled. Formally modelling a measurement can be complicated and is thus not a task an application designer would want to undertake. At the same time, the same measurement may be relevant to different applications, so once a measurement has been modelled we would like to be able to reuse it for different applications. In order to do so, measurement specifications must be made independent of specific applications. We can achieve this by using models of the relevant aspects of target applications–we call these *context models*[2] [31]–in the definition of measurements. They will then be applicable to any system model that can be viewed as an instance of the context model used in the definition of the measurement. We have thus ensured that the definition of measurements can be done independently of the usage of those measurements, and vice-versa. We can now separate two roles in the development process: The *measurement designer*, who creates a library of measurements, and the *application designer*, who uses these measurements to annotate application models with non-functional specifications.

The basic idea of existing development processes–especially approaches based on the model-driven architecture (MDA) [19]–is the refinement of system models from an abstract view of the system to a model close to the real implementation. The application designer creates, and thinks about, functional models at different levels of abstraction. He should be able to do so for non-functional models, too. This paper is about how different context models can be used to represent different *levels of abstraction* for a measurement, and how this can be leveraged to provide tool support for the application designer's refinement of non-functional specifications. Two ideas form the basis for this: (a) we require the measurement designer to describe the refinement relations between different context models as transformations, and (b) we apply these transformations to the more abstract measurement definitions to create refined versions. The application designer can then reuse these refinements as prompted by refinements in the functional model of the system. We distinguish two kinds of non-functional refinement: *structural refinement* and *measurement refinement*, which will be explained later in Sect. 3.

This paper is an extended and refined version of [32,33]. It focuses on modelling issues related to measurement refinement. In Sect. 2 we give a short introduction to our overall development process and the

specification languages we use, which form the context of this work. The following sections describe measurement refinement and the related models in more detail: from the application designer's view (Sect. 3), from the measurement designer's view (Sect. 4), and from a more technical, tool-oriented perspective (Sect. 5). Section 6 presents a prototype implementation of tool support for our concepts. We use a simple example application with response time constraints throughout the paper to illustrate our approach. Finally, the conclusion points out the most important arguments of our work as well as issues for further research.

## 2 A process for component-based systems with non-functional properties

Figure 1 gives an overview of our overall software development process for non-functional properties. After the requirements analysis the application designer begins to model the system. This includes modelling of non-functional properties by specifying non-functional constraints and attaching them to components and connectors. The application designer switches between modelling–and refining–non-functional properties of the components (labelled "Application Modelling" in the figure) and of the components' environment (called "Environment Modelling" in the figure).

Our approach separates measurement definition from measurement usage–that is, 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 specification of the system fincluding its non-functional properties.

Our process comprises the following steps:

1. Definition of measurements at different levels of abstraction, including provision of transformation rules for context models by the measurement designer (see Sect. 4). The measurement designer can do so independently of application development and even at a far earlier time. Remember that a measurement is simply a non-functional dimension that must be constrained to form a non-functional property.
2. Use of measurements during the specification process by the application designer. The application designer constrains measurements and binds these constraints to elements of the functional model.
3. Tool-supported refinement of measurements. The application designer chooses one out of several

---

[2] Note that these models are unrelated to models from context-aware computation. They merely represent the context of a measurement definition.
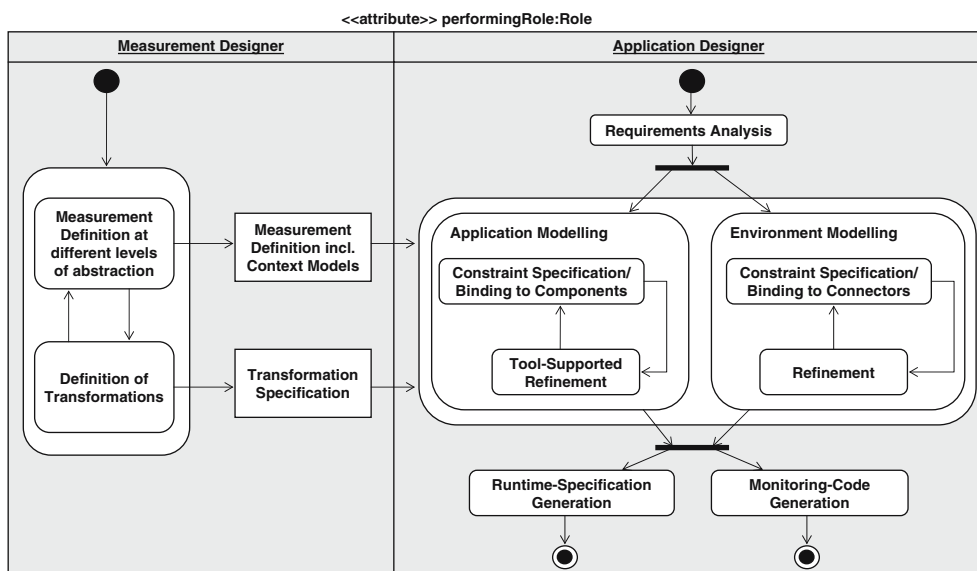
Fig. 1 Development process for non-functional properties overview

refined measurements. These have been previously provided by the measurement designer together with an informal description of each measurement (see first item).

4. Modelling and refinement of connectors between components during the assembly process. The application designer uses connectors to model the influence of the container on non-functional properties of the application.

The resulting non-functional specification is used for a variety of purposes. Apart from generating code for runtime monitoring of QoS parameters, its main use is in providing a base for QoS contract negotiation and resource reservation in the running system–the component container.

A third role–the *component developer*–is involved in the software development process (Fig. 2). His task is to provide appropriate component implementations. It is essential to test the non-functional properties of these component implementations in a test container [24]. The component developer then translates the test results into a non-functional specification for each component implementation. For this purpose he uses the predefined measurements from the measurement repository. The component implementations and the respective non-functional specifications are stored in a component implementation library. The application designer can query the library to find component implementations fulfilling the non-functional requirements of the application system specification. The verification of the components' functionality is beyond the scope of this paper.

We use two specification languages: for functional modelling we primarily use the component model element from UML 2.0 [29] extended with a stereotype for interfaces which allows us to distinguish between operational interfaces, offering a set of operations to be invoked, and streaming interfaces for data-flow based communication. We added graphical representations for modelling streaming ports as shown in Fig. 3: interfaces marked up as ≪sources≫–that is, ports emitting data packages–are displayed as outgoing double arrows (–≫), ≪sinks≫—ports accepting data packages–as incoming ones (≫—), respectively. Connections between these ports are depicted as double lines (=). For the non-functional specification we use CQML$^+$ [31] an extension of (CQML) (the Component Quality Modelling language) [1].

CQML$^+$ builds on quality characteristics, which essentially are definitions of measurements. Quality statements are used to specify constraints on characteristics. Both quality characteristics and quality statements are parametrised and can, therefore, be reused in different contexts. To actually attach the non-functional specification to the functional one, CQML$^+$ provides the construct of quality profiles. In such a profile current parameter values–for example, streams or operations of the component to which the non-functional constraint is applied–replace the formal parameters of quality statements. Quality statements can be associated to a component as offers (`provides`), requirements (`uses`), or resource demands (`resources`). CQML$^+$–as CQML–uses a slightly modified version of the object constraint language (OCL) [27] for the specification of the formal meaning of a characteristic.
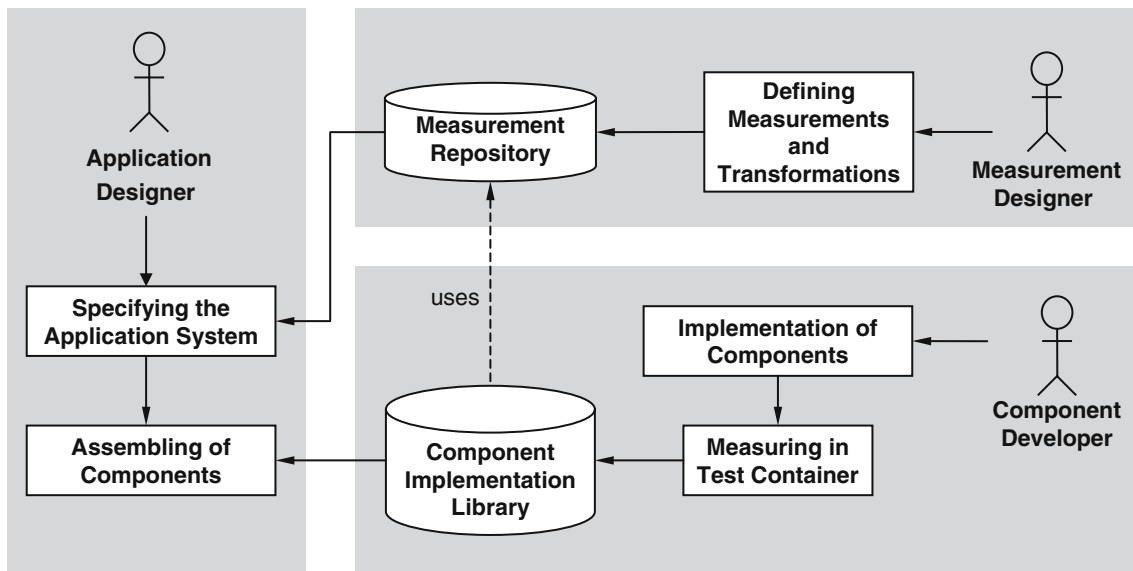
Fig. 2 Roles involved in the development process for non-functional properties

Fig. 3 UML extensions for graphical annotation of non-functional properties
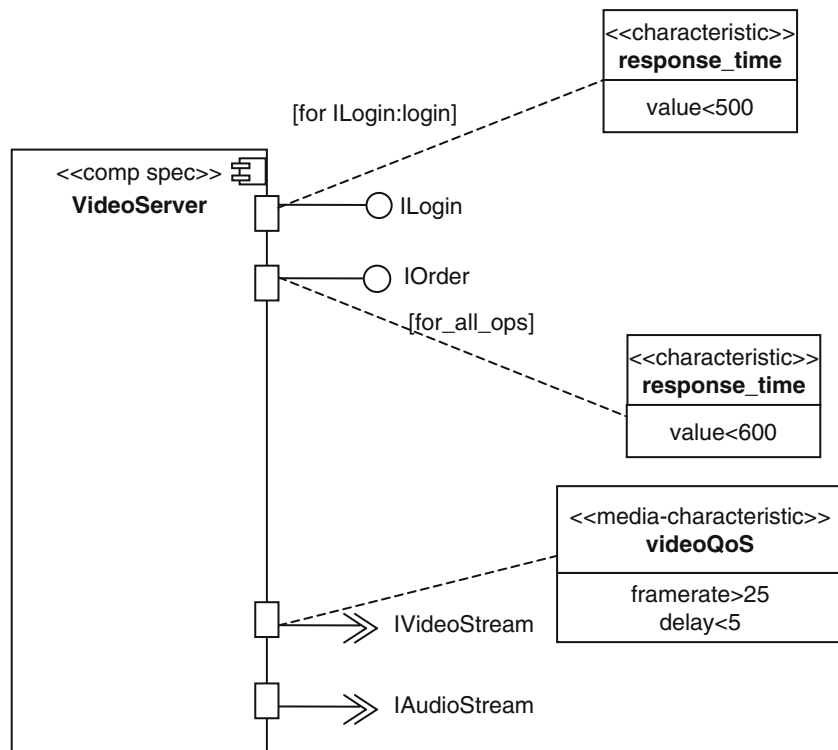
For example, Listing 1 shows a CQML$^+$ snippet describing CD-quality audio transmission. It begins with a quality characteristic specification of the `sample_rate` measurement. The `domain`-clause specifies the range of possible values for the measurement–all real numbers greater or equal zero. The unit of measurement is 'samples per second'. The OCL expression in the `values`-clause specifies that the current sample rate is determined by counting the number of events that occurred for a flow of samples in the last 1,000 ms. Next, it

defines a quality statement `cd_audio` that constrains `sample_rate` to the typical CD-audio rate for one channel. Finally, this quality statement is associated with the `audioOut` port of component `AudioPlayer`.

CQML$^+$ is a textual language comprising both measurement definition and measurement usage. For graphical modelling of measurement definitions we plan to use ideas proposed in [30]. For measurement usage we have defined a graphical notation allowing to attach constraints to parts of the functional model (Fig. 3).

Listing 1 A sample CQML⁺specification of CD-quality audio transmission

```
1   quality_characteristic sample_rate (af : Flow) {
        domain: numeric real [0..) samples/second;

        values: af.events.eventsInRange (1000)->size();
    }

6   quality cd_audio (af: Flow) {
        sample_rate (af) >= 44100;
    }

    profile data_delivery for AudioPlayer {
11      profile good {
            provides cd_audio (audioOut);
        }
    }
```

Here the definition of the measurement is secondary. As a simple example imagine a `VideoServer` component providing ports for login and ordering and also streaming ports for outgoing audio and video streams. To constrain all operations offered via the `IOrder` interface we only need the name of the measurement (here `response_time`) and the value constraining it. As depicted we concentrate these information in one graphical element called `characteristic`. It is also possible to define complex measurements, which we call `media-characteristic`s. These are template measurement sets which are normally used together to characterize a media stream. In our example the streaming port `IVideoStream` is constrained by a media-characteristic `videoQos` defined by the predefined values for `framerate` and `delay`. The internal tool representation uses CQML⁺, merging measurement definition and constraints into one specification. We will explain this in Sect. 6.1.

For a more in-depth explanation, we describe the individual models as seen from the application designer's view as well as from the measurement designer's view in the following sections.

3 The application designer's view

The application designer obtains a target specification from the requirements analysis. Using this artefact, he begins modelling an adequate system which fulfils the customer's requirements. He creates a functional model of the system, tagging non-functional aspects to it using a system modelling tool supporting graphical modelling. As he progresses in the development, the functional model gets more and more detailed. Correspondingly, the application designer must also refine the non-functional specification. We distinguish two kinds of non-functional refinement:

1. *Structural refinement*: The application designer adds new model elements, as the functional model gets more refined. In this process, he may have to reassign non-functional property specifications that had been tagged to one model element to some newly added model element–or he may even have to distribute them to several new elements. For example, at a very early stage the application designer of a video server application may have modelled the complete application as one monolithic component, also tagging any non-functional specifications–for example, response time constraints–to this big component. Later, he refines the component by decomposing its functionality into several subcomponents. In this step, he will also need to refine the non-functional properties tagged to the monolithic component by determining which of the subcomponents have to provide each non-functional property.

2. *Measurement refinement*: With this type of refinement the application designer uses a more precise interpretation of the meaning of a certain measurement. For example, he may wish to start out thinking about response time simply as the time between start and end of an operation call. Later he may wish to make more precise statements about response time. Figure 4 shows his options: the time between (1) the reception of a request and the sending of the corresponding response, or (2) the reception of a request and the reception of the corresponding response, or (3) the sending of a request and the sending of the corresponding response, or (4) the sending of a request and the reception of the corresponding response.
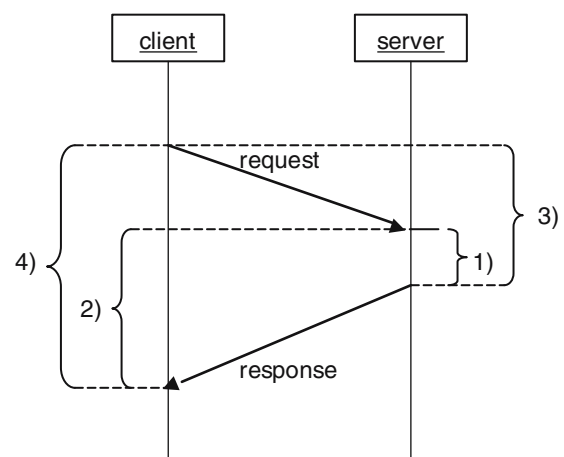
Fig. 4 A sequence diagram excerpt showing different kinds of response time specification
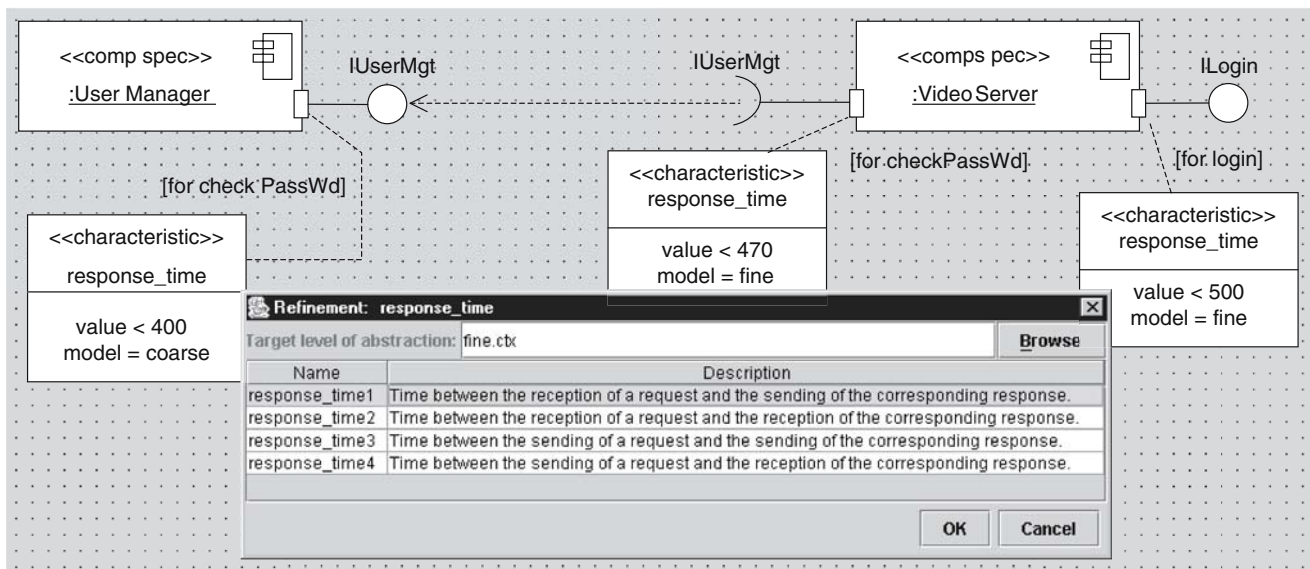
Fig. 5 Sample screen shot

This paper focuses on measurement refinement. Structural refinement of non-functional properties is a complete research area of its own and thus outside the scope of this paper. As a simple example imagine the login mechanism of our `VideoServer` component using another component `UserManager` that manages user data. At an early stage of development the application designer decides that the video server component provides an interface `ILogin` and uses an interface called `IUserMgt`. The `UserManager` provides this interface `IUserMgt`. For operations of these interfaces he can specify different response times depending on the internal execution times of the components. This corresponds to Step 2 in Sect. 2.

However, so far he has not thought about response time in detail, but only as the time between start and end of an operation call. Here, our mapping support is applied. If he wants to refine the response time of the operation `IUserMgt::checkPassWd` used by the `VideoServer`, the application designer can use the CASE tool to refine this non-functional aspect. The tool provides four different kinds of refined response times using a library where the information about the mapping is stored. Depending on what the application designer wants to model, he will choose one of the refined response times and the tool will update the internal model representation and tag the treated characteristic as refined. This is Step 3 from Sect. 2. Figure 5 shows what the screen could look like after these two steps. It shows the two components `VideoServer` and `UserManager`, their used and provided interfaces, and the attached non-functional constraints. Some of these have already been refined—this can be seen from the

line 'model = fine'.[3] The application designer has just opened the refinement dialogue for the last non-functional constraint to be refined and selected one of the possible refinements. Note that the application designer is completely shielded from the formal intricacies underlying the different response time definitions.

Once non-functional specifications have been created and connected with a functional specification, it becomes important to have analysis tools allowing for determination of various properties of the system. One property for which analysis is very important and helpful is to determine whether a component satisfies the non-functional demands of another component. For this it is necessary to compare the `used` properties of the "client" component with the `provided` properties of the "server" component.

In the example such an analysis becomes necessary between the `VideoServer` and the `UserManager`. The `VideoServer` requires the response time for `checkPassWd` to be less than 470 ms, while the `UserManager` provides a response time for `checkPassWd` of less than 400 ms. Analysis can conclude that the offered response time constraint is stronger than the required constraint. Thus, the two components can safely be plugged together.

After a refinement of the response time, the situation may well be different. For example, the application designer may have chosen variation 4 (cf. Fig. 4) for refining response time in `VideoServer` and variation

---

[3] `fine` is the name given to the context model by the measurement designer (see Sect. 6.2). The line showing this name can be hidden from diagrams.

1 for `UserManager`. This corresponds to the principle of locality in component-based software engineering: no component specification makes constraining statements about anything beyond its own boundaries.

Trying to analyse whether these two components can cooperate yields no result as the two variations cannot be compared directly. This is not a shortcoming of the analysis, however, but a lack of the model. The application designer needs to add information about the delay of the communication channel between the two components. In other words, the refinement of the non-functional constraint prompts a refinement in the functional model: the designer needs to consider aspects of the communication between the components.

Communication between components is modelled using connectors in Architecture Description Languages (ADLs) [22]. This concept can be extended to provide non-functional properties of communication (cf., e.g., [9, 16,35]). In our example, we might model the container-induced delay for communication to be 50 ms. Given this additional information, the analysis tool should then be able to conclude that the two components can safely be used together. This concludes Step 4 as described in Sect. 2.

In order to allow the application developer to concentrate on the business logic of his application, it seems reasonable to provide him with a library of connectors for different aspects of the container and of distribution. He would simply select an appropriate connector from this library—or build a chain of connectors to combine non-functional effects of different connectors like distribution and encryption—and plug it into his model.

4 The measurement designer's view

In the previous section we have had a look at the application designer's view. Now, let's have a look behind the scenes. This is where the measurement designer has done his work to make handling of non-functional properties easy for the application designer. He has specified individual measurements using CQML$^+$, defined context models for each measurement specification and level of abstraction, and performed transformations to provide measurement specifications at different levels of abstraction. This corresponds to Step 1 in Sect. 2.

Each CQML$^+$ specification—and in particular each definition of a quality characteristic—is written relative to what in [1] is called a computational model. We prefer the term *context model*, as it is really a model of the context of the characteristic definition—that is, it comprises the elements necessary for specifying the semantics of the characteristic. For each context model and each com-

ponent model to be used, there needs to exist a mapping relating the concepts of the component model to concepts in the context model. For each concept of the component model (e.g., the concept of component itself) we need to identify the concept in the context model which represents it.

As we have shown in Sect. 3, at different stages in the development cycle it is helpful to use characteristics defined at different levels of abstraction. In order for this to be possible, we need to define context models at all these levels of abstraction. In effect, each context model represents the specification of a specific level of abstraction. This is different from what was proposed in [1], where one computational model was used for every CQML specification, independent of level of abstraction. Instead, we use multiple context models to represent different levels of abstraction.

Figures 6 and 7 show two examples of context models. Figure 6 shows a rather coarse—or more abstract—context model. All that one can talk about are components, interfaces, and operations on the static side and component instances and operation calls between instances on the dynamic side. For each operation it is possible to access the history of invocations of this operation. Each operation call connects two operations, one in the used interface of the calling component instance (`caller`) and one in the provided interface of the called component instance (`callee`). Although only structure is shown in the figure, each context model also has a behavioural aspect captured in a transition system. These specifications have been left out for lack of space. The term 'dynamic' in the diagrams refers to classes of which instances are created in the course of executing the transition system, while 'static' refers to those classes whose instances remain fixed over a complete run.

Simple as it is, this context model already allows us to define the response time of an operation. Listing 2 shows the corresponding CQML$^+$ definition. The `domain` clause defines response times to be real values given in milliseconds. The `values` clause defines how response time values can be measured. It relates to the context model, using the start and end time of an operation call, which are stored in the attributes `start` and `end`, respectively.

The context model in Fig. 7 is much more detailed. It represents a much lower level of abstraction. In particular, it contains event sequences `SE` and `SR` for each operation. For each operation in a used interface, `SE` (short for "service emission") contains events fired whenever a request for an operation call was issued by the calling component; `SR` (short for "service reception") contains one event per result that was received by the calling
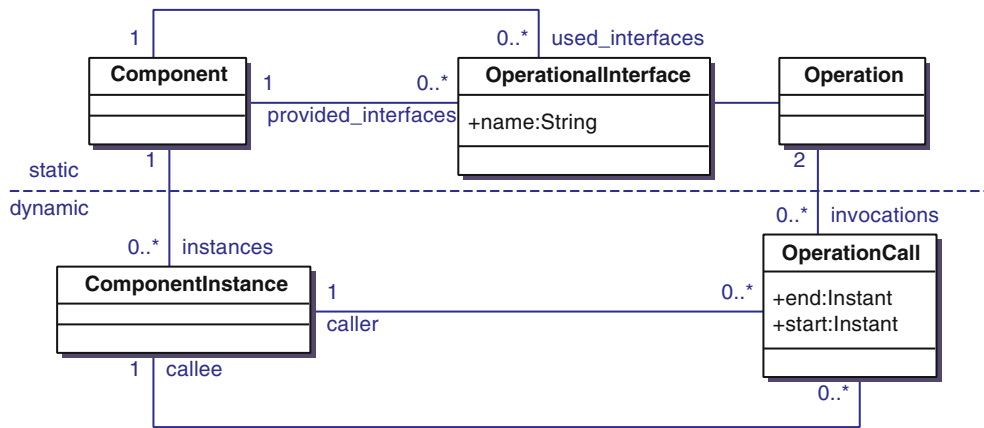
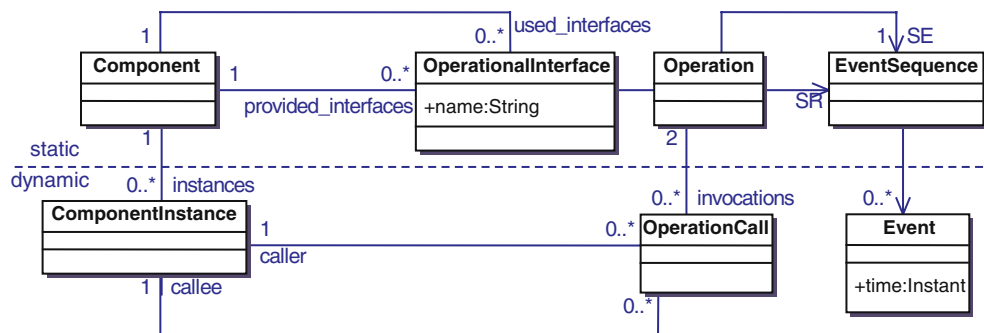**Fig. 6** Abstract context model

Fig. 7 More specialized context model

Listing 2 Abstract response time definition

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

       values: op.invocations ->last().end - op.
           invocations ->last().start;
6    }
```

component. On the other hand, for each operation in a provided interface, SR contains one event per request received, and SE one event per result sent out from the called component. This context model is already very close to Aagedal's [1] computational model.

To perform the interactive refinement described in Sect. 3, we must specify the transformation between these two models. We need to say for each model element in the coarser model which model element(s) it should be mapped to in the finer model. This can be specified using a transformation language based on XML [5]. The transformation language and algorithm will be explained in more detail in Sect. 5.

After designing the finer context model and the transformations, the measurement designer who specified the response time characteristic in Listing 2 uses a transformation tool to apply the transformations to his specification of response time, and to generate refined versions of response time for the more detailed context model. Section 6.3 discusses in detail the implementation of the transformation tool.

Listing 3 shows two of the four resulting versions of response time.[4] Note that the numbers appended to the characteristics' names correspond to the numbers from Fig. 4. The relationship between the more abstract response time definition and the newly created refined versions is stored as another transformation in the transformation specification. It remains the task of the measurement designer to give a clear textual explanation of the differences between the various types of response time, so that they can be used easily by an application designer. Of course, the measurement designer can also define additional measurements which could not be defined at the higher level of abstraction.

---

[4] The remaining two versions have been left out for lack of space.

Listing 3 Refined versions of response time definition

```
     —— Time between receipt of request and sending of
          response
     quality_characteristic response_time1 (op:
          Operation) {
       domain: numeric real [0..) milliseconds;
4
       values: —— The following has been substituted for
              —— op.invocations->last().end
              (let op1 : Operation
                      = op.invocations->last()
9                         .operation
                          ->select (
                            operationalInterface
                            .component
                            .provided_interfaces
14                          ->contains (
                              operationalInterface
                              )
                          )
                 in
                 op1.SE->last().time()) —
              —— The following has been substituted for
19            —— op.invocations->last().start
              (let op1 : Operation
                      = op.invocations->last()
                          .operation
                          ->select (
24                          operationalInterface
                            .component
                            .provided_interfaces
                          ->contains (
                              operationalInterface
                              )
                          )
29               in
                 op1.SR->last().time());
     }


34   —— Time between sending of request and receipt of
          response
     quality_characteristic response_time4 (op:
          Operation) {
       domain: numeric real [0..) milliseconds;

       values: —— The following has been substituted for
39            —— op.invocations->last().end
              (let op1 : Operation
                      = op.invocations->last()
                          .operation
                          ->select (
44                          operationalInterface
                            .component
                            .used_interfaces
                          ->contains (
                              operationalInterface
                              )
                          )
49               in
                 op1.SR->last().time()) —
              —— The following has been substituted for
              —— op.invocations->last().start
              (let op1 : Operation
54                      = op.invocations->last()
                          .operation
                          ->select (
                            opera\—tionalInterface
                            .component
59                          .used_interfaces
                          ->contains (
                              operationalInterface
                              )
                          )
                 in
                 op1.SE->last().time());
64   }
```

Furthermore, application designers may require additional refinement patterns which they could communicate back to the measurement designer who would then provide the appropriate context models and transformation specifications.

5 The transformation language

In the previous section we explained that the measurement designer specifies context models as well as the transformations between them, and uses these transformations to generate characteristic specifications at lower levels of abstraction from specifications at higher levels of abstraction. In this section we will look at the language used to describe the transformations as well as at the actual transformation algorithm.

We have defined an XML-based language for the specification of transformations between context models. It expresses mappings between elements of a more abstract and a more detailed context model. An excerpt from the transformation descriptor for our two sample context models can be seen in Listing 4. Note that some of the text in the transformation specification is OCL code. These pieces are code templates which are to be substituted for pieces of expressions in the more abstract model. We have omitted most of the trivial mappings, giving only the mapping for Component as an example. It can be seen that we distinguish two kinds of transformations:

1. *Classifier transformations* (cf. Line 2 in Listing 4) which are essentially type replacements.
2. *Feature transformations* which replace features—such as, attributes, navigations, or operation calls—from the more abstract model with expressions in the finer model. The measurement designer specifies expressions determining the value of features from the coarser context model in terms of the elements of the finer context model. This is used for features which are no longer present in the finer model. For example the transformation definition on Line 8 defines expressions that can be used to determine the value denoted by the start attribute of the OperationCall classifier in the coarse model. The fact that there are two target expressions indicates that this aspect of the model has been enriched with information in the refinement.

We are aware that there may be other types of transformation, but so far all examples we have looked at could be successfully handled with these two types. When transforming the specification of a characteristic, the

Listing 4 Sample transformation descriptor. The XML code has been slightly simplified to enhance readability

```
1    <refinement_xform from="coarse.xmi" to="fine.xmi">
       <transform classifier="Component">
         <target classifier="Component"/>
       </transform>

6    ...

       <transform feature="OperationCall::start"
           ownerRef="owner">
         <target_expression>
           let op : Operation
11              = owner.operation->select (
                    operationalInterface
                      .component
                      .provided_interfaces
                      ->contains (operationalInterface
                        )
                  )
16           in
                op.SR->last().time()
         </target_expression>
         <target_expression>
           let op : Operation
21              = owner.operation->select (
                    operationalInterface
                      .component
                      .used_interfaces
                      ->contains (operationalInterface
                        )
                  )
26           in
                op.SE->last().time()
         </target_expression>
       </transform>

31     <transform feature="OperationCall::end" ownerRef=
           "owner">
         <target_expression>
           let op : Operation
                = owner.operation->select (
                    operationalInterface
                      .component
36                    .provided_interfaces
                      ->contains (operationalInterface
                        )
                  )
             in
                op.SE->last().time()
41       </target_expression>
         <target_expression>
           let op : Operation
                = owner.operation->select (
                    operationalInterface
                      .component
46                    .used_interfaces
                      ->contains (operationalInterface
                        )
                  )
             in
                op.SR->last().time()
51       </target_expression>
       </transform>
     </refinement_xform>
```

transformation tool applies the transformations described by each `transform`-tag to each usage of the element/feature specified by attribute `element` in the specification of the characteristic. Because there is some indeterminism in the mappings, the transformation will result in more than one version of response time. In one generated version the choice of target expression must be

consistent for every `transform`-tag. Multiple occurrences of a feature in the original expression must be replaced by the same expression in the refined version.

There is some difference in the way classifier and feature transformations are handled. While classifier transformations are simple replacements of types by another type, feature transformations require some more work: here the transformation rule defines a template expression that is to be substituted for the expression referencing the feature. Each expression referencing some feature has the general form `owner.feature`, where `owner` can be any expression and `feature` is the name of a feature. During the transformation, the owner part of this expression is inserted into the target expression at the places indicated by the identifier declared to be the `ownerRef` (see Line 8 of Listing 4) before the whole expression is substituted. Another issue to be taken into consideration is uniqueness of names. Names defined in the target expression template may clash with names defined or visible in the expression that is being transformed. To avoid such clashes, all names defined in `let`-statements in the target expression template are appended the smallest positive number that makes them unique.

The response time specifications in Listing 3 have been generated from the definition in Listing 2 using the algorithm and the sample transformation descriptor above. The numbers correspond to Fig. 4. Note how the `start` and `end` expressions have been replaced by the corresponding target expressions. All combinations of target expressions have been used in generation. However, to save space, only the two most important versions have been included in this paper.

6 A toolkit for the specification and refinement of non-functional properties

Specification of non-functional properties of application systems is complex and error-prone. It is therefore desirable to provide tool support to the application developer and the other roles participating in the development of an application. In this section, we present a toolkit which supports the specification concepts we have been discussing so far.

We require the tool to provide support for all roles in the development process. At the same time, we want to be able to provide tailor-made support for each role, and to be flexible to add further functionality as our research progresses. We have therefore chosen to create a kit of largely independent tools, which interoperate using a common repository. Figure 8 shows an overview of the

Fig. 8 Overview of the
COMQUAD toolkit
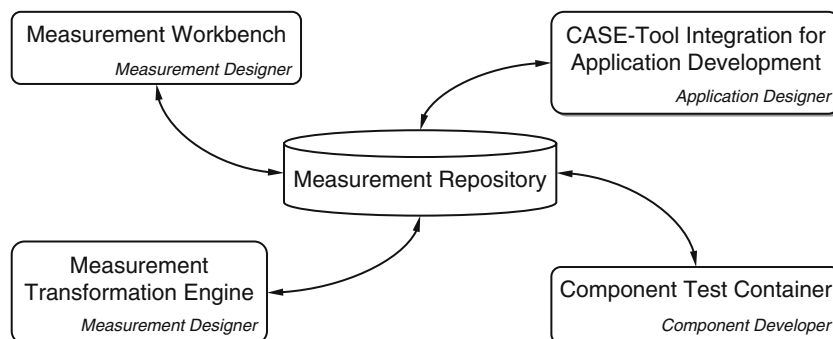
toolkit we have realised so far. The individual tools are as follows:

Measurement workbench: a tool supporting the definition of new measurements, as well as their classification into taxonomies of non-functional properties, by the measurement designer. This classification can be used to simplify search and retrieval later on.

Measurement transformation engine: a tool supporting the refinement of measurements as described in the previous section.

Component test container: a tool for component developers which can be used to determine the non-functional properties of actual component implementations.

CASE tool integration for application development: this is the counterpart to the measurement workbench and the transformation engine. It provides access to the measurements previously defined by the measurement designer to the application designer, and enables him to use these measurements to constrain models of the application under development.

All tools interact using a central *measurement repository*, which contains both the CQML$^+$ specifications and some additional information to simplify search and retrieval of measurements. This repository has been developed using the Netbeans Metadata Repository (MDR) [21], which is a repository generation and management engine based on meta-modelling [8], the OMG's Meta-Object Facility (MOF) [28], and the Java Metadata Interfaces standard (JMI) [10]. We have developed a MOF-based meta-model for CQML$^+$, and generated an MDR repository from this meta-model. A CQML$^+$-parser reads CQML$^+$ specifications in plain text and transforms them into instances of the abstract syntax in the repository.

The following subsections first discuss the structure of the repository, and then in turn look more closely at the tools provided to the measurement designer and

the application designer. The component test container is not treated in this paper; [24] explains some of the required concepts and presents a prototype based on Enterprise Java Beans (EJB) components.

6.1 A meta-model for CQML$^+$

Because we are using a meta-model–based repository framework, we needed a meta-model for non-functional specifications. This meta-model is comparatively complex, because it supports the complete range of CQML$^+$-specifications, and additionally provides ways of structuring specifications such that they are easily accessible for application developers. The latter structures are not relevant in the context of this paper. Hence, we will not discuss them in further detail. Large parts of the meta-model have originally been developed in [39]. Additional work has been performed in [4,23].

The meta-model is composed of four packages, which can be seen in Fig. 9. Context models are represented by instances of the classes in the `contextmodels` package. Building on this, the `applicationmodels` package provides model elements for the connection to actual application models.

Actual CQML$^+$ specifications are represented as instances of the elements in `CqmlP`. Finally, the `management` package contains classes for the hierarchical classification of measurements, and the representation of refinements.

Figure 10 shows the key classes of the CQML$^+$ meta-model. All first-class elements are instances of sub-classes of `ModelElement`. These are: categories, profiles, resource specifications, quality statements, and quality characteristics. For the purposes of this paper, the quality characteristics are the most interesting. They are composed from an invariant, a domain clause, defining the type, and a values clause giving the semantics in the form of an OCL expression. Most of these elements can be seen in their concrete syntax in the example Listing 2. OCL expressions are stored in the
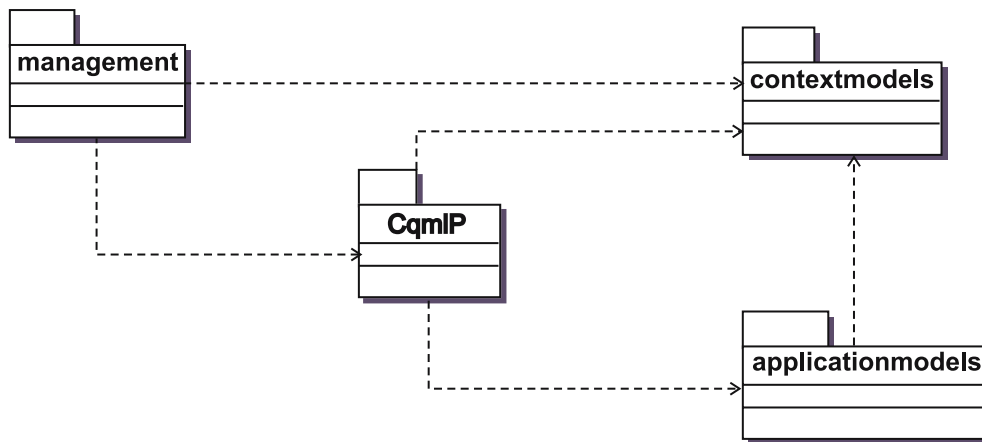
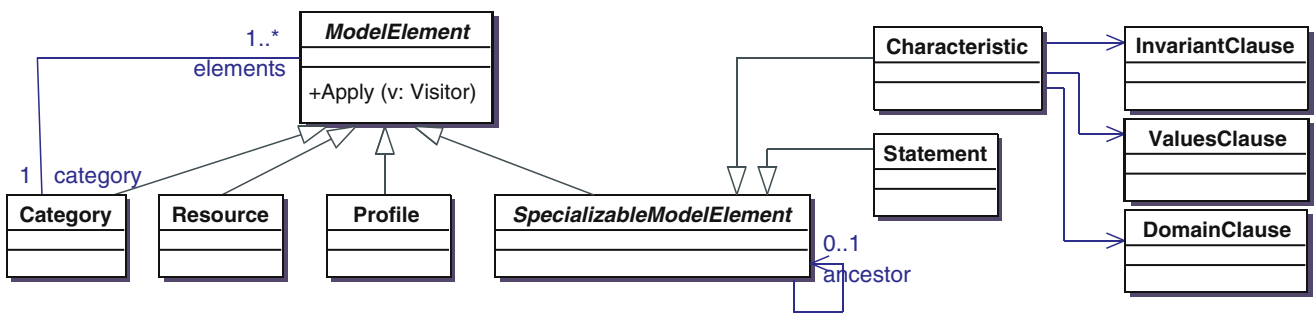**Fig. 9** Packages of the measurement meta-model

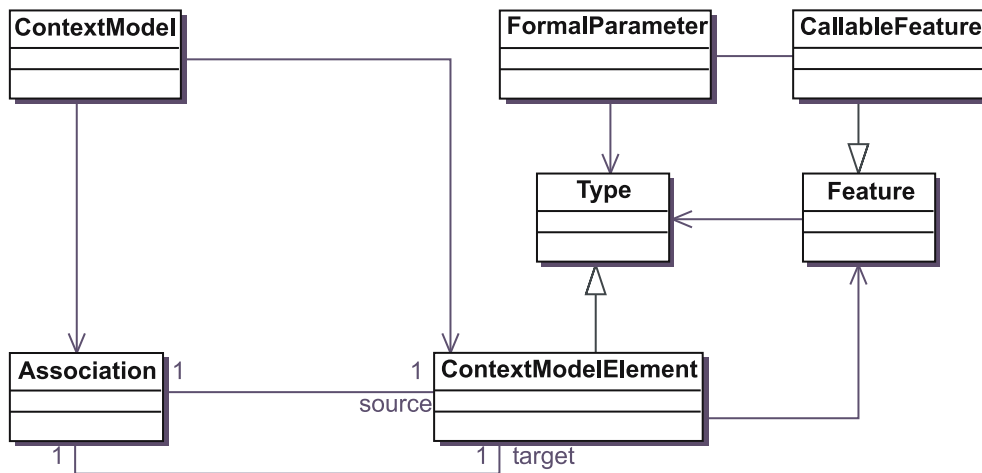**Fig. 10** Core classes of the CQML$^+$ meta-model

Fig. 11 The meta-model for context models

repository as instances of a slightly adjusted version of the OCL 2.0 meta-model. These adjustments where necessary, because OCL as it is used in CQML$^+$ does not refer directly to UML models, but to context models, instead. Every model element provides an operation `apply (v: Visitor)` which can be used to add functionality to the meta-model by implementing additional visitors [11].

Context models are represented in a separate package of the meta-model. Figure 11 shows the key classes of this package.[5] Each context model has its own instance of `ContextModel`, which serves to bundle all context

---

[5] We only show the classes modelling the static parts of the context model. The dynamic part requires additional structures for its representation.
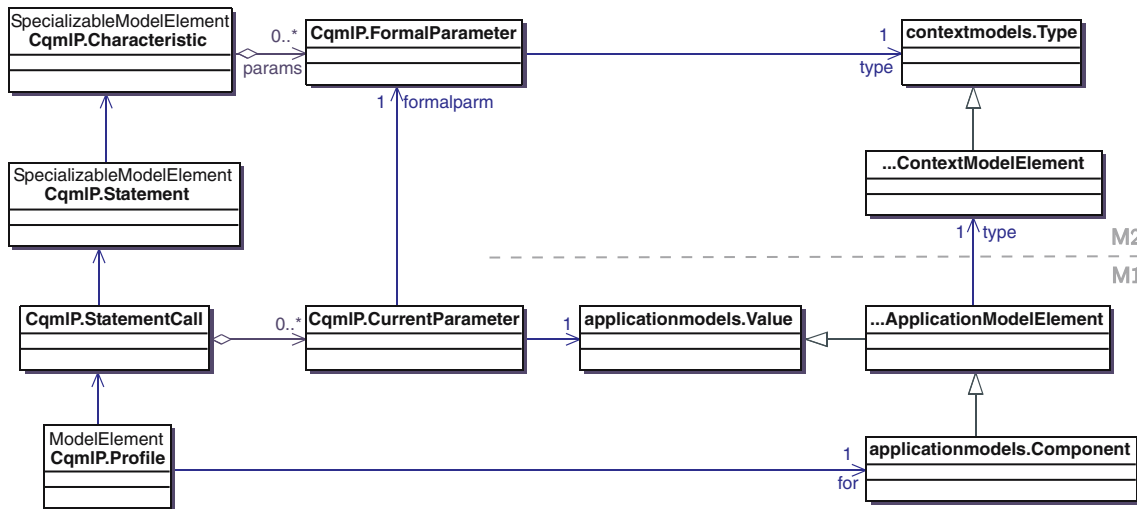
Fig. 12 Elements for the complete specification. It can be seen that specifications reference elements from the context model (i.e., the functional meta-modelling layer) as well as the application model level (i.e., the functional model layer). The dashed line indicates the boundary between these meta-modelling layers

model elements and associations in this context model. Each characteristic has a link to its context model. The actual elements of a context model are all instances of `ContextModelElement`—we have used class boxes in the example context models in Figs. 6 and 7. Context model elements may be connected through generalisations (not shown in the figure) or `Associations`. Context model elements, like classes in UML, have features, which can be callable features. Callable features are operations on meta-model elements that can be used from characteristic definitions. They should not be confused with business operations, which are application specific. An example for a callable context model feature is an operation `eventsInRange (time: int)` defined on `EventSequence`, which counts the number of events in the last `time` milliseconds. Features have a `Type`, which can be another context model element. In addition to context model elements, types can also be the standard primitive types.

Figure 12 summarises how a CQML$^+$ specification is represented in the meta-model. The figure is only a conceptual overview, there are actually some more classes involved, but we have abstracted from these for the sake of clarity. The left side of the figure shows the abstract syntax of the actual CQML$^+$ specification, beginning with the `Characteristic`, which is being used in the `Statement` to formulate constraints over measurements. Theses statements are then invoked (using `StatementCall` instances) from a `Profile` to bind the constraints to a specific `Component`. Components are referenced by their corresponding `Application-ModelElement`, which is a view on the functional

application model. Formal parameters of characteristics (and statements) reference a `ContextModelElement` as their `Type`, current parameters reference an `ApplicationModelElement` as their `Value`. Looking at this scenario from the perspective of functional specification using UML, we find that the context model layer roughly corresponds to the M2 layer of the functional specification (i.e., the UML meta-model), and that the application model layer roughly corresponds to the M1 layer (i.e., the actual UML application model). This means that the CQML$^+$ specification bridges meta-model layers. Indeed, this is not an issue that arises because of our specific way of modelling the abstract syntax of CQML$^+$, but it rather is a general issue with languages like CQML$^+$. Because they want to support the definition of new measurements in the language itself, they need to provide access to the M2 layer. Because they want to support constraints on actual models, they need access to the M1 layer. This mixing of meta-levels considerably adds to the complexity of a CQML$^+$ specification. The new role of *measurement designer* solves this problem by separating the usage of the two meta-modelling layers, without compromising the flexibility of the language.

6.2 Supporting the measurement designer: the measurement workbench

To support the *measurement designer* in creating and maintaining libraries of measurements, we developed the *measurement workbench,* a tool based on the reposi-
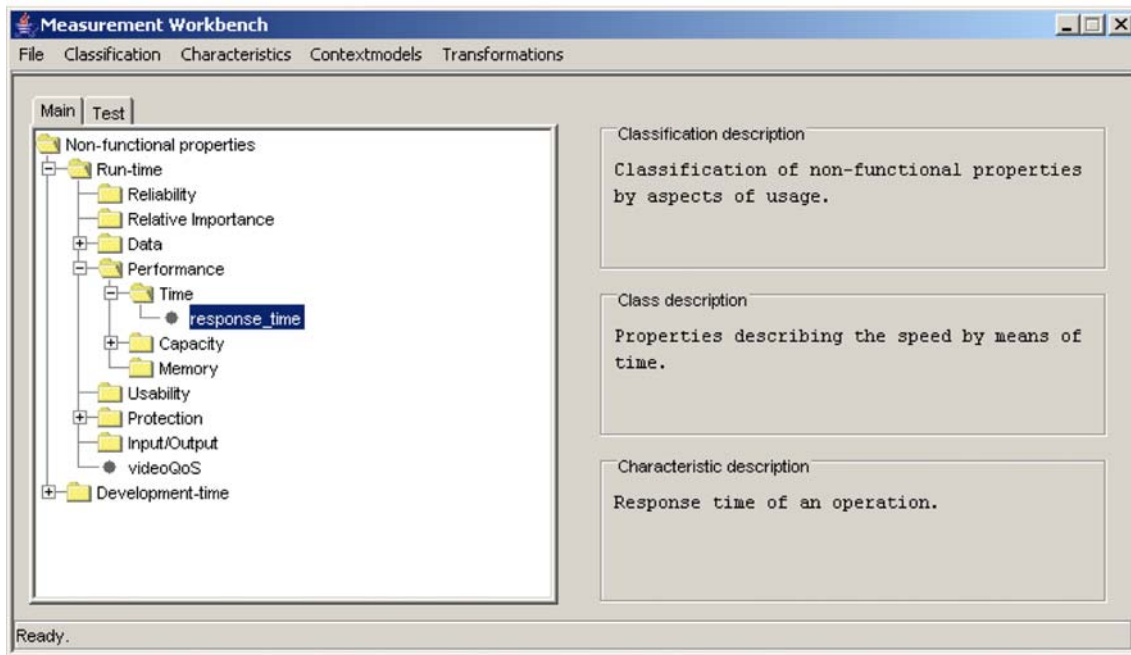
Fig. 13 Screen shot of the measurement workbench

Table 1 Mapping of UML concepts to context model meta-elements as performed by the measurement workbench when importing context model definitions

UML concept	Context model meta-element
Class	ContextModelElement
Attribute	Feature
Operation	CallableFeature
Association	Association

tory structures presented in the previous section. Figure 13 shows a screen shot of the measurement workbench's central dialogue. The dialogue shows a structured overview of the measurement repository in the left tree panel. On selection of a measurement, corresponding information are displayed in the right pane.

To add a new measurement, the measurement designer first needs to provide to the tool the context model to be used. The measurement designer can use an arbitrary case tool to design a UML representation of the context model, which he then saves as an XMI file. The measurement workbench automatically recodes the context model into an instance of the context model meta-model shown in Fig. 11 when the measurement designer imports the context model using the corresponding menu option. Table 1 shows how the UML concepts are translated. The table completely lists all concepts recognised by the measurement workbench. All other contents of the context model—in particular the dynamic specification—are ignored.

Now the measurement designer can add new measurements by selecting 'New Characteristic' from the 'Characteristics' menu. This opens the dialogue window in Fig. 14 where the measurement designer can enter all information regarding the new measurement. The top compartment contains information which is used to classify the measurement for later retrieval. Based on work reported in [15,20,37] we have defined a set of generic dimensions that can be used to characterise measurements. We plan to report on this characterisation in another publication. In the second compartment, the measurement designer enters an intuitive description capturing the meaning of the measurement in terms understandable by the application designer. It is this description that the application designer will use mainly to decide for a measurement to use when specifying the non-functional properties of his application. The next section is where the measurement designer puts the complete CQML$^+$-specification of the measurement. Last but not least, the measurement designer selects the context model relative to which he has defined the measurement. Once the measurement designer has finished entering the definition of the new measurement, he is ready to store the information in the repository. Instead of entering each measurement via the graphical user interface, the measurement designer can also describe them in an XML file (based on a simple DTD we defined) and have the measurement workbench read in this script. This is particularly convenient when large

Fig. 14 Screen shot of the measurement workbench's measurement editor dialogue
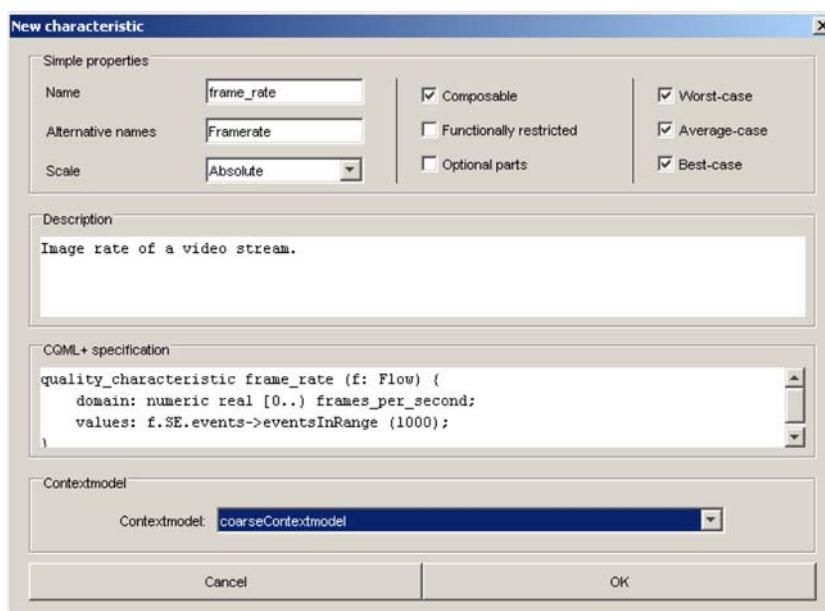
amounts of measurement specifications must be put into the repository.

6.3 Supporting transformation: the measurement transformation engine

Now, let us discuss how we support the transformation of measurements described in previous sections of this paper. We have implemented the measurement transformation engine, a tool physically integrated into the measurement workbench to allow the measurement designer to perform measurement transformations. We will begin by examining the steps the measurement designer performs to transform a measurement. Further on, we will discuss some aspects of the implementation of the transformation.

To perform the response time transformation, the measurement designer must first enter the abstract measurement definition from Listing 2. As discussed above, he needs to load the context model in Fig. 6 into the workbench and then add the measurement definition, making sure it is connected to the context model just loaded. Next, he loads the new context model (cf. Fig. 7) into the workbench, and then uses the 'Transform' option from the 'Transformations' menu to define the transformation. This opens another dialogue where the measurement designer selects the source measurement, target context model and transformation descriptor, and finally starts the transformation.

What happens behind the scenes, when the measurement designer starts the transformation? First, the transformation descriptor is read into the workbench and represented internally as an instance of a simple transformation description meta-model. The OCL parts (i.e., the target expressions of feature transformations) are parsed into instances of the OCL 2.0 meta-model using the Dresden OCL toolkit parser[6] [14] and checked for type correctness against the target meta-model. The actual transformation is then performed by a set of visitors (see Fig. 15), which we use to traverse the abstract syntax graphs representing the original specification and the transformation descriptor and to incrementally build the transformation result. Just as there are different input documents, there also exist different visitors, one for each type of input: The `TransformationVisitor` is used to extract the target expressions from the transformation descriptor, and to manage alternative transformations for the same source element. The `CharacteristicVisitor` walks through the complete measurement definition and extracts the individual parts to be transformed. It uses `OCLVisitor` instances to actually transform OCL expressions. Finally, the `CharacteristicGenerationVisitor` goes over all the resulting fragments and composes them into the resulting measurement definition(s).

Determining when the same alternative must be used in transforming two subterms and when different alternatives may be used can be rather difficult. The objec-

---

[6] For technical reasons this is currently only partly true. At the time of writing this paper, the OCL 2.0 parser of the toolkit is in the last throes of development. We are currently using an older version of the parser, which parses OCL 1.x expressions and creates abstract syntax trees. We are, however, planning to switch to the new parser when it becomes available. For the sake of clarity, we have chosen to represent the algorithm as though it already used the OCL 2.0 abstract syntax format.
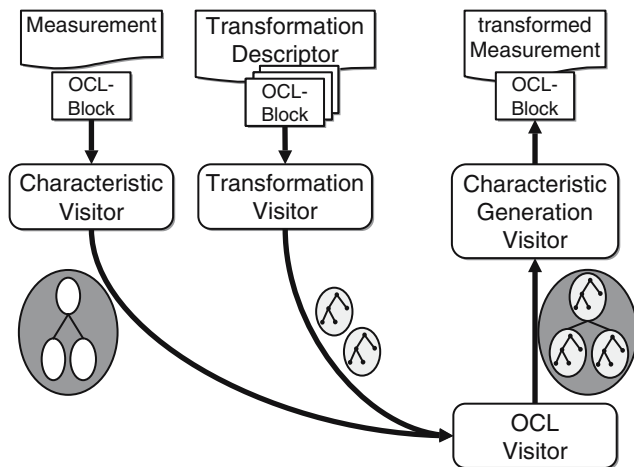
Fig. 15 Visitors used in the transformation of measurements. Abstract syntax graphs (ASG) of OCL blocks from the measurement definition are translated using the ASGs from the target expression definitions

Table 2 Mapping of graphical model elements to CQML$^+$

Graphical Model Element	CQML$^+$ Code

<<characteristic>> response_time / value<500	`quality GeneratedStmtID`
	`        (op: Operation) {`
	`    response_time(op) < 500;`
	`}`

<<characteristic>> response_time / value<500 / [for op]	`profile GeneratedProfileID`
	`        for ComponentName {`
	`    provides GeneratedStmtID (op);`
	`}`

tive is to ensure that references to the same object in terms of the abstract context model will be transformed to references to the same object in the refined context model. Because to perform this analysis in full is prohibitively complex, we perform analysis on the type level only; that is, for one transformation result we transform references to the same feature using the same target expression alternative.

6.4 Supporting the application designer: CASE tool integration

The application designer's task is to provide a sufficient specification of software systems. For this, he needs tool support for the design of non-functional system aspects, as well as functional aspects. As an example, we have extended ArgoUml [3] to provide the means for modelling non-functional properties using the concepts explained in this paper. In this section we will describe the application designer's view of the measurement repository and we will also have a closer look at the interaction between the internal model representation and the graphical notation used by the application designer.

As shown in Fig. 16 we use a layered architecture to separate the graphical interpretation and the internal representation of our model elements. The graphic layer provides the user interface. Designing non-functional aspects of applications is a complex task, which is best approached from different viewpoints. The application designer can choose between different types of diagrams:

Specification diagrams for specifying the functionality of individual components,

Implementation diagrams for the specification of functional and non-functional properties of component implementations,

Requirement diagrams for the description of non-functional requirements of an implementation of a component specification,

Assembly diagrams for describing the connections between components forming an application system.

Both the implementation diagram and the requirement diagram are used to specify non-functional properties of the application. As mentioned before, the application designer can use the information about non-functional measurements previously defined by the measurement designer and placed in the measurement repository. To this end, we provide a search interface as part of our workbench. It provides several retrieval strategies and is responsible for their coordination and execution.

The second main part of the **COMQUAD** CASE tool is the model layer. Here, we have to distinguish between functional and non-functional model elements. Functional elements model a specific functionality whereas non-functional elements describe non-functional properties of the modelled functionality. To realise the functional model, we reuse the NSUML-based API [25] provided in the ArgoUml project. To support non-functional model elements we integrated the measurement repository which was developed using MDR. ArgoUml can only display model elements defined in NSUML. To overcome this and to connect the two repositories, we defined proxy elements working as adapters. So we are able to visualize the non-functional elements defined on the MDR-side of the repository. In addition, we need a delegate mechanism in the other direction. Application model elements have a delegate association to a

Fig. 16 Architecture of the
COMQUAD CASE tool
(development based on
ArgoUml)
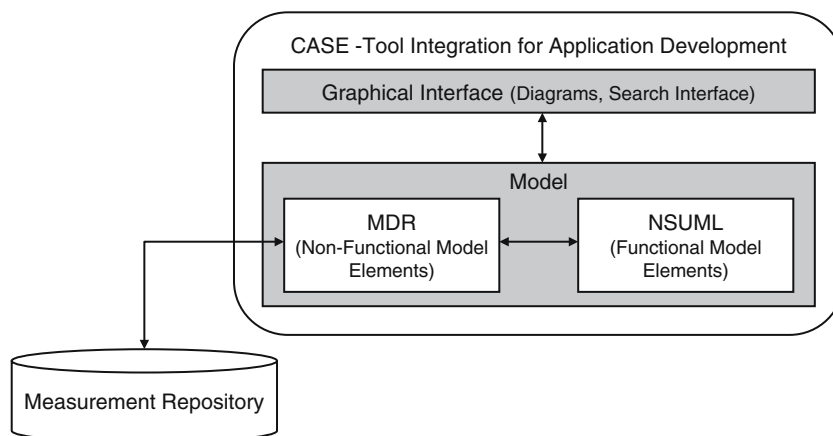
NSUML model element to realize the connection between CQML$^+$ specification and application model.

Finally, we will have a look on the mapping between the graphical model elements and the internal model representation. Let us take our example. Imagine, the application designer wants to constrain an operation of a component port with a response time less than 500 ms. For this, we provide an appropriate menu option for placing new constraints on a diagram. When the application designer selects this option, a dialogue opens and he is able to search in the measurement repository for a characteristic matching his query. In our case he selects `response_time`. The tool will then draw the graphical element as seen in Table 2 with the value set to zero. The application designer can now enter the appropriate value and select '<' as the comparison relation. Last he has to connect the constraint to the selected port and select the appropriate operation. Internally, the repository transforms this model element to the CQML$^+$ code in the second column of Table 2. To actually store it in the repository structure the code is mapped to our meta-model. The CQML$^+$ definition of the `response_time` measurement is taken directly from the measurement repository. The association between the non-functional and the functional model element defines the business operation and the port to which the constraint is applied. This can be seen in the second row of Table 2.

7 Related work

VEST [38] is a design toolkit for component-based systems which focuses on non-functional properties. It uses an extended notion of aspects [18] to allow *en-bloc* modifications to the non-functional specifications of individual components, thus effecting changes to the global non-functional specification of a system. Our work does not use aspects, although they could probably be combined with our approach. The major difference between our approach and the VEST approach is that we use context models at different levels of abstraction, while all work in VEST is tied directly to the component model provided by the target environment (Boeing's Bold Stroke in this case).

Model-Driven Architecture (MDA) [19] is an important current development driven mainly by the Object Management Group (OMG). Transformation between models is at the heart of this technology. Our work fits well into this larger view, although to the best of our knowledge we are the first to apply model transformations to measurement refinement. The new Query/Views/Transformations specification for which a request for proposals [26] has been issued by the OMG will be of great importance for our work. We can use the concept of views to relate context models and application models, and we can use the transformation technologies defined to simplify the implementation of our transformation tool. Simmonds et al. [36] describe an MDA technology for the creation of QoS-aware applications. The main focus is on the transformation of application models and weaving-in of non-functional aspects. Refinement of non-functional specifications is not considered. CoSMIC [12] is an MDA tool suite for supporting model-driven middleware. The tool supports only application development, deployment and configuration, but no refinement of non-functional models.

QCCS [34] describes a methodology for the development of contract-aware components. This methodology covers only the application design. Our refinement step can be used both in requirements analysis and application design. It is embedded in a process which reckons with non-functional properties from requirements to code [2,13]. QCCS also provides UML model transformation based on aspect-oriented design [17]. The

authors of [34] propose to weave non-functional constraints and functional aspects at application modelling time. In contrast, our methodology keeps non-functional and functional aspects separate until implementation time.

Another systematic approach to deal with non-functional system requirements (NFR) is the NFR framework [7]. It proposes a modelling framework for non-functional requirements inspired by the UML. NFRs are decomposed into sub-requirements (so-called *softgoals*) following different criteria: by type or by topic. An example for decomposition by type is the splitting of performance of a system into time-performance and space-performance. Decomposition by type is thus similar to our classification of non-functional properties. In contrast, decomposition by topic means that the functional system is decomposed into different components and the non-functional requirement for the whole system is now required to hold for some or all components. This is similar to our notion of structural refinement. The use of the NFR framework leads to an analysis of different non-functional requirements and how they depend on each other at the design level. It leaves open, how these requirements can be achieved by an application. Our approach, in contrast, is focused on modelling non-functional properties at design time and, thus, on the realisation of specific non-functional properties through a specific application design.

8 Conclusions and open questions

Non-functional properties must be considered throughout the development cycle of an application system. The application designer creates, and thinks about, functional models at different levels of abstraction. He should be able to do so with non-functional models, too. We have introduced the concept of explicitly defined context models of measurements which explicitly capture the level of abstraction of a measurement. Additionally, we enable tool support for the refinement of non-functional specifications by requiring the measurement designer to define transformations between context models and applying them to measurement definitions. We have discussed the way such transformations are specified and explained in detail the structure of a prototype implementation of a tool kit supporting application and measurement designer.

Furthermore, we have outlined a software development process which separates the roles of measurement designer and application designer. It is the measurement designer's responsibility to specify measurements, context models and transformations between context

models, all of which can then be used by the application designer when developing an application. Thus, the application designer is free to focus on the business logic.

The refinement process prompts for decisions when they are needed. We have indicated two points where this happens: (a) in the actual refinement step, where the application designer needs to choose between different refinements of a measurement, and (b) after a refinement has taken place, when the analysis tool cannot compare constraints on different refinements. In the latter case, the application designer will also need to refine the functional model by making explicit the effect caused by communication between components. We have shown how connectors can be used to model this. Defining these connectors, building libraries, and integrating the connectors into application models is still a research issue, although some approaches can be found in the literature. One important question, among others, is whether the usage of connectors we have sketched for response time also works for characteristics which are not time-related. On a more general note, we would like to propose the interaction of refinements to the functional and the non-functional model as an interesting research area.

It is important to point out, that, although we have explained our approach with two context models only, it is intended to be generic. For any one measurement there could be any number of context models and, correspondingly, any number of different levels of abstraction. How this large number of models can be managed in a way that further reduces the complexity for the application designer and makes choosing the next model for refinement easy, is an area for further research.

In this paper we have proposed an approach for the refinement of non-functional properties at the modelling level. Before this approach can be called practical, further studies are required: (1) we need to study more examples to improve our understanding of the limitations and capabilities of the approach; some related issues have been discussed in [40], (2) we need to provide support for the measurement designer in purge unneeded or meaningless generated refinements; one idea would be to specify dependencies between individual feature refinements so that certain combinations are excluded a priori, and of course (3) we need to perform an extensive case study to prove the applicability and practical gain of our approach.

This paper has focused on measurement refinement. Another important research topic is structural refinement. We plan to investigate this in our future work.

References

1. Aagedal, J.O.: Quality of Service Support in Development of Distributed Systems. PhD thesis, University of Oslo (2001)
2. Aigner, R., Pohlack, M., Röttger, S., Zschaler, S.: Towards pervasive treatment of non-functional properties at design and run-time. In: 16th International Conference on Software and Systems Engineering and their Applications (ICSSEA'03). Paris, CNAM-CMSL, 2–4 December 2003
3. ArgoUML website. http://argouml.tigris.org/
4. Bandelow, D.: Entwicklung einer CQML$^+$-Basisbibliothek(in German) Diplomarbeit, Department of Computer Science, Technische Universität Dresden (2004)
5. Bray, T., Paoli, J., Sperberg-McQueen, C.M., Maler, E.: Extensible markup language (XML) 1.0 (2nd edition) . W3C Recommendation (2000)
6. Bruel, J.-M. (ed) Proc Leading of the 1st international Workshop on Quality of Service in Component-Based Software Engineering, Toulouse, France. Cépaduès-Éditions (2003)
7. Chung, L., Nixon, B.A., Yu, E., Mylopoulos, J.: Non-Functional Requirements In Software Engineering. Kluwer, Dordrecht (2000)
8. Clark, T., Evans, A., Sammut, P., Willans, J.: Applied Metamodelling – A Foundation for Language Driven Development. version 0.1, published on-line at http://www.xactium.com/ (2004)
9. Demairy, E., Anceaume, E., Issarny, V.: On the correctness of multimedia applications. In: 11th EuroMicro Conference. on Real Time Systems. IEEE, June 1999
10. Dirckze, R.: Java metadata interface(JMI) specification, version 1.0. Java Community Process JSR 040 Final Specification, June 2002. http://www.jcp.org/
11. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, Reading (1995)
12. Gokhale, A., Natarajan, B., Schmidt, D.C., Nechypurenko, A., Gray, J., Wang, N., Neema, S., Bapty, T., Parsons, J.: Cosmic: an MDA generative tool for distributed real-time and embedded component middleware and applications. In: Proceedings. ACM OOPSLA 2002 Workshop on Generative Techniques in the Context of MDA. Seattle, WA, November 2002
13. Härtig, H., Zschaler, S., Aigner, R., Göbel, S., Pohlack, M., Pohl, C., Röttger, S.: Enforceable component-based real-time contracts – supporting realtime properties from software development to execution. Real Time Systems (to appear)
14. Hussmann, H., Demuth, B., Finger, F.: Modular architecture for a toolset supporting OCL. In: Evans, A., Kent, S., Selic, B. (eds) Proceedings of. 3rd International UML Conference. 2000 - The Unified Modeling Language. Advancing the Standard. volume 1939 of *LNCS*, pp. 278–293. Springer, Berlin Heidelberg Newyork (2000)
15. Information technology – Quality of Service: Framework. ISO/IEC 13236:1998, ITU-T X.641, 1998
16. Issarny, V., Bidan, C.: Aster: a framework for sound customization of distributed runtime systems. In: International Conference on Distributed Computing Systems. pp. 586–593. IEEE Computer Society, 1996
17. Jézéquel, J.-M., Plouzeau, N., Weis, T., Geihs, K.: From contracts to aspects in UML designs. In: AOSD Workshop on Aspect-Oriented Modeling with UML. Enschede, The Netherlands (2002)
18. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.-M., Irwin, J.: Aspect-oriented programming. In: Akşit, M., Matsuoka, S. (eds.) 11th European Conference. on Object-Oriented Programming, vol. 1241 of *LNCS*, pp. 220–242. Springer, Berlin Heidelberg Newyork (1997)
19. Kleppe, A., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture: Practice and Promise. Addison Wesley, Reading (2003)
20. Malan, R., Bredemeyer, D., Defining non-functional requirements. Bredemeyer Consulting, White Paper. http://www.bredemeyer.com/papers.htm, 2001
21. Matula, M.: Netbeans metadata repository, March 2003. http://mdr.netbeans.org/MDR-whitepaper.pdf
22. Medvidovic, N., Taylor, R.N.: A framework for classifying and comparing architecture description languages. In: Proceedings of. 6th European Software Engineering Conf. together with 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE97). pp. 60–76, Zurich, Switzerland, September 1997
23. Meyer, L.: Werkzeugunterstützung für Verfeinerungen nichtfunktionaler Eigenschaften In German. Großer Beleg, Department of Computer Science, Technische Universität Dresden, August 2004.
24. Meyerhöfer, M., Neumann, C.: TESTEJB – a measurement framework for EJBs. In: Crnkovic, I., Stafford, J.A., Schmidt, H.W., Wallnau, K. (eds.) Proceedings. 7th International Symposium on Component-Based Software Engineering (CBSE'04). number 3054 in LNCS, pp. 294–301. Springer, Berlin Heidelberg New York (2004)
25. NSUML and NSMDF website. http://nsuml.sourceforge.net/
26. Object Management Group. MOF 2.0 query, views, transformations request for proposals. OMG Document, April 2002. URL http://www.omg.org/docs/ad/02-04-10.pdf
27. Object Management Group. UML 2.0 OCL specification. OMG Document, October 2003. URL http://www.omg.org/cgi-bin/doc?ptc/03-10-14
28. Object Management Group. Meta object facility specification. OMG Document, October 2003. URL http://www.omg.org/cgi-bin/doc?ptc/03-10-04
29. Object Management Group. Unified modeling language: Superstructure version 2.0. OMG Document, July 2003. URL http://www.omg.org/cgi-bin/doc?ptc/03-07-06.pdf
30. Object Management Group. UML profile for modeling quality of service and fault tolerance characteristics and mechanisms. OMG Document, June 2004 URL http://www.omg.org/docs/ptc/04-06-01.pdf.
31. Röttger, S., Zschaler, S.: CQML$^+$: Enhancements to CQML. In Bruel [6], pp. 43–56
32. Röttger, S., Zschaler, S.: A software development process supporting non-functional properties. In: Proceedings of. IASTED International Conference. on Software Engineering (IASTED SE 2004). ACTA Press (2004)
33. Röttger, S., Zschaler, S.: Model-driven development for non-functional properties: Refinement through model transformation. In: Proceedings of . <<UML>> Conference, Lisbon, Portugal, 2004. volume 3273 of *LNCS*, pp. 275–289. Springer, Berlin heidelberg Newyork (2004)
34. Sassen, A.-M., Amorós, G., Donth, P., Geihs, K., Jézéquel, J.-M., Odent, K., Plouzeau, N., Weis, T.: QCCS: a methodology for the development of contract-aware components based on aspect oriented design. In: AOSD Early Aspects Workshop. Enschede, The Netherlands (2002)
35. Shaw, M., DeLine, R., Zelesnik, G.: Abstractions and implementations for architectural connections. In: 3rd International Conference. on Configurable Distributed Systems. IEEE Press (1996)
36. Simmonds, D.M., Ghosh, S., France, R.: An MDA framework for middleware transparent software development & quality of service. In Bruel [6], pp. 1–7
37. Sommerville, I.: Software Engineering. Addison-Wesley, Reading (2001)

38. Stankovic, J.A., Zhu, R., Poornalingam, R., Lu, C.: Yu, Z., Humphrey, M., Ellis, B.: VEST: An aspect-based composition tool for real-time systems. In: Proceedings of the. 9th Real-Time and Embedded Technology and Applications Symposium (RTAS'03), Toronto, Canada, pp. 58–69. IEEE Press (2003)
39. Tannhäuser, F.: Konzeption und prototypische Umsetzung eines Spezifikationswerkzeugs für CQML$^+$-Spezifikationen. Diplomarbeit, Department of Computer Science, Technische Universität Dresden, December 2003 (In German)
40. Zschaler, S.: Towards a semantic framework for non-functional specifications of component-based systems. In: Steinmetz, R., Mauthe, A. (eds) Proceedings of. EUROMICRO Conference. 2004, Rennes IEEE Computer Society (2004)

Steffen Zschaler obtained his Dipl.-Inf. from Technische Universität Dresden, Germany, in 2002. He is one of the co-organisers of the workshop on Models for Non-functional Aspects of Component-Based Software (NfC) at the MoDELS conference. Currently he is working as a research assistant in the Software Technology Group at Technische Universität Dresden. His research focuses on formal models of non-functional properties of component-based software.

Authors' Biographies

Simone Röttger received her Diploma in Softwaretechnology from Technische Universität Dresden, Germany, in 2001. She is working as a research assistant in the Software Technology Group at Technische Universität Dresden. Her research interests include software development for component based systems and the integration of non-functional properties in software development processes.