Consignes Hermes – update 2004

Chapter x

Real Time Components & Contracts

1. Introduction

In domains such as automotive or avionics, real-time and embedded systems are getting ever more software intensive. The software cannot any longer be produced as a single chunk, and engineers are contemplating the possibility of *componentizing* it. Various definitions exist for the notion of software component. We focus here on Szyperski's one [SZY 02]: "*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. A software component only exhibits its provided or required interfaces. This defines basic *contracts* between components allowing one to properly wire them.

In real-time and embedded systems however, we have to take into account many extra-functional aspects, such as timeliness, memory consumption, power dissipation, reliability, performances, and generally speaking Quality of Service (QoS). These aspects can also be seen as contracts between the system, its environment and its users. These contracts must obviously be propagated down to the component level. One of the key desiderata in component-based development for embedded systems is thus the ability to capture both functional and extra-functional

---

Chapter written by Jean-Marc Jézéquel, Univ. Rennes 1 & INRIA, Triskell Team @ IRISA with contributions by N. Plouzeau and O. Defour. This work has been partially supported by the Artist2 Network of Excellence.

properties in component contracts, and to verify and predict corresponding system properties [REU 03].

A contract is in practice taken to be a constraint on a given aspect of the interaction between a component that supplies a service, and a component that consumes this service [MEY 92]. Component contracts differ from object contracts in the sense that to supply a service, a component often *explicitly* requires some other service, with its own contract, from another component. So the expression of a contract on a component-provided interface might depend on another contract from one of the component-required interfaces. For instance, the throughput of a component A doing some kind of computation on a data stream provided by component B clearly depends on the throughput of B.

It is then natural that people resort 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 chapter is to show how MDA can be used in relation with real-time and embedded component based software engineering. Building on Model Driven Engineering techniques, we show how the very same contracts expressed in a UML [OMG 03] model 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.

The rest of the chapter is organized as follows. Section 2 details the notion of contract along the four levels defined in [BEU 99] and illustrates it on the example of a GPS (Global Positioning System) software component. 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.

2. Contract Aware Components: The four levels of Contracts

The term contract can very generally be taken to mean "component specification" in any form. This specification should tells us what the component does without entering into the details of how. A contract is in practice taken to be a constraint on a given aspect of the interaction between a component that supplies a service, and a component that consumes this service. In real life, contracts exist at different levels, from Jean-Jacques Rousseau's social contract to negotiable contracts. A now widely accepted classification of different kinds of contracts has been proposed in [BEU 99], where a contract hierarchy is defined consisting of four levels.

- Level 1: Syntactic interface, or signature (i.e. types, fields, methods, signals, ports etc., which constitute the interface).
- Level 2: Constraints on values of parameters and of persistent state variables, expressed, e.g., by pre- and post-conditions and invariants.
- Level 3: Synchronization between different services and method calls (e.g., expressed as constraints on their temporal ordering).
- Level 4: Extra-functional properties (in particular real-time attributes, performance, QoS (i.e. constraints on response times, throughput, etc.).

Before detailing these levels, let's introduce a running example for this Chapter in the form of a simple GPS receiver. A GPS device computes its current location from satellite signals. Each signal contains data which specifies the identity of the emitting 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 Figure 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()*).

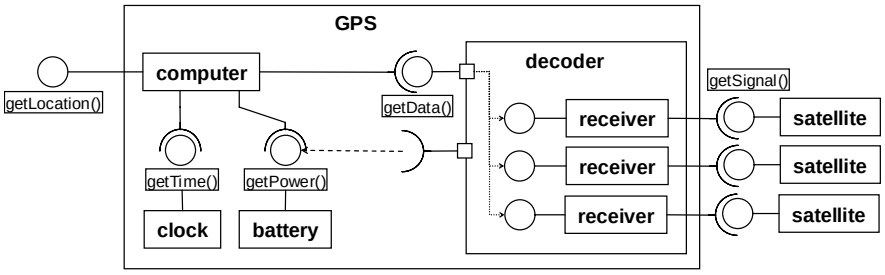

Figure 1. *The GPS component-based model*

2.1. Level 1 - Syntactic Interfaces

The syntactic interface of a component is a list of operations or ports, including their signatures (the types of allowed inputs and outputs), by means of which communication with this component is performed.

Generally speaking, a type can be understood as a set of values on which a related set of operations can be performed successfully. Once types have been defined, it is possible to use them in specifications of the form: if some input of type X is given, then the output will have type Y. Type safety is the guarantee that no run-time error will result from the application of some operation to the wrong object or value. A type system is a set of rules for checking type safety (a process usually called type checking since it is often required that enough information about the typing assumptions has been given explicitly by the designer or programmer, so that type checking becomes mostly a large bookkeeping process).

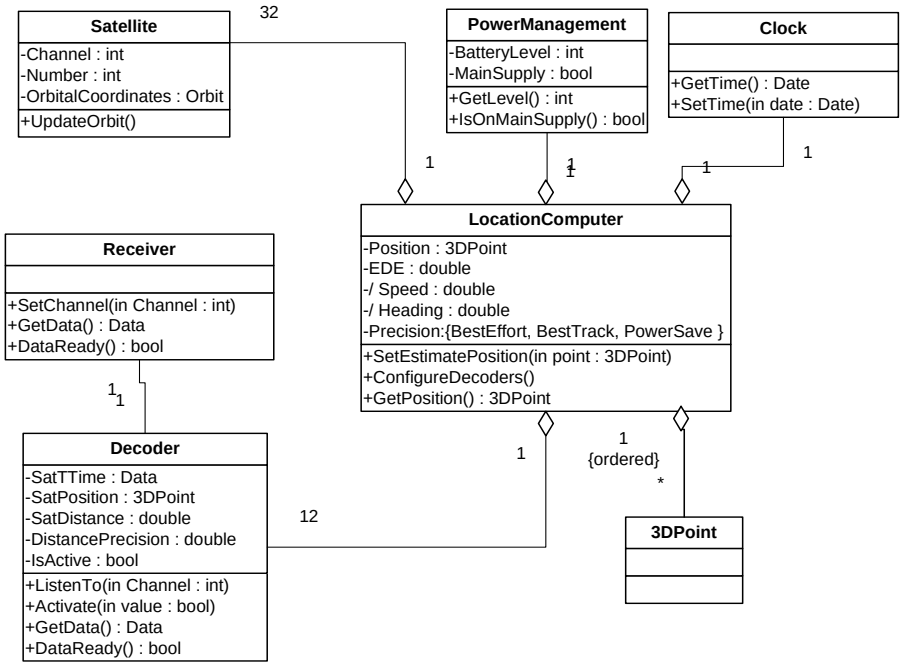

Figure 2. *Level 1 contracts in the form of a UML Static Diagram*

Static type checking is performed at compile- (or bind-) time and ensures once and for all that there is no possibility of interaction errors (of the kind addressed by the type system). Not all errors can be addressed by type systems, especially since one usually requires that type checking is easy; e.g., with static type checking it is difficult to rule out in advance all risks of division-by-zero errors.

Type systems allow checking substitutability when components are combined: by comparing the data types in a component's interface, and the data types desired by its environment client, one can predict whether an interaction error is possible (e.g. producing a run-time error such as "Method not understood").

Conformance is generally defined as the weakest (i.e., least restrictive) substitutability relation that guarantees type safety. Necessary conditions (applying recursively) are that a caller must not invoke any operation not supported by the service, and the service must not return any exception not handled by the caller. Conformance has a property called contravariance: the types of the input parameters of a service must conform in opposite to the types of its result parameters.

At first, the contravariant rule seems theoretically appealing. However, it is less natural than covariance (where parameter types conform in the same direction), often encountered in realworld modeling (animals eat food, herbivores are subtypes of animals, but they eat grass which is a subtype of food, not a supertype!), and is indeed the source of many problems. The most surprising one appears with operations combining two arguments, such as comparisons. If the contravariant rule is used, the type associated with equal for Child instances is not a subtype of the one of equal for Parent instances. As soon as this kind of feature is considered (and they are common), the contravariant rule prevents a subtyping relation between Child and Parent (see [CAS 95] for more details and solutions).

The conclusion is that as soon as one wants a minimum of flexibility for defining type conformance between a provided interface and a required interface, static type checking is no longer a simple bookkeeping process. So level 1 contracts do not have a very different nature that contracts of other levels. In some cases, they can be defined with restrictive rules to allow simple tools to process them, in other cases one could be interested in having more flexibility at the price of more complex tools for static checking, or even rely on runtime monitoring. A concern in component-based design of embedded systems is that runtime monitoring of interface types may be desirable for building reliable systems, and because one cannot completely trust component implementations. If components are deployed at run-time, the check for substitutability must be performed with available computing resources.

2.2. Level 2 - Functional Properties

Functional properties are used to achieve more than just interoperability. Level 2 contracts are about the actual values of data that are passed between components through the interfaces, whose syntax is specified at Level 1 (the preceding section). Typical properties of interest are constraints on their ranges, or on the relation between the parameters of a method call and its return value. Formalisms at level 2, for instance the OCL (Object Constraint Language dedicated to UML [WAR 98]), provide means for describing partial functions or relations by means of invariants, pre- and postconditions. It is also customary to include at level 2 properties of a persistent state of a component. In level 2 contracts, method executions are considered as atomic, which means they are appropriate for components with sequential or totally independent interactions.

There exist a number of tools using constraints for run-time monitoring which generate exceptions in case of violation of interfaces at run-time. This is the case for example in Eiffel, for JML annotations of Java and for several tools for .NET. Run-time monitoring assumes that level 2 contracts are executable, and might incurs a

nontrivial cost. Many frameworks use assertions in a test phase, often using a constraint language.

Some research tools already exist for the static checking of level 2 contracts, based for instance on the use of theorem provers. Such tools can be useful for applications requiring a high level of trust. Component producers might want to use them to provide highly dependable component implementations conforming to contracts.

The main problem of level 2 specifications is their applicability to distributed systems, due to the absence of means to express interactions as non atomic or to express explicit concurrency. This can be improved by considering additional level 3 specifications. In practice, level 2 specifications can be used mainly for a single level of components and when non-interference between transactions can be guaranteed by construction (in general by sequentializing access to components).

2.3. Level 3 - Synchronization Properties

Level 3 contracts are about the actual ordering between different interactions at the component interfaces. Level 3 specifications provide the following facilities:

- Description of transactions (input/output behaviors) not necessarily as atomic steps.
- Explicit composition operators avoid the obligation to provide an explicit input/output relation taking into account all potential internal interactions. This has the further advantage that a restricted use of a component does indeed allow to derive stronger properties (only the actually occurring interactions need to be taken into account, not all hypothetical ones).
- Many level 3 formalisms (see [MCH 94] for a survey) allow one to express explicit control information, which makes the expression of complex, history dependent input/output relations much easier;

Indeed, formalisms at level 3 need explicit composition and communication primitives. Most useful in this context are UML hierarchical state machines (StateCharts) and Sequence Diagrams. The expression of interfaces of complex components is made possible due to explicit composition. In this case, the verification of component properties and of system properties is of the same nature. However, system verification can easily become intractable for systems consisting of many complex components (cf. the state explosion problem).

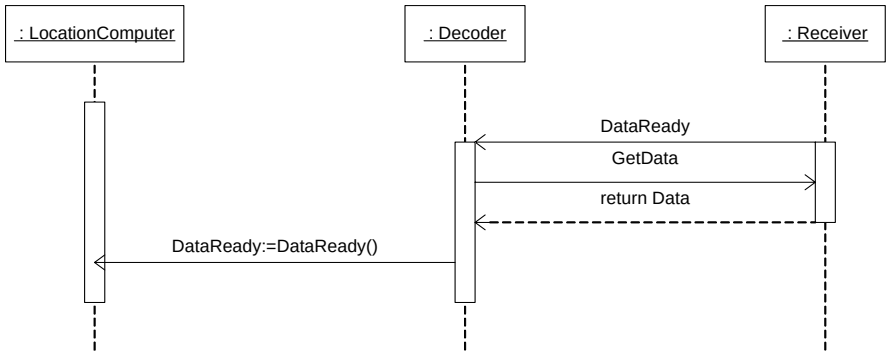

Figure 3. *Level 3 contracts in the form of a UML Sequence Diagram specifying an ordering of events*

2.4. Level 4 - Quality of Service

A quality of a system can in general be considered as a function mapping a given system instance with its full behavior onto some scale. The scale may be qualitative, in particular it may be partially or totally ordered, or the scale can be quantitative, in which case the quality is a measure. The problem of realizing systems that have certain guaranteed qualities, also known as their quality of service (QoS), involves the representation of such qualities in design models or languages and techniques to implement and analyze them as properties of implemented system instances [AAG 01].

While some definitions of QoS include concepts such as security, where the scale is not a measure, we here focus on quantitative measures, especially on those related to time. In this area, there is a common further classification of system requirements, distinguishing between hard real-time requirements, where the quality of any implemented system instance must lie in a certain interval, and soft real-time requirements.

For instance, the GPS application contains several time out constraints: 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 *getDatal()* service (resp. the satellite to complete its *getSignal()* service), then we have the two following formulae:

$$ThetaR = ThetaS + 2 , \quad (2)$$

$$ThetaD = \max (ThetaR) + 3 . \quad (3)$$

Embedded systems designers are usually facing many challenges if they strive for systems with predictable QoS. One key issue is that encapsulation of QoS properties inside a component is very difficult. In order to be useful in component based systems, contract languages must include facilities for expressing properties typical of components, that is, their context dependencies. A component provides a service under a given contract only if the surrounding environment offers services with adequate contracts. Such dependencies are much more complex than the traditional pre/postcondition contract scheme of object oriented programming. In the most general case, a component may bind together its provided contracts with its required contracts as an explicit set of equations (meaning that offered QoS is equal to required QoS). Therefore, a component oriented contract language should include constructs for:

- expression of QoS spaces (dimensions, units);
- primitives bindings between these spaces and the execution model (bindings to observable events, conversion from discrete event traces to continuous flows, definition of measures);
- constraint languages on the QoS spaces (defining the operations that can be used in the equations, form of these equations).

An example of such a language is the QoSCL [DEF 04] that extends the UML2.0 components metamodel with the following notions:

- *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.

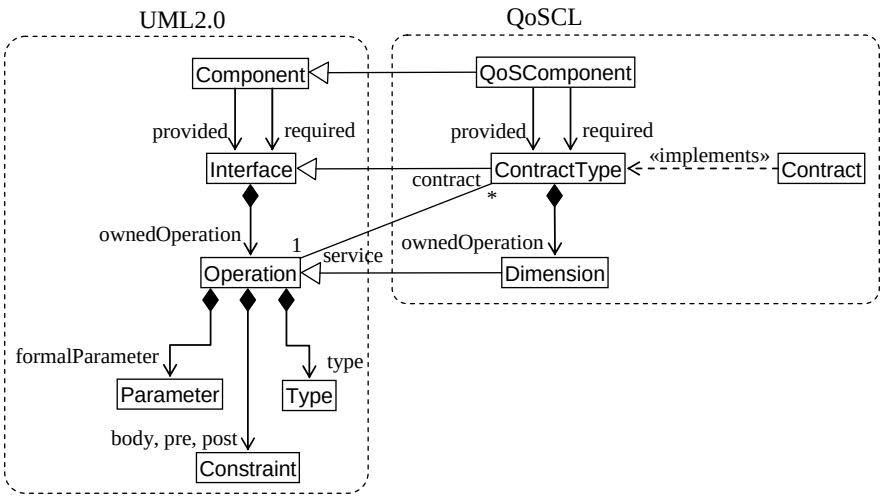

Figure 4. *The QoSCL metamodel*

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

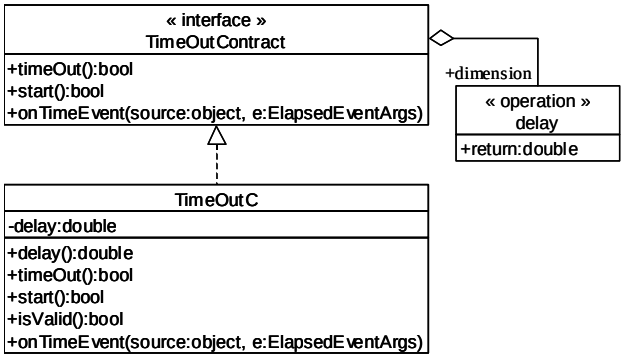

Figure 5. *The TimeOut contract with QoSCL*

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 [MEU 98].

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 [MEY 92]. Protocol validation extends this to the synchronization contract types [MCH 94]. 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 [HO 02].

3.2 Aspect Weaving

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 (a model transformer called MTL [VOJ 04]) is used to merge the monitor part of the component with its service part (see the result in Figure 6).

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.

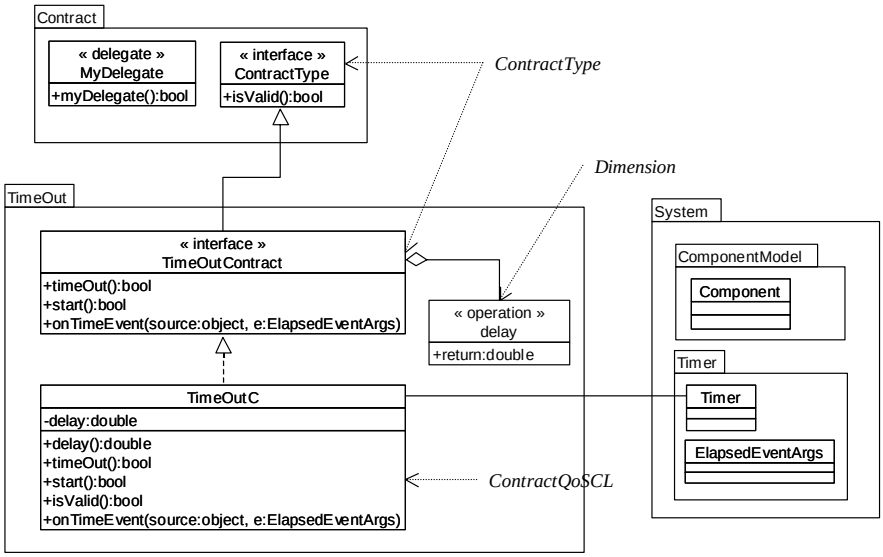

Figure 6. *TheTimeOut contract model for .Net*

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.

.

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 §2.4 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. Figure 7 below represents the computer component (Figure 1.) refined with contractual properties and their dependencies:

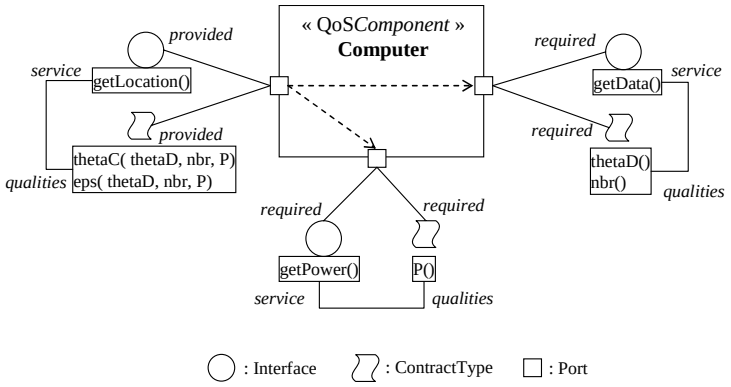

Figure 7. *Quality attributes and dependencies specification of a component*

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 [WAR 98] 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 (Figure 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.

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

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:

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

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 (***Figure 8.*** The) presents this rule:

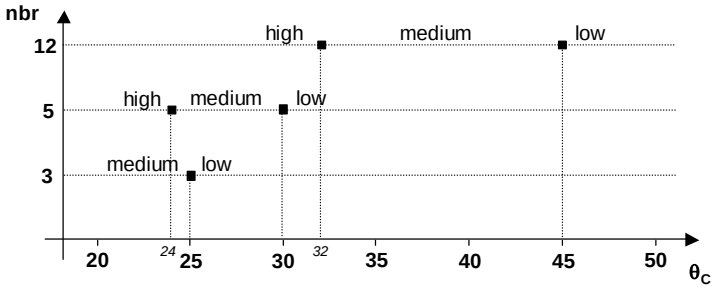

Figure 8. *The rule that binds the Eps, Nbr and ThetaC dimensions*

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 (***Figure 7.***) 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 ***Figure 8.*** The) 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:

```
satellite( [ ThetaS ] ) :-
  ThetaS $>= 15, ThetaS $=< 30.

battery( [ P ] ) :-
  P $>= 0,
  P $=< 15.

receiver( [ ThetaR, ThetaS ] ) :-
  ThetaR $>= 0, ThetaS $>= 0,
  ThetaR $= ThetaS + 2.

decoder( [ ThetaD, ThetaS, Nbr ] ) :-
  ThetaD $>= 0, ThetaS $>= 0,
  member( Nbr, [3,5,12]),
  receiver( [ ThetaR, ThetaS ] ),
  ThetaD $= ThetaR + 3.

gps( [ ThetaC, Eps, ThetaS ] ) :-
  ThetaC $>= 0, Eps = high, ThetaS $>= 0,
  computer( [ ThetaC, Eps, P, ThetaD, Nbr ] ),
  decoder( [ ThetaD, ThetaS, Nbr ] ),
  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. Conclusion

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, as exemplified with the Software Engineering Institute (SEI) promotion of its Predictable Assembly from Certifiable Components (PACC) initiative [WAL 03]: how component technology can be extended to achieve predictable assembly, enabling runtime behavior to be predicted from the properties of components. In mission-critical component based systems, it is indeed 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 chapter 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.

References

[AAG 01] AAGEDAL J.O.: "Quality of service support in development of distributed systems". Ph.D thesis report, University of Oslo, Dept. Informatics, March 2001.

[BEU 99] BEUGNARD A., JÉZÉQUEL J.M., PLOUZEAU N. AND WATKINS D.: "Making components contract aware" in Computer, pp. 38-45, IEEE Computer Society, July 1999.

[CAS 95] CASTAGNA G.: "Covariance and Contravariance: Conflict without a Cause". ACM Transactions on Programming Languages and Systems, 17(3), pp. 431-447, May 1995.

[DEF 04] DEFOUR O., JÉZÉQUEL J.M., PLOUZEAU N. "Extra-functional contract support in components". in Proc. of International Symposium on Component-based Software Engineering (CBSE7), May 2004.

[HO 02] HO WM, JÉZÉQUEL J.-M., PENNANEAC'H F., Plouzeau N. "A toolkit for weaving aspect oriented UML designs" in Proceedings of 1st ACM International Conference on Aspect Oriented Software Development, AOSD 2002, Enschede, The Netherlands, April 2002.

[MCH 94] McHALE C.: "Synchronization in concurrent object-oriented languages: expressive power, genericity and inheritance". Doctoral dissertation, Trinity College, Dept. of computer science, Dublin, 1994.

[MEU 98] MEUDEC C.: "Automatic generation of software test cases from formal specifications". PhD thesis, Queen's University of Belfast, 1998.

[MEY 92] MEYER B.: "Applying design by contract" in IEEE Computer vol. 25 (10), pp. 40-51, 1992.

[OMG 03] OBJECT MANAGEMENT GROUP: "UML Superstructure 2.0", OMG, August 2003.

[REU 03] REUSNERR R.H., SCHMIDT H.W., POERNOMO I.H.: "Reliability prediction for component-based software architecture" in the Journal of Systems and Software, vol. 66, pp. 241-252, 2003.

[SZY 02] SZYPERSKI, C.: "*Component software, beyond object-oriented programming*", 2nd ed., Addison-Wesley, 2002

[VOJ 04] VOJTISEK D., JÉZÉQUEL J.-M. "MTL and Umlaut NG - engine and framework for model transformation" ERCIM News 58, 58, July 2004.

[WAL 03] WALLNAU K.: "*Volume III: A technology for predictable assembly from certifiable component"s*. SEI report n° CMU/SEI-2003-TR-009.

[WAR 98] WARMER J., KLEPPE A. "*The Object Constraint Language: Precise Modeling with UML*", Addison-Wesley, 1998.