On the requirements for quality composability modeling and analysis

Javier F. Briones, Miguel de Miguel, Juan Pedro Silva, Alejandro Alonso
Universidad Politécnica de Madrid
Email: {jfbriones, mmiguel, aalonso, psilva}@dit.upm.es

Abstract—**Real-time, embedded and safety-critical systems have to meet some quality criteria in order to provide certain reliance on its operation. The quality of a system depends on the complex composition of the quality of its subsystems. Quality composability depends on matchmaking the provided and required quality specifications. To allow for flexibility during the system design, we study composability as a configuration problem. We allow options of quality specifications to represent design choices, deployment choices, operation modes or component adaptability. This kind of assessments of system architectures is very important e.g., for COTS development. The contributions of this paper are: to study the modeling requirements to model composability analysis, to compare two modeling approaches, and to show how a model-driven environment can leverage composability assessments. The two modeling approaches, QoS-FT + OCL and MARTE + VSL, are used to attach quality specifications to system models. However, our ultimate goal is to evaluate these specifications, and we have implemented tool-support to evaluate composability using constraint satisfaction techniques.**

I. Introduction

Real-time, embedded and safety-critical systems are often dependable systems. Dependable systems are those on which reliance can justifiably be placed on the functionality they deliver. Reliance covers several aspects such as reliability, availability, safety and security; all of them can be characterized using non-functional properties. Engineering systems to the highest reasonably practicable standards of dependability is a great challenge. Failing to do it increases the likelihood of hazards and malfunctions caused by those systems. The consequences range from merely inconveniences, e.g. poor quality in a video application, to the loss of human lives. Only enough confidence can be placed on the system being developed when functional and non-functional specifications are completely defined and a rigorous development process is put in practice.

We claim that *preliminary quality assessments* should be part of this rigorous process. The goal is to determine beforehand whether the system architecture will be able to deliver the required quality. Engineers should study non-functional properties during the design of the system to estimate its probable dependability attributes. The goal is to check whether the system will be able to meet the requirements and additionally, recommend quality measures to increase reliance on the system (e.g., supplementary requirements).

Some system engineering methodologies are based on the encapsulation and specification of system entities (e.g., modules, components, services). We focus on the *composability* of the different entities from a non-functional point of view. It is a static analysis based on the following ideas: *i*) the assessment should not be tailored to a specific non-functional aspect, *ii*) there are two types of quality specifications: offered and required constraints, *iii*) composability is based on matchmaking offered and required constraints, *iv*) there are configuration choices where a choice is defined by some quality specifications, *v*) quality constraints shall allow characterizing any quality dependency in the system.

We formulate composability as a configuration problem aiming at the selection of the system configuration which allows satisfying all the selected required constraints. Configuration choices represent design alternatives, deployment options, operation modes or entities adaptability. At the design-phase, they all can be considered the same regarding our goal: *assessment of non-functional specifications*. The benefits of tackling the problem as a configuration one are: it is common in plenty of situations e.g., COTS, algorithm selection, run-time modes; it makes the problem treatable for complex composition scenarios by enforcing a discrete number of alternatives. In the drawbacks we can find: it might not be optimal when the number of options is not finite (it could try to find a good system configuration but in case there is no solution it is impossible to test all the alternatives); and it cannot be used to calculate the value of a property (it is good to check several values for a property but not to calculate the valid/optimal value). An unpublished paper [1] deals other composability approaches.

Model-based engineering takes models and system specifications as fundamental engineering products and allows reasoning on them. Enclosing non-functional specifications with architectural models has been typically used to improve system understanding. Nevertheless, we aim at evaluating those models using a model-driven environment. This environment is selected for different reasons: consistency and traceability between models and analyses are seamlessly achieved, information of the architecture could be used in the analyses, analyses can be performed within the modeling tool, coordination among the different engineers working in the system is eased.

In this work, we study how two modeling languages cope with composability: the QoS-FT and the MARTE UML profiles. We have already evaluated the QoS-FT profile to model non-functional properties in a way that enables further composability analysis. We create quality specifications using and extended version of the profile [2] and attach those specifications to architectural models written in UML. We have implemented tool-support [3] to extract quality information from the architectural models. It finds an admissible configuration that satisfies all the constraints using a search algorithm and custom OCL constraint evaluators. We are also running some experiments using constraint programming but it is not currently integrated in our model-driven framework.

We think our work is a good contribution for number reasons. Previous work in quality of service management do consider quality levels to group quality specifications but there is no offered-required matching. In the domain of web service procurement, they do find the best provider based on matching providers' specifications with the client's required specification, however, this problem is easier. Additionally, our work is a sound formalization aiming at evaluating quality composability, what should be a good step forward to leverage generic tools (instead of current ad-hoc developments) for analyzing non-functional issues. Our work is also a good proof of concept of two modeling languages for non-functional properties, and a tool has already been implemented to evaluate quality specifications.

After reviewing some related work, section III formalizes and formulates the composability assessment based on quality constraints and levels. Section IV describes the model-driven analysis process giving a basic example. Section V compares two modeling approaches to deal with composability analysis modeling.

II. Related work

Non-functional requirements are: availability, interoperability, maintainability, performance, recovery, reliability, scalability, security, safety, or usability. They can be categorized according to a *quality model* e.g.: ISO/IEC 9126-1 [4], IEEE Std 830-1998 [5], McCall and Matsumoto quality factors [6], Volere requirements taxonomy [7], or Firesmith requirements taxonomy [8].

Wherever possible, non-functional requirements have to be characterized quantitatively. Some of the benefits of quantifying requirements are: *i*) to challenge, and thus improve, requirements, *ii*) to formalize requirements in better ways e.g., using mathematical expressions, and *iii*) to enable validation and analysis of requirements. To specify requirements quantitatively: first we have to identify attributes of the requirements that are measurable, and then, to define a metric to express the attributes. Although quantifying requirements can be difficult, several techniques can help. In [9] two of the best techniques for quantification are discussed: Volere's method [7] and Planguage [10], although it is recognized that none of them solves all problems of requirements quantification.

Most of a system's non-functional requirements are unverifiable at subsystem-level and they can even be difficult to test at the system-level. For example, the end-to-end latency can only be estimated taking into consideration all the subsystems; the same stands for the reliability of a system but its computation may be more complex. However there is work in progress trying to predict non-functional properties of an assembly, such as a component-based system, from the known properties of their parts e.g., SEI's work in Predictability by Construction and its initiatives Prediction-Enabled Component Technology (PECT) and Predictable Assembly from Certifiable Components (PACC). Crnkovic et al. [11] classify non-functional properties according to how they can be derived from the properties of the components involved. They can be: directly composable, architecture-related, derived, usage-dependent or system-environment context. However, the authors acknowledge that it might be difficult to model these dependencies.

As soon as we *i*) create a quality model, *ii*) quantify non-functional specifications as much as possible, *iii*) and characterize dependencies among the different non-functional properties, we can carry out the composability study as formulated in this paper. There still exists the key issue of offered and required constraints matchmaking. We use a symmetric model (as recommended in [12]) to model quality specifications where neither offered nor required constraints are restricted to single values. We have found in the literature two approaches, called *metrics*, to perform the matchmaking namely, matching [13] and conformance [12]). *Matching* signifies that there has to be at least one value of the offer that satisfies the requirements; *conformance* means that requirements are satisfied for every value of the offers. In [14] the concept of conformance is extended for situations where non-functional required constraints are not carefully defined. For instance, a required availability such as $0.95 < A < 0.98$ is poorly defined since an offered availability $A \geq 0.999$ does always satisfy the requirements but it is not included in the range. So far, we focus on the conformance metric deliberately, as it seems the only acceptable to deal with dependability. In other words, critical systems where dependability is the main attribute, require properly defined non-functional specifications.

There exists a similar work in the domain of web service procurement by Ruiz Cortes et al. and extended by Kritikos et al. Their goal is to find the best provider based on matching providers' offered specifications with the client's demanded specifications (what we call required). This problem is obviously far more simple as there is just one configuration point. Their papers also lack the thorough formalization done in this paper. In our opinion, it is a very practical work that uses constraint satisfaction [12], [14] and mixed-integer programming [15] to create a solver.

III. Composability assessment

A. Composability based on quality constraints and levels

Let's consider the following system entities: *a*) a service to perform image processing outputs data of different qualities according to the available processing and memory resources, *b*) the component of a control system has two operation modes, each one having a different failure rate, and *c*) a software driver for a motion-detection sensor whose accuracy depends on the hardware used existing two options available. Image quality, resource availability, operation mode, failure rate, hardware characteristics and detection accuracy are *non-functional properties* that should be part of the quality specification.

To study non-functional composability we need offered (provided) and required quality specifications. An *offered constraint* characterizes (at least) a property whose value might fluctuate during operation provided that the offered constraint is met. That is, only the satisfaction of the offered constraint is ensured but not a steady value for the property (-ies) involved. On the other hand, a *required constraint* is a condition over the values taken by the properties. In the composition, we need to ensure that for every value that meets the offered constraint all the required constraints are satisfied. One of the key issues of evaluating quality specifications is the matchmaking between offered and required constraints. For instance, the first mode of entity *b)* has a value for its failure rate in the range $(0, 0.01)$, and the second mode in the range $(0, 0.05)$. There is a required condition to be held: failure rate < 0.03. This required condition is satisfied for any value of the first range, but this is not true for the second range. If the entity is configured using the second mode, the value of the property could be lower than 0.03, but it might be greater and this is not admissible.

Each of the three entities before has a configuration choice. The choice implies *a)* an adaptable entity, *b)* a deployment choice, and *c)* a design choice. Each option of the choice is called *quality level* and it is represented with the set of quality constraints. The constraints in the set cannot be set up independently. For instance, to set up an offered constraint there is a need to set up first a specific required constraint (e.g., for entity *a)* to provide high accurate data of there is an increase in demand of available resources). A system *configuration* is a selection of one quality level for each choice. We will also consider *configuration constraints* which represent high-level limitations to system configurations.

B. Concepts formalization

Let x_i be the variable representing the *non-functional property* i and $\mathrm{Dom}(x_i)$ the domain of the variable. Let p_i be a specific value taken for the variable x_i. Let's also define the tuples **x**, to represent the collection of non-functional properties in the system, and **p**, to represent an assignment σ of values to all the properties being considered in the problem (e.g., an observation of the system).

$$\begin{aligned}\mathbf{x} &= (x_1, \dots, x_I)\\ \mathbf{p} &= (p_1, \dots, p_I) = \sigma(\mathbf{x})\\ p_i &= \sigma(x_i) \in \mathrm{Dom}(x_i)\end{aligned}$$

Let y_j be the variable representing the *configuration choice* j and $\mathrm{Dom}(y_j)$ the integer domain of the variable. Let l_j be a specific option (level, mode,...) taken for the choice y_j. The tuples **y** and **l** represent the collection of configuration choices and a set-up of selected options respectively.

$$\begin{aligned}\mathbf{y} &= (y_1, \dots, y_J)\\ \mathbf{l} &= (l_1, \dots, l_J) = \sigma(\mathbf{y})\\ l_j &= \sigma(y_j) \in \mathrm{Dom}(y_j)\end{aligned}$$

Offered, required and configuration constraints represent dependencies among different variables i.e., *relations*. They can be modeled with *boolean mathematical statements*: $O(\mathbf{x})$, $R(\mathbf{x})$, and $F(\mathbf{y})$. Notice that the mathematical relations are: $O = \{\mathbf{x}|O(\mathbf{x})\}$, $R = \{\mathbf{x}|R(\mathbf{x})\}$ and $F = \{\mathbf{y}|F(\mathbf{y})\}$. An assignment of values **p** satisfies a (required) constraint when the mathematical statement ($R(\mathbf{x})$) is true so **p** belongs to the relation (R).

Let's consider the following three constraints: 'a response to some warning event shall have a soft deadline of 50 ms (for an optimally efficient reaction) and a hard deadline of 200 ms (to guarantee that no damage takes place)', 'a warning signal will be generated when temperature is over $100°C$; it shall be audible (at least 1 Pa.) and have a minimum inter-arrival time of 300 secs.', and 'using the component A of vendor *brand* implies deploying the component B of the same vendor'. Each constraint is of one type and they could be expressed with mathematical statements:

$$\begin{aligned}R(\mathbf{x}) = R(D_s, D_h) &\equiv (D_s <= 50 \wedge D_h <= 200)\\ O(\mathbf{x}) = O(p, t_{ia}) &\equiv (p >= 1 \wedge t_{ia} >= 300)\\ F(\mathbf{y}) = F(v_A, v_B) &\equiv (v_A = \text{brand} \Rightarrow v_B = \text{brand})\end{aligned}$$

The reader should note that, in the analysis, a set of constraints have to be satisfied together and some of the constraints could involve the same non-functional properties. For example, there the following conflicting constraints (where p is the price and bw the bandwidth) cannot be satisfied together $\neg\exists(bw, p)\ (R_1(bw) \wedge R_2(bw, p) = true)$:

$$\begin{aligned}R_1(\mathbf{x}) = R(bw) &\equiv (bw >= 2{,}048)\\ R_2(\mathbf{x}) = R(p, bw) &\equiv (p <= 1{,}000 \wedge p = 0.5 \times bw)\end{aligned}$$

A *quality level* can be represented using a collection of constraints. We could represent it as a triple L_j of sets: the offered constraints in the level, the required constraints, and the configuration constraints. Any element of the triple could be empty.

$$\begin{aligned}L_j &= (\{O_{j1}(\mathbf{x}), \dots\}, \{R_{j1}(\mathbf{x}), \dots\}, \{F_{j1}(\mathbf{y}), \dots\})\\ &= (L_j^O, L_j^R, L_j^F)\end{aligned}$$

The previous definition of quality level is very useful for analysis implementations (constraint evaluation) because it keeps constraints separated. However, aiming at formalizing the problem, it is more convenient to use the definition l_j based on logical conjunctions of statements.

$$\begin{aligned}l_j &= (O_{j1}(\mathbf{x}) \wedge \dots, R_{j1}(\mathbf{x}) \wedge \dots, F_{j1}(\mathbf{y}) \wedge \dots)\\ &= (\mathcal{O}_{l_j}(\mathbf{x}), \mathcal{R}_{l_j}(\mathbf{x}), \mathcal{F}_{l_j}(\mathbf{y}))\\ \mathbf{l} &= (\wedge_j \mathcal{O}_{l_j}(\mathbf{x}), \wedge_j \mathcal{R}_{l_j}(\mathbf{x}), \wedge_j \mathcal{F}_{l_j}(\mathbf{y}))\\ &= (\mathcal{O}_{\mathbf{l}}(\mathbf{x}), \mathcal{R}_{\mathbf{l}}(\mathbf{x}), \mathcal{F}_{\mathbf{l}}(\mathbf{y}))\end{aligned}$$

C. Problem formulation

Only when system engineers have defined non-functional properties, quality constraints and quality levels, they can carry out a composability study to check whether the different entities can be assembled from a quality point of view. The problem formulation is three-fold: *i*) is it possible to meet existing requirements given existing entities and levels?, *ii*) which configuration allows satisfying the requirements?, *iii*) which of these configurations is the best one according to certain criteria? There are several concepts related to the problem raised:

Configuration. A configuration is a set-up of levels. One level has to be selected for each configuration choice.

Admissible configuration. It is a configuration that ensures the composability condition given by Equation 2. The collection of every admissible configuration $\mathbf{l}$ is represented with S in Equation 1.

Optimal admissible configuration. Admissible configurations can be compared if there exists a function $f(\mathbf{l})|_{\mathbf{l}\in S}$ to compare them. If we suppose that the higher the function value the better the system configuration, it has to fulfill the following equation:

$$\mathbf{l}^* \in S \mid \forall \mathbf{l} \in S : f(\mathbf{l}^*) \geq f(\mathbf{l})$$

Valid observation. It stands for any assignment of values to the non-functional properties ($\mathbf{p}$) reachable within an admissible configuration.

Feasible region. It is the space created joining every valid observation.

An admissible configuration is the one that, belonging to the relation $\mathcal{F}_\mathbf{l}$, defines an offered space that is included in the required space it also defines.

$$S = \{\mathbf{l} \in \mathcal{F}_\mathbf{l} \mid \mathcal{O}_\mathbf{l} \subseteq \mathcal{R}_\mathbf{l}\} \tag{1}$$

Expanding the expression and defining the formalisms $\mathcal{O}(\mathbf{y}, \mathbf{x})$, $\mathcal{R}(\mathbf{y}, \mathbf{x})$ and $\mathcal{F}(\mathbf{y})$ to allow using the existential quantifier on $\mathbf{y}$, we get the admissibility condition as a *predicate logic boolean formula*.

$$\begin{gathered} S = \{\mathbf{l} \mid \mathcal{F}_\mathbf{l}(\mathbf{l}) \wedge \forall \mathbf{p}(\mathcal{O}_\mathbf{l}(\mathbf{p}) \Rightarrow \mathcal{R}_\mathbf{l}(\mathbf{p}))\} \\ \mathcal{O}(\mathbf{l}, \mathbf{p}) \Leftrightarrow \mathcal{O}_\mathbf{l}(\mathbf{p}) \\ \mathcal{R}(\mathbf{l}, \mathbf{p}) \Leftrightarrow \mathcal{R}_\mathbf{l}(\mathbf{p}) \\ \mathcal{F}(\mathbf{l}) \Leftrightarrow \mathcal{F}_\mathbf{l}(\mathbf{l}) \end{gathered}$$

$$\exists \mathbf{l} \forall \mathbf{p} : \mathcal{F}(\mathbf{l}) \wedge (\mathcal{O}(\mathbf{l}, \mathbf{p}) \Rightarrow \mathcal{R}(\mathbf{l}, \mathbf{p})) \tag{2}$$

D. Solver methods

After modeling the composability problem raised in this paper, it is possible to study different solver alternatives. The problem faced is not a *first-order logic*: there are variables that range over individuals (x_i) but also variables that range over sets of individuals ($y_j = l_j$ defines a set of individuals by means of $\mathcal{O}_{l_j}(\mathbf{x}), \mathcal{R}_{l_j}(\mathbf{x}), \mathcal{F}_{l_j}(\mathbf{y})$). We contemplate three key issues to decide the solver to use.

1) Cardinality of the domains of y_j. If they are all finite, a solver could use a tree-search algorithm to traverse the different configurations.
2) What the relation $\mathcal{O}_\mathbf{l}$ says about $\mathbf{x}$. It might define ranges $p_{i,min} \leq x_i \leq p_{i,max}$ or even isolated values $x_i = p_i$. This issue is very important since the required constraints must be satisfied for every value in the offer.
3) The type and complexity of the statement $R_{j*}(\mathbf{x})$. Linear functions might allow using linear programming techniques. Monotone functions should ease the evaluation of requirements for every value in the offer.

IV. Model-driven evaluation

A. Analysis process

Most model-driven dependability analysis follow the same steps: *i*) engineers annotate model elements of the system architecture with dependability attributes, *ii*) a model transformation extracts the required information from the architectural models building an analysis model, *iii*) the analysis is performed giving a pass/fail result or providing extra annotations onto the architecture. The analysis can be performed within the modeling tool or using an external dependability tool. This decision usually depends on how difficult the analysis execution is. However, it should be possible to launch the process from the modeling tool. Model-driven engineering techniques act as a tool-bridge. Tool integration is easier when the semantics used for the annotations correspond to the semantics of the analysis. Model transformations help to couple both semantics.

B. On a modeling tool

We [2], [3] use UML as the modeling language to specify the architecture of the system. One method [16] to extend UML is by means of a UML profile, using stereotypes and tagged-values to annotate model elements. We also need a language to define quality constraints. We use Rational Software Architect to create the architectural models. We export them into the Eclipse Modeling Framework. Atlas Transformation Language is used to build the analysis models from the information of the architectural models. The code to launch model transformations and execute the analysis is implemented using the Java programming language.

We have considered two approaches to model quality composability: QoS-FT + OCL and MARTE + VSL. The first approach uses the *UML Profile for Modeling QoS and Fault Tolerance Characteristics and Mechanisms* and the *Object Constraint Language*. In previous work [2], we have worked with these two languages to model quality adaptability. Nevertheless, only some of the concepts described in that work are needed to model a composability analysis. The second approach has not been implemented yet. It will use the *UML Profile for Modeling and Analysis of Real Time and Embedded Systems* and the *Value Specification Language*. MARTE considers three annotation mechanisms: (instance specification) slots, tagged-values, and constraints. While there exist many similitudes between the two modeling approaches, the MARTE mechanism that uses tagged-values is not provided by the QoS-FT profile. We will compare the two approaches in the section V.

C. On the analysis tool

Although this is not the purpose of this paper, we want to give a hint of the solvers used and briefly motivate the election according to the issues of section III-D. So far, we have only considered finite domains for y_j.

The first approach was to implement a solver based on a depth-first search algorithm. It was implemented in Java and it is integrated in the modeling environment. It traverses the search space, evaluating (using extensions of the Eclipse OCL plugin) at each node those required constraints that can be evaluated because all the variables in the constraint are defined. The solver requisites are: $O_{j*}(\mathbf{x})$ defining ranges of values and $R_{j*}(\mathbf{x})$ being monotone functions. It was complete and optimal for the problem formulated, it was not very efficient though.

Currently, we are using a solver based on constraint satisfaction (it uses the constraint programming language Comet). This solver implements constraint solving and constraint propagation techniques, managing constraints much better. The solver requisites are: $O_{j*}(\mathbf{x})$ do not have to be ranges but a set-up of $\mathcal{O}_{\mathbf{l}}(\mathbf{x})$ should ultimately define ranges of values, and $R_{j*}(\mathbf{x})$ are monotone functions.

We studied other alternatives e.g., linear programming and mixed integer programming. Its applicability highly depend on the type of constraints. However, constraint satisfaction allows pluging-in ad-hoc constraint evaluators and propagators. We do realize that other alternatives might suit much better under certain conditions.

D. Example

The best way to show what the analysis should do is using a basic example. It has three configuration choices: transmission driver, network reservation and network access technology. Each choice has different options (quality levels). Each option implies different types of constraints: offered, required and configuration. The space of the offers has to be included in the space of the requirements and the configuration has to meet configuration constraints.

```
Choice tx = {lo,hi}
Opt lo     = ({},{0<=rate<=1024},{},{})
Opt hi     = ({},{0<=rate<=4096, priority==guaranteed},{},{})

Choice rsvp = {bronze, silver, gold}
Opt bronze  = ({0<=rate<=2048, priority==best-effort},{},{})
Opt silver  = ({0<=rate<=2048, priority==guaranteed},{},{})
Opt gold    = ({0<=rate<=10240, priority==guaranteed},{},{})

Choice acc = {isdn, t1}
Opt isdn    = ({},{},{acc==isdn => rsvp==bronze})
Opt t1      = ({},{},{})
```

There are 12 ($2 \times 3 \times 2$) possible configurations, and it is easy to check that only 5 are 5 admissible:

```
Conf c1  = (lo, bronze, isdn)
Conf c2  = (lo, bronze, t1)
Conf c3  = (lo, silver, t1)
Conf c4  = (lo, gold, t1)
Conf c5  = (hi, gold, t1)
```

We are working with examples containing thousands of possible configurations and the solver successfully computes admissible configuration in a matter of seconds.

V. Composability analysis modeling

A. Modeling requirements

We have identified the modeling-support required to model a composability analysis:

1. *Non-functional properties types* to define the quality model. Engineers could use a standard quality model or create an ad-hoc model which could be reused in different projects.
2. *Non-functional properties* used in constraints as variables.
3. *Constraints on non-functional properties* might be implicit, or explicit when they use a specification language. Different constraints should be able to involve the same non-functional properties. There shall also be a mechanism to specify whether the constraint is offered or required.
4. *Quality level* which groups the constraints that must be satisfied together. A set-up of levels is the output of the analysis.
5. *Configuration choice.* They are variables of the configuration constraints.
6. *Configuration.* The system configuration may be defined by a set of active quality
7. *Configuration constraints.* If they are explicit, they can use the same mechanisms used for constraints on non-functional properties provided that all the variables in the constraints have to be configuration choices. levels.

Other modeling elements could be desired but we want to focus on the essential mechanisms required to analyze composability. For instance, to model the adaptability of a system architecture based on a given component model, we need to link a configuration choice with the adaptable component. Another desired feature would be to provide modeling-support for the attachment of non-functional properties to system elements.

B. Implicit or explicit constraints

Constraints are conditions for declaring some semantics between more than one model element (for instance, a delay between two different events). One of the key decisions to create the modeling-support for composability analyses is whether to use explicit or implicit constraints. An *explicit constraint* is an expression which uses a specification language and includes all the information required to evaluate the constraint. On the other hand, the information required to evaluate an *implicit constraint* is spread in several parts of the model. An example of explicit constraint is the attachment of a non-functional value with the purpose of fixing the value of the non-functional property. Another implicit constraint exists when some tasks (each one defining their periods and deadlines) are allocated in the same execution host, the host has an scheduler based on fixed priorities, and it is clear but not explicitly expressed that the tasks have to be schedulable. The following expressions are examples of explicit constraints. Mean and Jitter are constants defined somewhere in the model but unequivocally addressed; *FailureRate*, *VideoRate*, *D* and *C* are variables; rateMonotonic() and $\leq$ are functions.

$$FailureRate \leq 0.001$$
$$\text{Mean} - \text{Jitter} \leq VideoRate \leq \text{Mean} + \text{Jitter}$$
$$\text{rateMonotonic}((20\text{ms}, 5\text{ms}), (40\text{ms}, 10\text{ms}), (D, C))$$

To provide an open modeling framework i.e., not tailored to a particular dependability concern, explicit constraints shall be allowed. The main benefit of implicit constraints is usability. The two methods for constraints specification are not exclusive although the composability analysis tool takes explicit constraints as input so, a semantics-aware model transformation has to create explicit constraints from the implicit constraints' parameters.

C. Profiles comparison

This section compares the two approaches we have considered, QoS-FT + OCL and MARTE + VSL, focusing on satisfying the modeling requirements outlined in section V-A.

1. A *type of non-functional property* (transmission rate). QoS-FT uses a UML Classifier (communication-throughput) stereotyped with QoSCharacteristic; a characteristic might have several UML Propertie(s) (rate) stereotyped with QoSDimension. MARTE uses a TupleType/DataType (NFP_DataTxRate) stereotyped with nfpType. They are rather different approaches with two important differences: using Classifier versus using DataType, and the location of the attributes of the non-functional property. Note that property attributes (unit, direction) are defined in QoS-FT in the type of the non-functional property.

```
<<QoSCharacteristic>> Classifier communication-throughput
  <<QoSDimension>> Property rate
    : real {unit(bit/sec), direction(increasing)}

<<nfpType>> TupleType NFP_DataTxRate
  Property expr : VSL_Expression
  Property source : SourceKind
  Property statQ : StatisticalQualifierKind
  Property dir : DirectionKind
  Property mode : String[*]
  Property value : Real
  Property unit : DataTxRateUnitKind
  Property precisison : Real
```

The QoS-FT standard includes a QoSCatalog that, even though it is not part of the standard, could be reused in users' applications. On the other hand, MARTE does include non-functional properties types that can be reused.

2. A *non-functional property* (rate) of the previous type. QoS-FT uses an InstanceSpecification (throughput) of a QoSCharacteristic (communication-throughput) stereotyped as QoSValue. The QoSValue has to be attached to the modeling element it constrains. Only one property attribute (rate) is given because the rest are defined with the type. MARTE uses a Slot (rate) of a Property that should be previously created in the user model. In the definition of the property (NFP_DataTxRate) the default values for some tuple members can be given (max, decr).

```
<<QoSValue>> InstanceSpecification throughput
    : communication-throughput
  Slot rate: 2048

Classifier Network
  <<nfp>> Property rate
    : NFP_DataTxRate = (statQ=max,dir=decr)
InstanceSpecification nw : Network
  Slot rate = (value=2048,unit=b/s,source=req)
```

3. A *constraint on the non-functional property* (called two) previously defined. QoS-FT provides the stereotypes QoSOffered and QoSRequired to model quality constraints while MARTE provides the stereotype nfpConstraint using a tagged-value to identify whether it is a required or an offered constraint. We use OCL for QoS-FT (a UML Opaque-Expression) and VSL for MARTE (a VSL Expression) as specification language. The context of the OCL constraint is a QoSCharacteristic (communication-throughput), and the namespace of the VSL specification is a modeling element (Network).

```
<<QoSOffered>> Constraint two
  {context communication-throughput inv: rate <= 2048}

<<nfpConstraint>> Constraint two {kind=offered}
  {rate <= (2048,b/s)}
```

4. A *quality level* (bronze) including the previous constraint. In QoS-FT, it is modeled with a State stereotyped with QoSLevel. A state owns a constraint called stateInvariant that has to be true to be in the state. If we have to include several constraints we can use the stereotype QoSCompound-Constraint to group them. In MARTE, a constraint can have associated (using a tagged-value) one or more State(s) stereotyped with Mode. In those modes the constraint should be taken into account.

```
<<QoSLevel>> State bronze
  Constraint stateInvariant : [two,best-effort]

<<Mode>> State bronze
<<nfpConstraint>> Constraint two
  {kind=offered, mode=[bronze,silver]}
  {rate >= (2048,b/s)}
```

5. A *configuration choice* (rsvp) including the previous level. MARTE has a mechanism, called mode behavior, to specify a set of mutually exclusive modes. To create a configuration choice we need to create a StateMachine stereotyped with ModeBehavior. QoS-FT does not provide any mechanism to specify a configuration choice, but in our work [2] we augmented the profile with a stereotype called QoSExternal-Behavior which extends StateMachine as well.

```
<<QoSExternarBehavior>> StateMachine rsvp
  <<QoSLevel>> State submachineStates : [bronze,silver,gold]

<<ModeBehavior>> StateMachine rsvp
  <<Mode>> State submachineStates : [bronze,silver,gold]
```

6. A *configuration* (c2) can be defined by a set of selected quality levels. MARTE has the stereotype Configuration to model it. It extends UML Package or UML StructuredClassifier and has a collection of modes annotated as tagged-values.

For QoS-FT, we do not recommend the use of QoSCompoundLevel; it is a concept of the standard document but it does not have a stereotype associated and it does not have the desired semantics when used in combination with QoSCompoundConstraint. We do not have either a similar concept in our previous work.

```
<<Configuration>> Package c2 {mode=[lo,bronze,t1]}
```

The semantics of active state in UML is rather complicated and we would like to define constraints which use states and state machines to indicate when a quality level is selected. We propose: to model configurations as QoSCharacteristic(s) in QoS-FT, and to add propertie(s) to the configuration in MARTE. The main reason is that configuration constraints will be handled as constraints on non-functional properties. This proposal does not modify any concept of the profiles.

```
<<QoSCharacteristic>> Classifier conf
  <<QoSDimension>> Property tx : State
  <<QoSDimension>> Property rsvp : State
  <<QoSDimension>> Property acc : State
<<QoSValue>> InstanceSpecification c2 : conf
  Slot tx: lo
  Slot rsvp: bronze
  Slot acc: t1

<<Configuration>> Class c2 {mode=[tx,rsvp,acc]}
  Property tx : Mode = lo
  Property rsvp : Mode = bronze
  Property acc : Mode = t1
```

7. *Configuration constraint*. The definition of configuration constraints is straightforward if we use previous specifications. The context of the ocl expression is the QoSCharacteristic conf, and the namespace of the value specification is configuration itself.

```
<<QoSLevel>> State isdn
  Constraint stateInvariant : [cstr]
<<QoSRequired>> Constraint cstr
  {context conf
    inv: acc=isdn => rsvp=bronze}

<<Mode>> State isdn
<<nfpConstraint>> Constraint cstr
  {kind=required, mode=[isdn]}
  {acc==isdn? rsvp==bronze : true}
```

VI. Conclusion

We have formulated the composability problem as a configuration problem based on matchmaking offered and required quality constraints. As we deal with non-functional properties and constraints in a homogeneous and generic way, we do consider the formulation could leverage the creation of generic tools to analyze non-functional properties substituting adhoc developments. However, we recognize the existence of composability problems our formulation cannot solve. For instance, to compute resource availability or resource usage e.g., if two video streaming algorithms require a minimum bandwidth of 128 kbps it is not possible to calculate that the offered constraint of the network has to be: $bw >= 256$ kbps.

This paper also enumerates the requirements for modeling composability analyses and proposes and studies two approaches. Only the QoS-FT approach could not meet all the requirements and needed a small extension. Both approaches are approximately similar and the decision of which one to use should be based on two points: usability and the expressiveness of the specification language of the constraints. In our opinion, the MARTE approach is more usable and the QoS-FT allows more complex constraint specifications.

References

[1] J. F. Briones, M. A. de Miguel, J. P. Silva, and A. Alonso, “International conference on software composition,” 2010, submitted to.

[2] J. F. Briones, M. A. de Miguel, A. Alonso, and J. P. Silva, “Modeling quality of service adaptability,” in *Enterprise Distributed Object Computing Workshops (Advances in Quality of Service Management), International Conference on*, vol. 0. Los Alamitos, CA, USA: IEEE Computer Society, 2008, pp. 50–27.

[3] ——, “Quality of service composition and adaptability of software architectures,” in *12th IEEE International Symposium on Object component service-oriented Real-time distributed Computing*, 2009.

[4] ISO/IEC 9126-1, “Software engineering - product quality - part 1: Quality model,” ISO/IEC, Tech. Rep. ISO/IEC 9126-1:2001, June 2001.

[5] IEEE Std 830-1998, “IEEE recommended practice for software requirements specfcations,” IEEE, Tech. Rep. IEEE Std 830-1998, June 1998.

[6] J. A. McCall and M. T. Matsumoto, “Software quality measurement manual,” General Electric Company and Rome Air Development Center, Tech. Rep. RADC-TR-80-109 Part II, April 1980.

[7] S. Robertson and J. Robertson, *Mastering the Requirements Process*. Addison-Wesley Professional, Aug. 1999.

[8] D. G. Firesmith, “Common concepts underlying safety, security, and survivability engineering,” Carnegie Mellon Software Engineering Institute, Technical Note CMU/SEI-2003-TN-033, December 2003.

[9] N. Maiden, “Improve your requirements: Quantify them,” *IEEE Software*, vol. 23, no. 6, pp. 68–69, 2006.

[10] T. Gilb, *Competitive Engineering: A Handbook For Systems Engineering, Requirements Engineering, and Software Engineering Using Planguage*. Butterworth-Heinemann, Aug. 2005.

[11] I. Crnkovic, M. Larsson, and O. Preiss, “Concerning predictability in dependable component-based systems: Classification of quality attributes,” in *Architecting Dependable Systems III*, vol. 3549. Springer Berlin / Heidelberg, 2005, pp. 257–278. [Online]. Available: http://www.springerlink.com/content/dpc7hnegdgdfnn5a

[12] A. R. Cortés, O. Martín-Díaz, A. D. Toro, and M. Toro, “Improving the automatic procurement of web services using constraint programming,” *Int. J. Cooperative Inf. Syst.*, vol. 14, no. 4, pp. 439–468, 2005.

[13] S. Degwekar, S. Y. W. Su, and H. Lam, “Constraint specification and processing in web services publication and discovery,” in *ICWS ’04: Proceedings of the IEEE International Conference on Web Services*. Washington, DC, USA: IEEE Computer Society, 2004, p. 210.

[14] K. Kritikos and D. Plexousakis, “Evaluation of QoS-based web service matchmaking algorithms,” in *IEEE Congress on Services - Services 2008*, vol. Part I. Honolulu, Hawaii, USA: IEEE Computer Society, 6-11 July 2008, pp. 567–574.

[15] ——, “Mixed-integer programming for QoS-based web service matchmaking,” *IEEE Trans. Serv. Comput.*, vol. 2, no. 2, pp. 122–139, 2009.

[16] M. de Miguel, J. Briones, J. Silva, and A. Alonso, “Integration of safety analysis in model-driven software development,” *IET Software*, vol. 2, no. 3, pp. 260–280, 2008. [Online]. Available: http://link.aip.org/link/?SEN/2/260/1