QRML: A Component Language and Toolset for Quality and Resource Management

Freek van den Berg[1], Václav Čamra[2], Martijn Hendriks[1,4], Marc Geilen[1], Petr Hnetynka[2]
Fernando Manteca[3], Pablo Sánchez[3], Tomáš Bureš[2], Twan Basten[1,4]
f.g.b.v.d.berg@tue.nl, vakamra@gmail.com, martijn.hendriks@tno.nl, m.c.w.geilen@tue.nl, hnetynka@d3s.mff.cuni.cz
mantecaf@teisa.unican.es, sanchez@teisa.unican.es, bures@d3s.mff.cuni.cz, a.a.basten@tue.nl
[1]Electronic Systems, Eindhoven University of Technology, Eindhoven, Netherlands
[2]Distributed and Dependable Systems, Charles University, Prague, Czech Republic
[3]Grupo de Ingeniería Microelectrónica, Universidad de Cantabria, Santander, Spain
[4]ESI (TNO), Eindhoven, The Netherlands

***Abstract*—Cyber-physical systems (CPS) are complex, heterogeneous, and dynamic systems, spanning hardware and software components ranging from edge devices to cloud platforms. CPS need to satisfy many rigorous constraints, e.g., with respect to deadlines, safety, and quality, yielding a large configuration space where only a limited number of configurations meet the constraints and only a fraction are optimal regarding certain qualities. Finding the optimal configurations is hard, especially during runtime operation.**

We present QRML, the Quality and Resource Management domain-specific Language, and an accompanying toolset. QRML enables specifying heterogeneous hardware/software systems and their composition and configurations conveniently, automated reasoning about them, and generating implementation artifacts like quality and resource monitoring templates. A QRML model consists of a hierarchy of components. Component specifications express constraints and requirements, that may serve multi-objective quality and resource optimization and exploration purposes. The QRML toolset offers language support, visualizations, documentation generation, template-code generation, and constraint-solving support.

***Index Terms*—Quality and Resource Management, Domain-Specific Language, Component-Based Design, Multi-Objective Optimization, Monitoring, Cyber-Physical Systems**

I. Introduction

QRML– [**kar**-*uh*-m*uh* l] – a kind of chewy candy, commonly in small blocks, made from sugar, butter, milk, etc. [WordReference Random House Unabridged Dictionary of American English]

Cyber-Physical Systems (CPS) integrate cyber systems, human users, networks, and physical systems. A CPS needs context awareness, which is computationally intensive and challenging. Moreover, a CPS comprises increasingly complex distributed configurations, reflected in a growing number of sensors, actuators and other smart devices. Advanced image and video processing are increasingly used as sensors. All this leads to an immense number of dynamic system configurations in combination with heavy processing and communication loads. To make matters worse, a typical CPS needs to simultaneously satisfy many rigorous constraints, for instance with respect to hard deadlines and performance, safety, power dissipation, and information quality. Finding the best configurations that satisfy all constraints is hard, at design time, but especially during system operation at runtime.

Component-based design is widely used to tackle both design complexity and operational complexity of today's systems. The Component Object Model (COM) [2], the Common Object Request Broker Architecture (CORBA) [3], JavaBeans [4], the Open Services Gateway Initiative (OSGi) [5], and Service-Oriented Architecture (SOA) [6] are well-known component models. The approaches focus on software design and particularly address functional design and runtime interoperation. In CPS design, however, there is a need to integrally address a large variety of components covering hardware, software, networks, sensors, and so on. Moreover, it is important to address extra-functional aspects such as performance (latency, throughput), power dissipation, information quality, usage of processing, storage, and communication resources, and so on. These extra-functional aspects can be categorized into quality aspects (performance, power, information quality) and resource aspects (processing, storage, communication). Managing these quality and resource aspects in the broad sense is referred to as Quality and Resource Management (QRM). None of the component models in use today explicitly address QRM.

A mathematical component model targeting QRM was proposed in [7]. A component in that model has an interface consisting of six parts: input, output, provided budget, required budget, qualities, and parameters. Inputs and outputs capture functional and data dependencies between components, to the extent that they are relevant for QRM. Received video input may, for example, impact the needed processing budget and the processing latency. Provided and required budgets make the resource needs explicit, like the already mentioned processing budget. Latency is an example of a quality exposed by a component. It may, for instance, be used to reason about end-to-end latency of a video pipeline composed of multiple components. The configuration parameters in a component interface make explicit which aspects are externally controllable. Examples are the desired resolution or rate of a video

This work was done as part of the FitOptiVis project [1] funded by the ECSEL Joint Undertaking, grant H2020-ECSEL-2017-2-783162.

output or the configuration of a hardware accelerator. The component model of [7] can be used to formally specify QRM component models and resulting multi-objective constraint-optimization problems, such as the problem of finding the lowest-latency configuration that satisfies given constraints on rate and resolution of a video processing pipeline. For CPS, QRM optimization problems need to be addressed both at design time and during runtime operation, because, in the latter case, only then sufficient information on e.g. available resources, user requirements or experienced communication latency, is known. Constraints may be hard (e.g. in safety-critical systems) or soft. The component model provides the needed mathematical basis to specify constraint models that can be used for different solvers at design time or runtime managers built into CPS. It also provides a basis for developing runtime monitoring support that is needed to get an operational view on a system. The QRM component model is complementary to other component models in use; it does not address the functional aspects typically addressed by other component models. Instead, it provides an additional view on the system, the QRM view.

In this paper, we present a domain-specific language (DSL) for QRM, called QRML (for QRM Language), and an accompanying toolset. We aim to support component-based design and reasoning that focuses on QRM. We choose to do this through a DSL because it is lightweight and not limited to a single component model in use. A DSL integrates well with other component models, languages, and tools. State-of-the-art language workbenches such as Eclipse Xtext support the definition, evolution and extension of DSLs. We leverage these facilities to define QRML and support a number of QRM use cases. In particular, we support the definition and development of QRM component models, visualization of the compositional structure of those models, visualization of monitoring data about qualities and resource usage, solving QRM constraint problems by exporting models to a constraint solver, and a specialization towards Service-Oriented Architecture (SOA) including generation of C++ code templates for monitoring. These use cases illustrate the versatility of QRML and its toolset, complementing existing design methods and tools with a QRM view on the system. The language and toolset are set up to be extensible. QRML tools are available as an Eclipse package and through a web interface: https://qrml.org.

The paper is organized as follows. We present a running example in Section II that serves to motivate and illustrate the QRML language and toolset. Section III presents the architecture of the QRML language and its toolset, as well as the language syntax. Section IV presents the toolset functionality, covering the already mentioned use cases. Section V discusses related work. Section VI concludes.

II. Motivating example

We consider a Biometric Access Control (BAC, [8]) system that grants access to e.g. buildings, rooms, files, or data on the basis of face recognition, e.g., by analyzing the relative position, size, and/or shape of the eyes, nose, cheekbones, and jaw. Fig. 1 shows a component model of the BAC system following [7]. In line with the Y-chart philosophy [9], we define the BAC *application*, the BAC *platform* and connect them via a *mapping*. This separation of concerns facilitates modeling and reasoning about QRM. The BAC application performs three steps in a pipelined fashion: *face detection*: capturing an image and detecting a face; *face recognition*: identifying a person from the detected face; and, *access control*: looking up the access rights of an identified person.

A key platform component is the *smart camera*, a machine-vision system providing, in normal mode, image capturing and analysis. In advanced mode, it also extracts person IDs from captured images. A second platform component is the *compute platform*, either local or in the cloud, that can extract person IDs from images containing a face and it can look up access rights for a given person ID in a database.

Face detection is mapped to the smart camera and access control to the compute platform. Face recognition is mapped either to the smart camera or to the compute platform. The system user can control this via parameter fr.

In this simplified example, the BAC system has two qualities. First, the *end-to-end latency* is the accrued latency of the three application steps. It depends on the mapping and selected platform modes. Second, the *face recognition quality* is low when done by the smart camera, medium when done by the local compute platform, and high when done in the cloud. The user cannot directly control the selection of the local or cloud alternatives when face recognition is mapped to the compute platform (via parameter fr); the best fitting configuration depends on quality and latency requirements and may be selected at runtime by a quality- and resource manager.

III. QRML

A. Architecture

System developers facing QRM challenges may use the specification, visualization, reasoning and code-generation capabilities of the QRML toolset to support their development work. Moreover, functionality of the toolset may need to be extended to better support certain domain-specific aspects. The QRML toolset currently supports the following use cases: testing the layout.

- Specifying a QRM view on a system in the form of a QRML model.
- Visualizing the modular and compositional structure of a QRML model and visualizing monitoring data.
- Solving multi-objective constraint problems specified by a QRML model.
- Generating code templates from QRML models.
- Extending QRML with domain-specific capabilities.

Language-workbench technology [10] is a good match to realize these use cases as it provides metamodeling, construction of editing environments, and definition of execution semantics. Furthermore, many language workbenches have support for extensibility. Examples of mature language workbenches are Eclipse Xtext [11] and Jetbrains MPS [12]. We have realized the toolset using the Eclipse Xtext tooling.

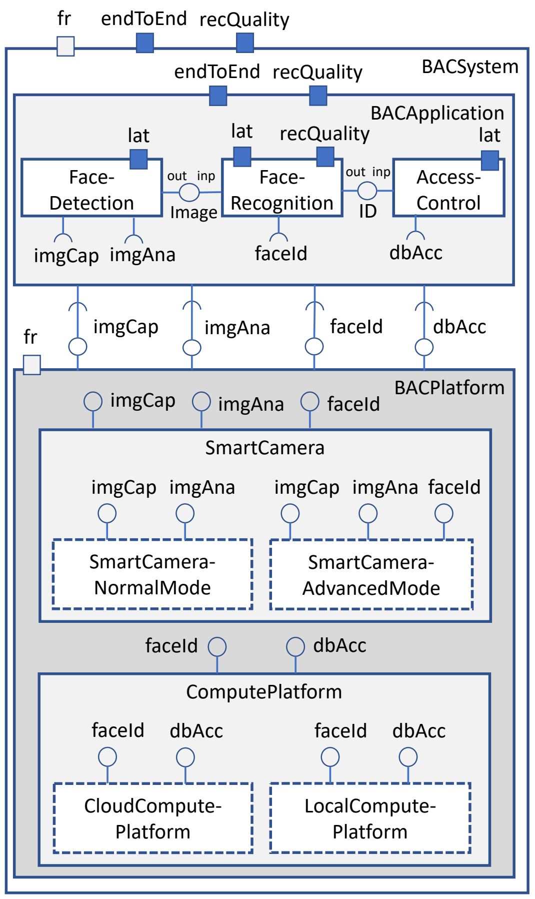

Fig. 1: The BAC component composition, displaying budget and I/O dependencies, qualities and parameters.

Fig. 2 shows the architecture of the key toolset components. The QRML DSL and the QRML-core language form the basis. QRML is intended as a front-end language usable by system designers. QRML-core is a fully expressive subset of QRML that omits constructs that make modeling easier, but not more expressive. QRML-core targets tool developers.

Specific views on QRML and on QRML models are supported for visualization and constraint solving. An extension of QRML targets SOA, including support for monitoring. We use extensibility mechanisms provided by the language workbench. SOA support is achieved by extending QRML to SDSL, the Service-oriented Design Specification Language. The chosen architecture allows other extensions targeting specific application domains or specifying different views and supporting different use cases for QRML models.

Transformations connect models in the different toolset languages and provide access to functionality such as constraint solving. First, we have an endogenous model-to-model transformation that transforms a QRML model to a QRML-core model. Second, we have exogenous transformations that transform (extended) QRML(-core) models into specific views. Visualization support is provided for the full QRML language by generating a JSON description that can be visualized in the standalone QRMLvis React component. QRMLvis integrates

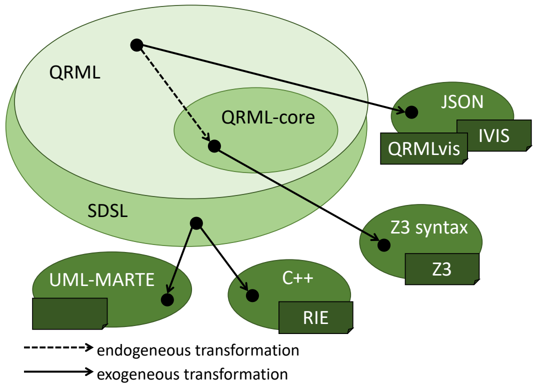

Fig. 2: The architecture of the QRML language and toolset.

with the IVIS framework [13] for visualizing monitoring data. QRML-core models can also be exported to the Z3 SMT solver [14] to solve QRM constraint models derived from a QRML model (like finding feasible BAC configurations within a given latency constraint). SDSL specializes QRML for SOA, including support for specifying monitors. From SDSL models, UML-MARTE model templates and C++ monitoring templates can be generated. While SDSL provides a QRM view on the system, UML-MARTE models may be used for design activities such as performance and schedulability analysis. The generated C++ implementation templates can be used in the RIE library, that provides a SOA implementation platform. The transformations illustrate various ways to connect QRML models to external functionality.

B. The language

The key concept in QRML is the *component*. A component has an interface consisting of six parts: *inputs*, *outputs*, *required budgets*, *provided budgets*, *qualities*, and *parameters*. Fig. 1 illustrates these concepts. Inputs and outputs capture functional data transferred between components, such as images and IDs in the BACApplication component. Required and provided budgets capture the resource dependencies between components. The FaceDetection component, and, hence, BACApplication, needs for instance an image-capturing resource budget 'imgCap' that is provided by the SmartCamera component in BACPlatform. Parameters, like the 'fr' parameter in the BAC example, expose controllable aspects of a component, to be used, for instance, by end users or at runtime by a quality- and resource manager (a QRM), whereas qualities (latency, recognition quality) expose metrics of interest to users or to a QRM. Component compositions as shown in Fig. 1 specify I/O connections and budget matches, possibly controlled by the parameters, and they make derived qualities explicit. Component compositions have a well-defined mathematical semantics (see [7]) in the form of a constraint model of feasible configurations that can be used to optimize exposed qualities.

We use the BAC system example to introduce QRML and its toolset. Fig. 3 shows the modular structure of the QRML BAC specification as a class diagram. Tables I to VII illustrate

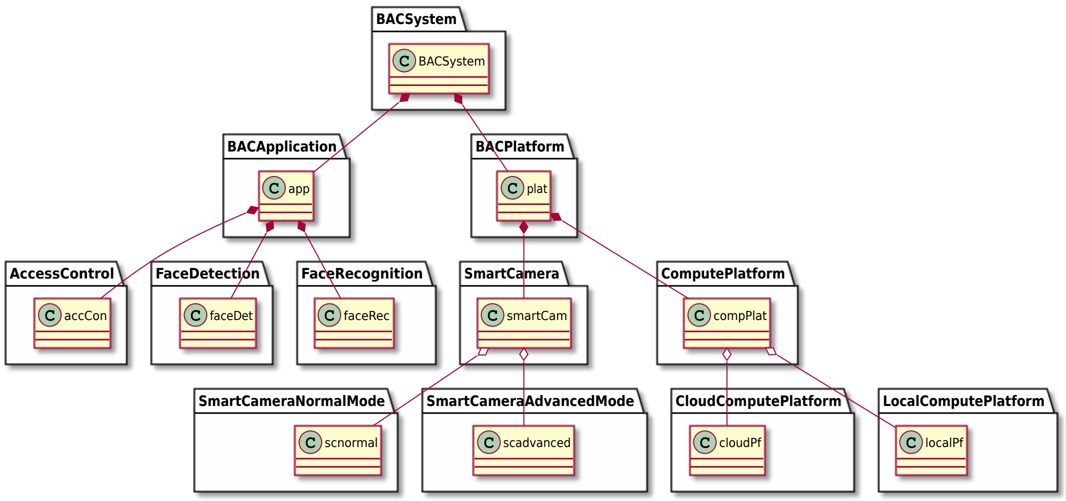

Fig. 3: The modular specification structure of the BAC system

TABLE I: The QRML code of FaceRecognition

```
1.  component FaceRecognition {
2.    inputs Image inp;
3.    outputs ID out;
4.    requires FaceIdentification faceId;
5.    quality Latency lat; lat.latency==faceId.latency;
6.    quality Quality recQuality; recQuality.qual==faceId.qual;
7.  }
```

TABLE II: The QRML code of BACApplication

```
1.  component BACApplication {
2.    component FaceDetection faceDet;
3.    component FaceRecognition faceRec;
4.    component AccessControl accCon;
5.    faceDet.out outputs to faceRec.inp;
6.    faceRec.out outputs to accCon.inp;
7.    requires ImageCapturing imgCap; imgCap==faceDet.imgCap;
8.    requires ImageAnalysis imgAna; imgAna==faceDet.imgAna;
9.    requires FaceIdentification faceId; faceId==faceRec.faceId;
10.   requires DatabaseAccess dbAcc; dbAcc==accCon.dbAcc;
11.   quality Latency endToEnd; endToEnd==faceDet.lat.latency +
    faceRec.lat.latency + accCon.lat.latency;
12.   quality Quality recQuality; recQual-
    ity.qual==faceRec.recQuality.qual;
13. }
```

TABLE III: The QRML code of SmartCamera

```
1.  component SmartCamera {
2.    alternatives {
3.      component SmartCameraNormalMode scnormal;
4.      component SmartCameraAdvancedMode scadvanced;
5.    }
6.  }
```

TABLE IV: The QRML code of SmartCameraAdvancedMode

```
1.  component SmartCameraAdvancedMode {
2.    supports ImageCapturing imgCap;
3.    supports ImageAnalysis imgAna { latency==50; };
4.    supports FaceIdentification faceId { qual==low; latency==20; };
5.  }
```

TABLE V: The QRML code of BACPlatform

```
1.  component BACPlatform {
2.    parameter faceRecPlatform fr;
3.    component SmartCamera smartCam;
4.    component ComputePlatform compPlat;
5.    supports ImageCapturing imgCap; imgCap==smartCam.imgCap;
6.    supports ImageAnalysis imgAna; imgAna==smartCam.imgAna;
7.    supports FaceIdentification faceId;
8.    fr.plat==smartCam → faceId==smartCam.faceId;
9.    fr.plat==compPlat → faceId==compPlat.faceId;
10.   supports DatabaseAccess dbAcc; dbAcc==compPlat.dbAcc;
11. }
```

TABLE VI: The QRML code of BACSystem

```
1.  main component BACSystem {
2.    parameter faceRecPlatform fr;
3.    component BACApplication app;
4.    component BACPlatform plt;
5.    plt.fr.plat==fr.plat;
6.    app.imgCap runs on plt.imgCap;
7.    app.imgAna runs on plt.imgAna;
8.    app.faceId runs on plt.faceId;
9.    app.dbAcc runs on plt.dbAcc;
10.   quality Latency endToEnd;
11.   endToEnd.latency==app.endToEnd.latency;
12.   quality FaceIdentification recQuality;
13.   recQuality.qual==app.recQuality.qual;
14. }
```

the BAC model. The figure and LaTeX sources for the tables were automatically generated.

First, we define the application. Component FaceRecognition (Table I) receives images as input (Line 2) and outputs person IDs (Line 3). It has two qualities, viz., the latency for recognizing a face (Line 5) and the recognition quality (Line 6). The quality values are determined by (Lines 5, 6) the received faceId budget (Line 4).

Type definitions, such as Image, ID and FaceIdentification, are given in Table VII. QRML supports type definitions that decompose a type in terms of primitive types, e.g., booleans, integers, reals and enumerations, and other (hierarchical) types. The budget keyword is used for budget types, a channel keyword for input and output types, and a typedef otherwise.

Components FaceDetection and AccessControl are specified analogously to the FaceRecognition component. Component BACApplication (Table II) combines instances of the mentioned components (Lines 2–4) in a pipeline (Lines 5, 6), yielding four required budgets (Lines 7–10): image capturing, image analysis, face identification and database access. Moreover, it has two qualities, viz., the accrued latency of the three

TABLE VII: The QRML code of the TypeDef statements

```
budget ImageCapturing {}
budget ImageAnalysis { integer latency; }
budget FaceIdentification { integer latency; Quality qual; }
budget DatabaseAccess { integer latency; }
channel Image { }
channel ID {}
typedef faceRecPlatform{enumeration{smartCam,comPlat} plat;}
typedef Quality { enumeration { low, medium, high } qual; }
```

TABLE VIII: Quality values for platform components

	SmartCamera		**ComputePlatform**	
	Normal	**Advanced**	**Local**	**Cloud**
imgAna.latency	25	50	-	-
faceId.qual	-	low	medium	high
faceId.latency	-	20	50	100
dbAcc.latency	-	-	50	100

individual components (Line 11) and the recognition quality inherited from component faceRec (Line 12).

Second, we define the platform. The SmartCamera component (Table III) provides two alternative modes, specified again as components. SmartCameraAdvancedMode (Table IV) provides three budgets (Lines 2–4) for image capturing, image analysis, and face identification, respectively. SmartCameraNormalMode is specified similarly but lacks face identification support. Table VIII shows the quality values for both modes. The SmartCamera component provides as budgets the union of the provided budgets of its two alternatives.

Similarly, components CloudComputePlatform and LocalComputePlatform form alternatives of ComputePlatform. Table VIII also provides their quality values. Component BACPlatform (Table V) aggregates SmartCamera and ComputePlatform. It has a parameter ‘fr’ (Line 2) that determines the internal matching of the ‘faceId’ budget (Lines 8, 9). Table VIII illustrates that a higher recognition quality comes at the price of a higher latency. The parameter is not inherited by component ComputePlatform, implying that the choice for the local or the cloud platform variant is not controllable. A QRM will have to ensure that a platform configuration is selected that fits with any given latency and quality requirements.

Third and finally, we define main component BACSystem (Table VI) that maps the application to the platform by aggregating application and platform component instances (Lines 3, 4), mapping the required to provided budgets (Lines 6–9), exposing the two application qualities (Lines 10–13), and introducing parameter ‘fr’ for platform selection (Lines 2, 5). QRML has a keyword ‘main’ that allows modelers to identify a main system component, as done in this example.

As explained earlier, a QRML model can be transformed into a QRML-core model. QRML-core is a fully expressive QRML subset designed to facilitate tool development. The channel and budget keywords, for instance, are replaced by typedef in QRML-core. These keywords facilitate the modeler, but are not needed to, e.g., check type consistency.

IV. The QRML toolset

QRML tools are available as an Eclipse package and through a web interface: https://qrml.org. This section provides an overview of the current status of the toolset.

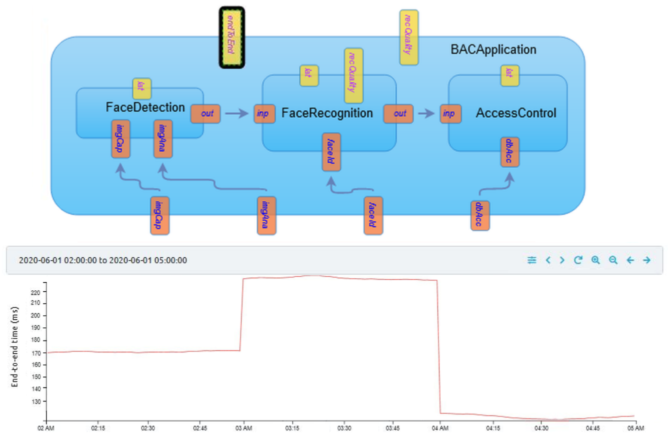

Fig. 4: Visualization of a QRML model in QRMLvis (top) and a latency graph (bottom) in the IVIS framework

A. Language support

Language support enables a designer to create valid QRML models in an efficient way. QRML provides language support in near-real-time, i.e., several checks are performed immediately after the system designer makes any change to a model.

- Keyword highlighting, syntax checking, input completion, and checks for unused objects.
- Reference checking: checks if references to elements in the model are correct.
- Check for naming conflicts: prevent duplicate names so that references are unambiguous.

B. Visualizations

The QRML toolset is equipped with two kinds of visualizations. One transforms the component hierarchy of a QRML model into a class diagram compiled using PlantUML [15]; see Fig. 3. The second visualization is a JSON-based web widget [16], QRMLvis, that visualizes the I/O connections and budget matches in the compositional structure of QRML models. The top part of Fig. 4 shows the BACApplication component of the running example visualized in QRMLvis.

QRMLvis can be reused throughout the QRML toolset. Integrated in the QRML editor, it provides visual feedback as the component architecture is being developed. It is also integrated in the IVIS framework [13] which is a data visualization and data processing framework similar to Grafana [17]. Fig. 4 shows in fact an IVIS snapshot. The bottom part of the figure shows a line graph for the development of the ‘endToEnd’ (latency) quality of the BACApplication component over time. The quality is selected in the QRMLvis visualization. This example graph uses synthetic data, but it illustrates how data received from monitoring a system in operation can be integrally viewed in the component architecture of the system.

C. Documentation generation

The QRML toolset automatically generates documentation in Markdown [18] and LaTeX [19] format, which can be transformed into other formats to serve many purposes. QRML

Listing 1: SDSL model of the VideoTrace monitor

```
monitor VideoTrace{
  provider unican;
  event Performance{
     string com;
     float timeStamp;
     float latency;
  }
}
```

Listing 2: Monitor instantiation for FaceRecognition

```
 component FaceRecognition {
      monitor VideoTrace mon1;
 }
```

Listing 3: Code generated for monitor VideoTrace

```
#include <RIE.h>
using namespace RIE_Methodology;

class VideoTrace: public RIEMonitor {
public:

  void Performance(string com,
   float timeStamp,float latency);
  VideoTrace(string inst);
  VideoTrace();
  virtual ~VideoTrace();

  vector<string> eventNames;
  string providerName
 };
```

allows to include documentation in `/*doc ...*/` delimiters. Tables I to VI illustrate LaTeX documentation that is generated, whereas the QRML documentation on the website [20] is generated using the Markdown generation.

D. Extending QRML

As explained in Section III-A, QRML can be extended with domain-specific capabilities. One concrete extension is the SDSL extension targeting SOA. Both the QRML language and the toolset have been extended for this. A key design choice is that such an extension should not modify QRML, e.g., to facilitate proper version control. Therefore, we have decided to include options to override the QRML grammar and toolset. In this way, it is possible to leave the QRML language and toolset intact and only specify wherever an addition is needed. This enables extensions that all share the common QRML core.

E. Code generation

QRML has been specialized to support SOA in the form of the SDSL DSL (see Fig. 2). In SOA, components provide services to other components and communicate through well-defined interfaces. SDSL adopts this paradigm and provides monitoring support as well. SDSL adds several keywords to the QRML syntax: the 'interface' keyword for interface support and 'monitor', 'event', and 'provider' keywords for monitoring support. Listing 1 shows a monitor specification declaring the tracer that provides the traces and the event type to monitor. Listing 2 shows a monitor instantiation.

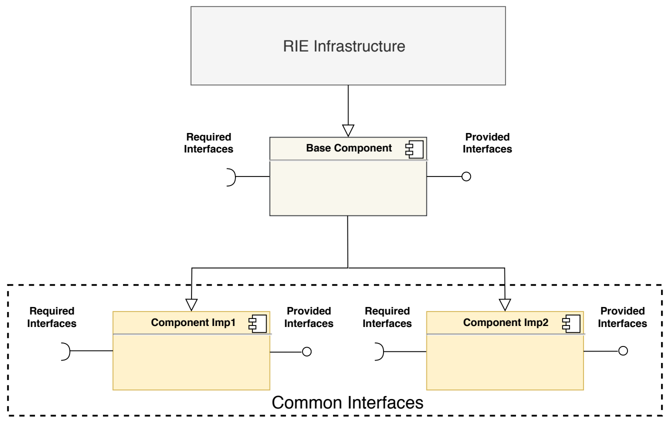

Fig. 5: Code generation for the RIE framework.

We developed a plain-code code generator that produces C++ templates for the Runtime reconfigurable Implementation of Embedded systems (RIE) library. RIE adheres to the SOA paradigm, providing support for runtime component management and monitoring as well as for runtime system reconfiguration and edge deployment. The runtime reconfiguration allows modifying component functionality, resource allocation (e.g. changing a CPU-based implementation to include a HW accelerator) or deployment (e.g. selecting an edge component instead of a local implementation).

For every component in an SDSL model, the C++ code generator creates a C++ base class. Component communications are modeled with C++ virtual functions (services) that are grouped in interface classes [21]. For every interface, the generator creates a base interface class and a remote version for edge deployment. The component class inherits from the RIE library and from interface classes that model provided services. An SDSL component may have several alternatives, such as the SmartCamera component in the BAC model. Classes are generated for each alternative. These alternatives inherit from the base class for the wrapping component and share interfaces with this component (provided and required services). The RIE library provides methods for accessing components and for modifying their set points at runtime. It also supports runtime addition of component alternatives. The implementation of components and the interaction between them can be flexibly added and adapted by the designer. Classes for edge deployment are implemented using synchronous remote procedure calls for interaction. Fig. 5 illustrates the code-generation approach for SDSL.

The code generator also generates code for monitors. Listing 3 shows the C++ code generated for the monitor of Listing 1. The generated code has a method (Performance) corresponding to the declared event. A component generates a monitoring event when calling that function. Once a system is operational, monitored data can be collected and visualized, in, e.g., the aforementioned IVIS visualization framework, providing users with an integral component-based view on the developed system and its runtime operation. Also the open-source Linux LTTng tracer infrastructure is supported.

TABLE IX: Code-generator feature summary

SDSL/QRML element	Generated code
Component	A C++ class inheriting from a RIE component (to inherit runtime functions) and from provided interfaces. In case of component alternatives, they also derive from the base component. Provided services are declared and implemented in the class, since the component derives from the provided interface where those services are declared. Required interfaces are declared as instances of the interface in the component class.
Main component	C++ class that includes system component instances and the connections among provided and required interfaces. It corresponds to the root component of the specification hierarchy (optionally labeled with keyword main).
Interface (budget, input, output)	C++ class with all its services declared as pure virtual methods in the component base class. The component implementations will implement those service functionalities. The generator also provides edge-deployment oriented interface classes.
Qualities and parameters	The qualities and parameters are implemented as component class members.
Type	Specific C++ classes implementing these type elements.
Channel	QRML channels are types implemented with specific interfaces that provide streaming-data read services.
Monitor	C++ class in which all monitored events are defined as methods. The implementation of these methods depends on the tracer infrastructure (e.g., LTTng implementation).

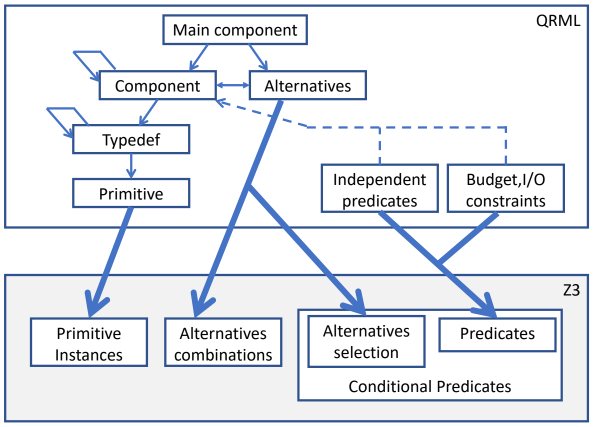

Fig. 6: The QRML to Z3 transformation

Table IX summarizes the transformations from an SDSL model to its C++ template-code implementation.

F. Constraint solving

A QRML model essentially specifies constraints on feasible system configurations that ensure that inputs and outputs, as well as provided and required budgets, match. These matching constraints are precisely defined in the semantic framework of [7]. A QRML model consists of components, possibly with alternatives; components use structured types, building on primitives, such as integers and booleans. Constraints are ultimately phrased in terms of values of primitive types. To find all feasible configurations, a QRML model is first transformed to a QRML-core model, which can then be exported to a constraint solver such as Z3 [14]. The transformations are programmed in Xtend. They are structured in several primitive transformations that e.g. eliminate channel and budget keywords, flatten hierarchical types, and translate budget and

TABLE X: The three elements of a Z3 specification

Primitives
(declare-const BACSystem_app_faceRec_faceId_latency Int) (declare-const BACSystem_app_faceRec_faceId_qual_qual) . . .
Alternatives
(assert (or (and ALT_BACSystem_plt_smartCam_scnormal ALT_BACSystem_plt_compPlat_cloudPf) . . .))
Constraints
(assert ((= BACSystem_app_endToEnd (+ (+ BACSystem_app_faceDet_lat_latency BACSystem_app_faceRec_lat_latency) BACSystem_app_accCon_lat_latency)))) (assert ((and (<= BACSystem_app_imgAna BACSystem_plt_imgAna)))) (assert (→ ALT_BACSystem_plt_smartCam_scadvanced (= BACSystem_plt_smartCam_scadvanced_scadvanced- _imgAna_latency 50)))

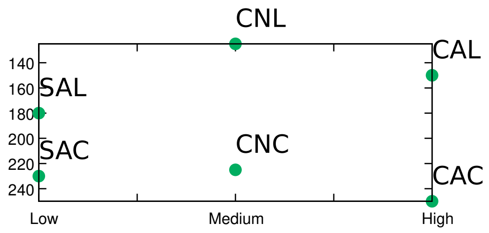

Fig. 7: Latency-quality trade offs for the BAC example.

I/O matches to constraints. The result is a QRML-core model that is very close to a Z3 constraint model, in QRML syntax. The final QRML-core to Z3 transformation involves then three key elements (cf. Fig. 6) as follows.

The first one is a translation of QRML primitive instances to Z3 primitive instances. Table X shows two examples.

Second, the alternatives in the QRML model can be combined in multiple ways, viz., four ways for the BAC example to select both SmartCamera and ComputePlatform alternatives. The Z3 model enumerates all these combinations using the or operator (as shown in Table X).

Third, the constraints need to be made explicit. Table X shows examples of (i) an independent predicate: the sum of three latencies; (ii) a budget constraint: app.imgAna runs on plt.imgAna; (iii) a conditional predicate: the imgAna latency for the SmartCameraAdvancedMode equals 50. The last predicate is conditional, because it is only applicable when the SmartCameraAdvancedMode alternative is selected.

The Z3 model can be analysed using the Z3 theorem prover [14]. Fig. 7 shows a GNUplot, with, for each configuration, the end-to-end latency on the y-axis and recognition quality on the x-axis. Configurations are encoded using three characters: (i) Face recognition runs on **S**martCamera or **C**omputePlatform; (ii) SmartCamera**N**ormalMode or -**A**dvancedMode; (iii) **L**ocal- or **C**loudComputePlatform. Configurations in which FaceRecognition runs on CloudComputePlatform have a better quality and those with LocalComputePlatform have a better latency. Configurations CNL and CAL dominate the other designs. So running FaceRecognition on the SmartCamera does not pay off in this particular example.

V. Related work

Recent work provides a component-interface-modelling framework for QRM [7]. It serves as a mathematical basis for QRML and introduces the notion of budgets, inputs, outputs, parameters and qualities as component-interface parts relevant for QRM. As already explained, QRML is complementary to widely used component models such as COM [2], CORBA [3], JavaBeans [4], OSGi [5], and SOA [6]. We have illustrated how QRML can be combined with SOA. It is also complementary to, and combines very well with, more CPS-related component models such as BIP [22] and Ptolemy [23] that have a formal semantics supporting verification. The formal underpinning of such models combines well with the mathematical basis of QRML. The differentiating aspect of QRML compared to all the mentioned approaches is its focus on QRM (and nothing more).

Besides component models, a wide range of general purpose languages and DSLs is in use, and is being developed, for systems engineering. The Systems Modeling Language (SysML, [24]) is a modelling language for systems engineering applications inspired by UML [25]. It supports the specification, analysis, design, verification and validation of a broad range of (systems-of-)systems. It does, however, not provide specialized support for QRM.

On the other side of the spectrum, we see specialized DSLs that provide performance evaluation support. An example is iDSL [26] that supports performance analysis for service-oriented systems. iDSL, like QRML, adheres to the Y-chart philosophy. iDSL models automatically transform into visualizations and Modest [27] process algebra code to specify semantics and to enable verification. QRML is inspired by such DSLs and builds on DSL technology to make it broadly accessible and ease integration with other methods, without losing its focus on QRM.

A key feature of QRML is its connection to monitoring and visualization. While frameworks such as Kibana [28] and Grafana [17] allow visualization of observed data and performance metrics, the integration of QRML with the IVIS data processing and QRMLvis visualization makes it possible to integrally relate the collected data and performance metrics to architectural elements. We chose the lightweight React [29] visualization framework because it allows easy integration in web environments and other tools. Another route for visualization would be through a full-featured editor built with a tool like Sirius [30]. This may be a future option, when the need arises for more enhanced model-development support.

VI. Conclusions

We have presented QRML, the Quality- and Resource Management Language, and its toolset. QRML is a DSL that targets QRM. The tools provide language and development support using Xtext language workbench technology. The QRML component model is complementary to component models focusing on functionality or other aspects of systems. QRML and its tools have been designed for extensibility, to facilitate specialization to specific domains and integration with other methods in use. We have shown how QRML can be specialized to SOA. QRML has a precise mathematical semantics that allows to formally reason about feasible system configurations and supports multi-objective optimization. QRML provides monitoring support through code generation and visualization. A next step is to couple QRML to a runtime quality- and resource manager based on the same component model. This would allow to configure such a runtime manager from the QRML models and build an operational view on the quality metrics and resource usage of the resulting system using QRML monitoring and visualization features.

References

[1] FitOptiVis team, “The FitOptiVis ECSEL project: highly efficient distributed embedded image/video processing in cyberphysical systems,” in *ACM Int’l Conf. on Computing Frontiers, CF 2019*. ACM, 2019.

[2] D. Box, *Essential COM*. Addison-Wesley Professional, 1998.

[3] “CORBA.” [Online]. Available: https://www.omg.org/spec/CORBA/

[4] “JavaBeans.” [Online]. Available: https://oracle.com/

[5] “OSGi.” [Online]. Available: https://osgi.org/

[6] T. Erl, *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall, 2005.

[7] M. Hendriks, M. Geilen, K. Goossens, R. de Jong, and T. Basten, “Interface Modeling for Quality and Resource Management,” 2020, arXiv:2002.08181 [cs.LO].

[8] “How does biometric access control work?” [Online]. Available: https://www.tedsystems.com/how-does-biometric-access-control-work/

[9] B. Kienhuis, F. Deprettere, P. van der Wolf, and K. Vissers, “The Y-chart approach,” in *Embedded processor design challenges*. Springer, 2002.

[10] M. Fowler, “Language workbenches: The killer-app for domain specific languages?” [Online]. Available: https://martinfowler.com/articles/languageWorkbench.html

[11] “Xtext.” [Online]. Available: https://www.eclipse.org/Xtext/index.html

[12] “MPS.” [Online]. Available: https://www.jetbrains.com/mps/

[13] “IVIS.” [Online]. Available: https://github.com/smartarch/ivis-core

[14] L. de Moura and N. Bjørner, “Z3: An Efficient SMT Solver,” in *Proc. TACAS*. Springer, 2008.

[15] “PlantUML: A tool to draw UML diagrams based on textual descriptions.” [Online]. Available: https://plantuml.com/

[16] “QRMLvis.” [Online]. Available: https://github.com/smartarch/qrmlvis

[17] “Grafana.” [Online]. Available: https://grafana.com/

[18] Y. Xie, J. J. Allaire, and G. Grolemund, *R markdown: The definitive guide*. CRC Press, 2018.

[19] L. Lamport, *LATEX: a document preparation system: user’s guide and reference manual*. Addison-wesley, 1994.

[20] “QRML.” [Online]. Available: https://qrml.org/

[21] H. Posadas, P. Peñil, A. Nicolás, and E. Villar, “Automatic synthesis of communication and concurrency for exploring component-based system implementations considering UML channel semantics,” *Journal of Systems Architecture*, vol. 61, no. 8, pp. 341–360, 2015.

[22] A. Basu, M. Bozga, and J. Sifakis, “Modeling heterogeneous real-time components in BIP,” in *Proc. 4th IEEE International Conference on Software Engineering and Formal Methods, SEFM ’06*, 2006.

[23] J. Eker *et al.*, “Taming heterogeneity - the Ptolemy approach,” *Proceedings of the IEEE*, vol. 91, no. 1, 2003.

[24] M. Hause *et al.*, “The SysML modelling language,” in *Proc. 15th European Systems Engineering Conference*, vol. 9, 2006, pp. 1–12.

[25] N. Medvidovic, D. S. Rosenblum, D. F. Redmiles, and J. E. Robbins, “Modeling software architectures in the Unified Modeling Language,” *ACM TOSEM*, vol. 11, no. 1, pp. 2–57, 2002.

[26] F. van den Berg, A. Remke, and B. R. Haverkort, “iDSL: Automated performance prediction and analysis of medical imaging systems,” in *European Workshop on Performance Engineering*. Springer, 2015, pp. 227–242. [Online]. Available: http://i-dsl.org/

[27] A. Hartmanns and H. Hermanns, “The Modest Toolset: An integrated environment for quantitative modelling and verification,” in *Proc. TACAS 2014*. Springer, 2014, pp. 593–598.

[28] “Kibana.” [Online]. Available: https://www.elastic.co/kibana

[29] “React.” [Online]. Available: https://reactjs.org/

[30] “Sirius.” [Online]. Available: https://www.eclipse.org/sirius/