Extra-functional contract support in components

Olivier Defour, Jean-Marc Jézéquel, Noël Plouzeau

INRIA-Rennes, Campus universitaire de Beaulieu, Avenue du général Leclerc
35042 Rennes Cedex, France

{olivier.defour, jean-marc.jezequel, noel.plouzeau}@irisa.fr
http://www.irisa.fr/triskell

Abstract. According to Szyperski, "*a software component is a unit of composition with contractually specified interfaces and explicit context dependencies only*". But it is well known that these contractually specified interfaces should go well beyond mere syntactic aspects: they should also involve functional, synchronization and Quality of Service (QoS) aspects. In large, mission-critical component based systems, it is also particularly important to be able to explicitly relate the QoS contracts attached to provided interfaces with the QoS contracts obtained from required interfaces. In this paper we propose a language called QoSCL (defined as an add-on to the UML2.0 component model) to let the designer explicitly describe and manipulate these higher level contracts and their dependencies. We show how the very same QoSCL contracts can then be exploited for validation of individual components and also validation of a component assembly, including getting end-to-end QoS information inferred from individual component contracts, by automatic translation to a Constraint Logic Programming language. We illustrate our approach with the example of a GPS (Global Positioning System) software component, from its functional and contractual specifications to its implementation in a .Net framework.

1 Introduction

In Szyperski's vision [1], "*a software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third-party*". In this vision, any composite application is viewed as a particular configuration of components, selected at build-time and (re-)configured at run-time.

This point of view is now widely adopted in components middleware technologies such as Corba Component Model (CCM) [2], Entreprise JavaBean (EJB) [3] or Microsoft .Net/Com [4]. In these various middleware, a software component is a binary executable code deployed in an environment which manages it (*EJBContainer* for EJB or *component home* for CCM). A software component only exhibits its provided or

required interfaces, hence defining basic *contracts* between components allowing one to properly wire them. But it is well known that these contractually specified interfaces should go well beyond mere syntactic aspects: they should also involve functional, synchronization [5] and Quality of Service (QoS) aspects [6]. In large, mission-critical component based systems, it is also particularly important to be able to explicitly relate the QoS contracts attached to provided interfaces with the QoS contracts obtained from required interfaces.

The aim of this article is to present a QoS contract model (called QoSCL for QoS Constraint Language), allowing such QoS contracts and their dependencies to be specified at design-time in a UML2.0 [7] modeling environment. We then show how the very same QoSCL contracts can be exploited not only for validation of individual components but also validation of a component assembly, including getting end-to-end QoS information inferred from individual component contracts, by automatic translation to a Constraint Logic Programming [8].

The rest of the paper is organized as follows. Using the example of a GPS (Global Positioning System) software component, Section 2 introduces the interest of modelling components with their contracts and their dependencies, and describes the QoS Constraint Language (QoSCL). Section 3 discusses the problem of validating a component assembly, including getting end-to-end QoS information inferred from individual component contracts by automatic translation to a Constraint Logic Programming (CLP). This is applied to the GPS system example, and experimental results are presented. Finally, Section 4 presents related works.

2 QoS specification

2.1 The common QoS features

Before to introduce the metamodel of any specification language, it is important to understand the semantic of the concepts that are handled. What are the concepts that we want to specify, their semantic, their features, properties and relationships ?

The quality of a service is defined as a set of extra-functional properties. Consequently, an extra-functional property is an intrinsic qualitative dimension of a service. An extra-functional property especially is a valuable quantity. This aspect implies 1that:an extra-functional property is ever associated to means of measurement;

2. as any valuable quantity, an extra-functional property can be constrained;
3. the effective quality of a service is the value of its extra-functional properties, compared to a reference totally ordered scale of values: the quality levels.

A quality level is a set of values, denumerable or not, and usually bounded. For instance, the *getLocation()* service provided by the GPS component, shown in Fig. 1 below, can be qualified by two extra-functional properties: the precision **eps** of the estimated position, and the response time q_c. The precision is defined in the finite set

{low; normal; high}, with the total following order: low < normal < high. The response time is a positive real value, dedicated to be less than a specified time out value.

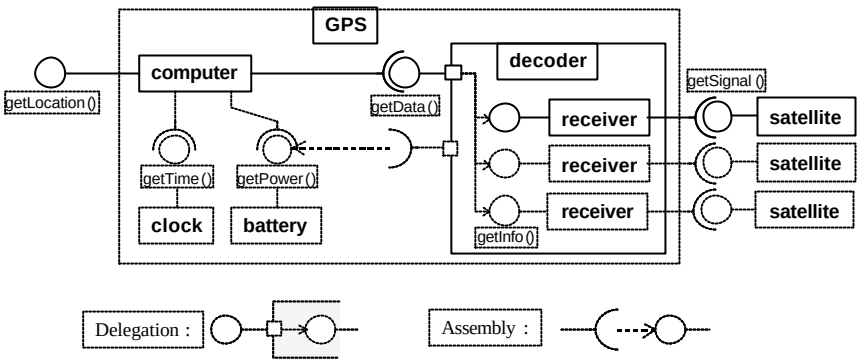

Fig. 1. Component-based model of a GPS device

Any artefact that realizes the GPS component model must implement not only the *getLocation* service, but also a means of measurement for its two intrinsic extra-functional properties **eps** and q_C. Consequently, it is useful to consider an extra-functional property as the specification of its means of measurement: the semantic of what is measured is defined at the component model level refined with QoS specifications.

A QoS specification languages, such as QML [9] for example, is dedicated no only to specify extra-functional properties, but also a set of constraints. Usually, these constraints are checked at run time, in order to renegotiate contracts. At design time, the constraints are used to specify either qualitative requirements for required services, or provided quality levels for provided services. At the assembly level between two components, we must check that the quality level of the provided service is compliant with the qualitative requirements.

For instance, the GPS component is connected to a set of satellite components that provide the required *getSignal* services. Let q_S be the response time of the *getSignal* service provided by the satellite components. A signal emitted by a satellite has a duration of 15 ms and, so, the GPS receives a complete signal in a time greater than 15 ms and less than 30 ms. That is a quality level of the provided *getSignal* service. But the GPS can also requires that the response time to complete the signal must be less than 20 ms: in this case, the constraint on the response time of the *getSignal* service is not compliant with the provided QoS level.

An other main feature of the QoS is the extra-functional dependency. A service provided by a component usually depends on the a subset of services that it requires. A component-based model refined enough, as the Fig. 1, shown the functional chain of dependencies that binds a provided service to required ones. On this example, it is easy to explain the functional dependencies that binds the *getLocation* service to the *getSignal* services, through the assemblies and the delegations.

As the extra-functional properties are intrinsic features of a service, the functional dependency implies an extra-functional dependency. On our example, consider the

GPS component: it is obvious that the *getLocation* service will be completed after the *getSignal* services. This relationship can be explained as a constraint that binds q_C and the whole of q_S:

$$q_C \geq \max(q_S) . \quad (1)$$

A more detailed analysis of the behavior of the GPS (Fig. 1), and particularly of the functional chain of dependencies that binds the *getLocation* service to the *getSignal* service, implies that:

$$q_R = q_S + 2 , \quad (2)$$
$$q_D = \max(q_R) + 3 , \quad (3)$$
$$q_C = q_D + \mathbf{nbr} * \log (\mathbf{nbr}) . \quad (4)$$

where :

- q_S is the response-time of the *getSignal* service, *i.e.* the time spent by a receiver to acquire a complete signal from a satellite,
- q_R is the response-time of the *getInfo* service, i.e. the reception time of a signal (q_S) more the time spent to demultiplex the signal (2ms),
- q_D is the response-time of the *getData* service, *i.e.* the maximum of the q_R, more the decoding time of demultiplexed signals (3ms).
- **nbr** is the number of active *receivers* (i.e. the number of powered receivers that get a signal). This number has an impact on the response-time of the *getLocation* service (q_C) because of the interpolation algorithm used.

This example clearly illustrates how the functional dependency of services can be translated into an equivalent extra-functional chain of dependencies.

2.2 The QoSCL metamodel

In the previous section, we have introduced the concepts of quality, level of quality and extra-functional dependency. We propose to extend the UML2.0 metamodel with these concepts. This specific metamodel, called QoS Constraint Language (QoSCL), allows a designer to specify:

- the intrinsic qualities of a service,
- the required or provided quality levels of a service
- the extra-functional dependency of a provided service's quality on a set of qualities defined on required services.

The Fig. 2 below shown the QoSCL extension of the UML2.0 metamodel. We have defined four new classes:

- **ComponentQoSCL**: extends *Component*. In addition of its functional provided and required interfaces, it has specific provided and required *ContractTypes*.
- **ContractType**: is a specialized *Interface*, which all elements of its *ownedOperation* attribute are typed as *Dimension*. Moreover, a provided (resp. required) *ContractType* is associated to only one *Operation* that belongs to a provided (resp. required) *Interface*.

- **Contract**: is a class encapsulated by a component that implements a *ContractType*. The implementation of the *Dimensions* depends on each component, and each contract has its own renegotiation behaviour. This behaviour can be specified with state charts, activity diagrams, etc…
- **Dimension**: extends *Operation*. It specifies the means of measurement of the effective quality level of an extra-functional property and, by assimilation, the property itself. The semantic of its three constraints is:
    - *body*: the set of possible values for this extra-functional property;
    - *pre*: the required quality level, which must be compliant with the *body* constraint;
    - *post*: the provided quality level, which must be compliant with the two previous.

  A *Dimension* has optionally *ContractTypes* as parameters that represents its extra-functional dependency with other qualities.

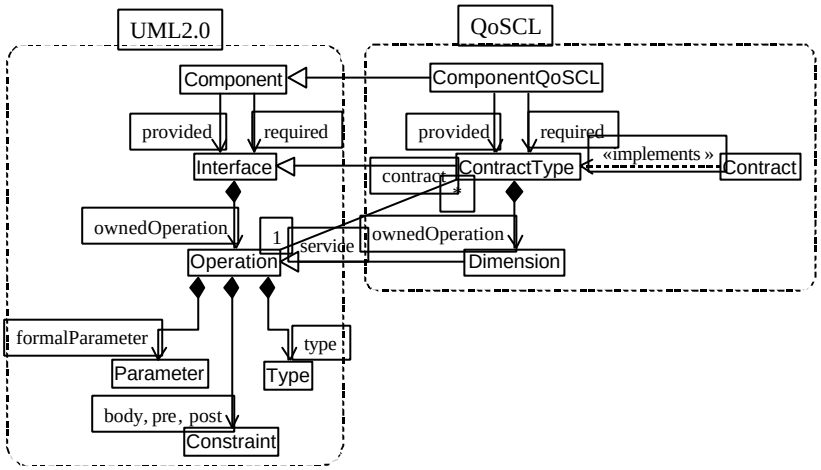

Fig. 2. The QoSCL metamodel

The Fig. 3 below shown the QoSCL model of the GPS component. The *getLocation* and *getSignal* services are connected their respective *ContractTypes*. The *thetaC Dimension* depends on the *getSignal ContractType* (attribute *parameter*). The *thetaS Dimension* has three constraints:

- its *body* specifies that its value must be positive;
- its *post* specifies that the *getSignal* service provided by the *Satellite* component is completed in more than 15ms and less than 30ms;
- its *pre* specifies that the GPS component requires this service with a response time less than 15.5ms.

The *thetaC* post-condition ensures that the response time of the *getLocation* service is less than 24ms. This service depends on the *getSignal* service, and, as we have underline yet, this dependency implies an extra-functional dependency between their respective extra-functional properties. That is the reason why the qualitative level

requirement on *thetaC* (i.e. its post-condition) implies a requirement on *thetaS* (i.e. its pre-condition). This mechanism will be more detailed in the following chapter of this paper.

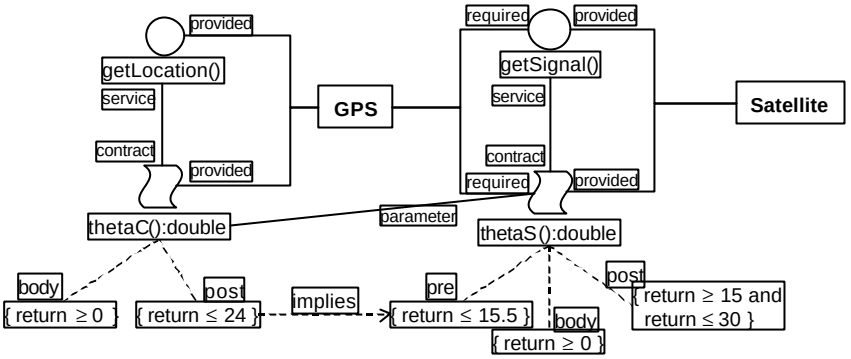

Fig. 3. The QoSCL model of the GPS

The QoSCL model is a platform independent component model, specifying abstract QoS properties and behaviors (*ContractType* and *Dimension*). It can be used to implement a concrete atefact, including monitors for the extra-functional properties that implement the specified behavior (*Contract*). Such implementations have already done: for instance, in the QCCS European project [10], an aspect weaver has been implemented in order to weave QoS properties into a functional platform independent model. A model transformation [11] translates this model into a .Net platform dependant model that implements a set of monitors and renegotiable contracts.

3 Predict the QoS levels

At this point, we have a QoS specification language, QoSCL, that allows to refine a UML2.0 component model with QoS properties. This language can support the implementation of the QoS concepts such as monitors and contracts, thanks to aspect weaving techniques followed by a model transformation for example. However, at design time, it is also possible to implement a tool that predicts the QoS levels according to the QoSCL specifications.

3.1 QoS levels prediction features

In the beginning of the QoS concepts section (§ **2.1**), we have said that a QoS property is a valuable quantity. It supports a set of constraints and it is bonded to others QoS properties via the extra-functional dependencies (**1**,**2**,**3**,**4**). So the first important point to underline is that QoSCL allows to *declare* a set of QoS properties and their relations.

The second important point is the form of the declared *relations*: constraints (required quality levels), mathematical equations (**1**-**4**) or empirical rules. These two last forms are shown on the two figures below (Fig. 4, Fig. 5):

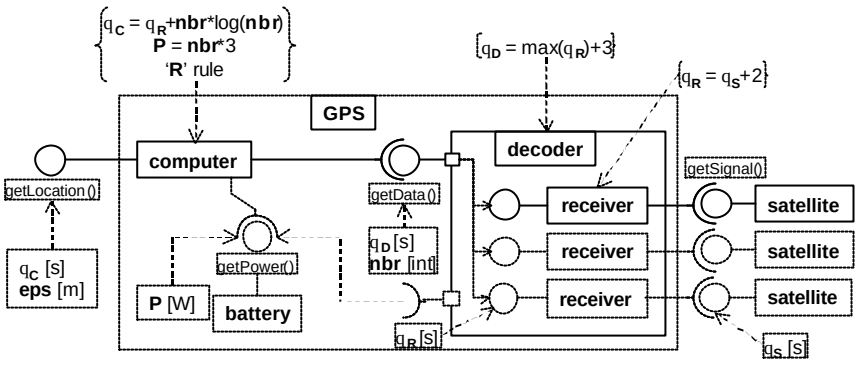

Fig. 4. GPS component model refined with its declared QoS properties and relations

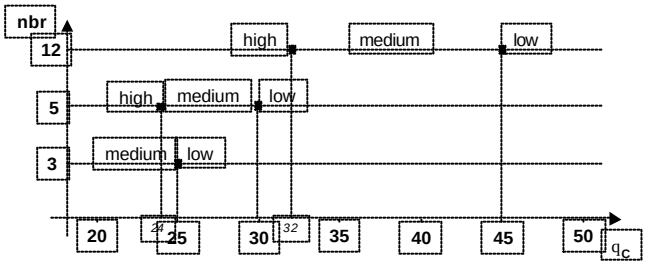

Fig. 5. The 'R' rule that binds three QoS properties (eps, nbr and q_C)

where **P** is the reactive power consumed by the *computer* and *decoder* components. It is an extra-functional property of the *getPower* service.

Our GPS model has twelve receivers (only three shown in the previous figures). However, all the receivers are not powered, in order to manage the power consumption, and only a subset of them receive a clear satellite signal. This subset depends on the environment: for example, this set is empty in a tunnel that blocks the signals transmission. The **nbr** property is equal to the number of powered receivers that have a clear signal.

The third important point of the QoS levels prediction is the nature of the information that we want to handle and to give at the designer. The QoS properties belong to valued sets, and that is this high-level information will be handled. Indeed, for engineers, a symbolic relation that binds a set of variables is not intuitive: it not allows an immediate knowledge. The only form that a human actor like a designer can easily understand is a *valued set, i.e.* either a finite set of values or a numerical interval.For instance, on the Fig. 3 diagram, the designer knows that the response time q_S of the *getSignal* service belongs to the real interval [15; 15.5]. An empty set would have

implied an error, i.e. that the service would not be provided with a quality level compliant with the set of declared constraints and dependencies.

At last, we want to allow the designer to study the influence of a specific quality level requirement on any QoS property on the others. In other words, the designer wants to study the propagation of an information through a network of relations. This mechanism is more complex than an evaluation of a set of functions.

For instance, we present below two information propagations through the network of relations shown in Fig. 4:

$$q_S \in [15; 30] , \mathbf{nbr} \in \{3,5,12\} \Rightarrow \\ q_R \in [17; 32] , q_D \in [20; 35] , q_C \in [21.43; 47.95] . \quad (\mathbf{5})$$

$$\mathbf{eps} = \text{high}, \mathbf{P} \leq 15, \mathbf{nbr} \in \{3,5,12\} \Rightarrow \\ q_C \in [23.49; 24], q_D \in [20; 20.5], q_R \in [17; 17.5], q_S \in [17; 17.5] . \quad (\mathbf{6})$$

As a consequence of this last point, the propagation of information through the network of relations implies the conformance of the pre, post and body conditions of each extra-functional property defined on an assembly.

3.2 Implementation details

To resume our problem, QoSCL allows a designer to *declare* a set of extra-functional properties, bonded between them into a *network of relations.* Theses relations are either *numerical constraints* (the quality levels), *mathematical (non-linear) functions* (**2**,**3**,**4**) or *empirical rules* (Fig. 5). The designer wants to select *any* extra-functional properties and to observe the *propagation* of a qualitative level requirement on the others properties through the network. The result that he obtains must be under the same form as its requirement, i.e. a *valued set*.

Finally, we want the following features:

1. a declarative technology;
2. a network of relations that binds a set of variables;
3. relations as constraints, mathematical non-linear functions or rules;
4. information as valued sets (finite sets or numerical intervals).

The only technology that covers the whole of these four points is the Constraint Logic Programming (CLP) [12] including a real interval arithmetic solver, denoted CLP(R) [13]. The tools that implement this particular technology are few: PrologIV [14] and Eclipse [15]. There exist also interval arithmetic solvers only, such as RealPaver [16]. These software handle intervals, which are reduced by considering the constraints. The result is the most reduced interval including the solution, if there exists. Here is an example [*ref p4*]:

```
>> X ~ cc(0,pi), sin(X) = cos(X).
>> X ~ oo(0.739085, 0.7390852).
```

It is important to note that many CLP software declare to integrate a CLP(R) solver, like SICStus-Prolog [17] for example. However, their solvers are not based on the interval arithmetic, but on a symbolic computation, which does not handle interval and fails to compute non-linear constraints:

```
|?- {X > 0, X < 3.14159, sin(X) = cos(X)}.
    {X>0},
    {X<3.14159},
    clpr:{-(cos(X))+sin(X)=0.0)}.
```

The QoS specifications are written in QoSCL. Although this language has all the features of a CLP compliant language, it was not originally dedicated to this specific use. Implementing a dedicated CLP solver engine, with its interval arithmetic solver, is not an easy task. Because of the common features, it is more judicious to transform QoSCL into an existing CLP language, using a MOF compliant model transformation such as MTL [11]. The result of a transformation of the GPS component-based model with QoSCL into a PrologIV program for example is:

```
00- %% CONTRACTUAL DEPENDENCIES
01- >> receiver( ThetaR, ThetaS) :-
02-      ThetaR ~ ThetaS + 2.
03-
04- >> decoder( ThetaD, ThetaS) :-
05-      receiver( ThetaR1, ThetaS),
06-      receiver( ThetaR2, ThetaS),…,
07-      max( X, [ThetaR1, ThetaR2,…] ),
08-      ThetaD ~ X + 3.
09-
10- >> computer( ThetaC, Eps, P, ThetaD, Nbr) :-
11-      ThetaC ~ ThetaD + Nbr * log( Nbr ),
12-      P ~ Nbr * 3,
13-      rule( Eps, P, Nbr).
14-
15- >> rule( medium, P, 3) :- P =< 25.
16- >> rule( low, P, 3) :- P > 25.
17- >> rule( high, P, 5) :- P =< 24.
18- >> rule( medium, P, 5) :- P ~ oc(24,30).
19- >> rule( low, P, 5) :- P > 30.
20- >> rule( high, P, 12) :- P =< 32.
21- >> rule( medium, P, 12) :- P ~ oc(32,45).
22- >> rule( low, P, 12) :- P > 45.
23-
24- %% PROPAGATION OF VALIDITY DOMAINS
25- >> P =< 15, Eps = high, ThetaS ~ cc(15,30), in(
Nbr, [3,5,12]),
```

```
26-     decoder( ThetaD, ThetaS),
27-     computer( ThetaC, Eps, P, ThetaD, Nbr).
```

The three first predicates (*receiver/2*, *decoder/2* and *computer/5*) reify the dependencies of the extra-functional properties defined on each component. The *rule/3* predicate is used by the *computer/5* predicate in order to bind the *Eps*, *P* and *Nbr* variables. The lines 25 to 27 is a request (**6**), which is made up of two parts:

1. line 25: the *ThetaS* (resp. *Nbr*, *P*, *Eps*) variable belongs to the [15;30] (resp. {3,5,12}, [0;15], {high}). These numerical constraints are either induced by the specifications and the environment, or by a designer who wants to know the global extra-functional impact of a specific stress applied on one or more properties.
2. line 26-27: represents the connection between the components, extracted from the component-based model. The extra-functional properties can be shared at this level between two components, like the *ThetaD* variable is shared by the decoder and the computer. It is also a request that the designer can answer. The answer computed by the inference engine is:

```
29-     ThetaS ~ cc(15, 15.5),
30-     Nbr = 5,
31-     ThetaD ~ cc(20, 20.5),
32-     ThetaC ~ cc(23.49, 24),
33-     Eps = high,
34-     P = 15.
```

The result is a piece of PrologIV code, which is not very easy to understand. The semantic of this result and the chaining of the rules by the inference engine are explained in the following section.

3.3 Inference and result

In a CLP software, it is important to underline the fact that the use of numerical predicates (+/3, -/3, ...) on a variable implies that this variable belongs to the real R or the integer N sets by construction. However, this kind of information induces few new knowledge by propagation and reduction of the intervals. That is the reason why, in line 25, we stress some properties in order to reduce their validity domains. These constraints are either provided by the specifications of the device ($\mathbf{q}_S \in [15;30]$, $\mathbf{nbr} \in \{3;5;12\}$) or by designer's requirements ($\mathbf{P} \leq 15$, $\mathbf{eps} = \text{high}$).

The computation of the current location has a power cost **P**, which depends on the number of active receivers **nbr** according the simple formula:

$$\mathbf{P} = \mathbf{nbr} * 3 . \qquad (7)$$

According to (**7**) and the requirement that the power must be less or equal than 15, we obtain that the number of active receivers must be equal to 3 or 5:

$$\mathbf{nbr} \in \{3; 5\} . \quad (\mathbf{8})$$

The lines 15 to 22 express the rule that binds the precision **eps**, the response time q_C and the power **P** of the *getLocation* service (Fig. 5). According to this rule, the constraints (**7**) and (**8**), and the fact that eps is required to be high, it is obvious that:

$$\mathbf{nbr} = 3, \quad (\mathbf{9})$$

$$\mathbf{P} = 15 . \quad (\mathbf{10})$$

The relationship that binds the response time q_C to the response time q_D and the number of active receivers **nbr** is:

$$q_C = q_D + \mathbf{nbr} * \log (\mathbf{nbr}) . \quad (\mathbf{11})$$

The second term (**nbr***log(**nbr**)) is the time spend by the computer to interpolates the position from the data. Since the validity domain of q_D and the value of **nbr** are known, we have:

$$q_C \in [23.49; 24] . \quad (\mathbf{12})$$

This interval can be propagated again through the response time chain of dependencies (Fig. 4), and finally we obtain the result on q_D and q_S:

$$q_D \in [20; 20.5] , \quad (\mathbf{13})$$

$$q_S \in [15; 15.5] . \quad (\mathbf{14})$$

That is the result obtain by the CLP software (l. 29-34).

4 Related works

QoS and extra-functional properties are not a new concept in the component-based software engineering [1][5]. Many authors have developed languages dedicated to extra-functional specifications: SMIL [18], TINA-ODL [19], QIDL [20], QDL [21] and so on. An interesting analysis of these languages and their common features is done by Aagedal [22]. He concludes that the QoS Modeling Language (QML) [9] is the most general language for QoS specification. Indeed, the concepts defined in QML are representative of specifications languages dedicated to the extra-functional aspects. These concepts have been integrated into QoSCL.

QML has three main abstraction mechanisms for the QoS specification: *contract type*, *contract* and *profile*. A contract type represents a specific QoS aspect, such as reliability for example. It defines valuable *dimensions*, used to characterize a particular QoS aspect. A contract is an instance of a contract type and represents a particular QoS specification. Finally, QoS profiles associate contracts with interfaces.

With QML, it is possible to write a contract which specifies that an operation must be completed with a delay less than 30 seconds:

```
type TimeOut = contract {
  delay : decreasing numeric s;
}

TimeOutImpl = TimeOut contract {
  delay < 30;
}

TimeOutProfile for ComputerI = profile {
   from getSignal require TimeOutImpl
}
```

In spite of its generality, QML, as the other languages mentioned above, does not have explicit dependencies between extra-functional properties. The properties are considered as independent quantities, evaluated at run time by monitors. But that does not match reality: we have shown that extra-functional properties have dependencies in the same way as a provided service depends on a set of required services. QoSCL makes these dependencies explicit.

The QoS specifications can be used either to implement contracts evaluation and renegotiation at run time, or to evaluate *a priori* the global quality of the assembly at design time.

G. Blair has proposed a specific architecture in order to manage component composition, based on a reflective component model [23][24]. In fact, the component reflection is dedicated to access at properties value and structural information of the composite. A set of rules, called *StyleRule*, manages the adaptation of the composite to its environment. In fact, these rules can be considered as model transformations used to reconfigure the composite: properties values, connections between components and graph actions.

In the various models shown, the extra-functional properties are not explicitly defined, neither their dependencies *a fortiori*. Consequently, it is not possible to predict at design time the global quality of the assembly.

Moreover, according to us, the use of rules in order to dynamically configure an assembly at run time is dangerous. Indeed, the authors of [25], which have implemented a declarative approach for adaptative components, underline the limit of such approach: the set of rules which governs the adaptation must respect completeness and uniqueness.

Completeness guarantees that for every possible situation there exits at least one adaptation action, and uniqueness ensures that this action is unique. The authors indicates that the two following properties can be enforced by use of the CLP. However, not only these two properties are not compatible with non-linear constraints [25], but also the extra-functional dependencies can be non-linear (see formula #4).

That is the reasons why we have chosen to implement the contracts evaluation and renegotiation in imperative language with a weaving aspect technology. Like Genβler and Zeidler [26], or in the Itacio tool [27], the use of CLP is kept till the design time, to check the validity of an assembly according to a set of rules (consistency rules,

contractual rules, etc..). However, none of them exhibits a component model with explicit extra-functional dependencies.

At last, researchers of the Software Engineering Institute at Carnegie Mellon University (USA) have underlined the importance to integrate more analysis technology in components-based software [28]. We think that this integration must be considered at the highest level: the component model. QoSCL (specification) and its dedicated tools (validation and renegotiation) presented in this paper are a contribution in this sense.

5 Conclusion and future work

In mission-critical component based systems, it is particularly important to be able to explicitly relate the QoS contracts attached to provided interfaces of components with the QoS contracts obtained from their required interfaces. In this paper we have introduced a language called QoSCL (defined as an add-on to the UML2.0 component model) to let the designer explicitly describe and manipulate these higher level contracts and their dependencies.

An important feature of QoSCL is that this language distinguishes four degrees of conformance for component-based diagram refined with extra-functional properties:

- 1st degree: the functional conformance. That is the well-known conformance between provided and required typed interfaces.
- 2nd degree: because the QoSCL *ContractTypes* are interfaces which operations are *Dimensions*, the extra-functional conformance is very close of the previous degree, applied to the *ContractTypes* of a component.
- 3rd degree: each extra-functional property is a valuable quantity, constrained by body, pre and post conditions. At the assembly level, the conformance of these three constraints must be checked.
- 4th degree: the quality levels of the whole of the extra-functional properties must be compliant with the networks of (non-linear) relations.

The other main feature of QoSCL contracts is that they can be exploited for:

- validation of individual components, by automatically weaving contract monitoring code into the components for example;
- validation of a component assembly, including getting end-to-end QoS information inferred from individual component contracts, by automatic translation into a CLP(R) language.

Both validation activities builds on the model transformation framework developed at INRIA (cf. http://modelware.inria.fr). Preliminary implementations of these ideas have been prototyped in the context of the QCCS project (cf. http://www.qccs.org) for the weaving of contract monitoring code into components part, and on the Artist project (http://www.systemes-critiques.org/ARTIST) for the validation of a component assembly part. Both parts still need to be better integrated with UML2.0 modelling environments, which is work in progress.

Acknowledgments

This work has been partly founded by the ARTIST IST project. The authors thank Jean-Philippe Thibault for the discussions.

References

1. "Component software, beyond object-oriented programming", 2nd ed., by C. Szyperski. Addison-Wesley, 2002
2. "CORBA Components, v3.0", adopted specification of the OMG, June 2002.
3. "EJB 2.1 Specification Final Release", Sun, 2002.
4. "Applied Microsoft .Net framework programming" by J. Richter. Microsoft Press, January 23, 2002.
5. "Synchronization in concurrent object-oriented languages:expressive power, genericity and inheritance" by C. McHale. Doctoral dissertation, Trinity College, Dept. of computer science, Dublin, 1994.
6. "Making components contract aware" by A. Beugnard, J.M. Jézéquel, N. Plouzeau and D Watkins in Computer, pp. 38-45, IEEE Computer Society, July 1999.
7. "UML Superstructure 2.0", OMG, August 2003.
8. "Constraint logic programming" by J. Jaffar and Lassez J.L. in proceedings of 14th ACM Symposium on principles of programming languages (POPL'87), pp-111-119, ACM, 1987
9. "QoS specification in distributed object systems" by S. Frolund and J. Koistinen in Distributed Systems Engineering, vol. 5, July 1998, The British Computer Society
10. http://www.qccs.org, Quality Control of Component-based Software (QCCS) European project home page.
11. "Reflective model driven engineering" by J. Bézivin, N. Farcet, J.-M. Jézéquel, B. Langlois, and D. Pollet in Proceedings of UML 2003, San Francisco, volume 2863 of LNCS, pages 175-189. Springer, October 2003.
12. "Logical arithmetic" by J.G. Cleary in Future computing systems 2, n°2, pp-125-149, 1987
13. "Applying interval arithmetic to real, integer and Boolean constraints" by F. Benhamou and W.J. Older, in Journal of logic programming 32, n°1, pp-1-24, 1997
14. "PrologIV: reference manual and user's guide", PrologIA, Tech. Rep., 1994
15. "Hybrid Problem Solving in ECLiPSe", by F. Ajili and M. Wallace, §6 in "Constraint and integer programming: toward a unified methodology", pp. 169-201. Michela Milano. Kluwer Academic Publishers. October 2003.
16. "An interval component for continuous constraints". Journal of Computational and Applied Mathematics, 2003.
17. http://www.sics.se/sicstus/
18. "Synchronized Multimedia Integration Language 2.0 specification" by W3C: http://www.w3.org/TR/smil20
19. "TINA object definition language manual", TINA-C, report : TP_NM_002_2.2_96, 1996
20. "QoS, aspects of distributed programs" by C. Becker and K. Geiths, in proceedings of international workshop on aspect-oriented programming at ICSEE'98, Kyoto, Japan, 1998.

21. “Integration of QoS in distributed objects systems” by J. Daniels, B. Traverson and S. Vignes, in proceedings of IFIP TC6 WG6.1 2nd international working conference on Distributed Applications and Interoperable Systems (DAIS’99), Helsinki, Finland, pp. 31-43, 1999.
22. “Quality of service support in development of distributed systems” by J.O. Aagedal. Ph.D thesis report, University of Oslo, Dept. Informatics, March 2001.
23. “v-QoS project home page”: http://www.comp.lancs.ac.uk/computing/users/lb/v-qos.html
24. “A reflective and architecture aware component model for middleware composition management” by G.S. Blair, R. Silva Moreira and E.M. Carrapatose in 3d international symposium on Distributed Objects and Application (DOA’01), September 2001, Roma (It).
25. “ A declarative approach for designing and developing adaptative components” by P. Boinot, R. Marlet, G. Muller and C. Consel, in 15th international conference on Automated Software Engineering (ASE’01), pp-111-119, September 2000, Grenoble (Fr).
26. “Rule-driven component composition for embedded systems” by. T. Genssler and C. Zeidler in 4th ICSE workshop on component-based software engineering: component certification and system prediction, ICSE’01, Toronto (Ca).
27. “Itacio: a component model for verifying software at construction time” by A. Cernuda del Rio, J.E. Labra Gayo and J.M. Cueva Lovelle in international workshop on component-based engineering, ICSE’00, Mimerick (Ir).
28. “Packaging and deploying predictable assembly” by S.A. Hissam, G.A. Moreno, J. Stafford and K.C. Wallnau, in the proceedings of IFIP/ACM working conference on component deployment (CD2002), , pp. 108-124, Berlin, Germany, June 2002.