FMCO'03

An MDA approach to tame component based software development

Jean-Marc Jézéquel, Olivier Defour and Noël Plouzeau

IRISA - Université de Rennes 1

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

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

Abstract. The aim of this paper is to show how the Model Driven Architecture (MDA) can be used in relation with component based software engineering. A software component only exhibits its provided or required interfaces, hence defining basic *contracts* between components allowing one to properly wire them. 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. We thus introduce a QoS contract model (called QoSCL for QoS Constraint Language), allowing QoS contracts and their dependencies to be modeled in a UML2.0 modeling environment. Building on Model Driven Engineering techniques, we then show how the very same QoSCL contracts can be exploited for (1) validation of individual components, by automatically weaving contract monitoring code into the components; and (2) 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

Szyperski [22] remarked that while objects were good units for modular composition at development time, they were not so good for deployment time composition, and he formulated the now widely accepted definition of a software component: "*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 configured or re-configured at run-time, as in CORBA [15], or .NET [20].

A software component only exhibits its provided or required interfaces, hence defining basic *contracts* between components allowing one to properly wire them. 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.

It is then natural that people resorted to modelling to try to master this complexity. According to Jeff Rothenberg, "*Modeling, in the broadest sense, is the cost-effective use of something in place of something else for some cognitive purpose. It allows us to use something that is simpler, safer or cheaper than reality instead of reality for some purpose. A model represents reality for the given purpose; the model is an abstraction of reality in the sense that it cannot represent all aspects of reality. This allows us to deal with the world in a simplified manner, avoiding the complexity, danger and irreversibility of reality*." Usually in science, a model has a different nature that the thing it models. Only in software and in linguistics a model has the same nature as the thing it models. In software at least, this opens the possibility to automatically derive software from its model. This property is well known from any compiler writer (and others), but it was recently be made quite popular with an OMG initiative called the Model Driven Architecture (MDA).

The aim of this paper is to show how MDA can be used in relation with component based software engineering. We introduce a QoS contract model (called QoSCL for QoS Constraint Language), allowing QoS contracts and their dependencies to be modeled in a UML2.0 [13] modeling environment. Building on Model Driven Engineering techniques, we then show how the very same QoSCL contracts can be exploited for (1) validation of individual components, by automatically weaving contract monitoring code into the components; and (2) 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.

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, their contracts and their dependencies, and describes the QoS Constraint Language (QoSCL). Section 3 discusses the problem of validating individual components against their contracts, and proposes a solution based on automatically weaving reusable contract monitoring code into the components. Section 4 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. This is applied to the GPS system example, and experimental results are presented. Finally, Section 5 presents related works.

2 The QoS Contracts Language

2.1 Modeling component-based systems

In modelling techniques such as UML2.0 for example, a component is a behavioural abstraction of a concrete physical piece of code, called artifacts. A component has required and provided ports, which are typed by interfaces. These interfaces represent the required and provided services implemented by the modelled artifact. The relationship between the required and provided services within one component must be explicitly stated. The knowledge of this relationship is of utmost importance to the component-based application designer. In the rest of this section, we address this relationship using the example of a GPS device.

A GPS device computes its current location from satellite signals. Each signal contains data which specifies the identity of the emiting satellite, the time of its emission, the orbital position of the satellite and so on. In the illustrating example, each satellite emits a new data stream every fifteen seconds.

In order to compute its current location, the GPS device needs at least three signals from three different satellites. The number of received signals is unknown *a priori*, because obstacles might block the signal propagation.

Our GPS device is modeled as a component which provides a *getLocation()* service, and requires a *getSignal()* service from Satellites components. The GPS component is made up of four components:

- the decoder which contains twelve satellite receivers (only three are shown on Fig. 1). This element receives the satellite streams and demutiplexes it in order to extract the data for each satellite. The number of effective data obtained via the *getData()* service depends not only on the number of powered receivers, but also on the number of received signals. Indeed, this number may change at any time.
- The computer which computes the current location (*getLocation()*) from the data (*getData()*) and the current time (*getTime()*).
- The battery which provides the power (*getPower()*) to the computer and the decoder.
- The clock component which provides the current time (*getTime()*).

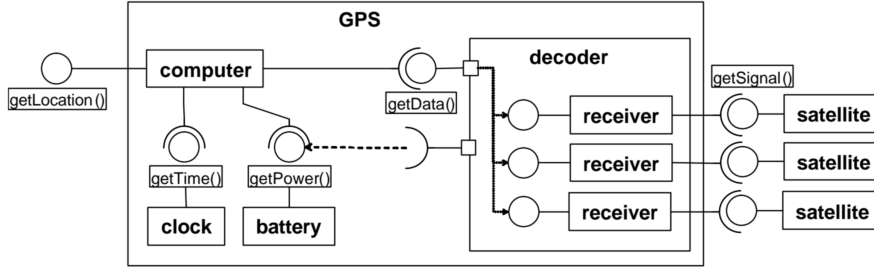

Fig. 1. The GPS component-based model

2.2 Contract aware components

In component-based models, the services are usually specified at a syntactic level. This level of specification is not precise enough. Indeed, a service can be unavailable according to the state of the environment and, reciprocally, the environment can be modified by the execution of a service.

Following [2] component contracts can be classified into four levels. The first level is the type compatibility. The second level adds pre/post-conditions: the operation's behavior is specified by using Boolean assertions for each service offered, called pre and post-conditions, as well as class invariants [14]. The third level adds synchronization constraints and the fourth level provides extra-functional constraints. To be more precise, we can build on the well-known idea of design-by-contract [12] negotiable contracts for components. These contracts ensure that a service will perform correctly.

In the previous section 2.1, we have said that a dependency relationship always exists inside one component between its provided and required services. A component provides its services inasmuch as its environment provides the services that it requires. All components always support this implicit contract. The extra-functional properties, which are intrinsic features of services, inherit this dependency relationship. The quality of a provided service depends on the quality of required services that it depends on. This fact is illustrated in our example.

The GPS application contains several time out constraints. For instance, the provided *getLocation()* service must ensure that it is completed in a delay less than 30s, whereas the *getData()* service must be completed in less than 25 s for example.

However, it is obvious that the time spent to acquire data from the decoder, denoted *TethaD*, has a direct impact on the global cost in time of the *getLocation()* service, denoted *ThetaC*. Not only *ThetaC* depends on *ThetaD*, but also on the number of active receivers, denoted *Nbr*, because of the interpolation algorithm implemented by the Computer component. *ThetaD* and *Nbr* are two extra-functional properties associated to the *getData()* service provided by the Decoder component. The relation that binds these three quantities is:

$$ThetaC = ThetaD + Nbr * \log(Nbr) . \quad (1)$$

Each receiver demultiplexes a signal, in order to extract the data. This operation has a fixed time cost: nearly 2 seconds. In addition, the demultiplexed signals must be transformed into a single data vector. This operation takes 3 s. If *ThetaR* (resp. *ThetaS*) denotes the time spent by the receiver to complete the *getDataI()* service (resp. the satellite to complete its *getSignal()* service), then we have the two following formulae:

$$ThetaR = ThetaS + 2\ , \qquad (\mathbf{2})$$

$$ThetaD = \max(ThetaR) + 3\ . \qquad (\mathbf{3})$$

There exist many QoS contracts languages which allow the designer to specify the extra-functional properties and their constraints on the provided interfaces only (see section 5). However, none of them allow specifying dependency relationships between the provided and required services of a component. To overcome this limitation we introduce the QoS Constraint Language (QoSCL). This language includes the fundamental QoS concepts defined in the well-known precursor QML [5]. It is the cornerstone to implement in a second time a QoS prediction tool.

2.3 Extra-functional dependencies with QoSCL

Our own contract model for extra-functional contracts extends the UML2.0 components metamodel. We designed the QoSCL notation with the following objectives in mind.

1. Since the extra-functional contracts are constraints on continuous values within multidimensional spaces, we wanted to keep the QML definitions of dimensions and contract spaces.
2. Since our extra-functional contracts would be used on software components with explicit dependency specification, we needed means to express a provided contract in terms of required contracts.
3. Since we targeted platform independent designs, we wanted to use the UML notation and its extension facilities.

We thus designed our extra-functional contract notation as an extension of the component part of the UML2.0 metamodel:

- *Dimension*: is a QoS property. This metaclass inherits the operation metaclass. According to our point of view, a QoS property is a valuable quantity and has to be concretely measured. Therefore we have chosen to specify a means of measurement rather than an abstract concept. Its parameters are used to specified the (optional) others dimensions on which it depends. The type of a Dimension is a totally ordered set, and it denotes its unit. The pre and post-conditions are used to specify constraints on the dimension itself, or its parameters.
- *ContractType*: specializes Interface. It is a set of dimensions defining the contract supported by an operation. Like an interface, a ContractType is just a specification without implementation of its dimensions.

- *Contract*: is a concrete implementation of a ContractType. The dimensions specified in the ContractType are implemented inside the component using the aspect weaving techniques (see section 3). An *isValid()* operation checks if the contract is realized or not.
- *QoSComponent* extends Component, and it has the same meaning. However, its ports provides not only required and provided interfaces which exhibit its functional behaviour, but also ContractTypes dedicated to its contractual behaviour.

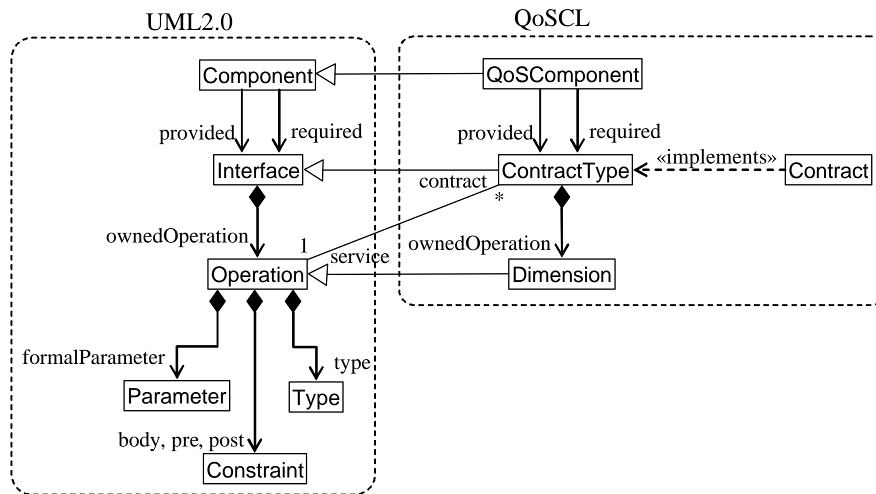

Fig. 2. The QoSCL metamodel

With the QoSCL metamodel, it is possible to specify contracts, such as the Time-Out contract useful for our GPS, as an Interface in any UML case tool:

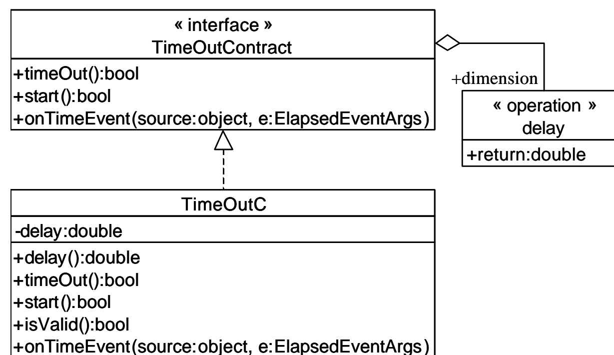

Fig. 3. The TimeOut contract with QoSCL

The QoSCL metamodel handles three specific aspects of contracts: dependency, composition, and adaptative behaviour. The dependency is the core of this work, and

our main contribution to enhance existing extra-functional contracts specification languages, such as QML. QoSCL makes it also possible to model a composite contract via generalization association. At last, like any abstract functional model, it is possible to implement different behaviors for the same Operation, such as a Dimension. Thus, the renegotiation of a contract can be implemented according to its environment. This behavior can be specified thanks the UML2.0 sequence diagrams, activity diagrams or state machine for instance.

3 Implementing contract-aware components

QoSCL allows the expression of functional and extra-functional properties in a software component. The declared properties are useful to the software designer because this gives predictability to a component's behaviour. However, this predictability is valid only if the component implementation really has the behaviour declared by the component. This implementation validity is classical software validation problem, whatever the kind of contracts used [11].

These problems are usually addressed by two families of techniques. A first family is based on testing: the system under test is run in an environment that behaves as described in a test case. An oracle observes the behaviour of the system under test and then decides whether the behaviour is allowed by the specification. A second family of techniques relies on formal proof and reasoning on the composition of elementary operations.

Standard software validation techniques deal with pre/post-condition contract types [12]. Protocol validation extends this to the synchronization contract types [8]. The rest of this section discusses issues of testing extra-functional property conformance.

3.1 Testing extra functional behaviour

Level 3 contracts (*i.e.* contracts that include protocols) are more difficult to test because of non-deterministic behaviours of parallel and distributed implementations. One of the most difficult problems is the consistent capture of data on the behaviour of the system's elements. Level 4 contracts (*i.e.* extra-functional properties) are also difficult to test for quite similar reasons. Our approach for testing level 4 contracts relies on the following features:

- existence of probes and extra-functional data collection mechanisms (monitors);
- test cases;
- oracles on extra-functional properties.

In order to be testable, a component must provide probe points where basic extra-functional data must be available. There are several techniques to implement such probe points and make performance data available to the test environment.

1. The component runtime may include facilities to record performance data on various kinds of resources or events (e.g. disk operations, RPC calls, etc). Modern operating systems and component frameworks now provide performance counters that can be "tuned" to monitor runtime activity and therefore deduce performance data on the component's service.
2. The implementation of the component may perform extra computation to monitor its own performance. This kind of "self monitoring" is often found in components that are designed as level 4 component from scratch (*e.g.* components providing multimedia services).
3. A component can be augmented with monitoring facilities by weaving a specific monitor piece of model or of code. Aspect-oriented design (AOD) or aspect-oriented programming can help in automating this extension.

We have chosen this latter approach as our main technique for designing monitors. This choice was motivated mainly by the existence of "legacy" components from industrial partners [17]. From a software design process point of view, we consider that designing monitors is a specialist's task. Monitors rely on low level mechanisms and/or on mechanisms that are highly platform dependant. By using aspect-oriented design (AOD), we separate the component implementation model into two main models: the service part that provides the component's functional services under extra-functional contracts, and the monitor part that supervises performance issues. A designer in charge of the "service design model" does not need to master monitor design. A specific tool[1] (a model transformer) [24] is used to merge the monitor part of the component with its service part.
More precisely, a contract monitor designer provides component designers with a reusable implementation of a monitor. This implementation contains two items: a monitor design model and a script for the model transformer tool (a weaver). The goal of this aspect weaver is to modify a platform specific component model by integrating new QoSCL classes and modifying existing class and their relationships.

3.2 A practical example of weaving

As we have said in the last paragraph of section 2, QoSCL allows us to model the structural, behavioral and contractual components features. These three aspects can be specified using the dedicated UML2.0 diagrams. The QoS aspect weaver is a mechanism integrated into Kase, which:

- modifies the UML diagram (add new classes and associations)
- modifies the behavior of the targeted service

Thanks to QoSCL, it is possible to specify into Kase the contract types and their implementation such as TimeOut and TimeOutC (Fig. 4). According to our vision, detailed in the QoSCL section (§2.3), the TimeOut contract is an interface, which has a special operation denoting the "delay" dimension. The TimeOutC is a .Net class that implements the TimeOut interface. The value of the "delay" dimension is

[1] The Kase tool is developed by TU-Berlin with the support of the European Project "Quality Control of Component-based Software" (QCCS) [17].

implemented like a private attribute (*-delay:double*) and its related access/evaluation method (*delay():double*).

A QoS aspect not only specifies how the structural diagram will be modified, but also how the monitored part and the monitor cooperate: when does the timer start, when does it stop, who handles timeout, etc… This part of the aspect is specified using the Hierarchical Message Sequence Charts (HMSC) notation in the UML 2.0. Fig. 5 shows the behavior of a contractual service, called op(), as a HMSC diagram. The op() operation is the service which must verify a TimeOut contract. The op_c() operation is a new operation, which realizes the op() service and evaluates the TimeOut contract below (Fig. 5). This service has two possible behaviors, depending on whether the op() service finishes before or after the timer.

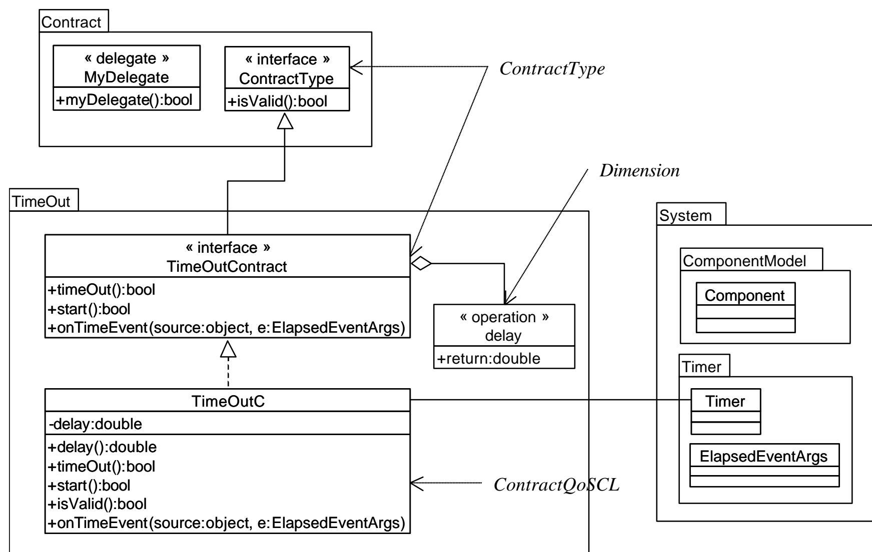

Fig. 4. TheTimeOut contract model for .Net

In addition of its structural (Fig. 4) and behavioral (Fig. 5) parts, a contractual QoS aspect has pre-conditions that must be met at weaving time. For example, a :Cl class abides a TimeOut contract under the condition that it implements the op() service of course. In our tool, the aspect is concretely weaved in the UML diagram by a Python script, which:

- checks the aspect pre-conditions;
- weaves the aspect if these preconditions are satisfied, and this weaving adds new classes, modifies constructors and operations, etc).

The QoS aspect weaver implemented in the Käse tool allows us to:

- specify a QoS aspect;
- implement an evaluation of this aspect for a targeted service.

According to the QoSCL point of view, contracts can be specified at design time as specialized interfaces. Therefore, connecting two components at binding time is

easy, using their respectively compliant required and provided interfaces. The QoS aspect weaver implemented in Käse allows to implement in C# any contract type.

In case of failure, an extra-functional contract can be renegotiated. For instance, a time out contract that fails too often obviously needs to be adjusted (alternatively the service bound to that contract has to be shut down).

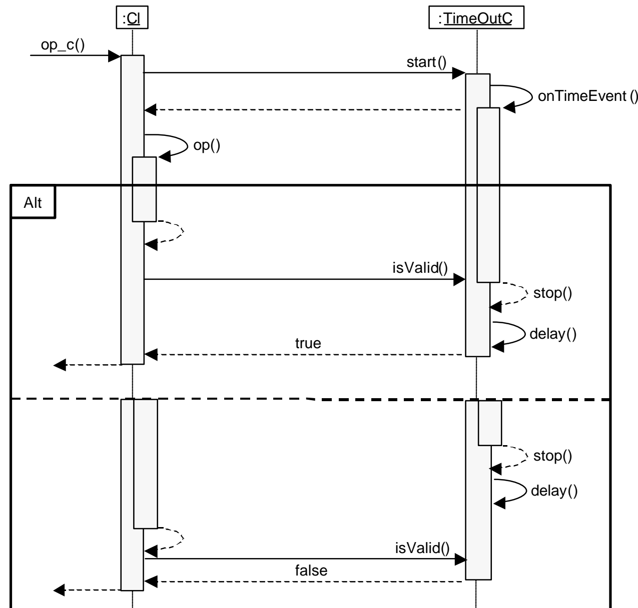

Fig. 5. Behavior of the op() service with evaluation of a TimeOut contract

3.3 Limitations of extra-functional property testing

The QoSCL notation and the monitor integration technique help the component designer to define and check extra-functional properties. However, application designers rely on component assemblies to build applications. These designers need to estimate at design time the overall extra-functional properties of a given assembly. Using the techniques presented above, they can perform a kind of integration testing. The tests aim at validating the extra-functional behavior of the assembly with respect to the global specification of the application. However, the application designers often have trouble to select and configure the components, make the assembly and match the global application behavior. Conversely, some applications are built with preconfigured components and the application designer needs to build a reasonable specification of the overall extra-functional behavior of the application.

4 Predicting extra-functional properties of an assembly

4.1 Modeling a QoS-aware component with QoSCL

QoSCL is a metamodel extension dedicated to specify contracts whose extra-functional properties have explicit dependencies. Models can be used by aspect weavers in order to integrate the contractual evaluation and renegotiation into the components. However, at design time, it is possible to predict the global quality of the composite software.

Predicting a behaviour is difficult. In the best cases, the behaviour can be proved but this. Otherwise, the behaviour is predicted with uncertainty. Since we want to predict the quality of a composite, i.e. la value of a set of extra-functional properties, this uncertainty will be translated into a numerical interval or an enumerated set of values, called validity domains.

The dependencies defined in QoSCL, which bind the properties, are generally expressed either as formulae or as rules. The quality of a service is defined as the extra-functional property's membership of a specific validity domain. Predicting the global quality of a composite is equivalent to the propagation of the extra-functional validity domains through the dependencies.

For instance, we have defined in section §2.2 a set of extra-functional properties that qualifies different services in our GPS component-based model. In addition, we have specified the dependencies between the extra-functional properties as formulae. This knowledge can be specified in QoSCL. The Fig. 6 below represents the computer component (Fig. 1) refined with contractual properties and their dependencies:

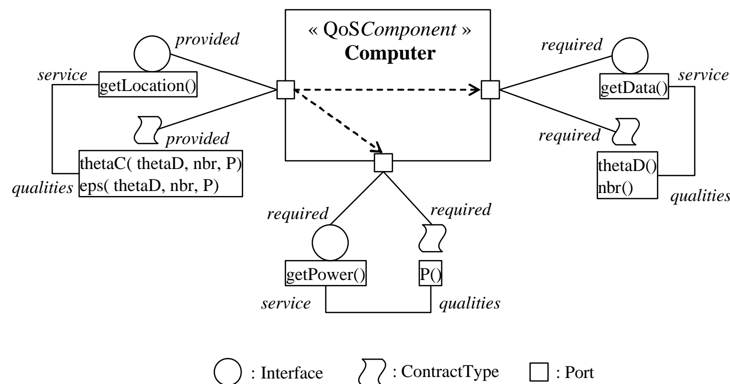

Fig. 6. Quality attributes and dependencies specification of a component

The rules that govern the connection between two (functional) ports are also valid for ports with required or provided ContractTypes. Thus, a port that requires a service with its specific QoS properties can only be connected to another Port that provides this service with the same quality attributes.

Specifying the QoS properties of required and provided services of a component is not enough to predict the quality of an assembly at design time. Additional information must be supplied:

- constraints on the value of the QoS properties are needed to get the parties to negotiate and to agree; they explain the level of quality required or provided for a service by a component;
- the dependency between these values is an important kind of relationship; it can be described either as with a function (for instance: *ThetaC* = *ThetaD* + *Nbr* * log(*Nbr*) **(1)**) or with a rule (if *Nbr* = 3 and *Eps* = medium then *ThetaC* ≤ 25).

In other words, these constraints can be stated as OCL [14] pre and post-conditions on the Dimensions. For instance:

```
context Computer::thetaC( thetaD : real, nbr : int,
P : real) : real
    pre: thetaD >= 0 and P >= 0
    post: result = thetaD + nbr * log( nbr ) and P =
    3*nbr
```

At design time, the global set of pre and post-conditions of all specified Dimensions of a component builds a system of non-linear constraints that must be satisfied. The Constraint Logic Programming is the general framework to solve such systems. Dedicated solvers will determine if a system is satisfied, and in this case the admissible interval of values for each dimension stressed.

4.2 Prediction of the GPS quality of service

In this section we present the set of constraints for the GPS component-based model (Fig. 1). A first subset of constraints defines possible or impossible values for a QoS property. These admissible value sets come on the one hand from implementation or technological constraints and on the other hand from designers' and users' requirements about a service. The fact that the *Nbr* value is 3, 5 or 12 **(2)**, or *ThetaC* and *ThetaD* values must be real positive values (**3-4)** belongs to the first category of constraints. Conversely, the facts that *Eps* is at least medium **(5)** and *P* is less or equal than 15mW **(6)** are designers or users requirements.

$$\mathbf{Nbr \in \{3, 5, 12\},} \quad (2)$$
$$\mathbf{ThetaC \geq 0,} \quad (3)$$
$$\mathbf{ThetaD \geq 0,} \quad (4)$$
$$\mathbf{Eps \in \{medium, high\},} \quad (5)$$
$$\mathbf{P \leq 15.} \quad (6)$$

Secondly, constraints can also explain the dependency relationships that bind the QoS properties of a component. For instance, the active power level *P* is linearly dependent on the *Nbr* number of receivers according to the formula:

$$P = 3 * Nbr. \quad (7)$$

Moreover, the time spent by the getLocation() service (*ThetaC*) depends on the time spend by the *getData()* service (*ThetaD*) and the number of data received (*Nbr*), according the equation **(1)**. Lastly, a rule binds the precision *Eps*, the time spent to compute the current position *ThetaC* and the number of received data (*Nbr*). The following diagram (Fig. 7) presents this rule:

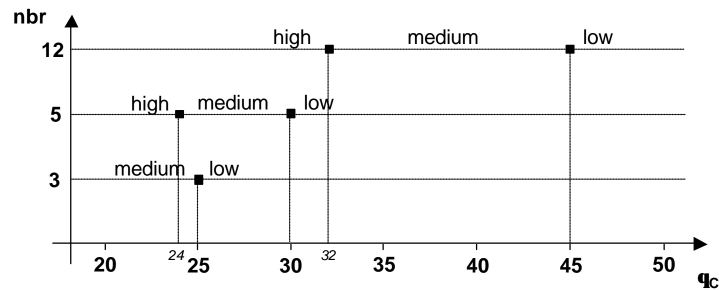

Fig. 7. The rule that binds the Eps, Nbr and ThetaC dimensions

All these constraints, expressed in OCL syntax, can be translated into a specific CLP-compliant language, using a Model Transfomation [24]. For instance, we present below the result of a such transformation applied to the computer QoSComponent (Fig. 6) and its OCL conditions (using the Eclipse™ syntax):

```
01- computer( [ ThetaC, Eps, P, ThetaD, Nbr ] ) :-
02-       ThetaC $>= 0, Eps = high, P $>= 0,
03-       ThetaD $>= 0, member( Nbr, [3,5,12]),
04-       ThetaC $>= 0, ThetaD $>= 0,
05-       ThetaC $= ThetaD + Nbr * log(Nbr),
06-       P $= Nbr * 3,
07-       rule( Eps, ThetaC, Nbr).
08-
09- rule( medium, ThetaC, 3) :- ThetaC  $=< 25.
10-       rule( low, ThetaC, 3) :- ThetaC  $> 25.
11- rule( high, ThetaC, 5) :- ThetaC  $=< 24.
12- rule( medium, ThetaC, 5) :- ThetaC $>24,
13-       ThetaC $=< 30.
14- rule( low, ThetaC, 5) :- ThetaC  $> 30.
15- rule( high, ThetaC, 12) :- ThetaC  $=< 32.
16- rule( medium, ThetaC, 12) :- ThetaC  $> 32,
17-       ThetaC $=<45.
18- rule( low, ThetaC, 12) :- ThetaC  $> 45.
```

The first line (01) indicates the QoS properties bound by the component. The two following lines (02, 03) are the constraints on the admissible values for these QoS properties, and lines 05 to 07 are the dependency relationships (**1-7** and Fig. 7) that bind them.

For each component, it is necessary to check its system of constraints, in order to compute its availability. The result of such request is the whole of admissible values for the QoS properties of the component. Thus, for the computer component, the solutions for the admissible QoS properties values are enumerated below:

ThetaC	*ThetaD*	*Eps*	*P*	*Nbr*
[3.49 .. 24.0]	[0.0 .. 20.51]	high	15	5
[12.95 .. 32.0]	[0.0 .. 19.05]	high	36	12

The requirement about the estimated position (*Eps* = high) implies that:

- the number of data channels must be either 5 or 12,
- consequently, the active power is either 15 or 36mW,
- and the response times of the *getLocation*() ands *getData*() services are respectively in the [3.49; 32.0] and [0.0; 20.51] real intervals.

At this time, the designer knows the qualitative behavior of all of its components. It is also possible to know the qualitative behavior of an assembly, by conjunction of the constraints systems and unification of their QoS properties.

The following constraint program shows the example of the GPS component:

```
19- satellite( [ ThetaS ] ) :-
20-       ThetaS $>= 15, ThetaS $=< 30.
21-
22- battery( [ P ] ) :-
23-       P $>= 0,
24-       P $=< 15.
25-
26- receiver( [ ThetaR, ThetaS ] ) :-
27-       ThetaR $>= 0, ThetaS $>= 0,
28-       ThetaR $= ThetaS + 2.
29-
30- decoder( [ ThetaD, ThetaS, Nbr ] ) :-
31-       ThetaD $>= 0, ThetaS $>= 0,
32-       member( Nbr, [3,5,12]),
33-       receiver( [ ThetaR, ThetaS ] ),
34-       ThetaD $= ThetaR + 3.
35-
36- gps( [ ThetaC, Eps, ThetaS ] ) :-
37-       ThetaC $>= 0, Eps = high, ThetaS $>= 0,
38-       computer( [ ThetaC, Eps, P, ThetaD, Nbr ] ),
39-       decoder( [ ThetaD, ThetaS, Nbr ] ),
40-       battery( [ P ] ).
```

Similarly, the propagation of numerical constraints over the admissible sets of values implies the following qualitative prediction behavior of the GPS assembly:

ThetaC	*ThetaS*	*Eps*
[23.49 .. 24.0]	[15.0 .. 15.50]	high

The strong requirement on the precision of the computed location implies that the satellite signals have to be received by the GPS component with a delay less than 15.5 s. In this case, the location will be computed in less than 24 s.

5 Related work

In the Component-Based Software Engineering community, the concept of predictability is getting more and more attention, and is now underlined as a real need [4]. Thus, the Software Engineering Institute (SEI) promotes its Predictable Assembly from Certifiable Components (PACC) initiative: how component technology can be extended to achieve predictable assembly, enabling runtime behavior to be predicted from the properties of components. The ongoing work concentrates in a Prediction-Enabled Component Technology (PECT) as a method to integrate state-of-the-art techniques for reasoning about software quality attributes [23].

In the introduction of the SEI's second workshop on predictable assembly [21], the authors note that component interfaces are not sufficiently descriptive. A syntax for defining and specifying quality of service attributes, called QML, is defined by Frolund and Koistinen in [5], directly followed by Aagedal [1]. The Object Management Group (OMG) has developed its own UML profile for schedulability, performance and time specification [16]. These works emphasize the contractual use of QoS properties, and constitute the fundamental core of QoS specifications.

In the previous approaches, a QoS property is specified as a constant: they do not allow the specification of QoS properties dependency relationships. In contrast, Reussner proposes its parameterized contracts [18]: the set of available services provided by a component depends on its required services that the context can provide. This concept is a generalization of the design-by-contract [11]. The same author has published in 2003 a recent extension of its work dedicated to the QoS [19]. He models the QoS dependency with Markov chains where:

- the states are services, with their QoS values;
- the transitions represent the connections (calls) between components, i.e. the architecture of an assembly;
- the usage profile of the assembly is modeled by probabilities for calls to a provided service. Usage profiles are commonly modeled by Markov chains since Cheung [3] or Whittaker and Thomason [25].

From an assembly model and its usage profile, it seems possible to generate the associated Markov chain and to predict the QoS level of provided services. Conversely, it is not possible to invert the prediction process, in order to propagate a

particular QoS requirement applied on a provided service on the QoS properties of required services that it depends. Moreover, via the Chapman-Kolmogorov equation, the Markov processes handle only probabilities, and they are not able to reason about formal un-valued variables. For instance, it is impossible to compare the n^2 and n*log(n) complexity of two sort algorithms.

Constraints solvers over real intervals and finite domains have already been used in the context of the software engineering. For instance, logic programming techniques can generate test cases for functional properties. More precisely, this technique allows a more realistic treatment of bound values [10]. About the software functional aspect, many authors have successfully used the constraints logic programming, based on translations from the source code to test or its formal specification into constraints: the GATEL system [9] translates LUSTRE [7] expressions, and A. Gotlieb defines directly its transformation from C [6]. The works mentioned above focus on the functional aspects of software only, while our approach encompasses extra-functional properties.

6 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 notation 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 have shown how the very same QoSCL contracts can then be exploited for:

1. validation of individual components, by automatically weaving contract monitoring code into the components;
2. 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.

Both validation activities build 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.

References

1. Aagedal J.O.: "*Quality of service support in development of distributed systems*". Ph.D thesis report, University of Oslo, Dept. Informatics, March 2001.

2. Beugnard A., Jézéquel J.M., Plouzeau N. and Watkins D.: "*Making components contract aware*" in Computer, pp. 38-45, IEEE Computer Society, July 1999.
3. Cheung R.C.: "*A user-oriented software reliability model*" in IEEE Transactions on Software Engineering vol. 6 (2), pp. 118-125, 1980.
4. de Roever W.P.: "*The need for compositional proof systems: a survey*" in Proceedings of the Int. Symp. COMPOS'97, Bad Malente, Germany, Sept. 8-12, 1997.
5. Frolund S. and Koistinen J.: "*QoS specification in distributed object systems*" in Distributed Systems Engineering, vol. 5, July 1998, The British Computer Society.
6. Gotlieb A., Botella B. and Rueher M.: "*Automatic test data generation using constraint solving techniques*" in ACM Int. Symp. on Software Testing and Analysis (ISSTA'98), also in Software Engineering Notes, 23(2):53-62, 1998.
7. Halbwachs N., Caspi P., Raymond P. and Pillaud D.: "*The synchronous data flow programming language LUSTRE*" in Proc. of IEEE, vol. 79, pp.1305-1320, Sept. 1991.
8. McHale C.: "*Synchronization in concurrent object-oriented languages: expressive power, genericity and inheritance*". Doctoral dissertation, Trinity College, Dept. of computer science, Dublin, 1994.
9. Marre B. and Arnould A.: "*Test sequences generation from luster descriptions: Gatel*" in 15th IEEE Int. Conf. On Automated Software Engineering (ASE), pp. 229-237, Sept. 2000, Grenoble, France.
10. Meudec C.: "*Automatic generation of software test cases from formal specifications*". PhD thesis, Queen's University of Belfast, 1998.
11. Meyer B.: "*Object oriented software construction*", 2nd ed., Prentice Hall, 1997.
12. Meyer B.: "*Applying design by contract*" in IEEE Computer vol. 25 (10), pp. 40-51, 1992.
13. Object Management Group: "*UML Superstructure 2.0*", OMG, August 2003.
14. Object Management Group: "*UML2.0 Object Constraint Language RfP*", OMG, July 2003.
15. Object Management Group: "*CORBA Components, v3.0*", adopted specification of the OMG, June 2002.
16. Object Management Group: "*UML profile for schedulability, performance and time specification*". OMG adopted specification n° ptc/02-03-02, March 2002.
17. http://www.qccs.org, Quality Control of Component-based Software (QCCS) European project home page.
18. Reussner R.H.: "*The use of parameterized contracts for architecting systems with software components*" in Proc. of the 6th Int. Workshop on Component-Oriented Programming (WCOP'01), June 2001.
19. Reusnerr R.H., Schmidt H.W. and Poernomo I.H.: "*Reliability prediction for component-based software architecture*" in the Journal of Systems and Software, vol. 66, pp. 241-252, 2003.
20. Richter, J.: "*Applied Microsoft .Net framework programming*". Microsoft Press, January 23, 2002.
21. Stafford J. and Scott H.: "*The Software Engineering Institute's Second Workshop on Predictable Assembly: Landscape of compositional predictability*". SEI report n° CMU/SEI-2003-TN-029, 2003
22. Szyperski, C.: "*Component software, beyond object-oriented programming*", 2nd ed., Addison-Wesley, 2002
23. Wallnau K.: "*Volume III: A technology for predictable assembly from certifiable components*". SEI report n° CMU/SEI-2003-TR-009.
24. Weis T. and al.: "*Model metamorphosis*" in IEEE Software, September 2003, p. 46-51.

25. Whittaker J.A. and Thomason M.G.: "*A Markov chain model for statistical software testing*" in IEEE Transactions on Software Engineering vol.20 (10), pp.812-824, 1994.