Model-Driven Development for Non-functional Properties: Refinement through Model Transformation

Simone Röttger and Steffen Zschaler

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

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. Our basic idea is the separation of two different roles in the development process: the role of the measurement designer and the role of the application designer. The former provides a library of measurement definitions which is later used by the latter to annotate functional application models with non-functional property specifications. In this paper we define the notion of context models to allow the measurement designer to provide measurement definitions at different levels of abstraction independently of concrete applications.
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 concepts presented in this paper form the basis of a tool which we are currently developing.

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 present the models required by the methodology. Although they are directly applicable to Quality of Service properties only (such as response time, delay, memory usage), we believe that they can be extended to cover other non-functional 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 [8]. A measurement is a mapping from states, objects, or events of a physical system (e.g., an

[1] COMponents with QUantitative properties and ADaptivity at Dresden University of Technology and Friedrich-Alexander-University Erlangen-Nuremberg, Germany; supported by German Research Council; see www.comquad.org

implemented and running application) to a formal system (e.g., 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). By using models of the relevant aspects of target applications—we call these *context models*—in the definition of measurements, these definitions can be made independent of specific applications. 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. Non-functional specifications essentially constrain measurements applied to a functional model of a system. They are therefore application specific. From this duality it follows naturally to define two roles in the development process: The *measurement designer* and the *application designer*. The former creates a library of measurements, which are later used by the latter to annotate application models with non-functional specifications. This allows reuse of measurement specifications defined once.

The basic idea of existing development processes—especially MDA-based [12] approaches—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. We propose to use different context models to represent different *levels of abstraction* for a measurement. Requiring the measurement designer to define transformations between the different context models and applying them to measurement definitions forms the conceptual basis for providing tool support for the application designer's refinement of non-functional specifications. The application designer can then perform 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 [18]. It focuses on modelling issues related to measurement refinement. In Sect. 2 we give a short introduction to our overall development process, which forms 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). 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

Fig. 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

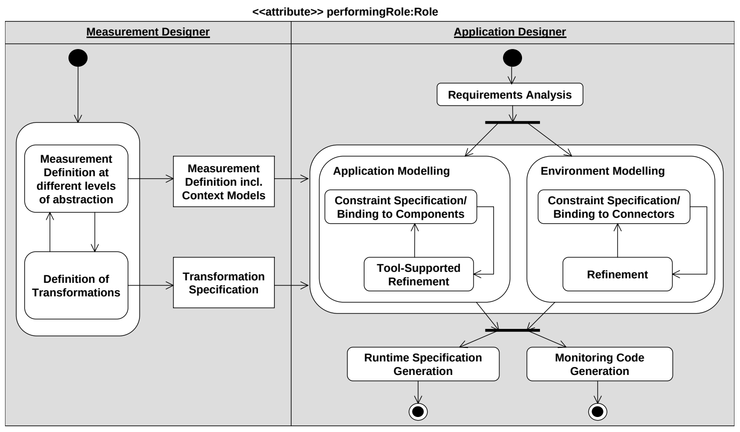

<<attribute>> performingRole:Role
Measurement Designer
Application Designer
Measurement Definition at different levels of abstraction
Definition of Transformations
Measurement Definition incl. Context Models
Transformation Specification
Requirements Analysis
Application Modelling
Constraint Specification/ Binding to Components
Tool-Supported Refinement
Environment Modelling
Constraint Specification/ Binding to Connectors
Refinement
Runtime Specification Generation
Monitoring Code Generation

Fig. 1. Development process for non-functional properties overview

of the components (called “Application Modelling” in the figure) and of the components’ environment (called “Environment Modelling” in the figure), using the concept of connectors for the latter part.

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 including its non-functional properties.

Our process comprises the following steps:

1. Definition of measurements at different levels of abstraction and 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.
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 different kinds of provided refined measurements. These have been previously provided by the measurement designer together with an informal description of each measurement.
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. Besides 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.

We use two specification languages: For functional modelling we primarily use the component model element from UML 2.0 [16] extended with a stereotype for interfaces which allows us to distinguish between operational and streaming interfaces. For the non-functional specification we use CQML$^+$ [17] an extension to CQML [1]. Finally, we use a more compact XML-based representation in the runtime environment.

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, operations or streams 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$^+$ is a textual language comprising both measurement definition and measurement usage. For graphical modelling of measurement definitions we plan to use ideas proposed in [7]. For measurement usage we have defined a graphical notation allowing to attach constraints to parts of the functional model. The internal tool representation uses CQML$^+$, merging measurement definition and constraints into one specification.

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 starts to model 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 non-functional specification needs to be refined, too. 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 some 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 component. Later, he refines the component by decomposing its functionality into several subcomponents. In this step, he will also

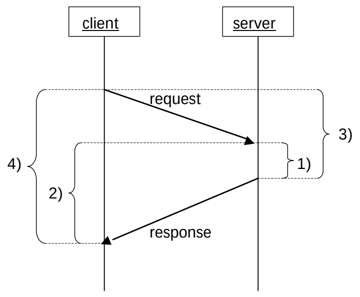


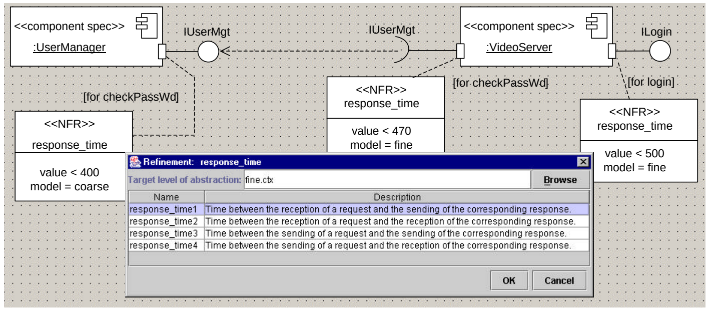
Fig. 2. A sequence diagram excerpt showing different kinds of response time specification

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 characteristic. 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. Fig. 2 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.

This paper focuses on measurement refinement. Structural refinement remains an open issue. As a simple example imagine a login mechanism of a `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 `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. We are working to extend an existing CASE tool to provide support for our graphical notation for non-functional properties.

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 asks 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 3 shows what the screen



Target level of abstraction: fine.ctx

Name	Description
response_time1	Time between the reception of a request and the sending of the corresponding response.
response_time2	Time between the reception of a request and the reception of the corresponding response.
response_time3	Time between the sending of a request and the sending of the corresponding response.
response_time4	Time between the sending of a request and the reception of the corresponding response.

Fig. 3. Sample screen shot

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`'. 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 `check-PassWd` to be less than 470ms, while the `UserManager` provides a response time for `checkPassWd` of less than 400ms. 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. 2) for refining response time in `VideoServer` and variation 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 or not these two components can work together yields no result as the two variations cannot directly be compared. 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) [14]. This concept can be extended to provide non-functional properties of communication (cf., e.g., [5, 9, 20]). In our example, we might model the container-induced delay for communication to be 50ms. 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—or build a chain of connectors to combine non-functional effects of different connectors like distribution and encryption—from this library 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 component 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 4 and 5 show two examples of context models. Figure 4 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.[2] For each operation it is possible to access the history of invocations of this operation. Each operation call connects two operations,

[2] 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.

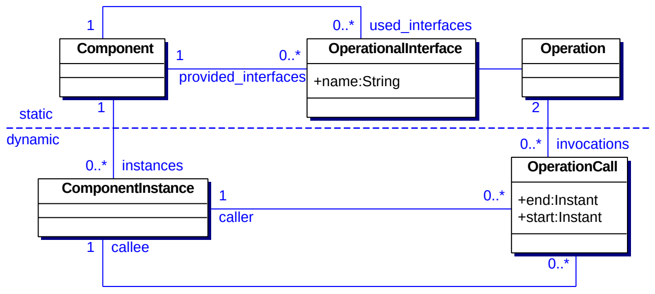


Fig. 4. Abstract context model

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

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

Listing 1.1. Abstract response time definition

one in the used interface of the calling component instance (`caller`) and one in the provided interface of the called component instance (`callee`).

Simple as it is, this context model already allows us to define the response time of an operation. Listing 1.1 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. 5 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 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 have to 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.

The term 'dynamic' in the diagrams refers to classes instances of which are created in the course of executing the transition system, while 'static' refers to those classes whose instances remain fixed over a complete run.

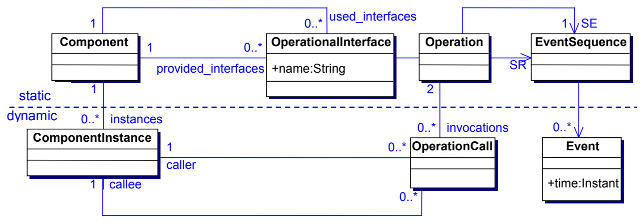


Fig. 5. More specialized context model

This can be specified using a transformation language (as defined for MDA, e.g., in [12]) based on XML [3]. 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 1.1 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. Currently we are working on the implementation of such a tool using the concepts presented in this paper.

Listing 1.2 shows two of the four resulting versions of response time[3]. Note that the numbers appended to the characteristics' names correspond to the numbers from Fig. 2. 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. 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 and 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 algorithm used by the transformation tool.

We have defined an XML-based language for the specification of transformations between context models. It expresses mappings between elements of a more abstract

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

```
-- Time between receipt of request and sending of response
 quality_characteristic response_time1 (op: Operation) {
   domain: numeric real [0..) milliseconds;

   values: -- The following has been substituted for
           -- op.invocations->last().end
           (let op1 : Operation
                  = op.invocations->last()
                         .operation
                         ->select (operationalInterface
                                   .component
                                   .provided_interfaces
                                   ->contains (operationalInterface)
                        )
             in
               op1.SE->last().time()) -
           -- The following has been substituted for
           -- op.invocations->last().start
           (let op1 : Operation
                  = op.invocations->last()
                         .operation
                         ->select (operationalInterface
                                  .component
                                  .provided_interfaces
                                  ->contains (operationalInterface)
                        )
             in
               op1.SR->last().time());
}

...

-- 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
           -- op.invocations->last().end
           (let op1 : Operation
                  = op.invocations->last()
                         .operation
                         ->select (operationalInterface
                                  .component
                                  .used_interfaces
                                  ->contains (operationalInterface)
                        )
             in
               op1.SR->last().time()) -
           -- The following has been substituted for
           -- op.invocations->last().start
           (let op1 : Operation
                  = op.invocations->last()
                         .operation
                         ->select (operationalInterface
                                   .component
                                   .used_interfaces
                                   ->contains (operationalInterface)
                        )
             in
               op1.SE->last().time());
}
```

Listing 1.2. Refined versions of response time definition

and a more detailed context model. An excerpt from the transformation descriptor for our two sample context models can be seen in Listing 1.3. Note that some of the text in the transformation specification is CQML$^+$ 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 1.3) which are essentially type replacements.
2. *Feature transformations* which replace features from the coarse model with expressions in the finer model. The measurement designer specifies expressions giving 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 transformation types, but so far all examples we have looked up could be successfully handled with these two types. When transforming the specification of a characteristic, the transformation tool applies the transformations described by each `transform`-tag to each usage of the element/feature specified by the `element` attribute 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 1.3) 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 1.2 have been generated from the definition in Listing 1.1 using the algorithm and the sample transformation descriptor above. The numbers correspond to Fig. 2. 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.

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

  ...

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

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

Listing 1.3. Sample transformation descriptor. The XML code has been slightly simplified to enhance readability

6 Related Work

VEST [22] is a design toolkit for component-based systems which focuses on non-functional properties. It uses an extended notion of aspects [11] 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) [12] is an important current development. 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 the Object Management Group (OMG) has issued a request for proposals [15] 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 implement our transformation tool. [21] describes 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 [6] 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 [19] 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]. QCCS also provides UML model transformation based on aspect-oriented design [10]. The authors of [19] 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.

7 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 refinement of non-functional specifications by requiring transformations between context models to be defined and applying them to measurement definitions.

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.

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

We are currently working on a tool set prototype to support our development process. This prototype implements the concepts presented in this paper. Context models are stored in a meta data repository [13], and we use model transformation techniques to describe the mapping between context model and application model.

Acknowledgements

We want to thank everybody in the COMQUAD project, Dr. Thomas Santen and the anonymous reviewers for their helpful comments on different versions of this paper.

References

1. J. Ø. Aagedal. *Quality of Service Support in Development of Distributed Systems*. PhD thesis, University of Oslo, 2001.
2. R. Aigner, M. Pohlack, S. Röttger, and S. Zschaler. Towards pervasive treatment of non-functional properties at design and run-time. In *16th Int'l Conf. on Software and Systems Engineering and their Applications (ICSSEA'03)*, Paris, France, 2–4 December 2003. CNAM-CMSL.
3. T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. Extensible markup language (XML) 1.0 (second edition), October 2000. W3C Recommendation.
4. J.-M. Bruel, editor. *Proc. 1st Int'l Workshop on Quality of Service in Component-Based Software Engineering, Toulouse, France*. Cépaduès-Éditions, June 2003.

5. E. Demairy, E. Anceaume, and V. Issarny. On the correctness of multimedia applications. In *11th EuroMicro Conf. on Real Time Systems*. IEEE, June 1999.
6. A. Gokhale, B. Natarajan, D. C. Schmidt, A. Nechypurenko, J. Gray, N. Wang, S. Neema, T. Bapty, and J. Parsons. Cosmic: An MDA generative tool for distributed real-time and embedded component middleware and applications. In *Proc. ACM OOPSLA 2002 Workshop on Generative Techniques in the Context of MDA*, Seattle, WA, November 2002.
7. I-Logix Inc., Open-IT, and THALES. UML profile for modeling quality of service and fault tolerance characteristics and mechanisms: Revised submission. OMG Document, May 2003. URL http://www.omg.org/docs/realtime/03-05-02.pdf.
8. Information technology – Quality of Service: Framework. ISO/IEC 13236:1998, ITU-T X.641, 1998.
9. V. Issarny and C. Bidan. Aster: A framework for sound customization of distributed runtime systems. In *Int'l Conf. on Distributed Computing Systems*, pages 586–593. IEEE Computer Society, 1996.
10. J.-M. Jézéquel, N. Plouzeau, T. Weis, and K. Geihs. From contracts to aspects in UML designs. In *AOSD Workshop on Aspect-Oriented Modeling with UML*, Enschede, The Netherlands, April 2002.
11. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *11th European Conf. on Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer, 1997.
12. A. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison Wesley Professional, April 2003.
13. M. Matula. Netbeans metadata repository, March 2003. http://mdr.netbeans.org/MDR-whitepaper.pdf.
14. N. Medvidovic and R. N. Taylor. A framework for classifying and comparing architecture description languages. In *Proc. 6th European Software Engineering Conf. together with 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE97)*, pages 60–76, Zurich, Switzerland, September 1997.
15. 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.
16. 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.
17. S. Röttger and S. Zschaler. CQML$^+$: Enhancements to CQML. In Bruel [4], pages 43–56.
18. S. Röttger and S. Zschaler. A software development process supporting non-functional properties. In *Proc. IASTED Int'l Conf. on Software Engineering (IASTED SE 2004)*. ACTA Press, 2004.
19. A.-M. Sassen, G. Amorós, P. Donth, K. Geihs, J.-M. Jézéquel, K. Odent, N. Plouzeau, and T. Weis. QCCS: A methodology for the development of contract-aware components based on aspect oriented design. In *AOSD Early Aspects Workshop*, Enschede, The Netherlands, 2002.
20. M. Shaw, R. DeLine, and G. Zelesnik. Abstractions and implementations for architectural connections. In *3rd Int'l Conf. on Configurable Distributed Systems*. IEEE Press, May 1996.
21. D. M. Simmonds, S. Ghosh, and R. France. An MDA framework for middleware transparent software development & quality of service. In Bruel [4], pages 1–7.
22. J. A. Stankovic, R. Zhu, R. Poornalingam, C. Lu, Z. Yu, M. Humphrey, and B. Ellis. VEST: An aspect-based composition tool for real-time systems. In *Proc. 9th Real-Time and Embedded Technology and Applications Symposium (RTAS'03), Toronto, Canada*, pages 58–69. IEEE Press, May 2003.