NQSL - Formal Language and Tool Support for Network Quality-of-Service Requirements

Christian Webel, Reinhard Gotzhein, and Joachim Nicolay

Department of Computer Sciences, University of Kaiserslautern, Kaiserslautern, Germany, {webel, gotzhein, j_nicola}@cs.uni-kl.de

Abstract. Network Quality-of-Service (QoS) is a central characteristic of the design of modern communication systems. Before designing and implementing communication systems, network QoS requirements and QoS mappings have to be specified and analyzed. In this paper, we provide language and tool support for this purpose. To specify network QoS requirements and QoS mappings, we define a formal description technique called *NQSL*, the *Network QoS Specification Language*. To support the efficient handling of NQSL specifications, we present a tool chain consisting of the Graphical NQSL Editor (GNE), the NQSL Analyzer (NA) for QoS domain reduction, and the NQSL-to-SDL Compiler (NSC) for the generation of SDL data and process types.

1 Introduction

The provision of *network Quality-of-Service* (*network QoS*) is one of the major challenges in the development of future communication systems. Network QoS comprises performance, reliability, guarantee, and scalability aspects on different levels of abstraction, as well as mappings between these levels. During *requirements analysis*, network QoS requirements are to be specified formally. In particular, the relevant network QoS aspects of a system are to be identified by defining QoS domains, QoS domains of adjacent levels are to be mapped, and subsets of QoS domains are to be selected. During *system design*, a QoS architecture has to be devised, QoS functionalities are to be identified, and QoS mechanisms to realize these functionalities must be supplied. Since QoS functionalities are placed on different system levels and have high interdependencies, and since the QoS status of a networked system may be subject to frequent changes both on application level and on resource level, the provision of network QoS is a highly complex task that requires a cross-layer approach in all development phases.

In previous work [1], we have introduced a formalization of network QoS. In particular, we have formalized the notions of QoS domain, QoS scalability, QoS mapping, and QoS requirements. Moreover, we have identified formal criteria to reduce QoS domains for consistency and tractability, based on utility and cost. In this paper, we build on and extend these results by providing language and tool support. To specify network QoS requirements and QoS mappings, we define a formal description technique called *NQSL*, the *Network QoS Specification*

Language. To support the efficient handling of NQSL specifications, we present a tool chain consisting of the Graphical NQSL Editor (GNE), the NQSL Analyzer (NA), and the NQSL-to-SDL Compiler (NSC). These tools relieve the system developer from several tedious and error-prone tasks, such as applying QoS mappings by hand or reducing QoS domains based on the definitions of utility and cost functions. Both tasks are required to evaluate and assess QoS mappings and the set of relevant QoS domain values on different levels of abstraction, and have to be repeated after each modification of the QoS requirements specification.

The remaining part of this paper is organized as follows: In Section 2, we survey related work. In Section 3, we summarize our formalization of network QoS requirements (cf. [1]). Section 4 introduces NQSL, the Network QoS Specification Language. In Section 5, the NQSL tools GNE, NA, and NSC are presented. Conclusions are drawn in Section 6.

2 Related Work

To cope with various requirements of system designs, user preferences, middleware, hardware, networks, operating systems, and applications, several QoS specification techniques have been proposed (see [2] for a classification):

- QML (Quality Modelling Language) [3] is focused on the specification of application layer QoS requirements. QoS requirements of lower layers, QoS scaling, and QoS mappings are not addressed.
- CQML (Component Quality Modeling Language) [4] adopts some of the fundamental concepts of QML, and also addresses dynamic QoS scaling. As QML, it is focused on the application layer. Since CQML is widely used, several tool kits exist, including front-end tools and parsers, *e.g.* [5].
- QDL (Quality Description Language) has been proposed as a part of the QuO (Quality Objects) framework [6] that supports QoS on the CORBA object layer. With QDL, it is possible to specify QoS requirements on application layer and on resource layer, and to define QoS scaling.
- The Quality Assurance Language (QuAL) is part of the Quality of Service Management Environment (QoSME) [7]. With QuAL, QoS requirements are specified in a process-oriented way. QoS-A (Quality-of-Service Architecture) [8] uses a parameter-based specification approach, including QoS adaptation and QoS mappings.

In summary, it can be stated that previous formal treatments of QoS address only some aspects of QoS requirement specification, focusing, for instance, on a subset of abstraction layers, or leaving out QoS mappings. Our work comprises the aforementioned issues and therefore provides a holistic, comprehensive formalization of network QoS requirements, across layers. Furthermore, we provide QoS tools beyond front-end tools, in particular, an analysis tool and a compiler.

3 Formalization of Quality-of-Service

In previous work [1], we have introduced a formalization of network QoS. In particular, we have formalized the notions of QoS domain, QoS scalability, QoS mapping, and QoS requirements. In this paper, we build on and extend these results. Therefore, we provide a survey of the formalization of network QoS in this section.

3.1 Formalization of Network QoS Requirements

The need for formalization of network QoS requirements arises from the fact that a precise description of network QoS between service user and service provider is needed to police, control, and maintain the data flow a user emits to the communication system. Further on, the mechanisms realizing these functionalities need a precise and well-defined description of QoS. Formalization of network QoS is done by firstly identifying the QoS domain, and secondly by describing the QoS scalability.

The *QoS domain* Q captures the QoS characteristics of a class of data flows, i.e. performance, reliability, and guarantee and is therefore defined as $Q = P \times R \times G$, where P is the performance domain, R is the reliability domain, and G is the guarantee domain. An element $q = (p, r, g)$ of Q is called *QoS domain value. QoS performance* describes efficiency aspects characterizing the required amount of resources and the timeliness of the service. The relevant efficiency parameters are included in the *QoS performance domain* P with $P = P_1 \times \ldots \times P_n = \prod_{i=1}^{n} P_i$, where $P_1, \ldots, P_n$ are performance subdomains. The *QoS reliability* describes the safety-of-operation aspects characterizing the fault behaviour (e.g., loss rate and distribution, corruption rate and distribution, error burstiness) and is defined as $R = \mathit{Loss} \times \mathit{Period} \times \mathit{Burstiness} \times \mathit{Corruption}$, with $\mathit{Loss} = \mathbb{N}_0$, $\mathit{Period} = \mathbb{R}_+$, $\mathit{Burstiness} = \mathbb{R}_+$, and $\mathit{CorruptionRate} = \{cr \in \mathbb{R} \mid 0 \leq cr < 100\}$. The *QoS guarantee* describes the degree of commitment characterizing the binding character of the service. QoS guarantee is formalized by the *QoS guarantee* domain as $G = \mathit{DoC} \times \mathit{Stat} \times \mathit{Prio}$, where $\mathit{Stat} = \{p \in \mathbb{R} \mid 0 < p \leq 1\}$, $\mathit{Prio} = \mathbb{N}$, and $\mathit{DoC} = \{\mathit{bestEffort}, \mathit{enhancedBestEffort}, \mathit{statistical}, \mathit{deterministic}\}$.

Varying communication resources require adaptive mechanisms to avoid network overload, and to scale the application service. The *QoS scalability* S describes the control aspects characterizing the scope for dynamic adaptation of the QoS aspects of a data flow (described by a QoS domain) to a certain granted network QoS. The *QoS scalability domain* S is defined as $S = \mathit{Util} \times \mathit{Cost} \times \mathit{Up} \times \mathit{Down}$, where $\mathit{Util} = \{u \mid u : Q \rightarrow [0, 1]\}$, $\mathit{Cost} = \{c \mid c : Q \rightarrow \mathbb{R}_+\}$, and $\mathit{Up}, \mathit{Down} \in \{x \in \mathbb{R}_+ \mid 0 \leq x \leq 1\}$. The elements of *Util* and *Cost* are called *utility functions* and *cost functions*, respectively. A utility function determines the usefulness of QoS domain values, a cost function c expresses the amount of needed resources, associating higher costs with scarcer resources. QoS domain values with the same utility (cost) ($\sim_{u(c)}$) are assigned to the same so-called *u(c)*-equivalence class of Q: $[x]_{u(c)} = \left\{q \in Q \mid q \sim_{u(c)} x\right\}$.

The *QoS requirements qosReq* define the set of valid QoS domain values and a QoS scalability value, and are formally stated as a triple (q_{min}, q_{opt}, s), where $q_{min},\ q_{opt} \in Q$ and $s \in S$. The QoS domain values q_{min} and q_{opt} specify a set $Q^{'} \subseteq Q$ of valid QoS domain values. To obtain $Q^{'}$, a preorder $\lesssim_u$ induced by the utility function is applied: $Q^{'} = \{q \in Q \mid q_{min} \lesssim_u q \lesssim_u q_{opt}\}$.

For consistency and tractability, we stepwise reduce the QoS domain Q. In a first step, we define the *reduced* QoS domain Q^u by selecting the best element of each u-equivalence class of Q regarding c, and by considering values from $Q^{'}$ only. Let m be the cardinality of $Q/{\sim_u}$, the quotient set of Q w.r.t. $\sim_u$, and let $[x]^i_u$ denote the ith element of $Q/{\sim_u}$ regarding $\lesssim_u$ (ith u-equivalence class). Then, $Q^u = \{q_1, \ldots, q_m\} \cap Q^{'}, \quad q_i = q \in [x]^i_u \mid \forall y \in [x]^i_u \,.\, q \lesssim_c y, \quad 1 \leq i \leq m$. A further reduction induces a derived QoS domain $Q^{u,c}$, discarding QoS domain values with higher cost, but less utility, $Q^{u,c} = \{q \in Q^u \mid \forall y \in Q^u \,.\, c(q) > c(y) \Rightarrow u(q) > u(y)\}$ [1].

3.2 Formal QoS Mappings

The mechanisms realizing QoS management tasks are typically embedded in the communication system, prevalent across layers, hiding complex tasks from the application. This leads to abstract QoS requirements on higher system layers, whereas on lower system layers, the level of detail increases. To rigorously relate the different viewpoints on network QoS, a well-defined translation of the requirements is needed, called *QoS mapping*. The QoS mapping can be decomposed into QoS domain mapping and QoS scalability mapping.

The QoS domain mapping $dm : Q_h \rightarrow Q_l$ is a function from a (higher layer) QoS domain Q_h to a (lower layer) QoS domain Q_l. The domain mapping dm may be defined using the auxiliary functions $dm_P : Q_h \rightarrow P_l$ (performance mapping), $dm_R : Q_h \rightarrow R_l$ (reliability mapping) and $dm_G : Q_h \rightarrow G_l$ (guarantee mapping). A detailed description of the three mapping subfunctions is given in [1]. In general, the QoS mappings are neither injective nor surjective.

The QoS scalability mapping is needed to apply control aspects characterizing the dynamic adaptation of QoS parameters on different system levels. A QoS scalability mapping sm is a set of four mapping functions sm_{Util}, sm_{Cost}, sm_{Up} and sm_{Down}, translating the different scalability domains into each other [1].

4 The Network Quality-of-Service Specification Language

In this section, we introduce *NQSL*, the *Network Quality-of-Service Specification Language* for the formal specification of QoS requirements. NQSL is directly derived from the formalization of network QoS in [1], which we have outlined in Section 3. It supports the specification of QoS domains and subdomains, QoS scalability, QoS mappings, and QoS requirements. The syntax of NQSL mainly adds keywords identifying concepts of network QoS, notation to specify

[1] For examples, see Section 5.2.

functions, and a set of basic data types. Due to this direct correspondance of the formalization of network QoS, which uses basic mathematical notation, and NQSL language elements, it is straightforward to associate a formal semantics with NQSL specifications.

To give a flavour of NQSL, we briefly present the language elements for the specification of QoS domains and subdomains. The complete definition of NQSL can be found in [9]. The syntax of NQSL is defined in Extended BNF (EBNF), using the usual notational conventions: non-terminals are written in angle-brackets `<non-terminal>`, terminals are enclosed by single quotes `'terminal'`, productions are declared in the form `<non-terminal> = expansion;`, square brackets enclose optional parts `[optional]`, and alternatives are separated by `|`.

As stated in Section 3.1, a QoS domain captures the QoS characteristics of a class of data flows, i.e. performance, reliability, and guarantee. A QoS domain (see List. 1) is identified by its *domain_ name* and defined by a *domain_ body* consisting of declarations of performance, reliability, and guarantee domains. A QoS subdomain is identified by a unique *name* and defined by a type, using basic data types (`Integer`, `Real`, `Enum`), tuples of data types (`Integer` × `Real`), or previously defined subdomains. Optionally, the domain of the data type can be restricted to a set of possible values.

Listing 1. NQSL: QoS domain and subdomain definition (excerpt)

```
<qosdomain_decl>       = 'QoSDomain' <domain_name> '{' <domain_body> '}';
<domain_body>          = <performance_domain> <reliability_domain>
                         <guarantee_domain>;
<performance_domain>  = 'Performance' '{' <partdomain_body> '}';
<reliability_domain>  = 'Reliability' '{' <partdomain_body> '}';
<guarantee_domain>     = 'Guarantee'   '{' <partdomain_body> '}';
(...)
<subdomain_decl>  = 'Subdomain' <subdomain_body>;
                  | 'Subdomain' <identifier>;
<subdomain_body>  = '{' <name_decl> <type_decl> [<typedomain_decl>] '}';
<name_decl>       = 'name'   ':' <identifier> ';';
<type_decl>       = 'type'   ':' <datatype_body> ';';
<typedomain_decl> = 'domain' ':' <typedomain_body> ';';
```

In Listing 2, an excerpt of the QoS domain *Video* is specified in NQSL. The performance domain consists of three subdomains. The first subdomain *Resolution* is defined as a tuple of integers, with the values restricted to the pairs $(320, 240)$, $(480, 360)$, and $(640, 480)$. The subdomains *Quality* and *FrameRate* have already been defined, and therefore are referenced.

Listing 2. NQSL specification: QoS domain *Video* (excerpt)

```
QoSDomain Video{
  Performance{
      Subdomain {
        name:   Resolution;
        type:   (Integer,Integer);
        domain: {(320,240),(480,360),(640,480)}; }
      Subdomain Quality;
      Subdomain FrameRate;
    }
  Reliability{  (...)   }
  Guarantee{    (...)   }
}
```

A QoS requirements `specification` is identified by a unique name and `uses` a QoS domain defined beforehand. It consists of a set of QoS requirement profiles `qosReq`, which are subdivided into a description of `minimum` and `optimum` QoS as well as `scalability`. Listing 3 gives an excerpt of the QoS requirement specification *VideoTransmission*. The specification consists of two QoS requirement profiles *Surveillance* and *Panorama*. In the example, the scalability aspect is shown, with utility and cost functions restricted to the performance domain. For instance, utility is defined by refering to performance subdomains *Resolution*, *Quality*, and *FrameRate*, with `Resolution.1` denoting the first tuple element. W.l.o.g. we assume that the needed transmission rate on *Hardware* layer would provide a good metric for the needed resources. The specification of optimum QoS can be found below (see Fig. 4).

Listing 3. NQSL specification: QoS scalability of *Video* (excerpt)

```
specification VideoTransmission uses Video {
    qosreq Surveillance {
      minimum{ (...) }
      optimum{ (...) }
      scalability {
        util = 0.1*((Resolution.1-160)/480) + 0.1*(Quality/75) +
               0.8*(FrameRate/25);
        cost Hardware = TransmissionRate;
        up = 0.2;
        down = 0.1; }
      }
    qosreq Panorama { (...)}
}
```

To determine the costs of a video data flow configuration on application level from the costs specified on hardware level, QoS domain mappings are used. In Listing 4, two domain mappings are specified, mapping the QoS domain *Video* to *Hardware* via *Middleware*. Note that for the subdomains *reliability* and *guarantee*, we assume identical mappings, therefore, no explicit QoS mappings are provided.

Listing 4. NQSL specification: QoS domain mappings

```
domainmapping from Video to Middleware{
  performance:
    NoOfFrames=ceil((160*Quality+3000)*(Resolution.1-160)/(160*1420));
    Period=1/FrameRate;
  reliability;
  guarantee; }
domainmapping from Middleware to Hardware{
  performance:
    TransmissionRate = NoOfFrames/Period*1512;
  reliability;
  guarantee; }
```

5 Tool Support for NQSL

In this section, we present our tool support for NQSL, consisting of the Graphical NQSL Editor (GNE), the NQSL Analyzer (NA), and the NQSL-to-SDL Compiler (NSC).

5.1 Graphical NQSL Editor

The *Graphical NQSL Editor (GNE)* is generated from a metamodel for network QoS, and implemented as a plugin for Eclipse IDE, using the *Eclipse Modeling Framework (EMF)* [10] and the *Graphical Modeling Framework (GMF)* [11]. Starting point is the *domain model* defined as a metamodel that is described in *ECore*, a UML-dialect and part of the Meta Object Facility (MOF) [12] that is limited to class diagrams. Based on this metamodel, EMF generates a rudimentary editor with basic functionalities such as creating or modifying objects. In the next step, GMF is used to generate a more sophisticated editor. Based on the Java classes generated by EMF and the domain model, GMF creates a graphical editor that is much more comfortable and intuitive to use. To this, the *graphical definition model* identifying graphical elements, *e.g.* figures, nodes, links etc., and the *tooling definition model* specifying the palette, creation tools, actions, etc. of the graphical elements are needed. These three models are bound by the *mapping definition model*. Based on this model, the *generation model* is obtained by a transformation step.

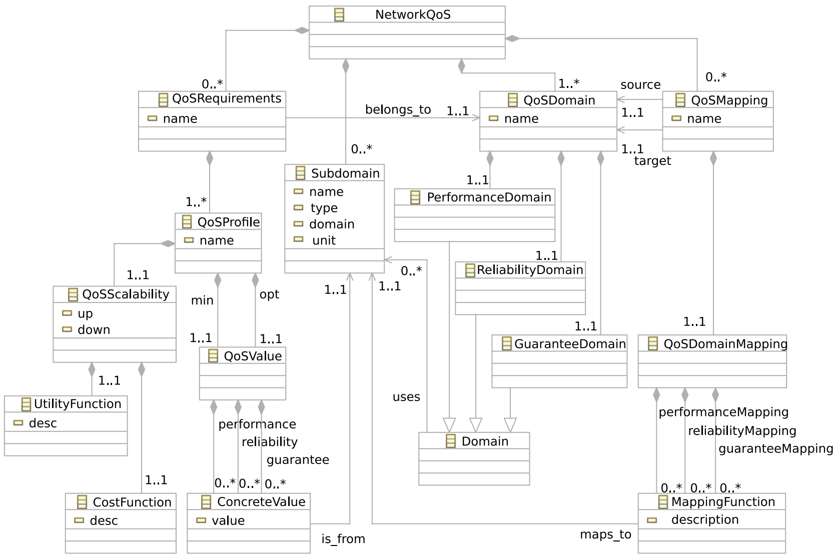


Fig. 1. Domain Model for the Graphical NQSL Editor

Starting point for the development of the graphical editor for NQSL is the domain model in Fig. 1. This metamodel is based on the formalization of network QoS surveyed in Section 3. The metamodel introduces a class *NetworkQoS*, which encapsulates QoS requirements and QoS mappings. Additionally, QoS domains and QoS subdomains are aggregated in this class; this way, a QoS subdomain can be used in different QoS domains, which is modeled by references. A QoS requirement specification is modeled by the class *QoSRequirements*, which in turn consists of a set of QoS profiles capturing different application scenarios. The relation between QoS requirements and QoS domain is modeled by a reference. To simplify the implementation, the domains of QoS performance, QoS reliability and QoS guarantee are collected in a superclass *Domain*. The *QoSScalabiliy* is modelled as described in the formalization. QoS scalability consists of *UtilityFunction*, *CostFunction*, and two thresholds *up* and *down*. The mappings for performance, reliability and guarantee are collected in *MappingFunction*. The QoS scalability mapping is not be explicitly modeled, as it is identical for all QoS specifications.

Following the formalization of network QoS, the Graphical NQSL Editor (GNE) consists of three parts: an editor for QoS domains and subdomains, an editor for QoS mappings between QoS domains, and an editor for QoS requirements. Fig. 2 shows the user interface of the *GNE domain editor*. A QoS domain is created by referencing previously built QoS subdomains. Notice that QoS subdomains can be used in several QoS domains.

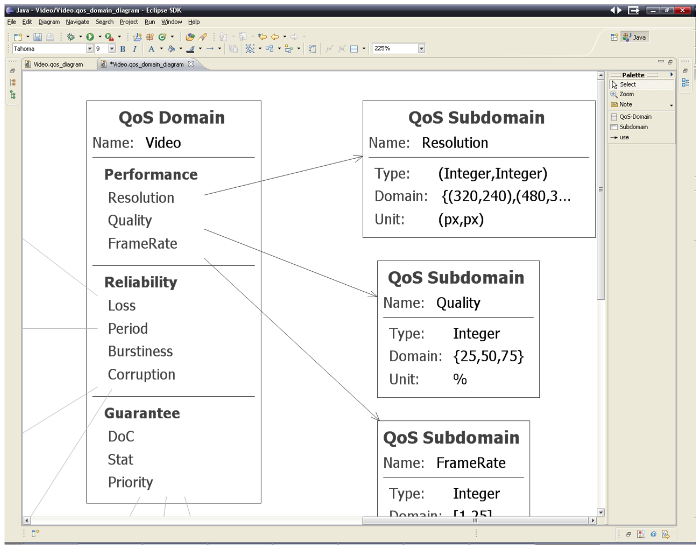
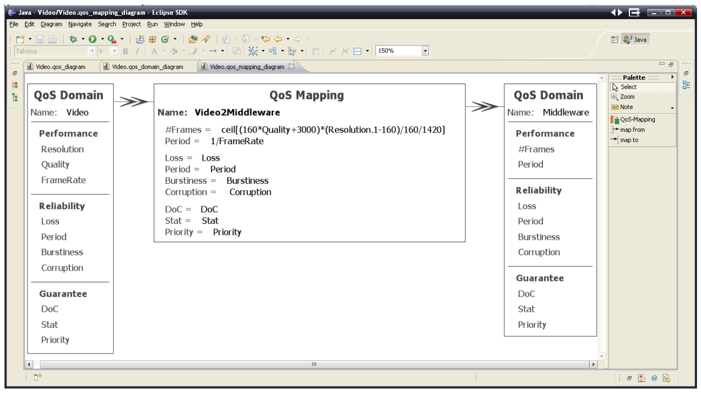


Fig. 2. GNE Domain Editor

The user interface of the *GNE Mapping Editor* is shown in Fig. 3. Two previously defined QoS domains *Video* and *Middleware* are related by specifying the QoS mapping *Video2Middleware*. If QoS subdomains of source and target domain are different, a customized mapping function has to be supplied. If some QoS subdomains are identical, *e.g.* priority or loss, a default mapping is generated. In addition, the direction of the mapping can be controlled by relations *map from* and *map to*.


Java - Video/Video.qos_mapping_diagram - Eclipse SDK
File Edit Diagram Navigate Search Project Run Window Help
Video.qos_diagram
Video.qos_domain_diagram
Video.qos_mapping_diagram
150%
Palette
Select
Zoom
Note
QoS-Mapping
map from
map to

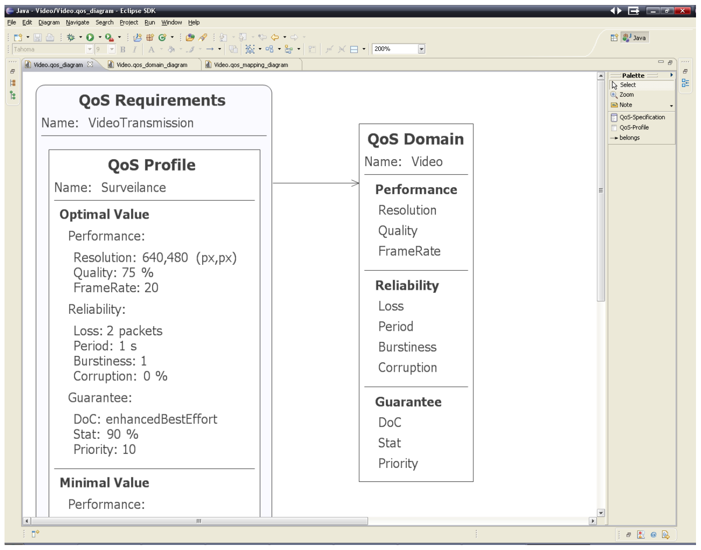
QoS Domain
Name: Video
Performance
Resolution
Quality
FrameRate
Reliability
Loss
Period
Burstiness
Corruption
Guarantee
DoC
Stat
Priority

QoS Mapping
Name: Video2Middleware
#Frames = ceil[(160*Quality+3000)*(Resolution.1-160)/160/1420]
Period = 1/FrameRate
Loss = Loss
Period = Period
Burstiness = Burstiness
Corruption = Corruption
DoC = DoC
Stat = Stat
Priority = Priority

QoS Domain
Name: Middleware
Performance
#Frames
Period
Reliability
Loss
Period
Burstiness
Corruption
Guarantee
DoC
Stat
Priority

Fig. 3. GNE Mapping Editor

Finally, the *GNE Requirements Editor* is used to define QoS requirements, consisting of a set of QoS profiles, on application level (see Fig. 4). First, a new QoS requirements specification is created and associated with a QoS domain. Then, QoS profiles can be added by specifying concrete minimum and optimum QoS, and a QoS scalability value. In the example, the QoS requirements specification *VideoTransmission* is associated with the QoS domain *Video* and consists of the QoS profile *Surveillance*.

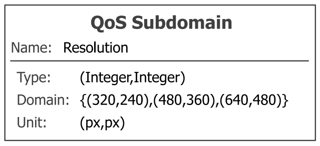

Java - Video/Video.qos_diagram - Eclipse SDK
File Edit Diagram Navigate Search Project Run Window Help
Video.qos_diagram
Video.qos_domain_diagram
Video.qos_mapping_diagram
200%
Palette
Select
Zoom
Note
QoS-Specification
QoS-Profile
belongs

QoS Requirements
Name: VideoTransmission
QoS Profile
Name: Surveilance
Optimal Value
Performance:
Resolution: 640,480 (px,px)
Quality: 75 %
FrameRate: 20
Reliability:
Loss: 2 packets
Period: 1 s
Burstiness: 1
Corruption: 0 %
Guarantee:
DoC: enhancedBestEffort
Stat: 90 %
Priority: 10
Minimal Value
Performance:

QoS Domain
Name: Video
Performance
Resolution
Quality
FrameRate
Reliability
Loss
Period
Burstiness
Corruption
Guarantee
DoC
Stat
Priority

Fig. 4. GNE Requirements Editor

For further processing, GNE supports the transformation of QoS domains, QoS requirements, and QoS mappings from XMI [13], which is the default data format, to NQSL. Since the data is available in a XML-based format, we used XSLT [14] for this transformation.Figure 5 shows the transformation of subdomain *Resolution* to the corresponding NQSL description.



QoS Subdomain

Name: Resolution

Type: (Integer,Integer)

Domain: {(320,240),(480,360),(640,480)}

Unit: (px,px)

(a) Subdomain *Resolution*

```
Subdomain {
 name:   Resolution;
 type:   (Integer, Integer);
 domain: {(320, 240),(480,360),(640, 480)};
}
```

(b) Result in NQSL

```
<xsl:for-each select="/qos:NetworkQoSDescription/subdomains">
  subdomain {
    name: <xsl:value-of select="@name" />;
    type: <xsl:value-of select="@type" />;
    domain: <xsl:value-of select="@domain" />;
  }
</xsl:for-each>
```

(c) Transformation in XSLT

Fig. 5. Transformation Process

5.2 NSQL Analyzer

Based on the definition of QoS domains on all abstraction levels, QoS mappings between them, and the QoS requirements on application level (see Section 5.1), the *NQSL Analyzer (NA)* performs QoS domain reductions and derives QoS requirements on communication level and on resource level. This relieves the system developer from filling in QoS requirements on lower levels and checking their consistency. Moreover, the analysis results provide feedback on QoS mappings, and support the assessment of utility and cost functions. Finally, they serve as input for generating fragments of the system design, e.g. QoS data structures and QoS scaling functionality based on QoS scaling tables.

To perform *QoS domain reductions*, the NQSL analyzer works in three steps, which are performed subsequently on all abstraction levels and for each QoS profile:

– In Step 1, the QoS domain is reduced to a set of u-equivalence class representatives. Here, the NQSL analyzer determines the u-equivalence classes of the QoS domain as induced by the utility function u of a QoS requirement profile (see Section 3.1). For each equivalence class, the QoS domain value with minimum cost according to the cost function c is kept as representative for that class, i.e. all other QoS domain values of that class are discarded. To limit memory needs of the system implementation later on, an upper bound for the number of equivalence classes can be set. At this point, the system developer obtains feedback about the distribution of the corresponding utility values in the interval $[0, 1]$. An example is given in Table 1. Note that to keep the presentation concise, we consider a very small QoS domain, comprising 9 values only (see Table 1(a)). A more realistic cardinality would be in the order of 10^3 to 10^5. Each QoS domain value consists of a tuple for resolution and values denoting JPEG quality and frame rate. Furthermore, the corresponding utilities and costs are shown. In the example, the utility of each QoS value is defined by the user, whereas the costs are calculated by means of the QoS performance mappings and cost function shown in Listings 3 and 4. Based on the utility, 5 equivalence classes $[x]_u$ are obtained. Selecting the QoS domain value with minimum cost in each equivalence class leads to a reduced QoS domain Q^u, as shown in Table 1(b).

Table 1. QoS Domain Q_{Video}

(a) *QoS domain values* q_{Video} *of QoS domain* Q_{Video}

value	**utility**	**cost**$[10^3]$	**value**	**utility**	**cost**$[10^3]$
((320,240),25,25)	0.1	175	((480,360),75,25)	0.5	750
((320,240),50,25)	0.1	275	((640,480),25,25)	0.7	525
((320,240),75,25)	0.3	375	((640,480),50,25)	0.7	825
((480,360),25,25)	0.3	350	((640,480),75,25)	0.9	1125
((480,360),50,25)	0.5	550			

(b) *equivalence partitioning* into 5 classes and keeping cost-optimal QoS domain values

$[\mathbf{x}]_\mathbf{u}$	**value**	**cost**$[10^3]$	$[\mathbf{x}]_\mathbf{u}$	**value**	**cost**$[10^3]$
$[0.1]_u$	((320,240),25,25)	175	$[0.7]_u$	((640,480),25,25)	525
$[0.3]_u$	((480,360),25,25)	350	$[0.9]_u$	((640,480),75,25)	1125
$[0.5]_u$	((480,360),50,25)	550			

– In Step 2, the number of QoS domain values is further reduced by applying the *cost criterion* (see also [1]). In general, it is possible that for QoS domain values q and q', $u(q) > u(q')$, while $c(q) \leq c(q')$. If this is the case, the QoS domain value q' can be discarded, as it is associated with higher or equal cost, but less utility. Discarding of QoS domain values is continued until for all remaining QoS domain values q and q', $u(q) > u(q')$ implies $c(q) > c(q')$. In the example, the QoS domain value $((480, 360), 50, 25)$ is discarded since $((640, 480), 25, 25)$ provides better utility at lower cost.
– In Step 3, the QoS domain values are further reduced by keeping only those QoS domain values that satisfy the QoS profiles of the QoS requirements, i.e. q_{min} and q_{opt}. First, the equivalence class representatives of q_{min} and q_{opt} are determined. From these representatives, the utility interval corresponding to the QoS requirements are obtained. Finally, all QoS domain values with a utility outside this interval are discarded. The remaining QoS domain values constitute entries of the QoS scaling table. In the example, we assume that

Table 2. Reduced Equivalence Classes

$\mathbf{[x]_u}$	**value**	**cost**$[10^3]$	$\mathbf{[x]_u}$	**value**	**cost**$[10^3]$
$[0.1]_u$	((320,240),25,25)	175	$[0.7]_u$	((640,480),25,25)	525
$[0.3]_u$	((480,360),25,25)	250	$[0.9]_u$	((640,480),75,25)	1125
~~$[0.5]_u$~~	~~((480,360),50,25)~~	~~550~~			

we have only one QoS profile with q_{min} in $[0.3]_u$ and q_{opt} in $[0.9]_u$. This leads to the QoS scaling table shown in Tab. 3.

Table 3. QoS Scaling Table

$\mathbf{[x]_u}$	**value**	**cost**	$\mathbf{[x]_u}$	**value**	**cost**
$[0.3]_u$	((480,360),25,25)	350	$[0.9]_u$	((640,480),75,25)	1125
$[0.7]_u$	((640,480),25,25)	525			

Thus, for every QoS profile of a QoS requirements specification, a QoS scaling table is generated. To derive *QoS requirements* on communication level and on resource level, the NQSL analyzer applies the corresponding QoS mappings to the set of QoS domain values remaining after QoS domain reduction. For each layer, the QoS requirement specification is derived by determining the utility of the resulting QoS domain values, and by selecting the QoS domain values with minimum and optimum utility. In the example shown in Tab. 4, the QoS profile is mapped to a corresponding QoS profile on middleware layer, described by the number of data frames required for the transmission of one picture frame, and the period between two picture frames, i.e. $Q_{Middleware} = NoFrames \times Period\ [s]$. To obtain the corresponding QoS domain values on middleware layer, the QoS mapping defined in Listing 3 has been applied. From these results, it follows that the minimum and maximum QoS domain values on middleware layer are $(10, 0.04)$ and $(32, 0.04)$, respectively.

Table 4. Results of QoS Mapping

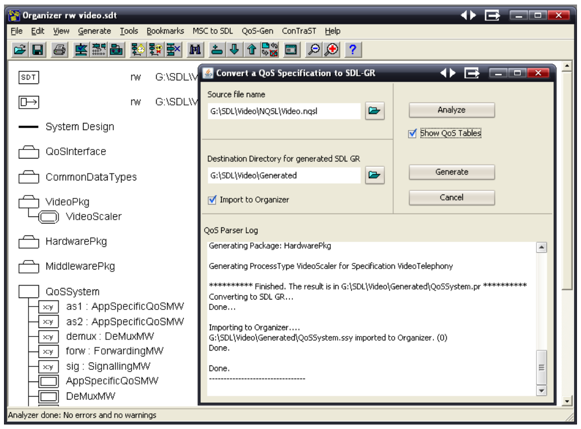
application layer			*middleware layer*		
$\mathbf{[x]_u}$	**value**	**cost**	$\mathbf{[x]_u}$	**value**	**cost**
$[0.3]_u$	((480,360),25,25)	350	$[0.3]_u$	(10, 0.04)	350
$[0.7]_u$	((640,480),25,25)	525	$[0.7]_u$	(15, 0.04)	525
$[0.9]_u$	((640,480),75,25)	1125	$[0.9]_u$	(32, 0.04)	1125

We have implemented the NQSL analyzer in Java. Currently, performance and guarantee mappings are supported by the tool.

5.3 NQSL-to-SDL Compiler

After finalizing the QoS requirements specification in NQSL, the developer turns to the specification of the system design. For the design, a QoS architecture has to be devised, required QoS functionalities such as access tests, resource reservation, traffic control, and scaling strategies are to be identified, and corresponding mechanisms to realize these functionalities must be provided. Furthermore, it has to be shown that the design satisfies the abstract QoS requirements.

Certainly, the design decisions that are to be taken here are far too complex to be automated entirely. However, it is feasible to generate fragments of the design. To start with, we have developed the *NQSL-to-SDL Compiler (NSC)* that translates QoS domains and QoS requirements specified in NQSL to corresponding data type definitions and QoS scaling process types based on QoS domain tables in SDL [15]. SDL, ITU-T's Specification and Description Language, is a formal design language for telecommunication systems that is widely used in industry and academia, with commercial tool support including graphical editors, analyzers, simulators, and SDL-to-C compilers. Fig. 6(a) shows a screen dump of the user interface of the NSC. As input, the tool accepts the NQSL output of the Graphical NQSL Editor. One output of the NSC are layer specific SDL packages, containing the according SDL data type definitions of the QoS domain. The NSC has been integrated into Telelogic TAU [16], the SDL tool suite of a commercial provider of SDL tools. It has been written in Java, using JFlex [17] for lexical analysis and CUP [18] for parsing. For better usability, we have integrated the NQSL Analyzer into the NSC user interface.



Source file name: G:\SDL\Video\NQSL\Video.nqsl

Destination Directory for generated SDL GR: G:\SDL\Video\Generated

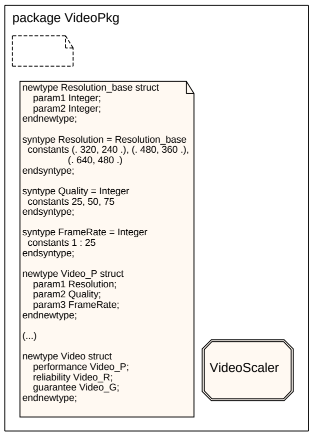
[✓] Show QoS Tables

[✓] Import to Organizer

```
Generating Package: HardwarePkg

Generating ProcessType VideoScaler for Specification VideoTelephony

********** Finished. The result is in G:\SDL\Video\Generated\QoSSystem.pr **********
Converting to SDL GR...
Done...

Importing to Organizer....
G:\SDL\Video\Generated\QoSSystem.ssy imported to Organizer. (0)
Done.

Done.
--------------------------------
```

```
Analyzer done: No errors and no warnings
```

(a) User Interface integrated in Telelogic TAU



```
newtype Resolution_base struct
    param1 Integer;
    param2 Integer;
endnewtype;

syntype Resolution = Resolution_base
  constants (. 320, 240 .), (. 480, 360 .),
                (. 640, 480 .)
endsyntype;

syntype Quality = Integer
  constants 25, 50, 75
endsyntype;

syntype FrameRate = Integer
  constants 1 : 25
endsyntype;

newtype Video_P struct
    param1 Resolution;
    param2 Quality;
    param3 FrameRate;
endnewtype;

(...)

newtype Video struct
    performance Video_P;
    reliability Video_R;
    guarantee Video_G;
endnewtype;
```

(b) Generated SDL data types

Fig. 6. NQSL-to-SDL Compiler

For a given QoS domain, the NSC generates an SDL package, *i.e.* a library that can be imported by SDL system specifications. The generated SDL package contains SDL data type definitions of all problem-specific QoS subdomains of the NQSL specification. QoS subdomains that are not problem-specific, such as *degree of commmitment* or *priority* of the QoS guarantee domain, are collected in the predefined SDL package *CommonDataTypes*, which s imported by every other SDL package generated by the NSC.

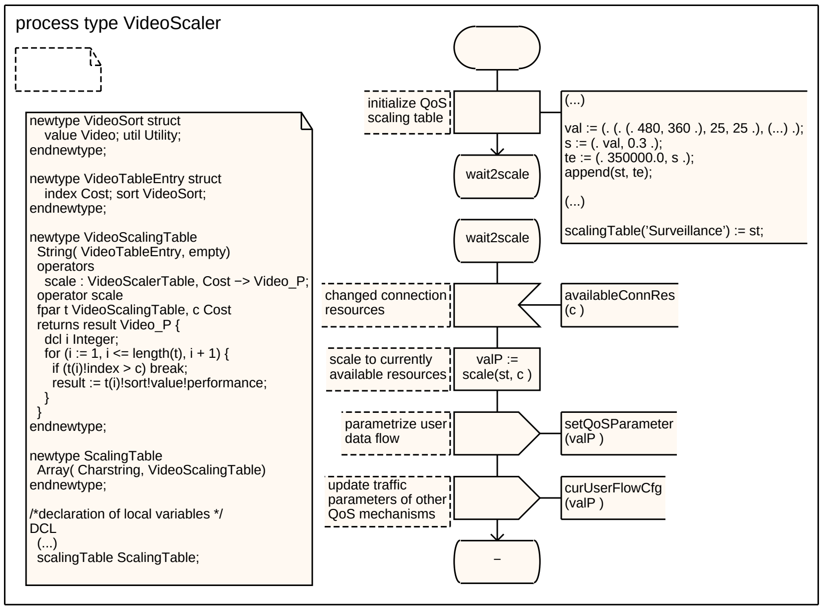
QoS domains and subdomains are mapped to SDL syntypes or newtypes, depending on their complexity. Fig. 6(b) shows the SDL data types generated for the QoS domain Q_{Video}, which are contained in the new SDL package *VideoPkg*. For the QoS domain Q_{Video}, the SDL data type *Video* (see bottom of Fig. 6(b)) is derived. This data type is structured into the performance data type *Video_P*, the reliability data type *Video_R*, and the guarantee data type *Video_G*. The subdomains *Resolution*, *Quality* and *FrameRate* of the performance domain *Video_P* are also defined within this package; the subdomains of the reliability and performance domain are imported from in the predefined SDL package *CommonDataTypes*.

Starting point for the translation of a QoS requirements specification to SDL is a reduced and optimized QoS domain as described in Sections 3.1 and 5.2. For this reason, a preceding NQSL analyzer run is mandatory. For every QoS requirements specification, an SDL scaling process type is automatically generated, consisting of a scaling algorithm and a scaling table. The scaling table aggregates the reduced QoS domain tables containing the optimal QoS domain values created for every QoS profile of a QoS requirements specification. The scaling algorithm selects the currently best QoS domain value under a given resource situation.

Figure 7 shows the process type *VideoScaler* generated by the NSC. The scaling algorithm is realized by an operator *scale* defined on the SDL data type *VideoScalingTable*. During the startup phase of the process, the tables for the QoS profiles of a specification are initialised according the output of the NQSL analyzer. If the resource situation changes as indicated by the input signal *availableConnRes*, the scaling operation is performed, and the user data flow configuration is updated. Further, the current application scenario can be chosen by another input signal (not shown in the figure).

6 Conclusions and Future Work

In this paper, we have presented NQSL, the Network QoS Specification Language, to formally specify network QoS. NQSL is derived from our previous formalization of network QoS with specific emphasis of scalability and cross-layer development. It provides language elements for specifying QoS domains, QoS subdomains, and QoS mappings. Further, QoS requirements can be defined by specifying QoS profiles, expressed by minimum and optimum QoS domain values and a QoS scalability value that consists of utility function, cost function, and two thresholds. To support the efficient handling of NQSL specifications, we


process type VideoScaler
initialize QoS scaling table
wait2scale
wait2scale
changed connection resources
availableConnRes (c)
scale to currently available resources
valP := scale(st, c)
parametrize user data flow
setQoSParameter (valP)
update traffic parameters of other QoS mechanisms
curUserFlowCfg (valP)
–

```
newtype VideoSort struct
   value Video; util Utility;
endnewtype;

newtype VideoTableEntry struct
   index Cost; sort VideoSort;
endnewtype;

newtype VideoScalingTable
  String( VideoTableEntry, empty)
  operators
    scale : VideoScalerTable, Cost −> Video_P;
  operator scale
  fpar t VideoScalingTable, c Cost
  returns result Video_P {
    dcl i Integer;
    for (i := 1, i <= length(t), i + 1) {
      if (t(i)!index > c) break;
      result := t(i)!sort!value!performance;
    }
  }
endnewtype;

newtype ScalingTable
  Array( Charstring, VideoScalingTable)
endnewtype;

/*declaration of local variables */
DCL
  (...)
  scalingTable ScalingTable;
```

```
(...)

val := (. (. (. 480, 360 .), 25, 25 .), (...) .);
s := (. val, 0.3 .);
te := (. 350000.0, s .);
append(st, te);

(...)

scalingTable('Surveillance') := st;
```

Fig. 7. Generated Process Type *VideoScaler* (excerpt)

have presented a tool chain consisting of the Graphical NQSL Editor (GNE), the NQSL Analyzer (NA), and the NQSL-to-SDL Compiler (NSC).

The work presented in this paper solves a number of problems of practical relevance. First, it is very important that network QoS requirements be specified formally. While there are several languages reported in the literature already, NQSL goes one step further by supporting the specification of network QoS requirements on all system layers, by including QoS scalability, and by supporting QoS mappings. Second, for practical usage, tool support is mandatory. Here, our tool chain supports editing, analyzing, and transforming NQSL specifications, relieving the system developer from several tedious and error-prone tasks, such as applying QoS mappings by hand or reducing QoS domains based on the definitions of utility and cost functions. To further increase usability, the SDL generator has been integrated into Telelogic SDL Suite. We are currently not aware of QoS tools with comparable functionality.

Our future work aims at extensions of our tool chain for QoS system development, and at SDL system designs satisfying formally specified network QoS requirements. The next step will be the extension of the NQSL-to-SDL compiler by architectural QoS concepts. Thereby, it will be possible to automatically generate a complete SDL system structure with QoS functionalities. Another step is the formal definition of a distributed resource management and scalability model for multiple data flows. This model will use the derived QoS profiles across layers to provide and manage QoS for different user data flows in a correct and efficient manner.

Acknowledgments. The work presented in this paper was in part carried out in the BelAmI (Bilateral German-Hungarian Research Collaboration on Ambient Intelligence Systems) project, funded by German Federal Ministry of Education and Research (BMBF), Fraunhofer-Gesellschaft and the Ministry for Science, Education, Research and Culture (MWWFK) of Rheinland-Pfalz.

References

1. Webel, C., Gotzhein, R.: Formalization of Network Quality-of-Service Requirements. In: Formal Techniques for Networked and Distributed Systems - FORTE 2007. Lecture Notes in Computer Science (LNCS) 4574, Springer (2007) 309–324
2. Jin, J., Nahrstedt, K.: QoS Specification Languages for Distributed Multimedia Applications: A Survey and Taxonomy. IEEE MultiMedia **11**(3) (2004) 74–87
3. Frølund, S., Koistinen, J.: QML: A Language for Quality of Service Specification. Technical Report HPL-98-10, pp. 63., Software Technology Laboratory, Hewlett-Packard Company (1998)
4. J. Ø. Aagedal: Quality of Service Support in Development of Distributed Systems. PhD thesis, University of Oslo, Oslo, Norway (2001)
5. Röttger, S., Zschaler, S.: Tool support for refinement of non-functional specifications. Software and Systems Modelling journal (SoSyM) **6**(2) (June 2007)
6. Vanegas, R., Zinky, J.A., Loyall, J.P., Karr, D., Schantz, R.E., Bakken, D.E.: QuO's Runtime Support for Quality of Service in Distributed Objects. In: Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98), The Lake District, UK (1998) 207–222
7. Florissi, P.G.S.: QoSME: QoS Management Environment. PhD thesis, Columbia University (1996)
8. Campbell., A.T.: A Quality of Service Architecture. PhD thesis, Computing Department, Lancaster University (1996)
9. Webel, C.: NQSL - A Specification Language for Network Quality of Service. Technical Report 368/07, Department of Computer Science, University of Kaiserslautern (2007)
10. Eclipse Foundation: Eclipse Modeling Framework Project (EMF). http://www.eclipse.org/modeling/emf/ (2007)
11. Eclipse Foundation: The Eclipse Graphical Modeling Framework (GMF). http://www.eclipse.org/gmf/ (2007)
12. Object Management Group, Inc.: Meta Object Facility (MOF) Specification. http://www.omg.org/mof/ (2000)
13. Object Management Group, Inc.: Xml metadata interchange (xmi) specification. http://www.omg.org/technology/documents/formal/xmi.htm (2007)
14. World Wide Web Consortium: XSL Transformations (XSLT). W3C Recommendation. http://www.w3.org/TR/xslt (1999)
15. International Telecommunications Union: Specification and Description Language (SDL). ITU-T Recommendation Z.100 (August 2002)
16. Telelogic AB: Telelogic SDL Suite and TTCN Suite. http://www.telelogic.com/products/tau/sdl/index.cfm (2007)
17. JFlex: JFlex - The Fast Scanner Generator for Java. http://jflex.de/ (2007)
18. CUP: CUP – LALR Parser Generator in Java. http://www2.cs.tum.edu/projects/cup/ (2006)