A Model-Driven Approach to Enable Adaptive QoS in DDS-Based Middleware

J. F. Inglés-Romero[a], A. Romero-Garcés[b], C. Vicente-Chicote[c], J. Martínez[b]

[a]Universidad Politécnica de Cartagena, ETSI Telecomunicación, Spain
[b]University of Málaga, ETSI Telecomunicación, Spain
[c]University of Extremadura, QSEG, Escuela Politécnica de Cáceres, Spain

***Abstract*—Critical and distributed systems need to be reliable and comply with the required performance at run-time. In this vein, Data Distribution Service for Real-Time Systems (DDS) provides developers with highly configurable middleware to control the end-to-end Quality of Service (QoS) of the applications through a wide range of attributes. However, dynamic and unpredictable environments pose a major challenge to these systems as their workload and resources may fluctuate significantly in time depending on the execution context. Developers usually find it difficult to choose and apply the right DDS QoS attributes, as once selected, they remain fixed during the whole execution of the system. They do not automatically change according to the execution context, e.g., to meet non-functional requirements related to performance or resource consumption. Moreover, changing the QoS attributes at run-time may lead to incompatibilities, since the configuration used by the different participants needs to be mutually consistent. In this paper, we propose a model-driven approach that enables the safe, automatic and transparent adaptation of the QoS attributes in DDS-based middleware, providing the best performance possible within the available resources at run-time. An example in robotics is presented to demonstrate the feasibility and the benefits of our proposal.**

***Keywords*—DDS, QoS, Model-Driven Engineering, Software Adaptation**

I. Introduction

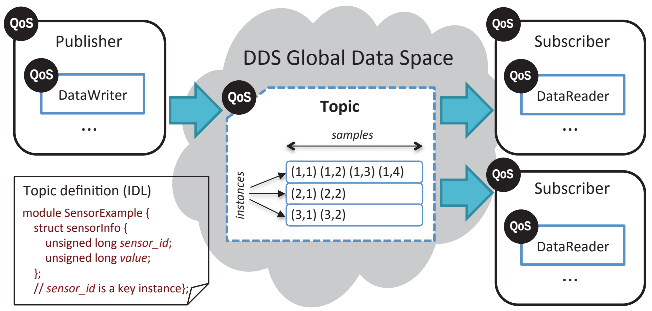
Software adaptation is becoming increasingly important as more and more applications need to cope with limited resources and changing conditions dynamically. Adaptation becomes particularly significant at the middleware level as, typically, the available resources (e.g., CPU, memory, network capacity, etc.) fluctuate over the time depending on how demanding the entities involved in the communication are. Thus, middleware-based communication systems not provided with appropriate adaptation mechanisms may eventually run out of resources, experiencing degradation or failing to comply with the expected performance. This is a major issue in critical distributed systems for which reliability is a must.

Data Distribution Service for Real-Time Systems (DDS) [1] has emerged as the first open international middleware standard directly addressing publish/subscribe communications for real-time and embedded systems. The DDS standard includes more than twenty Quality-of-Service (QoS) policies (each one with several attributes) to specify resource limitations for data queues, liveliness or reliability, among other features. This provides developers with a great flexibility to control the end-to-end QoS of the applications. However, the selection and appropriate configuration of the right QoS policies constitutes a great challenge in dynamic environments. The complexity of this task is twofold. On the one hand, the use of a static set of QoS policies, considered optimal under particular conditions, may be inadequate (or at least suboptimal) when these conditions change at run-time. On the other hand, changing the QoS policies at run-time may lead to incompatibilities, since the policies used by the different participants within the DDS architecture model need to be mutually consistent.

In this article, we describe a model-driven approach that enables the safe, automatic and transparent adaptation of the QoS policies in DDS-based middleware, providing the best performance possible within the available resources at run-time. In summary, the main contributions of this article are:

1. The concept of *communication templates* as a high-level abstraction mechanism to specify QoS policies in a distributed communication scenario. A communication template is a predefined configuration of QoS policies that covers a common use case for DDS-based systems. Although communication templates provide clear semantics about the kind of communication that occurs between entities (e.g., level of reliability or queue lengths), they are intended to be general enough to enable reusability and application independence.
2. The use of software adaptation to support *adaptive QoS policies* in communication templates. The values of these policies are purposely left open at design-time and dynamically adjusted, with a best-effort approach, to optimize non-functional properties, such as performance or resource consumption. Moreover, we provide designers with the means to specify the limits of the policies variability according to their applications and prevent the occurrence of incompatible configurations.
3. A model-driven process to create, refine, validate and instantiate communication templates in DDS-based applications.

The rest of the paper is organized as follows. Section 2 provides a background on DDS. Section 3 introduces the proposed model-driven process. Section 4 presents the implemented tools for modeling adaptive QoS. Section 5 describes the run-time process and shows the results of executing adaptive QoS on an example. Section 6 reviews related work and, finally, Section 7 draws some conclusions and outlines future work.



```
module SensorExample {
   struct sensorInfo {
      unsigned long sensor_id;
      unsigned long value;
   };
   // sensor_id is a key instance};
```

Figure 1: Overview of a DDS system.

II. DDS: Architecture and Components

DDS is a modern standard that introduces a software architecture, based on the publish/subscribe communication model, aimed at enabling scalable, real-time, dependable, high performance and interoperable data exchange. The standard is composed of a *Data-Centric Publish and Subscribe* (DCPS) model, and a *DDS Interoperability Wire Protocol* (DDSI). The former defines the DDS architecture, the entities involved in the communication (e.g., Publisher, Subscriber, Topic, DataReader, DataWriter, etc.), and a standard API, along with some profiles that enhance its use [1]. The latter defines a protocol that ensures interoperability across DDS implementations from different vendors [2].

The publish/subscribe model implemented in DDS does not use any central broker to avoid a single point of failure. Conversely, publishers and subscribers access to the so-called *global data space* to exchange information, as illustrated in Figure 1. Topics are the exchange information unit in DDS. They identify data of a particular type with an associated QoS. Topic types are defined using an *Interface Description Language* (IDL) and can be defined with a key that identifies different data flows (called *instances*). DDS also defines the concept of sample, which represents a topic instance over time. Figure 1 shows the definition of a topic type called *sensorInfo* considering *sensor id* as key. Topic instances have been represented as (*sensor id*, *value*) pairs.

DDS entities can be configured with a wide set of QoS policies (the standard defines twenty two types) and these policies must be compatible for the communication to take place. QoS policies specify different features associated with data delivery, data availability, data timeliness, resources, configuration, and entities lifecycle. For instance, there are policies for establishing resource limits for data queues (in terms of the number of samples, instances, and instances per sample), liveliness, reliability, deadlines and lifespan, among others. QoS policies are applied to a particular entity at run-time once it has been created and activated. Although some of these policies are immutable during the system execution, there are many others that can be modified at run-time (see the DDS specification [1] for detailed information).

III. Communication Templates for Adaptive QoS

When designers face a particular problem, they (consciously or unconsciously) tend to apply design patterns, i.e., well-proven solutions to recurring problems appearing in similar design contexts. This approach avoids errors and prevents designers from spending time and effort in reinventing solutions. Communication templates aim to support this approach in the domain of data distribution. In particular, we introduce a development process in which communication templates play a key role in assisting designers with the selection and configuration of appropriate QoS policies for DDS-based systems.

Let us introduce the proposed development process with an example that will be used throughout the paper. This example takes place in a shopping center where a social robot, called Gualzru, moves around promoting products and services. One of the main goals of this robot is to interact with people, showing them offers and product discounts that are displayed as commercials in an external panel screen. In order to be more effective, when Gualzru meets a potential customer, it creates a profile of the person based on his or her age and gender. Gualzru invites the person to follow it to the advertising panel, where the offers and products being displayed will be selected according to the profile.

The core functionality of this example is provided by four services depicted in Figure 2: (1) *Capturer*; (2) *Classifier*; (3) *DisplayManager*; and (4) *Panel*. The first three services are executed in the robot whereas the last one is executed in the advertising panel. The *Capturer* takes images from the robot camera and publishes them to the *Classifier*. The *Classifier* segmentates every image in order to obtain the face of a person (if any). Then, it classifies these data and assigns a gender and an age to the person. This information is sent to the *DisplayManager*, which is responsible for selecting and arranging the information that will be displayed according to the profile of the customer. Finally, the *Panel* receives this information and displays it in the panel screen.

Figure 3 shows the proposed model-driven process. The main steps are:

1. **DDS experts define communication templates**. Using the *Quality of Service Modeling Language* (QoSML), experts can create communication templates to cover recurring problems in publish/subscribe applications that, from their experience, hold certain properties. A template is created as a predefined configuration of QoS policies and designed to be reusable, i.e., to enable designers to exploit them as many times as needed in different applications. Figure 2 shows the use of three templates in the robotic example, each one for addressing a different type of communication. The *continuous data* template addresses processes in which data are constantly updated. It may be suitable for the transmission of the robot images, or any sensor data. The *events* template targets processes where some relevant events need to be managed, e.g., the detection and classification of a potential customer. And, finally, the *state information* template can be prescribed for processes in which data are sporadically updated and have long validity, e.g., the configuration of the Panel according to the person profile. In addition, Figure 2 includes the key QoS policies prescribed by each of these templates. This specification is based on the DDS use cases documented by Hunt in [3].

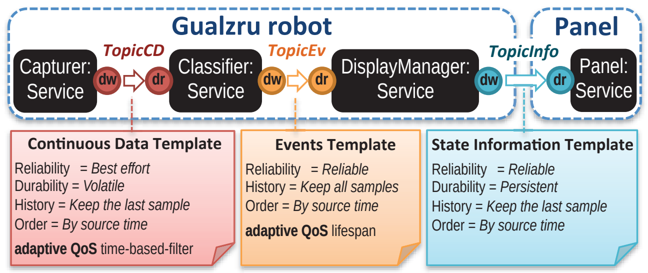


Figure 2: Overview of the example. Data writers and readers appear as *dw* and *dr*, respectively.

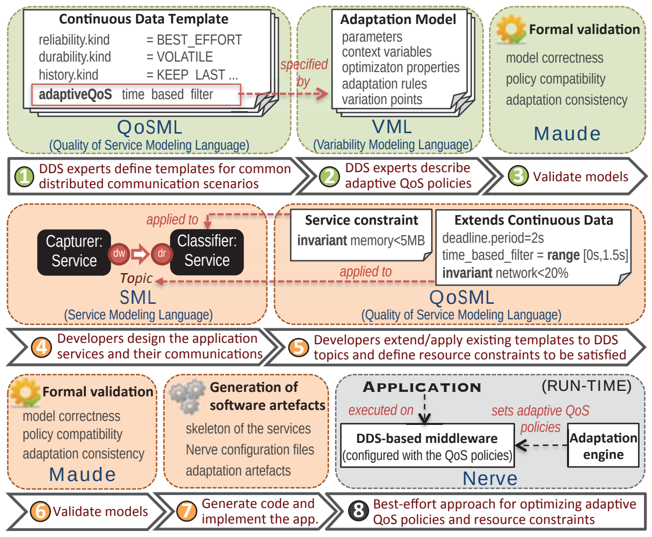

Continuous Data Template
reliability.kind = BEST_EFFORT
durability.kind = VOLATILE
history.kind = KEEP_LAST ...
adaptiveQoS time_based_filter
specified by
Adaptation Model
parameters
context variables
optimizaton properties
adaptation rules
variation points
Formal validation
model correctness
policy compatibility
adaptation consistency
QoSML
(Quality of Service Modeling Language)
VML
(Variability Modeling Language)
Maude
1 DDS experts define templates for common distributed communication scenarios
2 DDS experts describe adaptive QoS policies
3 Validate models
applied to
Capturer: Service
dw
dr
Classifier: Service
Topic
SML
(Service Modeling Language)
Service constraint
invariant memory<5MB
applied to
Extends Continuous Data
deadline.period=2s
time_based_filter = range [0s,1.5s]
invariant network<20%
QoSML
(Quality of Service Modeling Language)
4 Developers design the application services and their communications
5 Developers extend/apply existing templates to DDS topics and define resource constraints to be satisfied
Formal validation
model correctness
policy compatibility
adaptation consistency
Maude
Generation of software artefacts
skeleton of the services
Nerve configuration files
adaptation artefacts
APPLICATION (RUN-TIME)
executed on
sets adaptive QoS policies
DDS-based middleware (configured with the QoS policies)
Adaptation engine
Nerve
6 Validate models
7 Generate code and implement the app.
8 Best-effort approach for optimizing adaptive QoS policies and resource constraints

Figure 3: Proposed model-driven process.

2. **DDS experts specify adaptive QoS policies**. Apart from fixed configurations, communication templates can also declare adaptive QoS policies, i.e., policies that can be adjusted at run-time according to the context. For example, it may occur that the *Classifier* cannot handle all the images sent from *Capturer*, as the sending rate is variable depending on the camera parameters (such as changes in lighting or movement). In order to improve the efficiency of this communication, *continuous data* defines as adaptive the *time-based-filter* policy. This policy specifies the minimum separation period between subsequent samples, allowing the *Classifier* to control the flow of images, transparently to the application. Experts specify the adaptation using the *Variability Modeling Language* (VML).
3. **DDS experts validate the models**. Once the templates are completely defined and before making them available to developers, experts need to validate the QoSML and VML models. For this purpose, we provide a set of facilities implemented with Maude [4], as a formal verification framework.
4. **Developers design the application services**. Using the *Service Modeling Language* (SML), developers design the service architecture, i.e., (1) the services of the application; and (2) the DDS entities (i.e., data readers, writers and topics) for the publish/subscribe processes connecting these services.
5. **Developers apply communication templates and constraints.** From a repository of communication templates (and their corresponding adaptation models), developers select those that better fit their needs. Then, they assign templates to the different communications among the services in the SML model. Every DDS entity will be configured with the policies described in the templates. For instance, in Figure 2, *continuous data* is linked to the data topic *TopicCD*, consequently, the data writer in *Capturer* and the reader in *Classifier* are set accordingly to this template. In case a template does not fully comply with the requirements of the application, developers can use QoSML to extend it. In addition to setting policies, developers can refine adaptive QoS policies by reducing their variability range. For example, step 5 in Figure 3 illustrates how the range of the *time-based-filter* policy is limited to [0, 1.5] seconds. Thus, at run-time, the adaptation process will decide the best value in that range. If the adaptive QoS policy is set with a fixed value, the adaptation process will have no effect on it. Regarding resource constraints, QoSML allows developers to define invariants, such as the maximum allowed CPU, network or memory consumption. These invariants can target (1) a concrete service or (2) a particular communication process (defined by a data topic). At run-time, the adaptive QoS policies are adjusted to meet these constraints.
6. **Developers validate the models**. They need to validate their specifications similarly to step 3.
7. **Developers generate the code and implement the application services**. A transformation generates the application run-time artifacts to be executed in Nerve [5,6], a DDS-based middleware enabled with QoS monitoring and reconfiguration capability.
8. **The system execution**. Once developers deploy the services in the different machines, Nerve automatically instantiates and initializes all the DDS entities with the prescribed QoS policies and adjusts the adaptive QoS policies dynamically following a best-effort approach.

IV. Modeling Communication Templates

In this section we detail the modeling languages in the proposed model-driven process using the robotic example introduced in Section 3[1]. The first two subsections describe how DDS experts can create communication templates with QoSML and VML. After that, we address the use of SML and QoSML to allow developers to configure their applications by applying communication templates.

A. *Defining communication templates*

QoSML supports the definition of communication templates. Through these templates, DDS experts advise the use of certain QoS policies to address common use cases. We have created a textual editor for QoSML using the Xtext

[1] Additional material related to this work can be found in www.lcc.uma.es/~jmcruz/journal

framework [7]. This editor includes syntax checking, coloring and a completion assistant, among other features. Listing 1 shows the definition of the three communication templates in the robotic example, namely, *continuous data*, *events* and *state information*. Note that the QoS configurations prescribed by these templates have been specified based on the notions provided in [3].

Each template sets some QoS policies. For instance, *continuous data* (lines 17-21) establishes: (1) *BEST_EFFORT* delivery; (2) *VOLATILE* durability; (3) *KEEP_LAST* to set a queue of depth *1*; and (4) *BY_SOURCE_TIMESTAMP* samples order. It is worth noting that the base description of the DDS policies is also modeled with QoSML. Listing 2 shows an excerpt of this specification, where the declared QoS attributes (names, data types and default values) comply with the DDS standard [1]. Therefore, the policies omitted in the templates are set by default according to this description.

Apart from fixing the values of some policies, templates can also state adaptive QoS policies (`adaptiveQoS`). This is the case of the policies `time_based_filter` and `lifespan` in *continuous data* and *events* respectively (see lines 14 and 31 in Listing 1). While the former policy allows data flow control, `lifespan` can be used as a mechanism to control the memory usage. This policy handles the expiration time of the samples, beyond which they are removed from any queue. Highlight that this mechanism is perfectly consistent with communications based on events, since events normally become irrelevant as time passes. In addition, adaptive QoS policies need to be associated with a VML adaptation model to specify how they are adjusted at run-time. In particular, each adaptive QoS policy must be linked to a *variation point* (`vml::varpoint`), which is a type of variable defined in the VML model. For example, line 15 shows the mapping between the variation point `tmin` and the attribute `minimum_separation` of `time-based-filter`. The following subsection will describe how variation points are specified.

Also concerning the adaptation model, we can make VML parameters accessible in QoSML. This will enable application developers to easily tune the adaptation process when extending a template, without having to deal with VML. In this vein, QoSML includes general and specific primitives. On the one hand, as for the general ones, lines 9-10 show how `maxoverload` (defined in *continuousData.vml*) is accessible through the parameter `max_overload`. On the other hand, QoSML includes a specific primitive for the declaration of resources (`resourceDef`). For example, line 7 indicates that the parameter `maxmemload`, defined in *events.vml*, limits the maximum memory usage of the entities affected by the *event* template. This will allow developers to express constraints, such as MEMORY<20Mb, when extending the template. Note that the procedure to satisfy this constraint depends on the adaptation described in VML.

Another useful QoSML primitive is `appliedTo`. Although a template can potentially arrange all data readers and writers involved in the same publish/subscribe process, we can provide specific settings to a selection of readers and writers. E.g., line 13 limits the scope to the *READERS*.

```
1  import "ddsQoSPolicies.qos"
2  import "continuousData.vml" as vmlcdata
3  import "events.vml" as vmlevents
4
5  resourceDef CPU, MEMORY {
6     vml::param vmlcdata.maxcpuload limits max CPU;
7     vml::param vmlevents.maxmemload limits max MEMORY;
8  }
9  paramDef max_overload : number {
10   vml::param vmlcdata.maxoverload;
11 }
12 template continuous_data_template {
13    appliedTo READERS {
14      adaptiveQoS time_based_filter.minimum_separation
15           vml::varpoint vmlcdata.tmin;
16    }
17    set reliability.kind       to BEST_EFFORT;
18    set durability.kind        to VOLATILE;
19    set history.kind           to KEEP_LAST;
20    set history.depth          to 1;
21    set destination_order.kind to BY_SOURCE_TIMESTAMP;
22 }
23 template state_info_template {
24   set reliability.kind       to RELIABLE;
25   set durability.kind        to PERSISTENT;
26   set history.kind           to KEEP_LAST;
27   set history.depth          to 1;
28   set destination_order.kind to BY_SOURCE_TIMESTAMP;
29 }
30 template events_template {
31   adaptiveQoS lifespan.duration
32      vml::varpoint vmlevents.lifespan;
33   set reliability.kind       to RELIABLE;
34   set history.kind           to KEEP_ALL;
35   set destination_order.kind to BY_SOURCE_TIMESTAMP;
36 }
```

Listing 1: Templates definition using QoSML (file *templates.qos*).

```
qospolicy time_based_filter {
       minumum_separation : time = 0s;
}
qospolicy reliability {
       kind : enum {RELIABLE, BEST_EFFORT} = RELIABLE;
       max_blocking_time : time = 100ms; } ...
```

Listing 2: DDS policies with QoSML (excerpt of *ddsQoSPolicies.qos*)

B. *Specifying adaptive QoS policies*

DDS experts can use VML to specify the variability of the adaptive QoS policies declared in the communication templates. Like QoSML, we have created a textual editor for VML using the Xtext framework [7]. In previous work, VML was successfully applied in other application domains, such as robotics [8] or data visualization [9].

VML offers, among others, primitives for modeling: (1) the variation points of the system; (2) the context variables; and (3) a set of rules and properties that enables the computation of (1) based on (2). Aligned with *Dynamic Software Product Lines* (DSPL) [10], VML *variation points* (`varpoint`) represent points in the software where different variants might be chosen to derive the final system configuration at run-time. Therefore, variation points determine the decision space in the VML models, i.e., the answer to *what* can change. Listing 3 shows the VML model used for adapting the *time-based-filter* policy in *continuous data*. Recall this policy allows designers to adjust the flow of data received by a data reader. In the model, the variation point `tmin` (line 3) represents the minimum separation period between subsequent updates. Note that it has been declared as a number that takes values between 0 and 2 seconds with a precision of 5ms.

```
1  import "continuousData.dat" as objectives
2
3  varpoint tmin    : number [0:0.005:2]
4  context overload : number [0:0.1:100]
5  context cpuload  : number [0:0.1:100]
6
7  param maxcpuload  : number [0:0.1:100] := 100
8  param maxoverload : number [0:0.1:100] := 12
9  var state : enum {STEADY,OVERLOADED,OVERSIZED}:= STEADY
10
11 property resources minimizes {
12   objective := data objectives.resources
13   weight := 0
14 }
15 property performance maximizes {
16   objective := data objectives.performance
17   weight := 1 - resources.weight
18 }
19 rule rule1 : overload > maxoverload or cpuload > maxcpuload
20 implies {
21    state := OVERLOADED, resources.weight += 0.1
22 }
23 rule rule2 : timeFromLastUpdate(tmin)>200s and state=STEADY
24 implies {
25    state := OVERSIZED, resources.weight -= 0.1
26 }
27 rule rule3 : overload<=maxoverload and cpuload<=maxcpuload
28 and state=OVERLOADED implies {
29    state := STEADY }
30 rule rule4 : overload<=maxoverload and cpuload<=maxcpuload
32 and state=OVERSIZED implies {
33    resources.weight -= 0.1 }
```

Listing 3: VML model for time-based-filter (file continuousData.vml).

```
1  tmin = [0.0, 0.005, 0.01, 0.015, 0.02,...]
2  resources(tmin) = [0.0,0.29359,0.41061,0.48721,0.54331,...]
3  performance(tmin)=[1.0,0.92004,0.85697,0.80619,0.76382,...]
```

Listing 4: Excerpt of the file continuousData.dat.

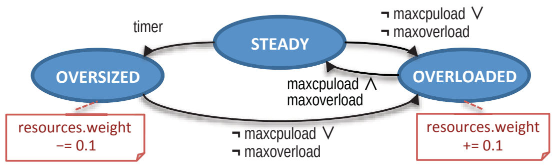
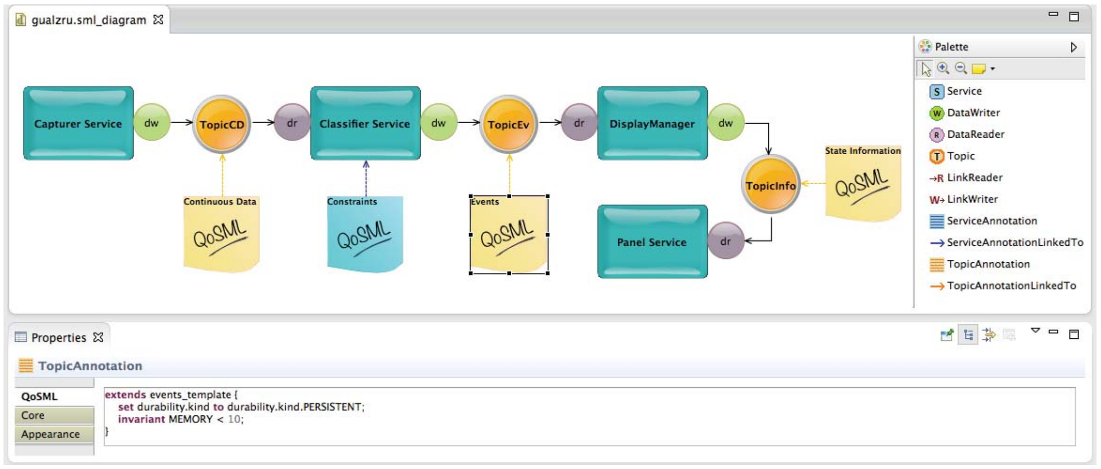


Figure 4: State diagram representing the modification of weights. Maxcpuload and maxoverload indicates the satisfaction or non-satisfaction (¬) of the CPU load limit and the overload limit, respectively.

Once variation points have been defined, we need to specify the *context variables* (`context`). These variables allow us to identify the situations in which variation points need to be adapted. For instance, Listing 3 includes the definition of the context variable `overload` (line 4) that indicates the percentage of incoming data that exceeds the capacity of a data reader. When a reader gets more data than it can handle, its overload will be greater than 0, which is a sign of inefficiency and waste of resources. Consequently, adaptation should avoid this situation keeping this value as low as possible by adapting `tmin`. For this, we need to define *how* variation points are set according to the context variables. In VML, this is achieved through *properties* (`property`) and Event-Condition-Action (ECA) *rules* (`rule`). On the one hand, properties specify features of the system that need to be optimized, i.e., minimized or maximized. Each property is defined using two functions: *objectives* and *weights*. While the first ones characterize the properties in terms of variation points (i.e., define the functions to be optimized), the second ones define the importance of each property in a given context (i.e., they weight the objective functions such that higher weighted properties have greater impact than lower weighted ones). On the other hand, the rules define relationships between context variables and variation points. These relationships might be direct (e.g., when we set a concrete value for a variation point) or indirect (e.g., when we change the weight of a property).

In the VML model shown in Listing 3, there are two properties: (1) `resources` (lines 11-14), aimed at minimizing resource consumption; and (2) `performance` (lines 15-18), aimed at maximizing performance. Note that the adaptation process will have to find the right balance between these two properties considering the current situation. In this case, minimizing resources and maximizing performance move `tmin` in opposite directions: decreasing `tmin` could imply more CPU consumption and network bandwidth but, at the same time, it could improve the throughput and, thus, the performance. The objective functions to be optimized can be described in VML through mathematical expressions or data import. The latter approach, used in Listing 3 (line 1), helps VML designers to exploit the advantages of widely used environments for numerical computing. In the example, we have modeled the mathematics of the VML properties through simulations and empirical data with Matlab [11]. The resulting mathematical model has been discretized to obtain a finite set of data that is imported in VML (see Listing 4). In addition, the weights are defined in the properties (lines 12 and 16) and updated in the rules (e.g., see line 21). For example, the weights of `resources` and `performance` are initialized to 0 and its complement, respectively. This means that the adaptation will start considering `performance` the most important property. Then, both weights will evolve in opposite directions depending on how `resources.weight` is updated in the rules. Before describing those rules, mention that we have introduced two parameters (lines 7-8) to help users customize the adaptation process. They are: `maxoverload` and `maxcpuload`, which indicates the maximum allowed percentage of overload and CPU load, respectively. The current value of CPU load is received by the context variable `cpuload`. Recall these parameters were already declared in the QoSML model in Listing 1 to allow developers to configure their values when extending a template. Note that the parameters are set to their default values in case developers do not determine them.

The rules in Listing 3 (lines 19-33) update the weights depending on three possible situations, expressed by the variable state. Namely, (1) *STEADY*, the constraints imposed by `maxoverload` and `maxcpuload` are met and then the weights are preserved; (2) *OVERLOADED*, at least one constraint is not satisfied, which, in order to reduce the load, a rule gradually increases the weight assigned to resources (this weight remains constant when it reaches its maximum value, i.e., 1); and (3) *OVERSIZED*, this case is reached after a period without updates, during which the system may have become suboptimal. Therefore, a rule gradually decreases the weight assigned to resources (this weight remains constant when it reaches 0). Figure 4 shows the state diagram inferred from these rules.



```
extends events_template {
    set durability.kind to durability.kind.PERSISTENT;
    invariant MEMORY < 10;
}
```

Figure 5: SML editor with the model developed for the example.

C. *Applying communication templates*

Once DDS experts complete the definition of their templates, they can make them available by uploading the QoSML models (and the associated VML files, if any) to an on-line repository. Then, application developers will be able to search and select the communication templates that better fit their needs from those available in the repository. However, before developers need to deal with communication templates, they have to model the service architecture of the application using SML. For that, we provide developers with a graphical editor implemented with GMF framework [12] (see Figure 5).

SML allows developers to define services and the DDS entities that connect them. Regarding the latter, we have considered a simplified representation of the DDS publish/subscribe model, in which some entities do not appear explicitly. As we can see in Figure 5, services can have attached several data readers and writers, depicted as *dr* and *dw* in the diagram. These elements need to be linked to a data topic to denote the type of data that is exchanged (e.g., *TopicCD* is used to connect *Capturer* and *Classifier*). It is worth noting that several data writers (acting as publishers) and readers (as subscribers) may be linked to the same topic. Moreover, developers can configure some DDS aspects of a selected entity in the property view of the editor. For example, data topics allow the configuration of the IDL file that defines them, and the partition and the domain of the readers and writers associated with the topic. Although DDS publishers and subscribers are omitted in the diagram, they are taken into account since we consider an implicit publisher or subscriber for each data writer or reader, resp.

SML allows annotating data topics and services with QoSML descriptions. Figure 5 shows annotations linked to each topic and the *Classifier* service. The content of these annotations is stated in Listing 5. It is also worth noting that the QoSML editor is conveniently embedded in the property view when we select any annotation element. This editor has been prepared to automatically import the QoSML models created by DDS experts (see Listings 1 and 2). Thus, application developers will be able to refer to the definitions they content, e.g., when extending a template.

```
TopicCD annotation:
1 extends continuous_data_template {
2    set deadline.period to 2s;
3    set time_based_filter.minimum_separation range(0s, 1.5s);
4    set max_overload to 10; }
TopicEv annotation:
5 extends events_template {
6    set durability.kind to PERSISTENT;
7    invariant MEMORY < 10; }
TopicInfo annotation:
8 applies state_info_template;
Classifier Service annotation:
9 invariant CPU < 15;
```

Listing 5: QoSML specification in the annotations of the SML model.

```
applies {
  set reliability.kind to RELIABLE;
  appliedTo READER "dr1" {
    set history.kind    to KEEP_LAST;
    set history.depth   to 10; }}
```

Listing 6: QoSML specification for a topic annotation with no template.

Although we can configure QoS policies without using templates (Listing 6 shows an example), annotating data topics revolves around communication templates, i.e., to enable either their application or their extension in case the application requirements are not met. In both cases, a template covers the entire publish/subscribe process defined by the data topic, i.e., it configures all the data readers and writers associated with the same topic. Line 8 in Listing 5 shows the application of *state information* in *TopicInfo*. This template fits well with the type of communication between *DisplayManager* and *Panel* and then it does not need any modification. Conversely, the *continuous data* template in *TopicCD* (lines 1-4) and the *events* template in *TopicEv* (lines 5-7) are extended.

There are four types of actions to extend a template: (1) developers can set new QoS values or overwrite existing ones. The *continuous data* template is completed with the *deadline* policy (see line 2) that introduces a real-time requirement in the communication between *Capturer* and *Classifier*; (2) developers can restrict the variation range of an adaptive QoS policy. For instance, line 3 establishes a period between 0 and 1.5 seconds for the *time-based-filter* policy. Thus, at run-time, the adaptation process will decide the best value in that range. In case a QoS policy is set with a fixed value, the adaptation process will have no effect on it. Besides, note that the new range will only be considered whether it is a sub-interval of the one originally defined for the VML variation point (see *tmin*, line 3 in Listing 3); (3) developers can set adaptation parameters, e.g. see *max_overload* in line 4. Recall this parameter was defined in the adaptation model as *maxoverload* and mapped in the template definition; finally, (4) developers can constrain the resources consumed by the data readers and writers associated with a specific topic. The *events* template includes an invariant to keep the memory lower than 10Mb (see line 7). This constraint will affect the data writer in *Classifier* and the data reader in *DisplayManager*. In particular, it indirectly sets the parameter *maxmemload* in their adaptation model according to the resource definition in Listing 1. When this constraint is not met in one of these entities, its adaptation process will limit the memory use by adjusting the *lifespan*.

Regarding service annotations, developers can declare resource constraints in services. For instance, the annotation linked to *Classifier* establishes that the CPU load of this service cannot reach or exceed the 15% of the node capability (see line 9 in Listing 5). To meet a constraint stated in a service, the entities that are susceptible to be adjusted are the readers and writers belonging to the service. In particular, those configured with a template that is able to impact on the resource the constraint expresses. Therefore, the invariant in *Classifier* has an effect on its data reader and not on its writer because only the former is able to adjust the CPU load by adapting the *time-based-filter* policy. This constraint would indirectly set the parameter *maxcpuload* in the adaptation model of *continuous data*, according to the resource definition in Listing 1. Finally, when there is a conflict between two constraints (one specified in a service and another one in a topic) that affect the same data reader or writer, the more restrictive prevails.

D. Semantic validation of the models

Apart from the syntactical correctness, which is basically checked by the editors, the models also need to be semantically validated. In this sense, we consider three essential aspects: *model correctness*, *policy compatibility* and *adaptation consistency*. Firstly, model correctness checks issues concerning the construction of the model itself, e.g., whether the policies are assigned in conformity with their data types, or ensuring that no invariants and ranges are used to extend a template without adaptive QoS policies. As for policy compatibility, it guarantees that the QoS values are consistent with each other according to the DDS standard. Finally, adaptation consistency avoids that readers and writers become incompatible at run-time due to QoS changes. To check these aspects, we have used Maude [4] as a formal verification framework.

To represent QoSML models with Maude, we have adopted an object-based programming approach with Core Maude, based on [13]. In particular, a model is created as a collection of Maude objects, which are record-like structures of the form < o : c | a_1:v_1,..., a_n:v_n >, where *o* is the object identifier, *c* is the class the object belongs to, a_i are attribute identifiers and v_i their corresponding current values. Listing 7 shows the translation of some QoSML descriptions into Maude.

Next, we provide some indications about the implementation in Maude. Concerning model correctness, it basically involves the verification of a set of conditions in the models, which is expressed in Maude as membership equational logic. More complex is assuring policy compatibility since it implies verifying that all the data writers and readers belonging to the same topic are mutually consistent. Communication templates (and its extensions) can prescribe a considerable number of configurations. Recall that their scope can go from targeting all the entities associated with the same data topic to particular settings for a single writer or reader. Therefore, it is necessary to check the validity of any connected pair writer-reader. For this purpose, we use the Maude *search* command to explore the space of possible configurations described in a QoSML model. It allows developers to find counterexamples, in which the configuration of two entities is invalid, analyze the problem and then fix it accordingly. In essence, to enable the search, we need: (1) a set of rewriting rules to produce different combinations of writer-reader instances configured according to the QoSML model; and (2) a set of equations to check the compatibility of these instances.

Finally, regarding adaptation consistency, we have adopted two different approaches. On the one hand, we simulate with Maude the VML model to predict unsuitable configurations of the adaptive QoS policies [14]. On the other hand, as each adaptive QoS policy has its variation range explicitly declared (in the definition of the variation point and occasionally in the template extension), we can check if the whole range is consistent with the rest of prescribed configurations. For instance, in Listing 5, the *time-based-filter* policy is constrained to the interval [0, 1.5] and the *deadline* policy is set to 2s. As these two attributes represent the minimum and the maximum separation period between subsequent samples, a *deadline* lower than the *time-based-filter* value would have been inconsistent. Maude equational logic can support such verifications.

```
qospolicy reliability { kind : enum {RELIABLE,BEST_EFFORT} =
RELIABLE; }
< 'o1 : QoSPolicy | name : "reliability", attrs : 'o2 >
< 'o2 : PolicyAttribute | name : "kind", datatype : 'o3,
  default : 'o6, policy : 'o1 >
< 'o3 : EnumType    | literals : ('o4, 'o5) >
< 'o4 : EnumLiteral | name : "RELIABLE", enumType : 'o3 >
< 'o5 : EnumLiteral | name : "BEST-EFFORT", enumType : 'o3 >
< 'o6 : EnumValue   | value : 'o4 >

template continuous_data_template { set reliability.kind to
BEST_EFFORT; }
< 'o11 : CommTemplate | name : "continuous-data",
  sentences : 'o12 >
< 'o12 : PolicyAssignment | policy : 'o2, value : 'o13 >
< 'o13 : EnumValue        | value : 'o5 >
```

Listing 7: Example of some QoSML sentences in Maude.

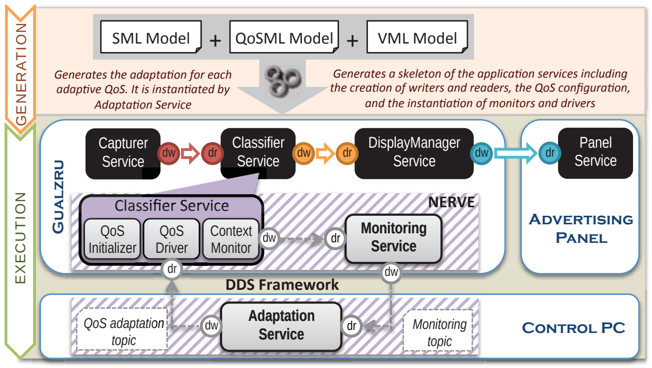

GENERATION
SML Model + QoSML Model + VML Model
Generates the adaptation for each adaptive QoS. It is instantiated by Adaptation Service
Generates a skeleton of the application services including the creation of writers and readers, the QoS configuration, and the instantiation of monitors and drivers
EXECUTION
GUALZRU
Capturer Service
Classifier Service
DisplayManager Service
Panel Service
dw
dr
Classifier Service
QoS Initializer
QoS Driver
Context Monitor
Monitoring Service
NERVE
ADVERTISING PANEL
DDS Framework
QoS adaptation topic
Adaptation Service
Monitoring topic
CONTROL PC

Figure 6: The model transformation and the run-time process.

V. Running Adaptive QoS

This section presents how the models are executed in Nerve [5,6], the DDS-based middleware on which adaptive QoS become effective. We conclude the section by illustrating the benefits of running adaptive QoS in the robotic example.

A. *Execution of adaptive QoS in Nerve*

In order to take the SML, QoSML and VML specifications into Nerve, we provide developers with a model-to-text transformation. Figure 6 outlines this transformation and the main elements taking place at run-time. It is worth noting that the modeling languages involved in the process are platform independent. They are aimed at raising the level of abstraction of dealing with QoS configuration and its adaptation. Therefore, among other benefits, it would allow targeting platforms different from Nerve through the development of new transformations.

In our process, the model-to-text transformation is divided into three steps: (1) the generation of the Nerve configuration file with the QoS values assigned to data writers and readers; (2) the generation of the code skeleton for each service. The developer will need to complete it with the application code; and (3) the generation of the artifacts that are run by the adaptation service.

Regarding the execution part, Figure 6 depicts the Nerve elements that come into action when a service has a writer or a reader with at least an adaptive QoS policy. The *adaptation service* is the “brain” that selects values for the adaptive QoS policies. This service executes one process for each adaptive writer or reader. In the robotic example, the *adaptation service* holds three processes: two for *Classifier* (it has a reader and a writer configured as a result of applying two communication templates with adaptation, i.e., *continuous data* and *events*) and one process for *DisplayManager* (its reader is set by the *events* template). Recall that *continuous data* does not prescribe writers any adaptive QoS policy, for that reason *Capturer* does not have adaptation. In the current implementation of the example, we have deployed the *adaptation service* in a separate node to reduce the system overhead (see *Control PC* in Figure 6). However, it would have been possible to distribute these three adaptation processes as needed, thanks to the support provided by the underlying DDS framework. Concerning the operation, the *adaptation service* has to solve the constrained optimization problem posed by the VML models according to the context situation. The current implementation basically consists of two parts: (1) a *Finite State Machine* to update the property weights (which is specified by the adaptation rules in the VML model, see Figure 4) and (2) an engine to compute the variation points through the optimization of the adaptation properties. For the latter, Nerve uses a constraint solver from the *G12 Constraint Programming Platform* [15].

Once variation points are determined, they are put into action through *QoS drivers*, which are responsible for configuring QoS policies at run-time. In addition, the *monitoring service* gathers context information from the application and sends it to the *adaptation service* (at a rate that is fully configurable). The monitoring topic may contain information about: (1) CPU and memory consumption from every service and network traffic in the computer (using the Sigar multiplatform-API [16]); (2) QoS from DDS data writers and readers; and (3) monitoring variables associated with a particular service (e.g., sender and receiver rates, overload, the processing time of an algorithm or tasks, etc.). Highlight that Nerve gives developers the option to implement application-specific monitors and drivers. For instance, it provides the possibility of targeting as variation points other attributes different from the DDS QoS policies, such as parameters for adjusting the transport protocol. Finally, when the system starts, the *QoS initializer* configures the QoS policies according to the Nerve configuration file. This file includes, in XML format, all the QoS settings defined to each data reader and writer of the application, i.e., (1) policies prescribed in a template and not overwritten (see Listing 1); (2) those overwritten in a template extension (see Listing 5); and (3) all the other policies with their default values (see Listing 2). Note that the adaptive QoS policies are initialized with default values.

B. *Adaptive QoS in action*

We have measured the performance of the example system with and without the *adaptation service*. Regarding the *continuous data* template, the dotted line in the left graph of Figure 7 shows the evolution of the *Classifier's* overload without adaptation. Note that the overload stays around 40%, which means that the *Classifier* is receiving more data than it can process. It also implies that a percentage of the data sent by *Capturer* is being overwritten in the *Classifier's* queue (*continuous data* prescribes a history length of 1). Therefore, there is a waste of resources in terms of network bandwidth and CPU consumption. Considering now adaptation, *continuous data* tries to mitigate this waste of resources by adjusting the *time-based-filter* QoS policy (observe the line with square markers in the left graph of Figure 7). The objective of this adaptation is to obtain an overload value lower than 10% (this percentage was set when the template was extended, see line 4 in Listing 5). Due to the high initial overload, the *adaptation service* gradually increases the *time-based-filter* QoS policy up to 0.09s. After a few seconds, the overload remains stable at 9.9% (note the solid line in the left graph of Figure 7). As a result, the *Capturer* sending rate and the *Classifier* receiving rate become more balanced.

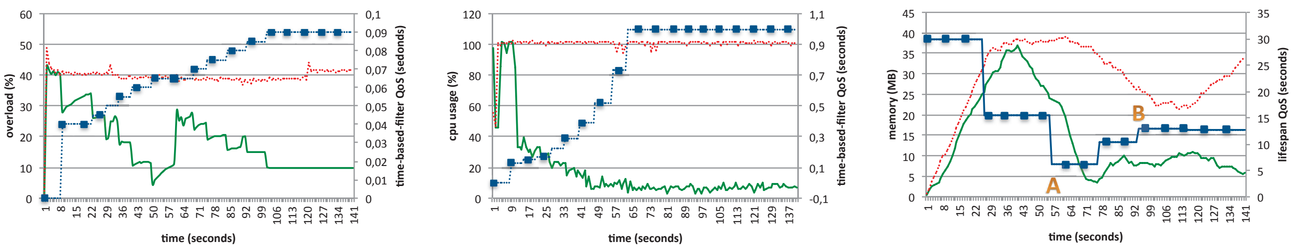


Figure 7: (Left axis) Solid/dotted lines meaning the Classifier's performance with (solid) and without adaptation (dotted). (Right axis) Lines with square markers representing the value of the QoS policy.

The middle graph in Figure 7 shows the evolution of the *time-based-filter* policy when we establish the invariant *CPU < 15%* in the *Classifier* service. This policy is adjusted to a more restrictive value than in the previous case, causing a decrease in the CPU load until it meets the established threshold. Highlight that invariants are addressed by the *adaptation service*, see the difference between the dotted and solid line in the middle graph of Figure 7. The adaptation assumes a best-effort approach for accomplishing resource requirements, thus, we cannot assure their inviolability as it depends on factors beyond the middleware, such as the implementation of the application services.

Regarding the application of the *events* template in the *Classifier* and *DisplayManager*, the right graph of Figure 7 shows the memory consumption of *Classifier*. When the *adaptation service* is disabled, the memory consumption grows up to 35-40 MB (see the dotted line). This is the result of prescribing data persistence (see line 6 in Listing 5), which is useful for debugging tasks, since the data writer in *Classifier* records all the results about the people detected. However, it implies that the memory consumption could grow indefinitely. Fortunately, the configuration of the *lifespan* QoS policy allows the *adaptation service* to control the size of persistent queues by changing the data expiration time. Initially, we have configured this QoS policy with a value of 30 seconds. The solid line in the right graph shows the evolution of the memory considering adaptation and the invariant: *memory < 10MB*. The *lifespan* decreases to 6.1 seconds at the execution time marked as A (see the line with square markers). This reduces the memory consumption to 3.5MB. However, the *adaptation service* has to optimize the overall performance by using all the available memory, that is, to adjust the *lifespan* to achieve a consumption as close as possible to the limit imposed by the invariant. Consequently, the *adaptation service* increases the lifespan value to almost 13 seconds at the execution time marked as B, which stabilizes the memory consumption between 5 and 10MB.

VI. Related Work

The concept of communication templates is not new. Outside DDS, the same idea was introduced by the *communication patterns* of Smartsoft [17], which provide fixed semantics for the most common communication scenarios. Furthermore, adaptation is becoming increasingly important in distributed systems, e.g. consider the growth of adaptive streaming technologies for optimizing the viewing experience of users [18]. In the DDS domain, among the efforts invested in trying to overcome the complexity of dealing with the end-to-end QoS of the applications, Real-Time Innovations Inc. (RTI) has proposed the *built-in QoS profiles* [19] in its DDS implementation. It consists in predefined configurations of QoS policies that developers can use to create DDS entities with specific QoS properties. They define three types of built-in QoS profiles: (1) *baselines*, which define default configurations for each DDS QoS policy; (2) *generics*, to represent simple communication features; and (3) *patterns*, which describe domain-specific use cases in terms of generics, including some configuration patterns documented by Hunt [3]. It is worth noting that developers can create new QoS profiles by extending the existing ones through XML configuration files. QoS profiles are similar to communication templates in that both provide an abstraction, which can be reused and extended, and allow developers to think about the behavior they want to achieve rather than how to configure each QoS policy individually. Apart from other differences, QoS profiles differ from communication templates in that they do not consider design-time validation (e.g., policy incompatibilities are only detected at run-time and informed to the user by means of exceptions).

Other DDS vendors have developed modeling tools, such as OpenSplice Modeler [20], for creating QoS configurations. These tools normally provide a graphical environment to (1) represent the DDS entities involved in the application; (2) set their QoS policies; (3) check the configuration; and (4) generate the implementation artifacts. Unlike our proposal, these approaches (including built-in QoS profiles) are generally tied to a particular vendor's technology and to the details of the DDS standard. They do not give any support for modeling adaptive QoS policies or mechanism to adjust resources dynamically. Besides, they also are deficient in development processes aimed at separating roles and concerns while encouraging reusability.

In the reminder of this section we review some work concerning dynamic QoS adaptation in the middleware. Hoffert et al. have developed ADAMANT (*ADAptive Middleware And Network Transports*) [21] to maintain QoS properties in dynamic environments for distributed real-time and embedded systems. Although ADAMANT uses DDS to propagate the monitoring information needed to determine adaptations, its target is not the DDS QoS policies. Conversely, ADAMANT puts the focus on selecting and configuring the transport protocol to address the QoS concerns at run-time. For this purpose, it uses several supervised machine learning techniques and a reconfigurable transport layer. Other research also copes with the adaptation of transport protocols, such as [22]. This seems to be a relevant issue that could be exploited together with the adaptation of QoS policies. We believe that the notion of communication template for adaptive QoS is not limited to DDS QoS policies and that our approach could also contribute to this issue. In addition, Hoffert et al. [23] propose the *Distributed QoS Modeling Language* (DQML) that helps developers to set and generate valid QoS configurations. However, this language does not include any abstraction comparable to our communication templates or elements to express and tune the adaptation at run-time.

Boonma et al. [24] present a DDS-based middleware for wireless sensor networks, in which the event routing protocol is adapted to satisfy QoS properties according to the available resources and the performance. While this work seems to obtain good results, it (as [21]) lacks in methods for modeling and validating the adaptation process. Regarding this concern, some researchers propose the use of formal methods to provide developers with rigorous tools for designing and testing the correctness of their systems. In this vein, Loulou et al. [25] present P/S-CoM, an approach developed with Z notation for supporting the correct modeling and the safe dynamic reconfiguration of the internal architectural style in a publish/subscribe middleware. Z notation is similar to Maude in that both enable the formal specification and verification of the adaptation, which is posed as a remarkable open issue [26].

Gray et al. [27] describe the *Adaptive Quality Modeling Language* (AQML) that allows modeling, simulating and generating QoS adaptation software. AQML defines three views: (1) QoS adaptation modeling, which specifies the adaptation logic through finite-state machines; (2) computation modeling, which describes the system architecture; and (3) middleware modeling, which includes the services and the system conditions (e.g., throughput or latency) provided by the middleware. Designers can express any parameter in the components in (2) and (3), which allows the adaptation logic in (1) (through in/out events and data in transitions and states) to observe and tailor the behavior of the system. The analysis of the adaptation is performed from a centric point of view using Matlab. An engine translates the QoS adaptation specifications defined in AQML into a Simulink/Stateflow model, which enables the evaluation of the state machine, e.g., to check the stability of the system or simulate the state transitions. As our approach, this work promotes the separation of concerns (e.g., the adaptation and the QoS configuration is set apart from the application logic). It also allows model validation and generates run-time artifacts, which enhances correctness and scalability, among other benefits. However, Gray et al. do not put special emphasis on the reusability and the separation of roles. Thus, application developers will need to deal with all the modeling details while, in our approach, they are able to reuse and refine models, e.g., they can specify resource constraints in a simple way without dealing with the adaptation logic.

Other research work is specifically aimed at managing the fluctuations in application workload and system resources. For example, Wang et al. [28] present a middleware mechanism to control the CPU utilization in distributed real-time and embedded systems. Unlike our proposal, where the adaptation assumes a best-effort role accomplishing resource requirements, approaches like [28] try to achieve a more accurate control. For this reason, these may tend to be tied to the application, the underlying technology or a particular setting, which would limit some of the principles that our modeling process promulgates, such as reusability. Even so, the insights resulted from this research may allow us to enrich the corresponding adaptation models with new strategies and algorithms.

VII. Conclusions and Future Work

In this paper we have described a model-driven approach for supporting the modeling, validation and generation of adaptive QoS configurations in DDS-based middleware. Our proposal revolves around the notion of communication template, i.e., an abstraction that represents a predefined configuration of (adaptive or fixed) QoS policies. Considering different roles, first, communication templates are created and then reused to configure the middleware. This process is supported using three modeling languages: the *Variability Modeling Language* (VML), the *Quality of Service Modeling Language* (QoSML) and the *Service Modeling Language* (SML), which are used to (1) specify the adaptation logic, (2) create/extends templates and define resource constraints, and (3) bind templates to publish/subscribe processes. Moreover, we have presented the validation of the models using Maude, aimed at preventing the occurrence of incompatible QoS configurations. Finally, after transforming the validated models into run-time artifacts in Nerve, the results of executing adaptive QoS policies were shown in an example.

For the future, some of the open challenges that we consider pivotal are listed next.

- **Adaptation consistency in complex systems**. Although we are able to validate to some extent our models with Maude, the verification of the global effects derived from a number of adaptive QoS policies running autonomously is still open. Note that each adaptive QoS policy seeks optimal values based on its local context only. Therefore, the challenge is twofold, on the one hand, to foresee side effects by providing tools capable of checking the interrelations among VML models and, on the other hand, to develop run-time mechanisms for coordinating and leading individual adaptations to a more global optimization of the system.
- **Extension of the QoS description**. We have designed communication templates not only to be used with DDS, but also to support different ways of describing QoS. In this sense, it would be interesting to extend our modeling languages to consider the adaptation of transport protocols, or to investigate the connection between communication templates and other high-level specifications, such as MARTE [29] or AADL [30].
- **Practical application to industry**. We plan to test our approach with more extensive case studies, which will allow us to incorporate the gained experience into our

research. Moreover, we are still working on improving the modeling tools presented in the paper. Although the current version is functional, there are some parts that need to be polished before making our tools available to everyone. For example, the automatic transformation to generate the Maude representation from the QoSML, VML and SML models is still under development. Currently, the Maude files are written manually.

Acknowledgment

This work has been partially supported by the Spanish Ministerio de Economía y Competitividad (TIN2015-65686-C5-4-R and TIN2015-69957-R). Juan F. Inglés-Romero thanks Fundación Séneca-CARM for a research grant (Exp. 15561/FPI/10).

References

[1] *Data Distribution Service for Real-time Systems (DDS)*, Object Management Group, 2007.

[2] *The Real-time Publish-Subscribe Wire Protocol DDS Interoperability Wire Protocol specification*, Object Management Group, 2009.

[3] G. A. Hunt, "DDS Use Cases: Effective Application of DDS Patterns and QoS", in *OMG's Workshop on Distributed Object Computing for Real-time and Embedded Systems*, 2006.

[4] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, J. Quesada, "Maude: specification and programming in rewriting logic," *Theorical Comp. Sci.*, vol. 285, no. 2, pp. 187-243, Aug, 2002

[5] J. Martínez, A. Romero-Garcés, J.P. Bandera-Rubio, R. Marfil-Robles, A. Bandera-Rubio, "A DDS-based middleware for quality-of-service and high-performance networked robotics," *Concurrency and Computation: Practice and Experience*, vol. 24, no. 16, pp. 1940-1952, 2012.

[6] A. Romero-Garcés, J. F. Inglés-Romero, J. Martínez, C. Vicente-Chicote, "Self-adaptive Quality-of-Service in distributed middleware for robotics," in *Proc. 2nd Workshop on Recognition and Action for Scene Understanding*, 2013.

[7] Xtext. (2017) [Online]. Available: www.eclipse.org/Xtext.

[8] A. Lotz, J. F. Inglés-Romero, M. Lutz, D. Stampfer, C. Vicente-Chicote, C. Schlegel, "Towards a Stepwise Variability Management Process for Complex Systems - A Robotics Perspective," *International Journal of Information System Modeling and Design*, vol. 5, no. 3, pp. 55-74, 2014.

[9] J. F. Inglés-Romero, R. Morales-Chaparro, C. Vicente-Chicote, F. Sánchez-Figueroa, "A Model-Based Approach to Develop Self-Adaptive Data Visualizations," in *Proc. 22nd Int. Conf. on Information Systems Development*, pp. 345-357, 2013.

[10] S. Hallsteinsen, M. Hinchey, S. Park, K. Schmid, "Dynamic software product lines," *IEEE Computer* , vol. 41, no. 4, pp. 93–95, 2008.

[11] Matlab. [Online] 2017. Available: www.mathworks.com.

[12] The Eclipse Graphical Modeling Framework. [Online] 2017. Available: www.eclipse.org/modeling/gmp/

[13] A. Boronat, J. Meseguer, "An algebraic semantics for MOF," *Formal Aspects Comput.*, vol. 22, no.3, pp. 269-296, 2010.

[14] J. F. Inglés-Romero, C. Vicente-Chicote, "Towards a formal approach for prototyping and verifying self-adaptive systems," *Lecture Notes in Business Information Processing*, Springer Berlin Heidelberg, vol. 148, pp. 432–446, 2013.

[15] G12 Constraint Programming Platform. [Online] 2017. Available: https://users.cecs.anu.edu.au/~jks/G12/

[16] The Sigar cross-platform API. [Online] 2017. Available: https://support.hyperic.com/display/SIGAR

[17] D. Stampfer, A. Lotz, M. Lutz, C. Schlegel, "The SmartMDSD Tollchain: An integrated MDSD Workflow and Integrated Development Environment (IDE) for Robotics Software". Journal of Software Engineering for Robotics, vol. 7, no. 1, pp. 3-19, July 2016.

[18] B. Li, Z. Wang, J. Liu, W. Zhu, "Two decades of Internet video streaming: A retrospective view," *Multimedia Computing, Communications, and Applications*, vol. 9, no. 1, pp. 33-53, 2013.

[19] Real-Time Innovations Inc. (RTI), "Built-in QoS profiles,". [Online] 2017. Available: http://blogs.rti.com/2014/02/11/built-in-qos-profiles

[20] Vortex OpenSplice Modeler. [Online] 2017. Available: http://www.prismtech.com/vortex/vortex-opensplice/tools/modeler

[21] J. W. Hoffert, A. Gokhale, D. C. Schmidt, "Timely Autonomic Adaptation of Publish/Subscribe Middleware in Dynamic Environments," *Int. J. Adaptive, Resilient and Autonomic Systems, vol.* 2, no. 4, pp. 1–24, 2011.

[22] J. H. Hwang *et al*., "DR-TCP: Downloadable and reconfigurable TCP," *J. Syst. and Soft.*, vol. 81, pp. 83–99, 2008.

[23] D. S. Joseph W. Hoffert, A. Gokhale, "DQML: A Modeling Language for Configuring Distributed Publish/Subscribe Quality of Service Policies," in *Proc. 10th Int. Symp. on Distributed Objects, Middleware, and Applications*, 2008.

[24] P. Boonma, J. Suzuki, "Self-Configurable Publish/Subscribe Middleware for Wireless Sensor Networks," in *Proc. 6th IEEE Conf. Consumer Communications and Networking*, pp. 1376–1383, 2009.

[25] I. Loulou, M. Jmaiel, K. Drira, A. H. Kacem, "P/S-CoM: Building correct by design Publish/Subscribe architectural styles with safe reconfiguration," *J. Syst. and Soft.* , vol.83, no.3, pp.412-428, 2010.

[26] M. Salehie, L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM Trans. on Autonomous and Adaptive Systems*, vol. 4, no. 2, pp. 1–42, 2009.

[27] J. Gray, S. Neema, J. Zhang, Y. Lin, T. Bapty, A. Gokhale, D. C. Schmidt, "Concern Separation for Adaptive QoS Modeling in Distributed Real-Time Embedded Systems," in *Behavioral Modeling for Embedded Systems and Technologies: Applications for Design and Implementation*, Information Science Reference, 2009.

[28] X. Wang, Y. Chen, C. Lu, X. Koutsoukos, "FC-ORB: A robust distributed real-time embedded middleware with end-to-end utilization control," *J. Syst. and Soft.*, vol.80, no.7, pp.938-950, 2007.

[29] *Modeling Analysis of Real-Time Enbedded Systems* (MARTE), Object Management Group, 2011.

[30] *Architecture Analysis and Design Language* (AADL), SAE standard AS-5506, 2004.