ECSEL2017-1-737451

FitOpTiVis

From the cloud to the edge - smart IntegraTion and OPtimisation Technologies for highly efficient Image and VIdeo processing Systems

Deliverable: D2.1 Component models, abstractions, virtualization and methods

Due date of deliverable: (31-05-2019)

Actual submission date: (12-06-2019)

Start date of Project: 01 June 2018 Duration: 36 months

Responsible: CUNI

Revision: final version

Dissemination level		
PU	Public	✓
PP	Restricted to other programme participants (including the Commission Service	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (excluding the Commission Services)	

DOCUMENT INFO

Author list

Author	Company	E-mail
Marc Geilen	TUE	m.c.w.geilen@tue.nl
David Millán	ITI	dmillan@iti.es
Carlo Sau	UNICA	carlo.sau@diee.unica.it
Tomas Bures	CUNI	bures@d3s.mff.cuni.cz
Petr Hnetynka	CUNI	hnetynka@d3s.mff.cuni.cz
Vaclav Camra	CUNI	camra@d3s.mff.cuni.cz
Pablo Sánchez	UC	sanchez@teisa.unican.es
Fernando Manteca	UC	mantecaf@teisa.unican.es
Martijn Hendriks	TUE	m.hendriks@tue.nl
Pablo Chaves	SCHN	Pablo.chaves@se.com
David Pampliega	SCHN	David.pampliega@se.com
Hossein Elahi	TUE	g.elahi@tue.nl
Shayan Tabatabaei	TUE	s.tabatabaei.nikkhah@tue.nl
Freek van den Berg	TUE	f.g.b.v.d.berg@tue.nl
Twan Basten	TUE	a.a.basten@tue.nl

Document history

Document version #	Date	Change
V0.1	18-10-2018	Starting version, template
V0.2	23-4-2019	Description of UML Marte, virtualization methods and models
V0.3	5-5-2019	Description of DSL and component abstractions
V0.4	9-5-2019	Added mathematical component model

V0.5	16-5-2019	Added introduction and reference architecture
V0.6	27-5-2019	Integrated changes to DSL and component model
V0.7	29-5-209	Unified style of figures with component architecture. Fixes in DSL examples.
V0.8	30-5-2019	Incorporated updates from UC and additional fixes in figures and DSL examples
V0.9	31-5-2019	Complete version for internal review
Sign off	3-6-2019	
V1.0	7-6-2019	Final version

Document data

Editor Address data	Name: Marc Geilen Partner: TUE Address: De Zaale, Eindhoven, The Netherlands Phone: +31-402473091

Distribution list

Date	**Issue**	**E-mailer**
12-06-2019	Final	fitoptivis-wp2@lists.utu.fi
		Patrick.vandenberghe@ecsel.europe.eu

Table of Contents

1. Executive summary

This report represents Deliverable D2.1 and documents the outcomes of the activities in WP2, Tasks 2.1 (Component Abstractions) and 2.2 (Virtualization Mechanisms) of the FitOpTiVis project during the first year of the project, starting from M4. The main objective of this deliverable is to establish the first version of the *reference architecture* to be used in the activities and use cases of the project. A conceptual architecture is introduced that describes the common elements in the work developed in the project. It provides *template solutions* that require further detailing and specialization for the individual use cases and application domains.

A *component abstraction* is defined with which platform and application components developed in the project can be uniformly *modelled* in terms of a defined set of *interfaces*. The intention of this abstraction is to define the common aspects only, and to position them in a common architecture but to leave room for domain-specific refinements to be made to specialize models, architectures and methods for the individual developments in the project.

Virtualization mechanisms are introduced pertaining to the architectural concepts and the modelling of virtual resources and their abstract budgets to achieve predictable and composable application behaviour and resource reconfiguration options. The methods of implementing virtualization in hardware and/or software are subjects of WP4.

The content of this deliverable contributes to achieve MS3 (Preliminary components and methods release with standalone assessment).

2. Introduction

2.1 Overview of requirements

In this section we describe the driving requirements of FitOpTiVis on the component abstractions.

The main purpose of the FitOpTiVis component model is to define video processing pipelines out of hardware and software components and to facilitate quality and resource management for such pipelines. The goal of the component model is to allow design space exploration and run-time adaptation. As such, the **component model should allow associating configuration parameters with components** (such as supported resolution, fps, etc.) and **allow reasoning about dependencies of these configuration parameters across components** (both hardware and software) in the pipeline.

The component model is going to be applied on new components, but also on existing components where the intellectual property protection does not allow detailed modeling of the internals of the component. As such the **component model should provide hierarchical abstractions** that allow a large scale from very fine-grained components (on the level of data processing tasks) to coarse-grained components (on the level of devices with embedded software).

The component model should be used by partners in the project and other scientific and industrial users that have no extensive background in component modeling or component-based architectures, as such the **component model should be easy to use**. This in particular means that it should **provide only constructs that are needed in the project and that the semantics of the constructs should be tailored to the needs and the domain of the project**. Additionally, the **component model should have textual notation**, which simplifies sharing the models and working with the models. This is facilitated with the introduction of a DSL.

2.2 Motivation

WP2 addresses Objective 1 of the FitOpTiVis project.

> ***Objective 1****: Template solutions for: component abstractions (covering video and imaging tasks and heterogeneous processing, storage and network devices and components); virtualization supporting scalability, portability and composability principles; multi-objective quality and resource management (support for run-time decision making, adaptation, (re-) distribution and upgrades).*

Image and video pipelines will be detailed into a reference architecture and a virtual platform consisting of abstract components. The architecture and models will emphasize multi-objective optimisation including performance and energy. The use cases will be built on top of a concrete version of the reference architecture.

The use cases, component applications and platforms in the project span a wide range of technologies, methods and tools. It is not possible to build a single integrated

hardware, software and tooling framework in which all activities are integrated, nor would this be desirable, since different domains have and need their specialized models, methods, tools, and hardware, software and middleware platforms.

The range of use cases and technologies in the project share common solutions and principles that are being explored and developed in the project. Those solutions and principles can be applied across many of the separate domains. These common solutions can only be effectively identified and developed when the individual developments are positioned within a shared framework, architecture and established common models and abstractions.

In this deliverable, we establish the first iteration of such solutions and principles in the form of a reference architecture and template solutions that capture the essential concepts and the common approach. A component abstraction is introduced that characterizes the aspects of components that are deemed most important to explicitly expose in FitOpTiVis. Those aspects are their input and output streams, their provided and/or required resource budgets, the configurations they support and the aspects of quality or cost that can be optimized.

2.3 Objectives

The main goal for this deliverable may be stated as follows.

Goal: establish a common reference for component abstraction and the concept of a virtual platform. The reference architecture will be provided in the form of template solutions for a flexible virtual platform built from the component abstractions and offering multi-objective run-time optimisation support for quality and resource management.

To realize this goal, the deliverable pursues the following objectives.

- [Section 3] Provide a reference architecture for the FitOpTiVis innovations. Introduce the common conceptual elements in the image and video pipeline systems of FitOpTiVis and their inter-relations.
- [Section 4] Provide a common component abstraction that describes the main aspects of the elements from which FitOpTiVis systems are built and provide a compositional model in which components can be composed into applications, platforms and systems.
- [Section 4.5] Provide means to model multi-objective quality and resource optimization and management (support for run-time decision making, adaptation, (re-) distribution and upgrades).
- [Section 5] Provide template solutions to define abstract components (covering video and imaging tasks and devices and components) using a domain-specific language (DSL).
- [Section 6] Provide virtualization mechanisms supporting scalability, portability and composability principles.
- [Section 7] Evaluate the reference architecture and template solutions for selected domain specific approaches and systems.

2.4 Relation to Other Work Packages

This section summarizes the relations between the work in WP2 and the work reported in this deliverable and other work packages and their deliverables.

- **WP1, Requirements and validation and result analysis**
  WP1 defines the use cases that are used to validate all project results. WP2 will explicitly consider the requirements derived from the use cases, in particular, the various types of components used in the cases and the quality metrics used to evaluate them.
  Deliverables D1.1 and D1.2 provide the detailed specifications of the use cases and requirements. This includes the components, the optimization metrics and relation to project Objective 1, the use of the reference architecture.
- **WP3, Design-time support**
  WP3 will develop model-based design-time methods with concrete models that are instances of the generic component abstraction in the template solution of WP2. The concrete design methods provide, besides the devices and components that are efficient, functional, etcetera, also the necessary information as required by the WP2 component abstractions and interfaces. Deliverable D3.1 provides first versions of the design-time optimization, deployment and programming strategies that will be aligned with the reference architecture in follow up steps.
- **WP4, Run-time support**
  WP4 will implement middleware and platform components that conform to the virtualization and quality and resource management approach (developed in WP2) of the FitOpTiVis reference architecture as instances of the template solution for run-time management.
  Deliverable D4.1 provides preliminary run-time models and support for energy, performance and other qualities. In subsequent steps, monitoring techniques will be used to provide an online view of the system status from the perspective of the reference architecture and its component model, e.g., the set points in which components are operating, their quality metrics, virtual platforms and budget allocations.
- **WP5, Devices and components**
  WP5 will develop high-performance, energy-efficient processing and communication devices and components that conform to the reference architecture. Their development specifically considers the key aspects that characterize components in the component model, reconfiguration, qualities, inputs, outputs and budgets. Human and machine-readable descriptions of the components are made using the DSL (Domain Specification Language) introduced in this deliverable that enables automated processing of the components at design-time or run-time.
  Deliverable 5.1 Components Analysis and Specification presents an analysis of the state-of-the-art of existing components for computation and networking. The inventory will be used to validate the proposed component abstractions to see if the properties and configurations of such components can be (accurately) modelled and the construction of systems from such components can be compositionally determined from the component models.

- **WP6, Use Cases and demonstrators**
  The template solutions of the FitOpTiVis reference architecture developed in WP2 will be used in the demonstrators planned in WP6 for the project use cases. Domain-specific and or use-case specific solutions will be shown, but they will be shown to be instances of a common architecture and to exhibit common principles and solutions.

2.5 Overview of the Document

The remainder of the deliverable is structured as follows. Section 3 introduces the reference architecture and the proposed solutions for the FitOpTiVis project, including the way it is envisioned to present template solutions. Section 4 presents the component abstractions that are used in the project to characterise the various platform and application components with which the project activities will be dealing. It includes a discussion of the literature, a conceptual model and a mathematical model that allows a precise abstraction and a framework to express multi-objective optimization goals. Section 5 defines the domain specific language for component models. (A full definition is given in Appendix A.) Section 6 discusses the conceptual modelling virtualization mechanisms, virtual platforms and quality and resource management architecture. Section 7 shows initial modelling efforts to validate the reference architecture and component abstractions currently defined.

3. Reference Architecture

3.1 Proposed Solution

The FitOpTiVis project considers a wide diversity of platforms, resources, applications, methods, tools and objectives. One could not aim for one solution to fit all needs and concerns simultaneously. It is important that commonalities between the approaches and use cases are found and exploited such that reusable results can be achieved and can be applied across different domains and use cases.

The FitOpTiVis project addresses the concern of diversity by the definition of a *reference architecture* that captures what is common to the use case of the project and to the domain of image or video pipelines for CPS on a heterogeneous network spanning the cloud to the edge. Individual use cases and individual developments in the project will consider their specialized version and additional detail, but they will be positioned with respect to the framework of the reference architecture to identify common problems and solutions and to ensure consistency and they can leverage the framework.

The *proposed solution* includes the following ingredients.

- A *reference architecture* is defined that captures the common assumptions and approaches in the project. The use cases will be positioned in terms of the elements of the reference architecture.
- A common *component abstraction* is defined that represents a shared abstract view of platform and applications components establishing their properties. It includes an abstract component model and compositions that build system models. The model focuses on aspects of data streams (re)configuration, virtualization, heterogeneity, resource sharing and (multi-objective) quality. It is anticipated that specialized application domains in the project have their own domain-specific refinements in the form of more detailed or additional models, but that these are consistent with the architecture.
- As the detailed models are domain-specific, likewise, the various solutions developed in the project will be diverse and domain-specific, whether they are run-time methods, design-time method, virtualization techniques, etcetera. The project intends to identify common solutions and general patterns of solution strategies in the form of *template solutions*, which are reusable elements amenable to be applied in different contexts.

The project defines a reference architecture as a common reference for component abstraction and the concept of a virtual platform. The overview of the reference architecture is visualized in Figure 1. More precise definitions are given in Section 4.3—4.5.

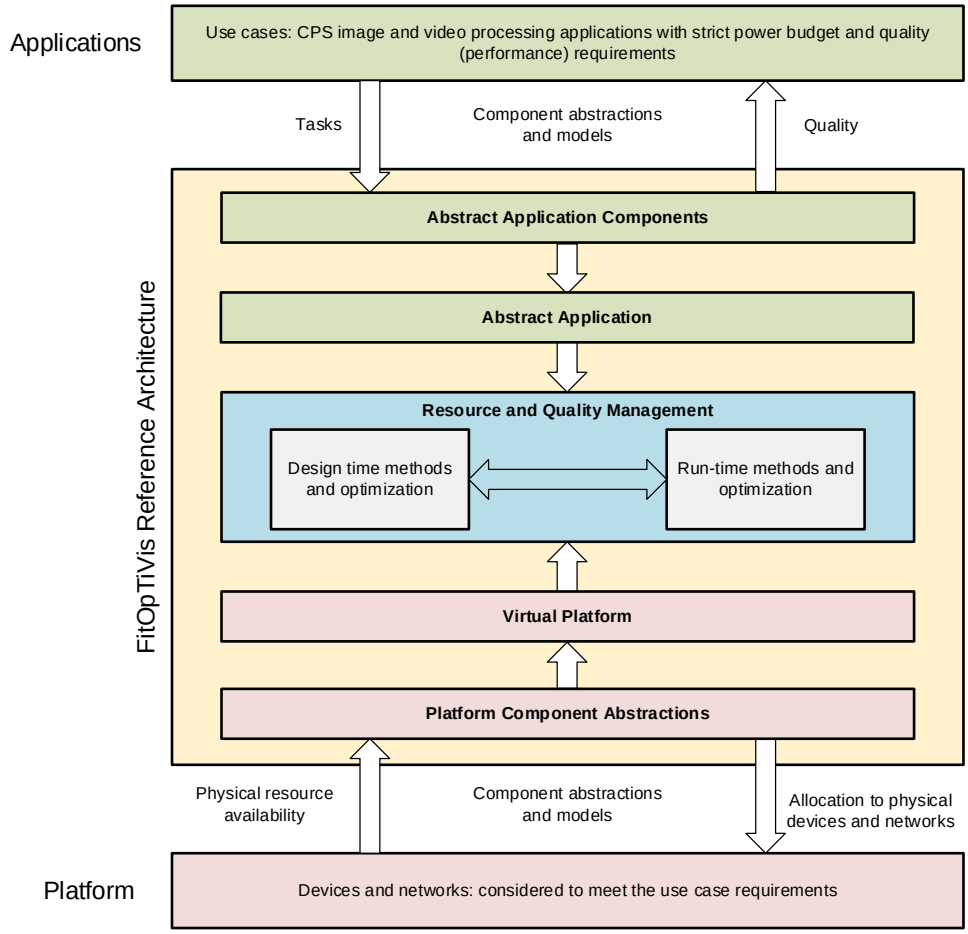


Figure 1: FitOpTiVis Reference Architecture

The top side shows concrete applications and the bottom side the concrete platform of a system under consideration. The reference architecture is concentrated in the yellow box between them. It deals with abstractions of applications and platform. Applications are considered as a collection of tasks. Applications can be potentially configurable at design-time reconfigurable at run-time. They are considered to have alternative configurations or set points that are explicit in the architecture model. We do not insist that all applications are (re)configurable. Some applications may have a trivial set of configurations, or just a single possible configuration. In general, configurations are characterized by different workloads on the execution platform and different quality provisions to the user of the application (for example, power consumption or latency). Different configurations are associated with a set of application parameters. In the framework, application tasks are modelled as application components, and the collection of applications components of an application are referred to as an *abstract application*.

The platform (bottom) side is abstracted as a virtual platform. It is assumed to be (re)configurable and resources can, in general, be shared by different applications. This does not mean that in all possible instances resources shall be shared by multiple applications. Resource virtualization is a particular emphasis of the FitOpTiVis project

and the architecture likewise assumes that resources provide virtual resources to applications in the form of a *virtual execution platform*.

The abstract applications and the abstract virtual platforms are brought together by a resource and quality management framework. This framework is responsible for finding feasible and optimal combinations of applications and platforms, and for finding the optimal set points of the components. The combinations need to match, which requires both application models and platform models have appropriate models of their resource requirements and resource provisions, respectively. As illustrated in the figure, part of this optimization work is done at design-time when components are designed and developed, and part of this work may be done at run-time when more information may be available, but less time to take and enforce any decisions.

We anticipate that large differences may exist in how the resource and quality management is implemented and executed in different domains, but the expected common approaches are captured in the architecture.

3.2 Template Solutions

The project intends to identify common solutions and general patterns of solution strategies in the form of template solutions, which are reusable elements amenable to be applied in different contexts. We identify what solutions are investigated and formulate initial concepts in this deliverable. They will be further detailed and evaluated in the project.

The project will specifically pursue the following classes of template solutions.

- The common conceptual model of the *component abstraction* itself. It is introduced in detail in Section 4. It defines what elements are commonly expected to be defined and how they are related. It also defines how components are composed into systems and how requirements on compositions are expressed, for instance the satisfaction of requirements, compatibility of inputs and outputs, and matching provision and requirements of resource budgets.
- A *domain-specific language* (DSL) that provides a human and machine-readable version of the abstract component models, their compositions and quality- and resource management requirements and objectives. The language supports the evaluation of completeness, uniformity and consistency of the many specific models that are made in the project. It also allows automation and tool support for common analysis and synthesis techniques (such as visualization or code generation) or model transformations. Detailed components in the project should be supplemented with a manifest description in this DSL. An initial version of this DSL is introduced in Section 5.
  It is anticipated that some of the specific application domains and use cases in the project will develop their own, specific refinements of the DSL in a ‘domain-specific DSL’ or ‘DSDSL’.
- A precise, *semantics of the component abstraction* is given in terms of a mathematical description of components and their composition operators. The composed system is provided with a semantics in terms of the constraints on the combined collection of configuration parameters and matching inputs and outputs, and well-defined multi-objective optimization objectives based on

ordering of budgets and qualities. The precise details of composition operators are expected to be domain specific. The overall problem of configuration, mapping and selection of set points can be captured in terms of a multi-objective constrained optimization problem. Conceptually and possibly in prototype experiments, generic constraint solving algorithms may be applied to find optimal solutions. It is expected however, that such solutions are insufficient in practice and domain-specific solutions and heuristics should be employed to determine good solutions in practice. The semantics is elaborated in Section 4.

- A *virtual platform model* that defines how resources and their sharing are modelled through virtual resources that are provided by virtualization mechanisms in their implementation. This allows, ideally, resources to be shared by multiple applications while providing well-defined resource budgets, to the individual applications and providing support for (re)configuration of virtual resources and their resource budgets. The virtual platform model is elaborated in Section 6.3.
- A *quality and resource management* (QRM) architecture describes how the required information, activities and responsibilities, such as optimization, monitoring, configuration, calibration, resource management, may be divided between different elements of the architecture and between design-time and run-time activities. The QRM architecture is introduced in Section 6.4.

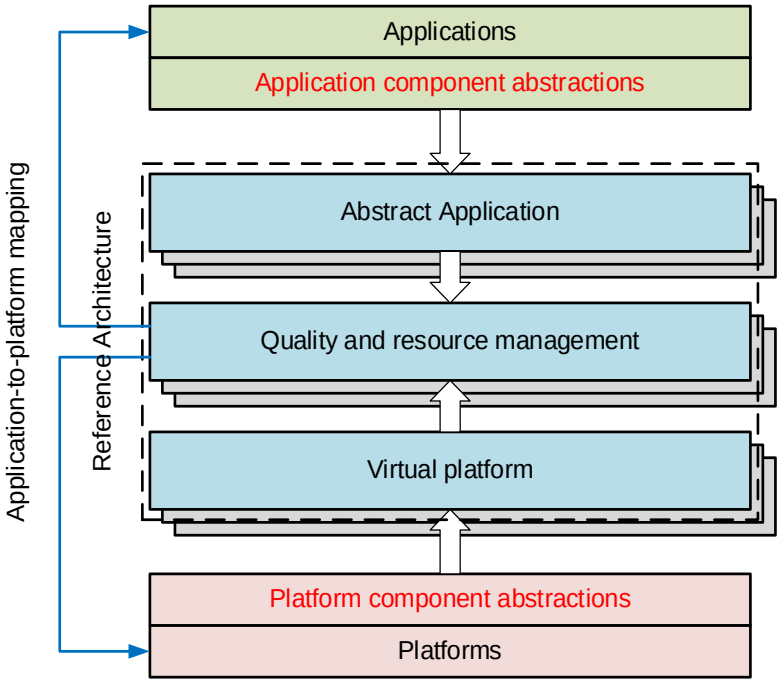


Figure 2: FitOpTiVis Template Solutions

Figure 2 illustrates how the reference architecture and template solutions are instantiated in different, more refined incarnations for different application domains. The definition of the reference architecture facilitates the relation to the work in other work packages as follows.

- WP3 develops *model-based design-time methods* with concrete models (grey) that are consistent with the generic component abstraction of the template solution (blue) of WP2. Ideally tool chains produce descriptions of the abstract component models automatically.
- WP4 creates *run-time management* solutions (grey) that are in line with the template quality and resource management (blue) laid down in WP2.
- WP5 creates *(re)configurable devices and components* (grey) and their component abstractions that are instances of the template solution (blue) of WP2
- WP6 demonstrates that detailed solutions in the 10 use cases.

The reference architecture and its template solutions are not expected to be directly used on any specific design problems or in any specific domain, use case or demonstrator. Instead, we expect the architecture and templates to be instantiated and specialized for a particular domain.

Specialized component models may be used that best characterize the application and or platform components that are common in a particular domain, such as timed dataflow models for real-time streaming data processing, or UML state diagrams for component-based, control-oriented software components.

Similarly, it is expected that different domains employ their own, specialized budget descriptions, specialized composition operators, specialized specification languages, mapping strategies, optimization strategies, and so forth.

Also, every domain typically has its own favoured analysis, and design-space exploration tools and methods and synthesis strategies.

We expect that the different use cases in the FitOpTiVis project will each use such a specialization of the architecture and solutions, but that they will all respect the overall architecture, which means in particular that they will follow the component model outlined in Section 4 and use the common DSL of Section 5, or a specialization thereof to model the system components. The applicability of the architecture to the various use cases should serve as a validation of the core concepts in the architecture.

4. Component Abstractions

In this section, we overview the basic component abstractions used in FitOpTiVis. We present the conceptual model and we introduce a mathematical model.

Though in the software engineering community and literature the "component" may refer to many different things, it is typically used in the sense of "a constituent part" of a system. For example, the UML 2 defines "component" as an entity with the following properties [OMG2017b]:

- *A Component represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment.*
- *A Component is a self-contained unit that encapsulates the state and behaviour.*
- *A Component specifies a formal contract of the services that it provides to its clients and those that it requires from other Components or services in the system in terms of its provided and required Interfaces. As such, a component serves as a type, whose conformance is defined by these provided and required interfaces (encompassing both their static as well as dynamic semantics).*
- *A Component is a substitutable unit that can be replaced at design-time or run-time by a Component that offers equivalent functionality based on compatibility of its Interfaces. As long as the environment is fully compatible with the provided and required Interfaces of a Component, it will be able to interact with this environment.*
- *A Component has an external view (or "black-box" view) by means of its publicly visible Properties and Operations.*
- *A Component also has an internal view (or "white-box" view) by means of its private Properties and realizing Classifiers* – i.e. internal architecture typically consisting of internal composition of components.

In FitOpTiVis, we follow this generally accepted view of a component and see a component as an abstraction of a hardware/software subsystem. We also identify important quality properties that are relevant to the FitOpTiVis subject matter of *quality and resource management* and make it possible to attach them to components or their constituents (interfaces in particular). This makes it possible to:

- reason, at *design-time*, about a system as of a composition of components and their configurations. This makes it possible to predict the overall properties of the system before the system is actually built.
- to relate, at *run-time*, monitored properties of a system to its constituents and thus to reason, still at run-time, about the system and to be able to adapt some of its parameters.

Compared to the traditional software engineering view of a component, the important distinction in FitOpTiVis is that the project takes a systems view, where a component can be realized by hardware, software or both. This makes it possible to describe a large span of options ranging from DSPs, FPGAs, to processing performed by GPU-accelerated cloud VMs (Virtual Machines) and containers.

Another important distinction of FitOpTiVis is its focus on quality and resource management. Related to this and to the fact that the hardware and software components

can be combined is that it provides both (1) a data processing workflow view (i.e., the steps in the video processing pipeline) and (2) the deployment architecture (i.e., that a software component runs on a particular hardware component).

In order to perform the design-time exploration and run-time adaptation, the component abstractions in FitOpTiVis need to allow for specification of configuration options and quality properties (along with budgets and costs).

The component abstractions are traditionally represented by a component model, which defines the component structure and meta-data and the component composition mechanism. In this section, we provide such a component model – called the *FitOpTiVis component model*. In Section 5, we then describe the textual representation (i.e., a domain specific language – DSL) that we developed for defining the components.

4.1 State-of-the-Art and Related Work

In this section we analyze several existing related components models on how they approach the FitOpTiVis requirements on the component abstractions. Based on this analysis, we build the component abstractions and bring them together as the FitOpTiVis component model and the DSL to describe them (as described further in this section and Section 5).

4.1.1 SysML

The Systems Modeling Language (SysML)[1] [OMG2017a] is a dialect of the Unified Modeling Language (UML) [OMG2017b]. Since its origins it has evolved into a standard for the Model-Based Systems Engineering (MBSE) applications. As such, it aims to unify all the various documents that are created during different stages of the software engineering process into a single document used by architects, developers, domain experts and maintainers alike. Therefore, it contains structures to allow all these various groups to express their view on the system.

Compared to UML, SysML removes Activity diagrams, Block definition diagrams and Internal block diagrams. On the other hand, it adds the Requirement diagrams, which provide modeling constructs for text-based requirements, and the parametric diagrams, which describe constraints among the properties associated with blocks. The parametric diagrams could be used to express various constraints, relations of qualities and the configuration parameters of described components. Importantly, the SysML allows description of both the software and hardware components and allocation of the former ones to the latter ones.

In relation to requirements of FitOpTiVis, SysML provides means for describing most of the required parts (components, their composition, etc.). However, having a significantly broader scope, **it is cluttered with many concepts that are not necessary for the FitOpTiVis objectives**, which makes it very difficult to use by partners. Similarly, the **lack of precise semantics** makes it difficult to directly use SysML for automatic design

[1] https://sysml.org/

space exploration and for run-time adaptation. Also, SysML **primarily uses graphical notations only** and the textual representation, which is based on XMI (XML Metadata Interchange), is intended for serialization only. This further decreases flexibility and leanness in design as opposed to the DSL descriptions we intend to use.

4.1.2 IEC 61131

The IEC 61131[2] is a microcontroller architecture standard made by the International Electrotechnical Commission. It has support for CPU instructions, functions (made from CPU instructions), sensor and actuator interfaces, human-machine interfaces, power-supply interfaces, and communication interfaces.

For our use case, it provides inspiration for components, and for specifying inputs and outputs. It has **no support for specification of configurable component quality properties** that could be used in design exploration and run-time adaptation of quality and resource aspects. Also, it does **not explicitly support hierarchical composition**.

4.1.3 IEC 61499

The IEC 61499[3] standard is an extension of the IEC61131 (Section 4.2.2) standard for distributed industrial automation systems. It adds support for event-based processing and composition of function blocks.

Like the IEC 61131 **it lacks support for component configurations and quality properties** needed for design exploration and run-time adaptation. Unlike IEC 61131, it does allow for the composition of function blocks.

4.1.4 AADL

The Architecture Analysis & Design Language (AADL)[4] [FGH2006] has been introduced in 2004 by Society of Automotive Engineers as a modeling language for model-based description and analysis of complex systems in terms of interactions of components. The AADL language does not limit design description to software components, but it covers also description of computational platform elements (e.g., processor or memory) and mapping of software components to hardware.

In AADL, components are divided into three main categories: (i) Application components, which are software components such as processes, threads, subprograms, (ii) Execution platform components, which are hardware components (e.g., processor, memory), and finally (iii) Composite components (also called systems) composing other components together (both hardware and software). Component interfaces (also called features) can be of several types – for data and event passing, method calls, and direct data access. Application components also have properties that

[2] https://webstore.iec.ch/publication/62427

[3] https://webstore.iec.ch/publication/5506

[4] http://www.aadl.info/

specify them, e.g., timing properties or constrains for binding executable threads to processors. AADL offers support for modes and switching among them. A mode in AADL is a distinct configuration of a component. Mode transitions are controlled by a state machine and enabled by events defined in behavior of components. Modes can specify different configurations of component composition, different call sequences, and multiple properties of components.

AADL supports many of the features required in our project – either directly or they can be modeled in terms of existing features. However, for our needs, AADL is **too complex** and **requires rather in-depth knowledge of component concepts** and would be **hard to use by non-expert users**.

4.1.5 Koala

Koala [OLK+2002] is a component framework developed by Philips and targets consumer electronics. The primary goal of Koala is to easily manage the complexity of embedded software used in consumer electronics and to handle the large diversity of such devices. The component model of Koala is heavily inspired by Microsoft's COM and Darwin [MK1996] component models. Koala offers hierarchically composed components and in addition to the primitive and composite components Koala also defines modules. A module is a basic compositional unit and from an implementation view, it corresponds to a single source code file. To handle diversity of devices, Koala offers diversity interfaces and switches. The diversity interface is a required interface intended for configuration, i.e., setting parameters. A switch is a module connecting several components together and its functionality is controlled through the diversity interface. Based on the values of the diversity interface parameters, the switch chooses which components are effectively connected.

For our needs, Koala does **not support description of hardware** components. The **quality properties along with budgets and costs would be very hard to model**.

4.1.6 ProCOM

The ProCOM [SVB+2008] component model distinguishes two levels of granularity – ProSys and ProSave. ProSave, the lower layer, operates with low-level passive (i.e., cannot initialize a new thread) and hierarchically structured components. Computation on this level is based on the pipes-and-filters paradigm; the functionality of the ProSave component is described as a set of services. The communication between components is realized by data ports (for passing data) and triggering ports (for passing signals). Each service contains one input port group and several output port groups. ProSys, the upper layer, describes a set of concurrent components, which are called subsystems in order to distinguish them from the lower-level ProSave components. These subsystems can run potentially on several computation hardware nodes. A ProSys subsystem is composed of a set of concurrent functionalities that can be either event driven or periodic. The only way for ProSys subsystems to communicate with each other is by sending asynchronous messages via channels. Channels are strongly typed and support multiple senders and receivers. A ProSys subsystem may be modeled as an assembly of ProSave components but can be also implemented directly or as a composition of other ProSys components. Behavior of components is formally specified by a formalism based on finite state machines.

As the Koala above, the ProCOM does **not model the hardware** components and also the **quality and budget properties would be very hard to model**.

4.1.7 Fractal (THINK and MIND)

The Fractal [BCL+2006] component model is a classical component model with hierarchically composed components. By itself, the Fractal is only an abstract specification and there exist multiple implementations targeting different domains. For our needs, the most related implementations are THINK [FSL+2002] and MIND[5], which both of them target development of embedded systems. As they are implementations of Fractal, the components are defined by their provided and required interfaces and they can be hierarchically nested. In addition to business interfaces, the components provide control interfaces via which it is possible to manage component lifecycle, configure components, etc.

The main difference to the THINK is that MIND supports for different hardware platforms explicitly expressed using descriptors. As in case of the component models above, Fractal also **lacks good support for component configurations and modelling quality and budget properties**.

4.1.8 SOFA 2 and SOFA-HI

SOFA HI [HBP+2009] is a profile of the SOFA 2 component framework [BHP2006] for development of high-integrity real-time embedded systems. SOFA 2 has a very similar set of features as Fractal, i.e., there are hierarchical components with provided and required interfaces. In SOFA 2, the components have explicitly defined their interface and implementation. Also, the connections among components are modeled via connectors, which are considered as first-class entities. SOFA 2 allows for modelling dynamic architectures (via reconfiguration patterns); SOFA HI restricts dynamism to mode switching.

For our needs, SOFA 2 does **not model explicitly the hardware** components and as above, **quality and budget properties cannot be easily modelled**.

4.1.9 BlueArX

BlueArX [KRS+2009] is a component framework developed and used by Bosch. It is intended for use in automotive domain, especially in embedded devices. BlueArX focuses on the design-time component model to support constrained domains considering various non-functional requirements while providing multiple views of a developed system. BlueArX uses a common hierarchical component model. The static view defines two types of components, an atomic component, which has an implementation, and a structural component, which are composed of other atomic and/or structural components. Components have interfaces dividable into two types – import and export interfaces. Connections between interfaces are implicit based on the interface names. These connections are implemented using a special type of variables

[5] http://mind.ow2.org/

called messages; a component specifies its message access properties in its interface description. The dynamic view consists of component scheduling specification, which contains mapping of services to periodic or event-triggered tasks and the order of services inside these tasks.

For our needs, the BlueArX does **not model hardware** components and also its **implementation is not available**.

4.1.10 AUTOSAR

AUTOSAR[6] is a software architecture for development of automotive electronic control units (ECUs). It defines composable components with explicitly defined interfaces, properties, configuration and adaptation management, etc.

For our needs, it supports (directly or indirectly) many of the required features. However, it is so closely tied to the automotive domain that it is **not easily applicable to another domain**.

4.1.11 UML-MARTE

As it has been commented in the SysML section, UML lacks the specific semantics required to fully support specification, modelling and design of current electronic embedded systems. The embedded system models need to reflect systems integrating multiple applications and diverse software platform components, e.g., embedded RTOS, middleware, drivers, etc. Similarly, current hardware architectures rely on multi-core processors, surrounded by many hardware devices for communication, storage, sensing, and actuation. In addition, several types of analysis are applied (e.g., schedulability, timed-simulations, etc.) which require to add additional information to the model, e.g., annotations of extra-functional properties related to timing, memory sizes, energy, etc. In this context, the standard MARTE profile was developed [OMG2018] to model and analyse real-time embedded systems, providing the concepts needed to describe real-time features that specify the semantics of this kind of systems at different abstraction levels.

For the needs of high-level specification MARTE is, however, **too complex**. It also **lacks simple textual notation** that would allow easy sharing of models. As such, we do not use MARTE as the first-line language, but rather, in one of the specific application domains, as a more detailed model to which specifications of some FitOpTiVis component model are translated to (see Section 7.3).

4.1.12 Mathematical Component models

4.1.12.1 Behaviour Interaction Priority BIP

To modelling heterogeneous real-time components, the BIP (Behaviour, Interaction, Priority) framework has been introduced in [BBS2006]. The lower level describes the behaviour of a component. The middle layer addresses the interaction between

[6] https://www.autosar.org/

components. The top-level describes the scheduling information. In BIP, systems are constructed from atomic components, which are finite-state automata extended with data and ports. Data transfer is the means of interaction between components. The algebra for this interaction is presented in [BS2008].

The first version of BIP systems is static, which means that components and interactions between them are fixed at design-time. To address dynamism of a real-time system, Dy-BIP, representing a dynamic extension of the BIP framework, is introduced in [BJM+2012]. Dy-BIP offers primitives to model dynamic architectures. Transition systems are used as the atomic primitives. Transitions are labelled with ports, action names, and constraints for interaction with other components. Each atomic component provides its own interaction constraints at each computation step.

The next generation of BIP is the Dynamic Reconfigurable BIP (DR-BIP) component framework capturing three types of dynamic changes, namely, different configurations of a component, creation, and deletion of components, migrations of components between predefined architectures. The formal definition of this model can be found in [BBB+2018].

The BIP model **only considers applications as components**. Therefore, it is not able to address dynamism on the hardware side. BIP focuses primarily on **functional behaviour and interaction rather than resource usage and aspects of quality**.

4.1.12.2 Contract-based frameworks

For simplicity, modularity and scalability, the design and verification should be performed at the component level. The correctness of component behaviour may depend on the behaviour of components with which it interacts. This method is referred to as *contract-based design*, because for decomposing systems into components it makes assumptions on the environment and in turn provides guarantees to the environment [CGP2008].

In their terminology, horizontal contracts are those for components at the same level of abstraction, representing different components of the system, while vertical contracts span different levels of abstraction of the same components [NSS+2012]. In contrast, the vertical relations in our terminology (see Section 4.4) refer to resource budgets between application and platform components, while horizontal composition to the exchange of data between components.

In the contract-based framework, each component has some implementations defining the behaviour of that component. These are usually deterministic and do not limit the environment. These are the main differences between contract (their component abstraction) and the implementation of a component. The guarantees of the model must be realized by an implementation. Components with contracts have an elegant compositional semantics in terms of sets of behaviours [NS2018]. When a contract guarantees more with fewer assumptions than another contract, the former is called a *refinement* of the latter. It may substitute for the former in any situation without violating any constraints. Similarly, we may consider the introduction of abstraction and refinement relations between component abstractions, for example if a component provides better qualities for less resource usage.

The contract-based frameworks provide a very generic framework but **lack the concrete syntax and support for specific constraints and compositions**. There is a **lack of tools supporting contract-based design**, except for the domain of formal verification. The contract-based frameworks do **not address dynamism** typical of reconfigurable and adaptive systems.

4.1.12.3 Multi-objective optimization techniques

In many fields of study such as business, economics and engineering, one often considers two or more objectives for optimization (for example, latency and cost minimization) along with constraints on such metrics. Constraints are non-negotiable limits on some properties, while objectives are the negotiable properties. For instance, we might have a constraint on the minimum frame rate of a streaming application and would like to trade off power consumption for latency.

To find optimal solutions to multi-objective problems, one often considers *Pareto optimality*, named after economist Vilfredo Pareto. He identified solutions that helped some people (some objective metrics) without hurting anyone else (other objective metrics) [P1971]. In FitOpTiVis, we intend to use the concept of Pareto optimality and additionally explore the use of the algebraic framework introduced in [GBTO2007] for compositional reasoning about optimality of systems of components. This approach allows us to describe the design decisions for composing components either by connecting inputs and output or by matching provided and required resource budgets. The mathematical component framework for quality and resource management, introduced in Section 4.5, shows how this algebra provides functionality for component abstractions, in more detail.

4.2 Basic terminology and definitions

Building on the analysis of the related work and on the experience with developing and extending various component models (ProCOM [SVB+2008], SOFA 2 [BHP2006], SOFA-HI [HBP+2009], Fractal [BCL+2006], DEECo [BGH+2013]), we define the basic abstractions of the FitOpTiVis component model as follows.

A **component** in the FitOpTiVis component model is the primary constituent of a system. A component can be a hardware component (e.g., a camera or a processing unit), a software component (e.g., a functional unit or a driver) or both (e.g., a smart camera). Components can have associated configuration parameters and can be composed together to form an architecture.

On the finest level of granularity, components can be divided to **platform components** and **application components**.

Platform components represent parts that are generic with respect to a particular application use. They provide computation means to execute actual data processing tasks. Examples of platform components include: Raspberry Pi board with Raspbian, Openstack node, a particular VM, FPGA.

Application components on the other hand represent the computation task specific to an application. Examples include: OpenCV-based routine, Docker image, routine on FPGA.

The relation between platform and application components is that an application component is hosted (runs on) on a platform component. In some cases, the same component can be both application and platform – e.g., a virtual machine component is hosted (as an application component) on an OpenStack node platform component but itself acts as a platform component for applications hosted on the virtual machine.

Application components can be connected to create data processing pipelines. Similarly, an application component may be connected to a platform component to signify the application component is hosted on the platform components.

The connection between components is realized through **bindings** between **component ports**.

Components can be **composed** to form larger components, e.g., **applications** or (virtual) **execution platforms**. In this respect, an **abstract application** is a composition of application component abstractions that provides functionality to a user. A **virtual execution platform** is a composition of virtual platform components that can run an application. An abstract application executes on a virtual execution platform, or virtual platform; a virtual platform exposes a collection of resource budgets to the abstract application. Abstract applications and virtual platforms collect all information needed for **quality and resource management**.

Components may have multiple **configurations**. A component **configuration** consists of configuration parameters (set points) that control characteristic properties of the components called **qualities**. Examples of configuration parameters include: fps, video frame resolution. Examples of qualities include: memory consumption, code size, processing speed. There are **trade-offs** between configurations and qualities – e.g., bigger video frame resolution requires more memory and leads to lower frame rate, an Openstack node can host VMs with 8GB of memory or up to twice as much VMs with 4GB of memory.

The configuration of a components (or some of its parameters) may be set at design-time (in case the configuration leads to recompilation of the component or reinstall of a component) or at run-time. The parameters that are **(re)configurable** at run-time are set via a dedicated run-time interface as developed in WP4.

An example of these component concepts is given below. We assume a smart camera component. As a black-box, this component combines both hardware and software in one package. Table 1 lists different configurations (rows). Each configuration is described by a combination of particular choice of configuration parameters (columns).

Table 1: Smart camera component configurations.

Mode	Frame rate (fps)	Biometric parameters	Faces	Raw frames
1	1	+	-	-
2	1	+	+	-
3	1	+	+	+

4	10	-	+	+
5	10	-	-	+
6	30	-	-	+

These example configuration options describe that the camera has selectable frame rate (1, 10, and 30 fps) and can detect biometric parameters and faces. It can also provide the entire video frame. However, only certain combinations of these parameters are possible. If for instance biometric parameters are to be provided by the camera, the frame rate is fixed to 1 reading per second.

When face detection is requested, the camera can process the video with up to 10 fps. The maximum frame rate of 30 fps is achievable only when both the detection of biometric parameters and faces is disabled.

In addition to these configuration parameters, the camera may consume different amounts of energy. Such energy consumption can be viewed as a quality parameter of the component. We assume that considering the provided features, frame rates and energy consumption, all its configurations are Pareto optimal.

For the smart camera example, frame rate, biometric parameters, faces and raw frames output are the configurable parameters that determine its set points. Frame rate and power consumption are considered its qualities.

This example modeled using our component abstractions is shown in Figure 3. The architecture of the example contains two application components – *sensor task* and *control host task* – and two platform components – *smart camera* and *cloud compute platform*. The components are composed together via two principal types of bindings – “provides data to” and “runs on”. In this example, the *sensor task* sends data from face recognition to *access control task*, which grants/denies physical access to identified persons. As the two tasks are application components, they need to be executed somewhere. The execution happens via a platform component. The composition of an application component with a platform component happens through the “runs on” binding. In the example below, the *sensor task* component runs on the *smart camera*, the *access control host task* runs on *cloud compute platform*.

Given the typical way of laying out the application components above the corresponding platform components, we also term the “provides data to” composition as *horizontal composition* and the “runs on” composition as *vertical composition*.

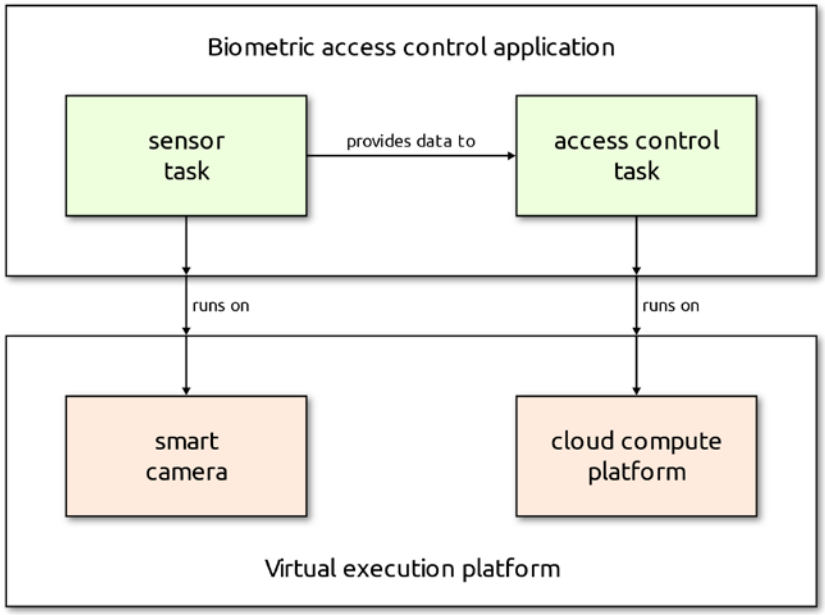


Figure 3: Example component model architecture of a biometric access application.

Application and platform components can be made at various granularities and / or hierarchically. For instance, the sensor task and the access control task can be abstracted together as the Biometric access control application. This application is in this case the top-level application component. In the same way, the smart camera and the cloud compute platform components can be abstracted as the Virtual execution platform, which is the top-level platform component.

Similarly, if needed a component may be further decomposed to an architecture of fine-level components. For instance, the sensor task can be decomposed to a pipeline of 4 tasks as shown in Figure 4. Note that as the sensor task itself has two ports (one for providing data to access control task and another for being hosted on the smart camera). The sub-components inside the Sensor task delegate to these ports on the outer boundary.

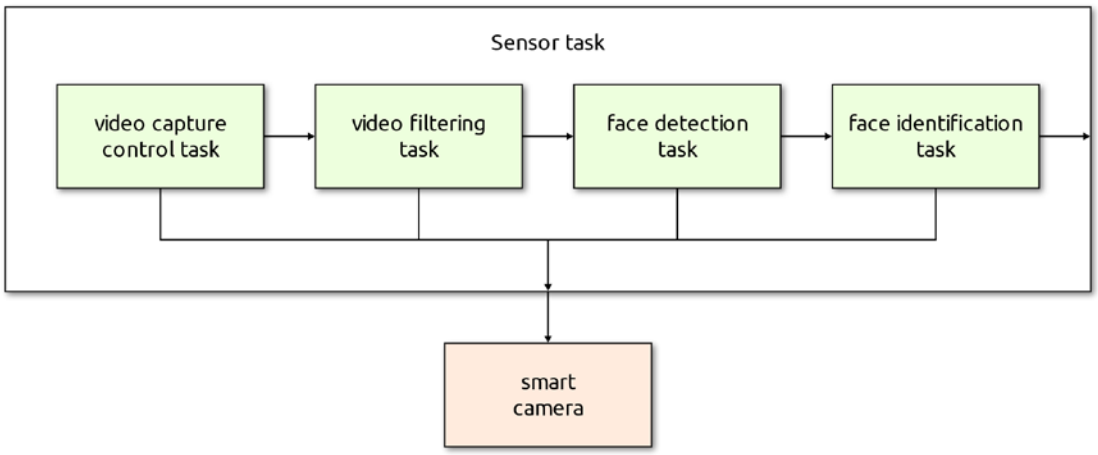


Figure 4: Example of the biometric access application with decomposed sensor tasks.

Components may also include any number of vertical layers as shown in Figure 5. The example depicts a pipeline that does part of the video processing in the cloud. Here the presence of the cloud creates two layers of platform components as the container/VM instance is virtualized on top of the cloud.

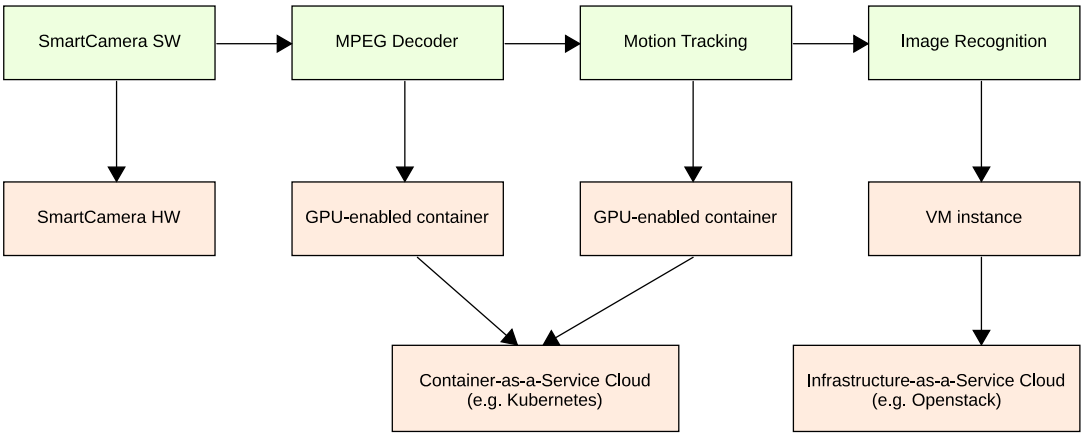


Figure 5: Example of a model with multiple platform layers.

4.3 Detailed description of the reference architecture model

The main constituent of the FitOpTiVis component model is the component. The component has a black-box and white-box view.

4.3.1 Black-box view

In the black-box view, the component exhibits multiple ports as shown in Figure 6. Every component provides the following six types of ports: *supports*, *requires*, *inputs*, *outputs*, *parameters* and *qualities*.

To graphically distinguish the nature of the ports, we exploit the graphical notation of UML 2 Component Diagrams.

We denote the *supports* port as vertically facing port terminated by a lollipop (circle). The *supports* port is the platform-component part of the composition between a platform and application component. It abstracts the provided resource budget.

The *requires* port is denoted by vertically facing port terminated by a socket (half-circle). The *requires* port is the application-component part of the composition between a platform and application component. The requires ports abstracts the required budget. It serves as an abstraction for settings of configuration parameters and reflecting qualities and costs in the application component.

The *inputs* port is denoted by horizontally facing port terminated by a socket. It represents the intake of data (typically a video stream).

The *outputs* port is denoted by a horizontally facing port terminated by a lollipop. This is a counterpart of the inputs port. It represents the egress of data (typically a video stream).

The *parameters* port exhibits the configuration parameters of the platform component. They are the parameters that can be set to determine the configuration in which the component operates.

The *qualities* port exposes the relevant qualities/costs of the platform component. It determines what aspects of quality and aspects of cost, which may vary across the different configurations, are exposed to be used to express constraints and requirements for quality and resource optimization purposes.

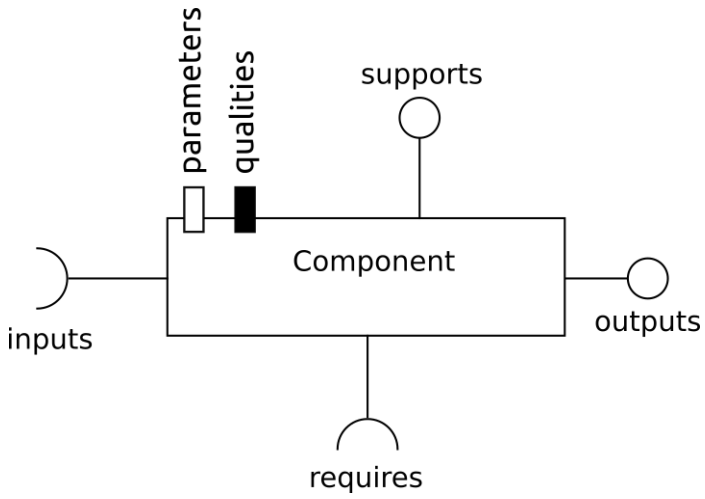


Figure 6 Ports of the components

Components can be connected together to form architectures. In this composition, only requires-supports and inputs-outputs connections can be formed, as shown in the figures below.

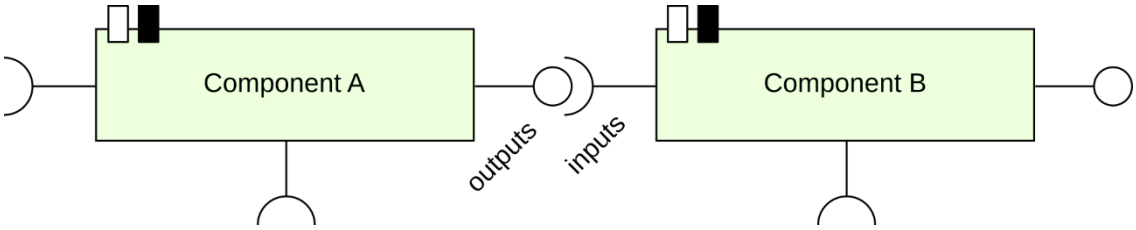


Figure 7: Horizontal composition by connecting inputs and outputs

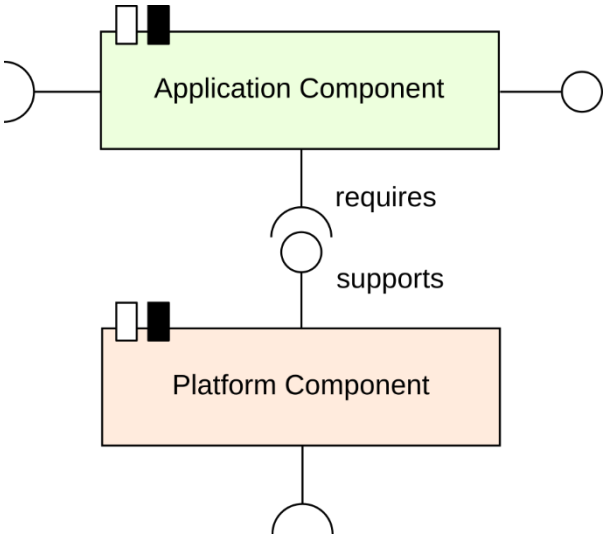


Figure 8: Vertical composition by connecting requires and supports ports

The ports further need to be compatible, e.g., video streams have to be of the same encoding, need to be related in terms of frame rate or resolution.

Similarly, when application components and platform components are composed, the budget supported by the platform component should match with the budget required by the application component. This does not necessarily mean that they need to be equal.

Budgets are not necessarily quantitative (scalar numbers) but may include diverse aspects and can be defined at different levels of abstraction.

- A budget may reflect the availability of a feature (e.g., security).
- A budget may include a level of guarantee (e.g., hard real-time vs soft real-time).
- Platform components may enforce budget restrictions on an application component, or monitor if an application stays within its budget.

To reflect this formally in the component model, each port is associated with an interface type. The interface type may define a number of properties that further characterize the contract between two interconnected components. The properties reflect the budgets, quality metrics, resource costs and configurations. The properties may be of different data types – numeric, Boolean, discrete.

When components are connected, a relation is established between properties of components. The connection can be made only between compatible ports. From this

perspective, two ports are compatible if they have compatible interface types and values properties (on the two respective ports) are also compatible. Trivially, if the ports have the same interface type and have the same property values, they are compatible.

4.3.2 White-box view

The white-box view allows modeling the internal structure of a *composite component*. This makes it possible to hierarchically elaborate a component as a composition of other components. The internals of a component are specified as an architecture of interconnected sub-components. This internal architecture follows the same rules as described in the previous section.

To align the internal architecture with the black-box view, the unconnected ports of the sub-components are delegated to ports on the outer boundary of the composite component. This is depicted in Figure 9, which shows 4 levels of nesting.

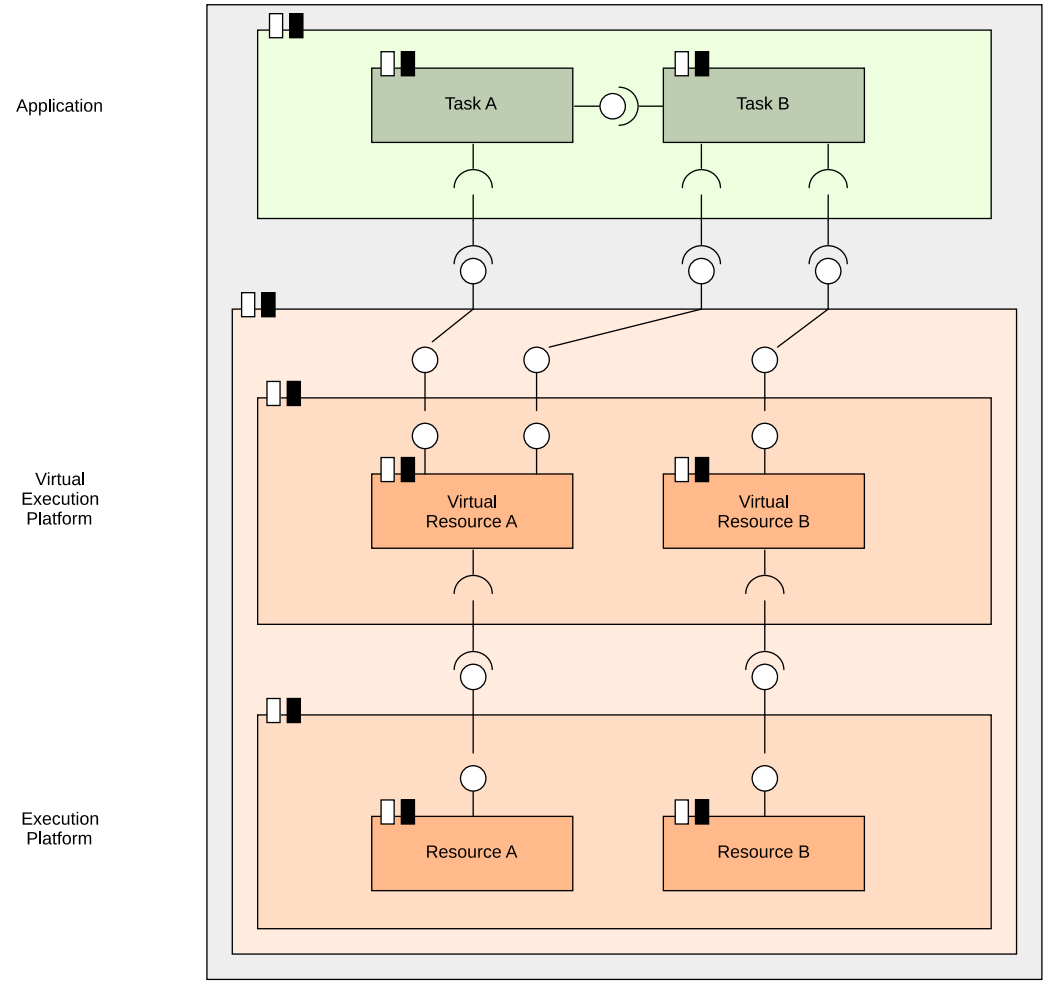


Figure 9: Component composition

On the first level, there is one component that is realized by both hardware and software. Internally, the component is split to an application component and platform component. The application component is internally modeled as two tasks. The platform component internally consists of two layers – the execution platform and the virtual execution platform on top of it. Both the execution platform component and the virtual execution platform component are internally modeled as collections of resources.

The component model thus allows decomposition to an arbitrary level of detail. Generally, the rule of the thumb is to go to such a level of detail that is necessary to model all quality properties, budgets and costs that need to be brought in in the design optimization phase (WP3) and the run-time adaptation (WP4).

4.3.3 Component configurations

An essential feature of the FitOpTiVis component model is that it explicitly captures the potential design space of component configurations. The component design space in FitOpTiVis is captured by component configurations, internal component properties (to reflect qualities, budgets, costs), and by constraints over the properties.

In particular, a component configuration reflects a discrete variant of the component. The configuration determines the ports (including their cardinality) and mapping of internal component properties to properties of ports. Furthermore, it determines the constraints over the properties. As such, the configuration determines both the black-box and the white-box view of a component.

Recalling the example with smart camera – the camera has 6 configurations as given in the table in Section 4.3. In configurations 1-3 it has an outputs port for providing biometric data; in configurations 2-4, it has an outputs port for providing face data; and in configurations 3-6, it has an outputs port for providing video data. In all configurations, the component has an internal property FPS (frames per second). The configuration defines constraints over the FPS property: it is 1 for configurations 1-3, 10 for configurations 4-5, and 30 for configuration 6. The constraints further bind the internal FPS property with the FPS property on the respective output ports.

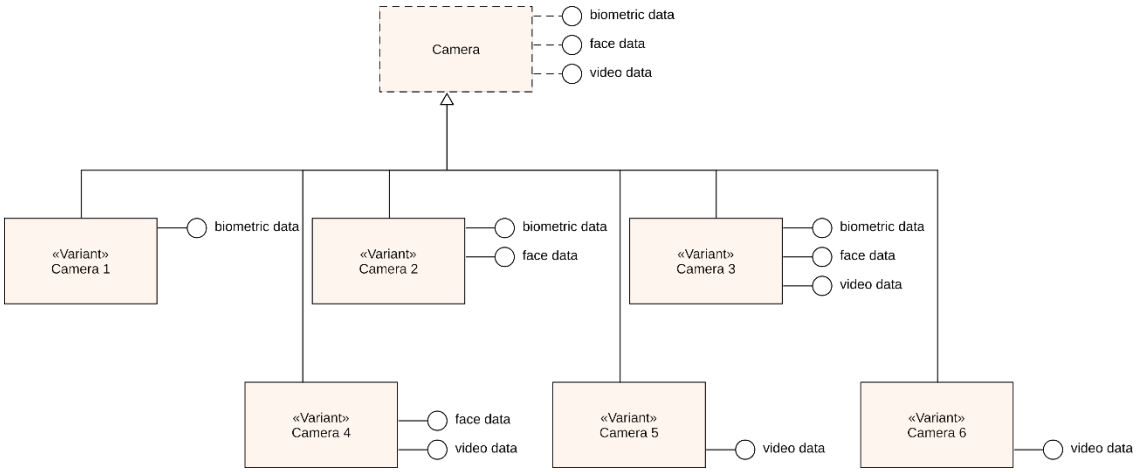


Figure 10: Smart camera configurations

Additionally, the configuration determines also the required budget / cost consumed. This comprises the energy consumption and GPU allocation, for example.

While in this example, the FPS was fully determined by the configuration, it is also generally possible to see the FPS property as a scale. The configuration then only specifies the permitted range and the dependency between the FPS and cost (e.g., the energy consumption).

This way, the component configurations, properties and constraints together with the component architecture form a design space. Searching the design space for the most fitting configuration and assignment of properties with respect to a cost (e.g., the energy consumption) can be seen as a constrained optimization problem.

4.3.4 Example: Component Abstraction in VR Use Case

In the context of use case UC-2, Virtual Reality, advanced virtual reality (VR) techniques require high performance computation under stringent latency requirements. Future low latency network services enable the use of remote acceleration of the computation on devices in the edge or the cloud to improve quality of a VR application. We assume in this example that the application and acceleration use OpenCL to define kernels that can be accelerated. Moreover, we assume through portable OpenCL solutions such as POCL [JSS+15] and POCL remote (investigated in FitOpTiVis, WP4) that the application may choose to use a remote OpenCL device for high quality results, or a local device when such a device is not available, or when accessing it would incur too much latency for the application.

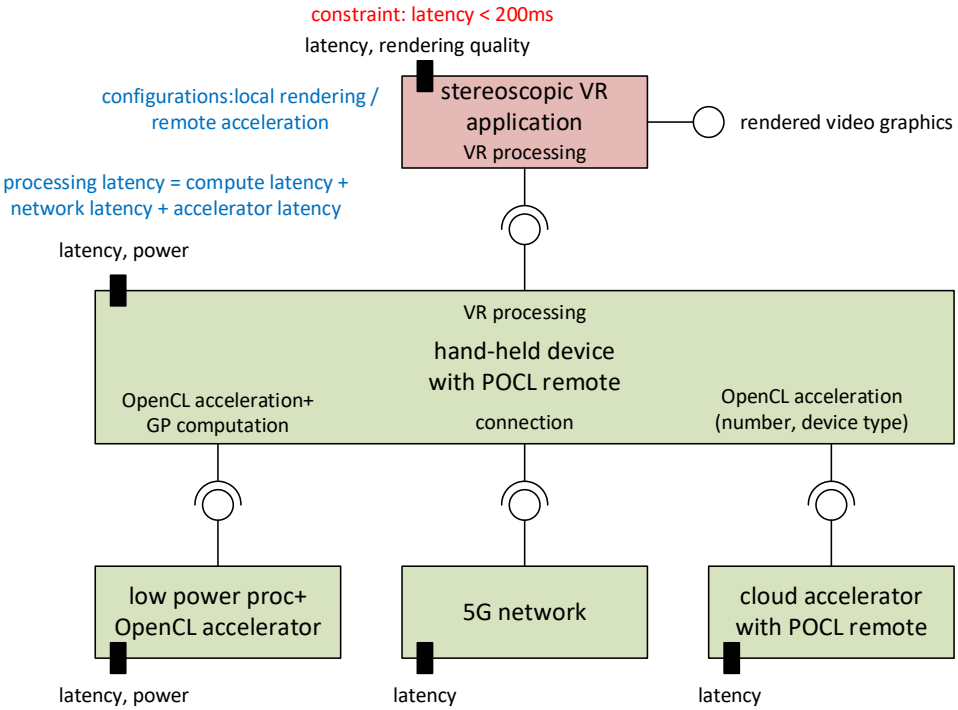


Figure 11: Model of the remote configuration of a stereoscopic VR application

The local device is assumed not to be able to deliver the same quality as the remote device. A (simplistic) model of this system in the component abstraction is given by three platform components, the hand-held device, the network and the cloud accelerator. The network offers communication service between the hand-held device and the accelerator. For the sake of this simple example, the budget provided by the network platform component is a connection, which is characterized by its latency only. The cloud service platform component has OpenCL devices of particular type(s) as its resources and the virtual resource it offers is defined by the number and type of the device.

The application supports two configurations, one corresponding to the local acceleration. It requires no budget from network or cloud service, but only an OpenCL device on the local platform. The application quality includes two aspects, the latency of the application, and the quality of the rendering. We assume that there is a constraint on the latency as stereoscopic VR applications may cause nausea when the latency is too large. The second configuration employs remote acceleration. It requires a network budget to realize a connection to the cloud service. The application latency depends on the network latency. It also requires a budget from the cloud accelerator. In this configuration the application delivers higher quality, but (possibly) at a larger latency.

The OpenCL standard includes device types and application models (kernels) but does not include an explicit resource management architecture. (Figure 12 illustrates the concepts and terminology of the OpenCL architecture.) Some of the properties of component may be determined through online monitoring or calibration, such as the latency of the network connection.

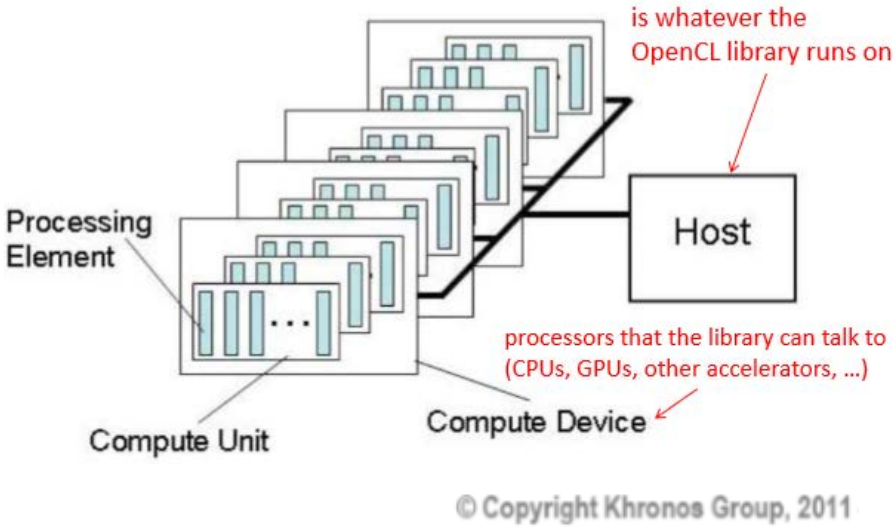


Figure 12: Concepts and terminology in the OpenCL architecture

Platform component abstraction: A platform is: "The host plus a collection of **devices** managed by the OpenCL framework that allow an **application** to share **resources** and execute kernels on devices in the platform."

The easiest way to characterize budgets from an OpenCL component is as a set of available devices by their types (CPU/GPU/accelerator), possible even including specific make and/or generation. Since OpenCL does not aim to provide performance portability of applications across various heterogeneous devices, applications are often tuned to a set of targeted GPU models or product families. This means that the "budgets" required by an application may need to be very specific to guarantee that a previously measured performance is reachable with the application. A more refined and challenging model would be to characterize the accelerator through generic performance metrics.

4.4 Mathematical Component Framework for Quality and Resource Management

In this section we give a precise, composition mathematical component model that can be used as an underlying model or semantics for the conceptual component abstractions that have been introduced. It introduces components with their configurations and properties in a multi-objective optimization setting and it defines compositions of components into larger components and systems that form the platforms, virtual platforms and applications of the reference architecture.

Components are the building blocks of our framework. Their configurations model the (re)configurable set points of each of them. The set points are characterized by their inputs and outputs and provided and required budgets and qualities. The qualities of the component refer to the properties that we want to optimize, for instance, latency, throughput, energy consumption, cost. We use concepts from the Pareto algebra framework defined in [GBTO2007] to capture quality and resource management in a compositional way.

4.4.1 Component Framework Definition

Within the mathematical component model, we assume the existence of the following sets:

- C_S is a set of component *configurations*,
- B is a set of *budgets* with a partial order $\preccurlyeq_B$,
- F is a set of *inputs* and *outputs* with a partial order $\preccurlyeq_F$,
- Q is a set of *qualities* with a partial order $\preccurlyeq_Q$.

Budgets, inputs and outputs, and qualities are all equipped with an ordering relation that distinguishes better values from worse values. For example, a smaller required budget is better than a larger required budget, and vice versa for provided budgets of components. The relation is assumed to be a partial order, which allows also certain values to be declared incomparable. For uniformity of the model we assume that also inputs and outputs have such an ordering. A 'better input', for instance, could be one that accepts a wider input data type. A better output could provide a wider set of output ports. Qualities of components are naturally also ordered to provide a basis for (multi-objective) optimization.

The configuration of a component is determined by the values of configuration parameters that can be set from outside of the component. We abstract from the

connection between parameters and the configurations in this section on the mathematical model. The sets B, F and Q are quantities in the sense of the Pareto algebra framework [GBTO2007]. Note that elements of these sets can have an arbitrarily complex (or arbitrarily simple) structure. Of particular interest for the FitOpTiVis components model is that they can be composed, hierarchically. A single budget can include, for example, both a processor and a memory budget.

When components are composed together into a new component, their inputs, outputs, budgets and qualities are combined. To formalize this, we require the existence of an addition (+) operation on B, F and Q that is monotone in the following sense:

$$a \preccurlyeq b \wedge c \preccurlyeq d \Rightarrow a + c \preccurlyeq b + d$$

which means that the composition of components agrees with the defined notion of better and worse, i.e., if one component is better than another, and we compose the better component with the same third component, the resulting composition should be better than the composition obtained from the worse component with the third component. This is a natural property to expect, but it needs to be formulated in the mathematical framework.

Note that we shall not define the details of the addition operator, as it is considered to be domain-specific. I.e., it depends on the types of components being used what the appropriate operator is. Some examples are given below.

We also require the existence of a $-$ operation on B and F that models what happens when budgets, inputs or outputs are (partially) satisfied/ consumed. We also need monotonicity for this relation:

$$b \preccurlyeq a \wedge c \preccurlyeq d \Rightarrow b - d \preccurlyeq a - c$$

Example 1 (*Budgets*). A storage budget can be modelled by a natural number: the number of bytes available for storage. Clearly, $(\mathbb{N}, \leq)$ is a partially-ordered set, and the usual $+$ and $-$ operations on $\mathbb{N}$ satisfy the monotonicity requirements.

Example 2 (*Inputs and outputs*). Let $V = \{video, audio\}$, to consider combinations of audio and/or video inputs or outputs. For this we define F as the powerset of , i.e., $F = \wp(V) = \{\emptyset, \{video\}, \{audio\}, \{video, audio\}\}$. The partial order is given by set inclusion: $f_1 \preccurlyeq f_2$ if and only if $f_1 \subseteq f_2$. Furthermore, $f_1 + f_2$ is defined as $f_1 \cup f_2$, and $f_1 - f_2$ is defined as $f_1 \backslash f_2$. Basic set theory gives us that these operations satisfy the monotonicity requirements.

We can now define a component as follows.

Definition 1 (*Component*). A component is a tuple (C, i, o, r, p, q), where

- $C \subseteq C_S$ are the configurations of this component,
- $i: C \to F$ gives the inputs for each configuration,
- $o: C \to F$ gives the outputs for each configuration,
- $r: C \to B$ gives the required budget for each configuration,
- $p: C \to B$ gives the provided budget for each configuration,
- $q: C \to Q$ gives the quality for each configuration.

A component *instance* is a component in combination with a specific configuration. A component instance thus gives concrete values for the input, output, required budget, provided budget and qualities.

The following defines relate our components to the corresponding concepts of Pareto algebra.

Definition 2 (*Component configuration space*). The component configuration space is the set $\mathcal{S} = F \times F \times B \times B \times Q$.

Definition 3 (*Component instance*). Let $M = (C, i, o, r, p, q)$, and let $c \in C$. The c-instance of M, denoted by $M(c)$, is the tuple $\big(i(c), o(c), r(c), p(c), q(c)\big) \in \mathcal{S}$. We let $C(M)$ denote the set $\{M(c) | c \in C\} \subseteq \mathcal{S}$.

To define a partial order on the component instances in the component configuration space S, we use the partial orders on F, B and Q.

Definition 4 (*Dominance*). Let $s_1 = (i_1, o_1, r_1, p_1, q_1) \in \mathcal{S}$, and let $s_2 = (i_2, o_2, r_2, p_2, q_2) \in \mathcal{S}$. We say that s_2 dominates s_1, denoted by $s_1 \preccurlyeq s_2$, if and only if $i_2 \preccurlyeq_F i_1 \wedge o_1 \preccurlyeq_F o_2 \wedge r_2 \preccurlyeq_B r_1 \wedge p_1 \preccurlyeq_B p_2 \wedge q_1 \preccurlyeq_Q q_2$.

A component instance is thus dominated by another component instance if it has at least the same input and required budget, and at most the same output, provided budget and qualities. In such a case one would argue that the configuration that leads to instance s_1 should never be preferred over the configuration of s_2 and should be considered redundant. The definition shows a fundamental distinction between how inputs and outputs and required and provided budgets are treated in the component scope. Loosely spoken, a component instance dominates another component instance if it requires less and provides more. Configurations that lead to dominated instances are ideally eliminated at design-time. Sometimes configurations turn out to be dominated in a particular run-time situation. In that case they may be eliminated at run-time in a quality and resource manager.

We lift the dominance relation to sets of configurations as follows. Let $S_1, S_2 \subseteq \mathcal{S}$. Then S_2 dominates S_1, $S_1 \preccurlyeq S_2$ if and only if for every $s_1 \in S_1$ there is some $s_2 \in S_2$ such that $s_1 \preccurlyeq s_2$. We say that $S \subseteq \mathcal{S}$ is *Pareto minimal*, denoted by $min(S)$, if and only if not $s_1 \preccurlyeq s_2$ for any $s_1, s_2 \in S$. Two configuration sets $S_1, S_2 \in \mathcal{S}$ are *Pareto equivalent*, denoted by $S_1 \equiv S_2$ if and only if they dominate each other, i.e., $S_1 \preccurlyeq S_2 \wedge S_2 \preccurlyeq S_1$. With these definitions on the configuration space and component instances we define the notion of a Pareto-minimal component.

Definition 5 (*Pareto-minimal component*). Let $M = (C, i, o, r, p, q)$ be a component. Its Pareto-minimal version, denoted by $min(M)$, is the smallest component (with the least number of configurations) that is Pareto equivalent to M.

Below we specify two composition operators for components. We show that they satisfy two properties that are needed for efficient compositional reasoning [GBTO2007]. The first property states that the Pareto-minimal component is a proper abstraction for the composition operators given in [GBTO2007] (denoted by $*$):

$$C(M_1 * M_2) \equiv C\big(min(M_1) * min(M_2)\big)$$

If a composition operator satisfies the equation above, then components can safely be minimized to its Pareto-optimal configurations during the composition process. This may result in an exponential reduction of the number of possible combinations for composition. The second property states that a composition operator preserves minimality:

$$C\big(min(M_1 * M_2)\big) = C\big(min(M_1) * min(M_2)\big)$$

When such an operator is applied to compose minimal components, then minimization after the composition is not needed.

Below we define the two composition operators and show that they satisfy the former property, but not the latter.

Definition 6 (*Horizontal Composition*). Let $M_1 = (C_1, i_1, o_1, r_1, p_1, q_1)$ and $M_2 = (C_2, i_2, o_2, r_2, p_2, q_2)$ be components. Their horizontal composition, denoted by $M_1 \Rightarrow M_2$, is a new component (C, i, o, r, p, q) where

- $C = C_1 \times C_2$
- $i(c_1, c_2) = i_1(c_1) + \big(i_2(c_2) - o_1(c_1)\big)$,
- $o(c_1, c_2) = \big(o_1(c_1) - i_2(c_2)\big) + o_2(c_2)$,
- $r(c_1, c_2) = r_1(c_1) + r_2(c_2)$,
- $p(c_1, c_2) = p_1(c_1) + p_2(c_2)$, and
- $q(c_1, c_2) = q_1(c_1) + q_2(c_2)$.

Note that the configurations of the new component include all combinations of the configurations of its constituent components. A practical implementation may need to address this combinatorial explosion of possibilities. The reduction to optimal configurations may help, but not generally solve this issue. Effective search strategies and (domain-specific) heuristics need to be applied.

The operator preserves Pareto equivalence, so it is fine to reduce component models to their Pareto minimal configurations.

Example 3 (*Horizontal Composition*). Using the input/output from Example 2 and the $+$ and $-$ operators defined in the example, we may compose a component with no inputs and the output $\{audio, video\}$ in configuration c_1 with another component with input $\{video\}$ and no outputs in configuration c_2. According to the definition of the composition we obtain a new component with input $i(c_1, c_2) = i_1(c_1) + \big(i_2(c_2) - o_1(c_1)\big) = \emptyset + (\{video\} \backslash \{audio, video\}) = \emptyset$ and output $o(c_1, c_2) = \big(o_1(c_1) - i_2(c_2)\big) + o_2(c_2) = (\{audio, video\} \backslash \{video\}) + \emptyset = \{audio\}$. I.e., all inputs are satisfied and the audio output remains available for further composition.

Definition 7 (*Vertical Composition*). Let $M_1 = (C_1, i_1, o_1, r_1, p_1, q_1)$ and $M_2 = (C_2, i_2, o_2, r_2, p_2, q_2)$ be components. Their vertical composition, denoted by $M_1 \Uparrow M_2$, is a new component (C, i, o, r, p, q) where

- $C = C_1 \times C_2$
- $i(c_1, c_2) = i_1(c_1) + i_2(c_2)$,
- $o(c_1, c_2) = o_1(c_1) + o_2(c_2)$,

- $r(c_1, c_2) = r_1(c_1) + \big(r_2(c_2) - p_1(c_1)\big)$,
- $p(c_1, c_2) = \big(p_1(c_1) - r_2(c_2)\big) + p_2(c_2)$, and
- $q(c_1, c_2) = q_1(c_1) + q_2(c_2)$.

The vertical composition operator preserves Pareto equivalence.

Example 4 (*Vertical Composition*). Vertical composition can be illustrated using the budget of Example 1, where memory requirements are modelled with a natural number indicating the number of bytes required. It is easy to see that the + en – operators keep track of the available memory and remaining memory requirements. Note that this rather simple model ignores relevant issues such as paging and fragmentation.

5. Domain Specific Language for the Component Abstraction

In this section we describe the proposed Domain Specific Language (DSL) for modelling FitOpTiVis applications and platforms. First, we describe it informally on a simple example (Section 5.1). Then, the complete grammar of DSL is explained in full detail (Section 5.2). The formal definition of the grammar in EBNF is available in Appendix A.

5.1 Example

As an example application, we use a simplified version of the video processing application introduced in Section 4.3 and depicted in Figure 13.

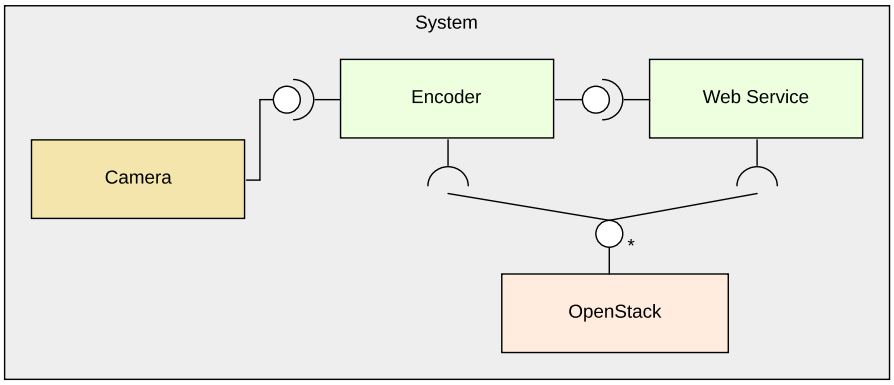


Figure 13: Example model of a video processing application.

The source of the video data is the Camera. It represents a component that is both software and hardware. The raw data stream from the camera is consumed by the Encoder component, which processes the raw data and passes the processed data to the Web Service component, which has a user interface for viewing videos by consumers. Both the Encoder and Web Service are software components and thus they need a platform component for execution. It is represented by the OpenStack component that provides virtual machines (note the asterisk at its supports interface, which means that there can be multiple interfaces of the same type, i.e., in our case, the single OpenStack component can provide several virtual machines).

The types of the horizontal interfaces, i.e., the types for inputs and outputs of components, are defined via the `channel` keyword. The types of vertical interfaces, i.e., the supports and requires of components, are defined via the `budget` keyword. Both kinds of the interfaces can define a number of properties. The listing below defines the types of interfaces (one vertical and two horizontal) for the example application.

```
budget VirtualMachine {
  property memory;
  property core_count;
}
```

```
channel VideoStream {
  property resolution;
  property fps;
  property encoding;
  property audioBitrate;
}

channel AudioStream {
  property audioBitrate;
}
```

The budget interface `VirtualMachine` represents a virtual machine supported by the OpenStack component and required by the software components. It defines its aspects of `memory` and `core_count` describing the virtual machine properties that are required or provided, respectively. The input/output interface channel `VideoStream` used between the software components and the camera has the properties `resolution`, `fps` and `encoding` describing the video stream passing through the ports of the component and `audioBitrate` describing the audio stream bitrate. The input/output interface `AudioStream` has only one quality, `audioBitrate`.

Component types are defined via the `component` keyword. Each component type can define its inputs/outputs and supported/required interfaces. The listing below shows the definition of the `Camera` component, which can operate in two modes, i.e., configurations. Either, it can operate as an audio-video source, or as only an audio source (only the microphone is used).

Either of the two configurations has a single output, which is parametrized with actual values for the interface's properties. In each configuration, the component defines also its own qualities and parameters that serve to further parameterize a corresponding configuration.

The specification assigns options for parameters or values for qualities. It further establishes a relation between properties of ports and the component's parameters and qualities. In this respect, it is important to note that the equals (=) sign in the examples does not denote an assignment, but an equality constraint.

```
component Camera {
  configuration AudioVideo {
    outputs VideoStream out {
      resolution = {width: 1920, height: 1080};
      fps = this.fps;
      encoding = "raw";
    };
    parameter fps in [25, 30];
    quality power_consumption = 5; /* Watts */
  }
```

```
  configuration AudioOnly {
    outputs AudioStream out {
      bitrate = 256;
    }
    quality energy_consumption = 2; /* Watts */
  }
}
```

Similarly, the definition below describes the Encoder component with one input, one output, and one required interface.

```
component Encoder {
  inputs VideoStream raw;
  outputs VideoStream encoded;
  raw.fps = encoded.fps;
  raw.resolution = encoded.resolution;
  encoded.encoding = “mp4”;
  requires VirtualMachine vm {
    memory >= 1024;
    core_count >= 1;
  }
}
```

The OpenStack component below has also defined properties. The component qualities are, like interface qualities, unspecified by default. The values will be specified or filled in later stages (e.g. when the entire system is being specified). Component qualities are also subject to constraints. Alternatively, the qualities can have an initial constraint. This constraint is by design equality-only.

Importantly, the `vm` supported interface can exist in several instances, i.e., it is defined as an array (with its size in the square brackets).

```
component OpenStack {
  parameter instance_count;
  parameter memory_per_instance_MB in [1024, 2048, 8192];
  parameter cores_per_instance in [1, 2, 4];

  quality memory_consumption = instance_count *
    memory_per_instance_MB;

  supports VirtualMachine vm[instance_count] {
    memory = memory_per_instance_MB;
    core_count = cores_per_instance;
  };
}
```

Finally, the `WebService` component consuming the processed video stream is quite straightforward.

```
component WebService {
  inputs VideoStream in;
  requires VirtualMachine vm {
    memory >= 1024;
    core_count >= 1;
  }
}
```

Lastly, there is a single composite component representing the whole application. It instantiates and connects all the component types described above. As the Encoder requires the input of the `VideoStream` type, the `Camera` component can be used only in its `AudioVideo` configuration.

```
system Application {
  component OpenStack os {
    node_count = 1;
  }
  component Camera camera {
    configuration = AudioVideo;
  }
  component Encoder enc;
  component WebService sink;
  camera.out outputs to encoder.raw;
  enc.encoded outputs to sink.in;
  enc.vm runs on os.vm;
  sink.vm runs on os.vm;
}
```

5.2 Specification

This section describes the created DSL formally. To show its syntax we use the EBNF (Extended Backus-Naur Form) notation. The complete set of syntax rules is presented in Appendix A.

The DSL is white-space insignificant and case-sensitive. Comments are written the same way as in C, C++, Java and other languages with roots in C. Thus, `//` starts a comment till the end of a line, while `/*` and `*/` surround multiline comments. Nested comments are not supported.

Identifiers (further denoted as <ID>) are sequences of characters, where the first character can be any letter or underscore (‘_’), followed by an unlimited number of letters, digits or underscores. Each identifier has to consist of at least one character.

Strings (further denoted as <StringLiteral>) are sequences of characters surrounded using either quotation marks (“”) or apostrophes (‘’). The backslash character (\) is used for escaping (like in C, Java and other languages).

A file written using our DSL comprises of 5 possible elements, which can repeat indefinitely. These 5 elements are import statement, budget definition, data channel definition, component definition and system definition.

```
<Model>: { <Element> }

<Element>: <Import>
| <BudgetDefinition>
| <ChannelDefinition>
| <ComponentDefinition>
| <SystemDefinition>
```

5.2.1 Import

Syntax:

```
<Import>: “import” “(“ <StringLiteral> “)” “;”
```

Example:

```
import(“interfaces.fit”);
```

The import element declares usage of definitions from another resource (e.g., from a file). The import will not transitively import other elements as specified by the target resource. In case of name collision, the element(s) in the current file take precedence.

5.2.2 Budget interface definition

Syntax:

```
<BudgetDefinition>:
      “budget” <ID> “{“ { <PropertyDefinition> } “}”
<PropertyDefinition>:
      “property” <ID> ‘;’
```

Example:

```
budget foo {
  property q1;
  property q2;
  property q3;
}
```

The budget definition defines a set of qualities of the particular budget. These qualities are visible to both producer and consumer of the interface and can be used for constraints.

5.2.3 Channel interface definition

Syntax:

```
<ChannelDefinition>:
      “channel” <ID> “{“ { <PropertyDefinition> } “}”
```

Example:

```
channel foo {
  property q1;
  property q2;
  property q3;
}
```

The channel definition is syntactically and semantically similar to budget definition, except the keyword “channel” is used instead of “budget” (see Section 5.2.2).

5.2.4 Component definition

Syntax:

```
<ComponentDefinition>: “component” <ID>
      “{“ ( <DefaultConfiguration> | <Configurations> ) “}
<DefaultConfiguration>: <ConfigurationBody>
<Configurations>: { <Configuration> }
<Configuration>: “configuration” <ID>
      “{“ <ConfigurationBody> “}”
```

Example:

```
component DefaultConfiguration {
  // configuration body
}
component MultipleConfigurations {
  configuration foo {
    // configuration body
  }
  configuration bar {
    // configuration body
  }
}
```

A component is defined by its possible configurations. In case a component has only a single (default) configuration, then the configuration keyword is omitted, and the component is defined directly. Otherwise, there is a list of possible configurations defined.

```
<ConfigurationBody>: { <ComponentRule> “;” }
<ComponentRule>: <SupportsPredicate>
      | <RequiresPredicate>
      | <InputsPredicate>
      | <OutputsPredicate>
      | <PropertyPredicate>
      | <SubcomponentPredicate>
      | <ConstraintPredicate>
```

The component configuration definition declares a set of predicates (constraints) that must hold for the component to be valid (e.g., once the system is being defined). Following sections describe all the possible predicates.

5.2.4.1 Interface usage predicates

Syntax:

```
<SupportsPredicate>: “supports” <InterfaceUsagePredicate>
<RequiresPredicate>: “requires” <InterfaceUsagePredicate>
<InputsPredicate>: “inputs” <InterfaceUsagePredicate>
<OutputsPredicate>: “outputs” <InterfaceUsagePredicate>

<InterfaceUsagePredicate>: <ID> <ID>  [ <ArrayIndex> ]
     [ <InterfaceUsageConstraints> ]
<ArrayIndex>: “[“ <Expression> “]”
<InterfaceUsageConstraints>:
     “{“ { <ConstraintPredicate> “;” } “}”
```

Example (when used in **`component`** a semicolon would follow after each line):

```
supports budget_type foo
requires budget_type bar
inputs channel_type baz
outputs channel_type qux {
  property1 = “foo”;
  property2 = 5;
  property3 < baz.property3;
}
```

All the interfaces are used in the same way – the syntax only differs in the used keyword.

The component can use multiple interfaces of the same type (e.g., it can have two different supports predicates for two different budget interfaces). In case one would want to use multiple, or variable amount of the same interface, the array syntax can be used. The expression inside the square brackets is a generic expression and can therefore be based on surrounding qualities. Note that only one-dimensional arrays are supported.

Once the interface port is declared, an optional constraint block may follow (`<InterfaceUsageConstraints>`). This block may contain any constraint defined in 5.2.7. All names of properties of the newly defined port are available in this block without the `<ID> “.”` prefix, and will take precedence over any other names, including the property names of the component. There is currently no way to use the hidden names.

5.2.5 Property predicates

Syntax:

```
<PropertyPredicate>:
     ( “property” | “quality” | “parameter” ) <ID>
     [ (“=” <Expression>) | (“in” <ArrayExpression> ) ] “;”
```

Example:

```
property foo
quality power_consumption
parameter fps
```

The qualities and parameters of components are defined using the same syntax as qualities of budget and channel interfaces. In addition to quality and parameters, we also allow defining general properties (using “property” keyword), which expresses the design uncertainty, whether the property is to be regarded as read-only quality or as a configurable parameter.

Since the components can constraint the values of qualities, parameters, and properties, and since the most common way of constraining a property is by equality, the language allows for direct specification of a single equality constraint using the “=” <Expression> syntax. For example:

```
property foo = 5;
```

is equivalent to

```
property foo;
foo = 5;
```

5.2.6 Subcomponent predicates

Syntax:

```
<SubcomponentPredicate>: “component” <ID> <ID> [ <ArrayIndex> ]
      [ <InterfaceUsageConstraints> ]
```

Example:

```
component another_component foo
component bar baz[10]
```

Each component can have several subcomponents’ instances defined. The first identifier denotes the type of the subcomponent, while the second identifier names the instance. Optionally, multiple components of the same type can be created using the array syntax.

The connecting of the interfaces (channels, budgets) is done using constraint predicates, which are explained in Section 5.2.7.

5.2.7 Constraint Predicates

Syntax:

```
<ConstraintPredicate>:  <BooleanExpression>
      | <AndPredicate>
      | <OrPredicate>
      | <ImplicationPredicate>
      | <RunsOnPredicate>
      | <OutputsToPredicate>
```

(Boolean expression are described later – together with other expressions – in Section 5.2.9.)

5.2.7.1 And-predicate

Syntax:

```
<AndPredicate>:  “all”  “[“  <ConstraintPredicate>  {  “,”
<ConstraintPredicate> } [ “,” ] “]”
```

Example:

```
all [ foo > 5, foo < 10 ]
```

The and-predicate contains a comma-delimited list of constraints that **all need to hold** in order for the predicate to hold. The list of predicates has to contain at least one predicate. The list can optionally end with a comma.

One does not need to use the and-predicate in the top-most level of component definition, as those predicates by default all have to hold. The and-predicate can, however, be used in more complex logical expressions using other composite predicate types (e.g., or-predicate, implication predicate).

5.2.7.2 Or-predicate

Syntax:

```
<OrPredicate>:  “any”  “[“  <ConstraintPredicate>  {  “,”
<ConstraintPredicate>}  [ “,” ] “]”
```

Example:

```
any [
  foo > 5,
  bar < 10,
  and [foo <3, bar > 5]
]
```

The or-predicate is syntactically the same as any-predicate. It holds if **at least one** of the predicates in the list **holds**.

5.2.7.3 Implication-predicate

Syntax:

```
<ImplicationPredicate>:
      <BooleanExpression> “=>” <ConstraintPredicate>
```

Example:

```
foo = 32   => bar < 5
```

The implication predicate can only be written using this left-to-right syntax, meaning that the **left** expression is **always the antecedent**, and the right implication is then the consequent. The implication predicate holds when either the consequent or the negation of the antecedent holds.

5.2.7.4 Runs on / Outputs to predicates

Syntax:

```
<RunsOnPredicate>:
      <QualityExpression> “runs” “on” <QualityExpression>
<OutputsToPredicate>:
      <QualityExpression> “outputs” “to” <QualityExpression>
```

Example:

```
component A a;
component B b;
a.budget_request runs on b.budget_provide;
a.out outputs to b.in;
```

The runs on and outputs to predicates are used to connect subcomponents in a component or components within a system. The `<QualityExpression>` denotes path to a quality and is described in Subsection 3.2.10.

The left side of the runs on predicate is the consumer of the interface (budget request), the right side is the provider of the interface (e.g., cloud component providing virtual machine budgets).

The left side of the outputs to predicate is the provider (data source), the right side is the consumer (data sink).

5.2.8 Expressions

Syntax:

```
<Expression>: <AdditiveExpression>
      | <InlineArrayExpression>
      | <InlineObjectExpression>
```

Additive expressions denote the common expression syntax found in other languages (unary operators, binary operators, precedence handling, brackets, literals) for

non-Boolean operations. The `AdditiveExpression` grammar can be found in Appendix A.

5.2.8.1 Inline arrays

Syntax:

```
<InlineArrayExpression>:
      “[“ [ <Expression> { “,” <Expression> }  [ “,” ]  ] “]”
```

Example:

```
[1, 2 + 3, “foo”]
```

The inline array expression denotes a tuple of values. It is used mostly in `<InExpression>`, which will be described in Section 5.2.9.2.

5.2.8.2 Inline objects (composite values)

Syntax:

```
<InlineObjectExpression>:
      “{“ [ <InlineObjectMember> { “,” <InlineObjectMember } 
          [“,”] ] “}”
<InlineObjectMember>:  <ID> “=” <Expression>
```

Example:

```
{ value1 = 5, value2 = [ “foo”, “bar” ] }
```

The inline object is used for composite values, and allows for tree-like structures to be composed. All data fields of a single object must have unique names, so as to not run into ambiguity issues.

5.2.9 Boolean expressions

Syntax:

```
<BooleanExpression>: <UnaryBooleanOperator> <BooleanExpression>
     | <ComparisonExpression>
          { <BinaryBooleanOperator> <ComparisonExpression> }
     | ‘(‘ <BooleanExpression> ‘)’
     | <InExpression>
```

The only supported unary Boolean operator is negation, which is written using the ‘!’ character. The binary Boolean operators are logical AND (‘&&’) and logical OR (‘||’). Other operators are not currently in the language, but can be added into the specification if the need arises later.

5.2.9.1 Comparison expressions

Syntax:

```
<ComparisonExpression>:

      <Expression> <ComparisonOperator> <Expression>
```

Example:

```
a + 2 < b - 3
```

Comparison is done using common syntax. Expressions are described later in Section 5.2.10. The supported comparison operators are less than (‘<’), less than or equal (‘<=’), greater than (‘>’), greater than or equal (‘>=’), equals (‘=’), not equal (‘!=’).

5.2.9.2 In-expression

Syntax:

```
<InExpression>: <Expression> in <Expression>
```

Example:

```
quality foo;
foo in [1, 3, “bar”]
```

The in expression consists of any expression on the left side, and array of values on the right side (the second expression must evaluate to an array). The in expression holds when the value on the left side exists in the array on the right side.

5.2.10 Quality expressions

Syntax:

```
<QualityExpression>: <ArrayAccessExpression>
      | <SubQualityAccessExpression>
      | <ID>
<ArrayAccessExpression>:
      <QualityExpression> “[“ <Expression> “]”
<SubQualityAccessExpression>: <QualityExpression> “.” <ID>
```

Example:

```
foo
foo.bar
foo[10]
requires virtual_machine vm;
vm.quality1
```

The quality expression denotes a path to some quality. It is used to walk through interfaces, arrays, and inline objects.

5.2.11 System

Syntax:

```
<System>: “system” <ID>
      “{“{(<SubComponentPredicate> | <ConstraintPredicate>)}“}”
```

Example:

```
system foo {
  component bar baz;
  component bar qux;

  baz.out outputs to qux.in
  baz.quality1 < 5;
}
```

The system is both semantically and syntactically similar to a composite component, except the system itself does not need to use budget or channel interfaces. The system is primarily composed of “component”, “runs on” and “outputs to” predicates. The system can specify additional constraints for any of the subcomponents.

6. Virtualization Mechanisms

6.1 Introduction

Platform virtualization is intended as the abstraction of a given physical platform to hide all the unnecessary details and to keep only the aspects that are relevant for the purpose of resource management. Virtualization allows to have a more practical and concise view of the available resources and, in turn, to take decisions to efficiently exploit them according to the required application to be executed. In FitOpTiVis, besides this general and quite common view, virtualization, is also seen as a set of methodologies to achieve predictable and composable application behaviour, and predictable resource configuration options. Focusing on WP2, virtualization mainly involves budget abstractions and exchange of information with application models.

By means of virtualization, in FitOpTiVis applications will not be mapped directly to physical platforms, but to virtual, abstract ones. This opens to the possibility of executing tasks without knowing the physical target device that is actually processing them, neither knowing if there are other tasks running on the same shared device. Virtualization in FitOpTiVis is intended also to abstract the concept of budgets. Budgets are abstract models of available resources, encompassing all the relevant aspects they offer to the user, and they constitute the interface between application components and physical resources. Budgets offered by the virtual platform must be detailed enough to let application models define a meaningful set of set points and qualities. Virtualization will also be exploited to achieve run-time reconfiguration and tuning, to properly and quickly adapt the system behaviour to modifications in requirements, applications, resources or environment.

Such virtualization approach has to face several challenges to be implemented. Two of the main issues are related to compositionality and budget realization. The former is the possibility of splitting virtual platforms to smaller virtual resources, each responsible of an independent budget. The latter deals with the fact that physical devices should be realized such that budgets provided by the corresponding virtual resources are independent, and that offered budgets can be effectively exploited by applications or resource management.

6.2 State-of-the-Art

Virtualization is an extremely investigated research topic and several technologies and standards are present in literature and are already commonly adopted in the practice. In FitOpTiVis, virtualization techniques are widely exploited. Deliverable 4.1, Appendix A gives an elaborate overview of the state-of-the-art in virtualization mechanisms and resource management based on virtualization techniques. In Deliverable 4.1 the emphasis is on their realizations and implementation frameworks. In this section we consider only the modelling aspect of virtualization and its relation to quality and resource management.

6.2.1 Virtualization Models

One of the main challenges of virtualization methodologies is related to the need to provide proper models that can exhaustively describe the underlying physical resources keeping, at the same time, the model lightweight and easy to be analysed. Note that, it is mandatory for the virtualization model to be representative of the resources (computing, memory, communication, etc.) offered by the corresponding physical entity. Abstract representations of parallel computing systems involving both hardware and software features have been surveyed yet in the 1990s [MMT1995], while examples of models envisioning a separation between hardware and software (or architecture and application) aspects are dated back to 2000s. Such separation demonstrated to be effective in mitigating the complexity of the system to reach higher productivity [GS2003], and has been formalized in [KDW+2002] by the introduction of the so-called Y-chart (see Figure 14).

Y-chart requires two different models for applications and architecture. These models are joined when mapping applications into architecture. By acting exclusively on models, it is possible to retrieve quantitative data from an analysis of each possible mapping and, according to them, to take decisions on the same design process. Here comes the importance of being representative for virtualization or architecture models: depending on the degree of fidelity of the model properties with respect to the ones of the underlying real entity, numbers coming from model analysis (after mapping) will be more accurate and, in turn, the decisions made will be more effective for the considered goal.

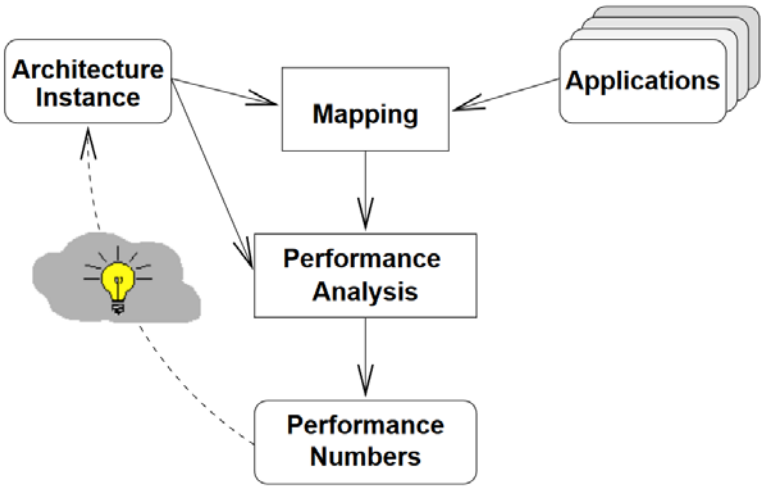


Figure 14 Y-chart: separation of models of architecture from models of applications [25].

Dealing with models, level of abstraction is always one of the parameters that has to be taken into consideration. It indicates how many details of the modelled entity are kept on the model and how many other are omitted. For instance, considering a hardware architecture, it is possible to adopt low-level models, such as transistor or logic gate level ones, or higher ones neglecting transient states and physical details, such as register transfer or transaction level ones. Abstraction level can be further increased: Electronic System-Level (ESL) [GHP+2009] is a coarser grain modelling for complex modern

devices involving millions of transistors, making it possible to perform early design analysis and explorations.

Besides hiding unnecessary details, the usage of incremental levels of abstraction facilitate model analysis: the lower the abstraction level is, the harder will be its manipulation. Given these considerations, a clear trade-off between accuracy and complexity is there. The higher the level you choose for your model the less will be its accuracy and fidelity, but on the other hand you will be able to simplify model analysis since the models are more lightweight. Choosing the proper level of abstraction depends on the purpose of the model and the way it is used and manipulated. Such a trade-off between levels of abstraction, or model lightweightness, and model representativeness/fidelity, virtualization mechanisms often provides subsequent stacked Y-chart models, providing an incremental abstraction [KDW+2002]. To find an optimal conjunction between architecture and application, designers go from the more abstract (less accurate) model, where analysis is fast and easy, and it is possible to perform lots of analysis and to explore lots of solutions, to the less abstract (more accurate) one, where analysis is slow and hard thus allowing the evaluation of few design points or configurations.

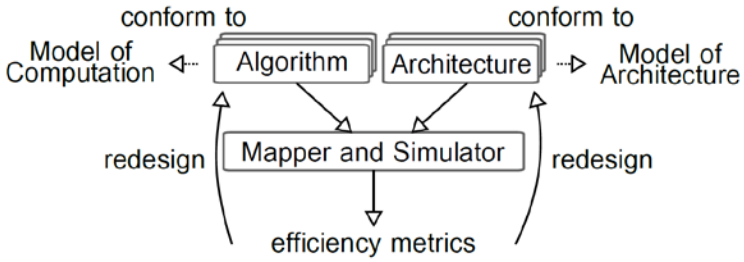


Figure 15 Y-chart defined for Models of Computation (MoCs) and Models of Architecture (MoAs) [KDW+2002].

In [KDW+2002] a first definition of virtualization model, here referred to as Model of Architecture (MoA) is provided, saying that it is *"a formal representation of the operational semantics of networks of functional blocks describing architectures"*.

Starting from the assumption that virtualization models are faithful representations of the underlying physical devices properties and to provide a more formal separation between MoAs and the models of computation (MoCs) used to represent applications (see ***Figure 15***), [PMD+2017] provided a new definition: a MoA is "*an abstract efficiency model of a system architecture that provides a unique, reproducible cost computation, unequivocally assessing a hardware efficiency cost when processing an application described with a specified MoC*". This definition not only puts emphasis on accuracy of model properties, but also clearly defines boundaries and connection points between models of architectures and models of applications. Moreover, it helps in understanding the FitOpTiVis meaning of virtual platform and abstract application, where the former provides amounts of budget, and the latter demands them. Nevertheless, there is a difference between FitOpTiVis budgets and costs in [KDW+2002], in the latter costs are always computable and reproducible, in FitOpTiVis case budgets may also be qualitatively assigned, rather than always formally computed.

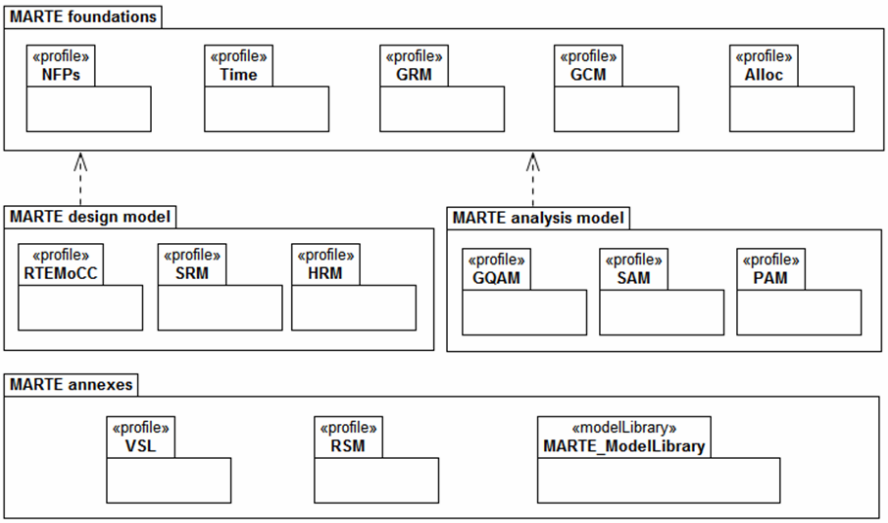


Figure 16 UML MARTE Design Model Package [29].

In literature, lots of examples of virtual models and platforms are present. Some languages, like the Architecture Analysis and Design Language (AADL) [FGH2006], allow to describe hardware besides software. Here, hardware components descriptions are intended to be simulated in time. UML MARTE [OMG2018] also involves software and hardware aspects, but this modelling standard is specifically conceived for real time embedded systems. UML MARTE offers the possibility to model non-functional properties, as shown in Figure 16 where, for instance, appear Schedulability Analysis Modeling (SAM) or Performance Analysis Modeling (PAM). Despite that, it does not take care of how they are extracted from the underlying hardware resources. High-level Virtual Platform (HVP) [CSC+2009] is an architecture virtual representation based on SystemC capable of executing tasks. These latter are described with a different model, called Communicating Processing, and are managed through dedicated task automata.

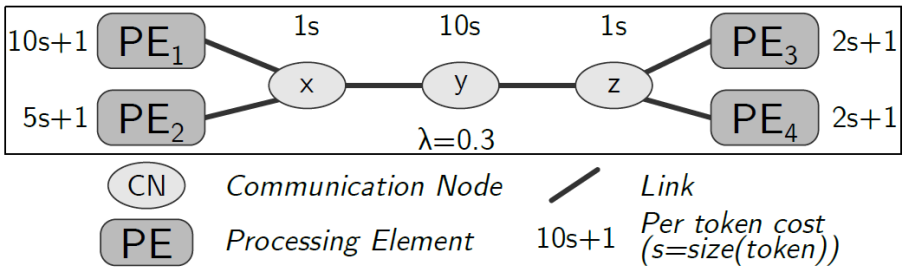


Figure 17 Example of the Linear System-Level Architecture Model (LSLA) [27].

Several works focused on the definition of models for the virtualization of non-functional properties of the physical architectures. Castrillon et al. [CL2014] defined a model where architectures are represented by a graph of processing elements. Edges connect processing elements and have an associated API to make tasks exchange data. In this context, both processing elements and edges API expose several non-functional properties. Grandpierre et al. [GS2003] focused on memory size and bandwidth properties in their virtualization model proposal. Here, the targeted physical platforms

are heterogeneous architectures where it is possible to simulate message passing and shared memory data transfers. Timing properties are instead deepened in the System-Level Architecture Model (S-LAM). Here distribute systems are targeted and communication is mediated by enablers like Random Memory Access (RAM) and Direct Memory Access (DMA). Kianzad et al. [KS2004] dealt with several parameters optimized together and represented with Pareto fronts. Their model, associated with a whole co-synthesis framework, involves processing elements and communication resources with a set of associated parameters: the former have area, price, idle power consumption, data and instruction memory size; the latter have idle power consumption, power consumption per unit of data and worst case transfer rate. The Linear System-Level Architecture Model [PMD+2017], depicted in Figure 17, is an additional virtualization model adopting linear equations to compute non-functional parameters, called costs, on a generic architecture. Here, differently from the previous cases, non-functional parameters are not limited to a pre-fixed set, while it adopts the same linear formula for computing them.

6.2.2 Virtualization for Quality and Resource Management

Even if the state of the art, but also the market, is full of solutions for virtualization, the support for run-time reconfiguration is only partially addressed. Existent virtualization mechanisms mainly focus on efficient management of resources and they only sometimes offer transparent adaptation of allocated resources or qualities according to the current workload. Moreover, they are usually oriented to servers and desktop machines in general, being thus not careful about power/energy consumption or available network bandwidth, parameters that became important dealing with embedded and cyber physical contexts. In particular, for the run-time reconfiguration of virtual platforms in literature, both the virtual reconfiguration and the corresponding physical solutions are a weak point at the moment.

Some research works explored the possibility of reconfiguring at run-time the underlying physical system according to its corresponding virtual entity. Cannella et al. [CDM+2012] proposed a virtualization mechanism for reactively and predictable migration of software tasks at run-time leveraging on dataflow models of application (see Figure 18). Their work targets multi-processor systems on chip (MPSoCs) with distributed memory and based on a network on chip (NoC) architecture. With the adopted application abstraction models, execution is driven by First-In-First-Out (FIFO) point-to-point communication links and to migrate a task from one source tile of the NoC to a destination one, authors propose to simply stop execution in the source tile, update predecessors and successors addresses with the ones of the destination tile, then this latter starts execution according to the related FIFOs state. This work proposes a solution for implementing reconfiguration according to an abstract application model, but they do not provide any virtualization reconfiguration and limit their approach to NoC based MPSoCs. [HBV+2016] introduces a piecewise linear performance model in which scheduling methods for processor sharing are abstracted into linear progress models where the rate of progression depends on the current set of active tasks on a processor.

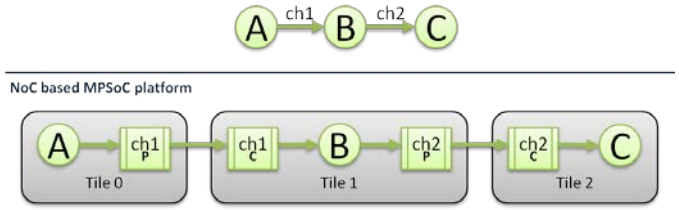


Figure 18: Example of abstract application (dataflow) mapping on the virtual platform in [CDM+2012]

Pelcat et al. [PDH+2014] also rely on dataflows, and in particular on Parameterized and Interfaced Synchronous Dataflows [DPN+2013] (PiSDF), to have an abstract view of the application to be executed. They virtualize the targeted MPSoC devices by means of a S-LAM model of architecture. According to application abstraction and architecture virtualization, the proposed tool, PREESM[7], generates code that can be statically or dynamically mapped onto the considered MPSoC. Run-time re-mapping is then possible by means of an online analysis of application execution on the architecture, also considering current timing and parameter data. SPIDER, a dataflow-based RTOS for MPSoCs, is taking care about run-time behaviour [HPD+2014]. As shown in Figure 19, the mapping is decided by a master core (global run-time, GRT) that sends job tokens to different slave cores (local run-time, LRT) within the MPSoC. Here authors also provide a way to take decisions according to the workload to minimize execution time, however they still do not consider again power/energy aspects.

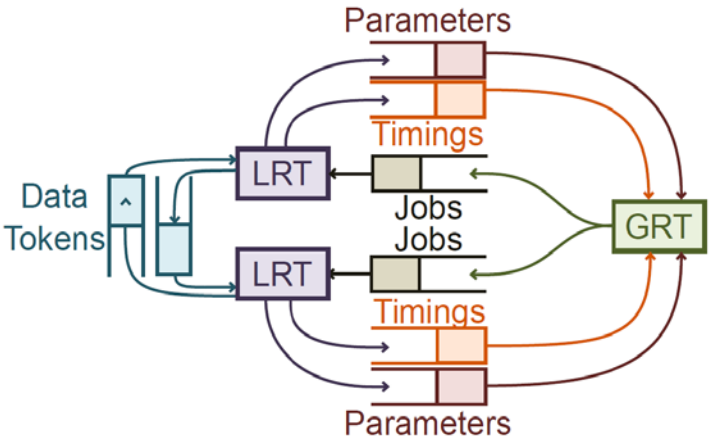


Figure 19: Spider environment for run-time re-mapping and monitoring of the executed dataflow application on MPSoCs.

Goossens et al. [GAC+2013] proposed CompSOC, a solution again based on dataflows and intended for NoC based MPSoCs. Basically, applications are deployed on virtual execution platforms hosted by the corresponding physical execution platforms. Applications are divided in tasks, while execution platforms, physical and virtual, are composed by resources. The core of CompSOC is the broker, a software entity capable of matching budgets available on physical resources (coming from a dedicated resource manager) and the available set points possible for applications, given in terms of

[7] https://preesm.github.io/

required qualities by a system manager (see Figure 20). In order to properly control virtual resources, a sequence of states is used for reserved, allocated, initialized and running cases. CompSOC revealed to be an effective solution for achieving efficiency, predictability and composability. However, again, no power/energy aspects have been considered so far.

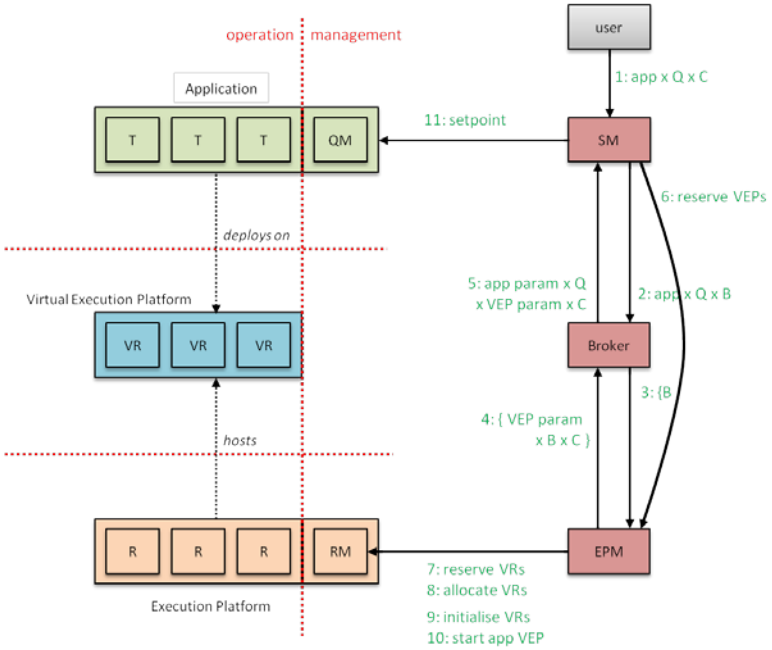


Figure 20: CompSOC approach overview: on the left operation related entities, on the right management ones.

According to the state of the art, virtualization seems to be a powerful instrument not only to achieve faster re-deploy of applications, easier backups, wider reproducibility, clean-up of the environment, as commonly delivered by commercial virtualization products, but also for improving efficiency of the underlying employed devices with a more tailored resource allocation. Here efficiency mainly deals with execution time, but also with power/energy and quality are important especially when dealing with embedded and cyber physical systems. The improvement of efficiency becomes even more effective if it is applied at run-time, thus reconfiguring on-the-fly virtual and, in turn, physical environment. In this sense, all the solution proposed in literature, are lacking some points in terms of metrics to be considered, e.g. power/energy, or in terms of limits in targeted applications and devices.

6.3 Virtual Platform Models

The virtual reconfigurable platform is one of the FitOpTiVis reference architecture parts, dealing with the abstraction of the, optionally reconfigurable, physical resources that are available for the execution of a certain application (see Figure 1). To have a representative and effective virtual platform, it is necessary to provide a model of the underlying physical resource(s), in particular by extracting and exposing to the reference architecture only the information useful to be aware of the context and to take decisions accordingly.

In FitOpTiVis, such information is represented by parameters, qualities and budgets. From the virtual platform point of view, parameters are used, when possible, to configure a virtual platform such that it provides certain qualities and budgets. Indeed, virtual platform components can be configurable and then expose different sets of qualities and budgets. These latter, from virtual platform components point of view, are intended as aspects (e.g. availability of specific features, level of guarantee, amount of memory, etc.) provided by the virtual platform component and they must meet the corresponding required budgets from the abstract application components.

By means of the domain specific language described in Section 5, it will be possible to describe different kinds of virtual platform models that could be adopted in different FitOpTiVis use cases or application fields. In the following, some examples of these virtual platform models will be proposed, trying to highlight the aspects that are compliant with the virtual reconfigurable platform view of the FitOpTiVis reference architecture.

6.3.1 Example Instance: Virtual Platform Models in CompSOC

In the CompSOC platform, *Component Bundles* (see Figure 21) are used to store component models (both application and platform models). For each component configuration, the Component Bundle contains its parameters, qualities, budget descriptor, and initial state. Configurations are determined by setting the parameter values. Qualities describe offered qualities of application components or costs of platform components. The Budget Descriptor, which has a hierarchical structure, describes the provided and/or required budget of a component. The initial state contains the data that is used to initialize the components (e.g., application data). It is not part of the component abstraction but needed in the platform to instantiate a platform and an application. Similarly, the bundle also includes the application instructions.

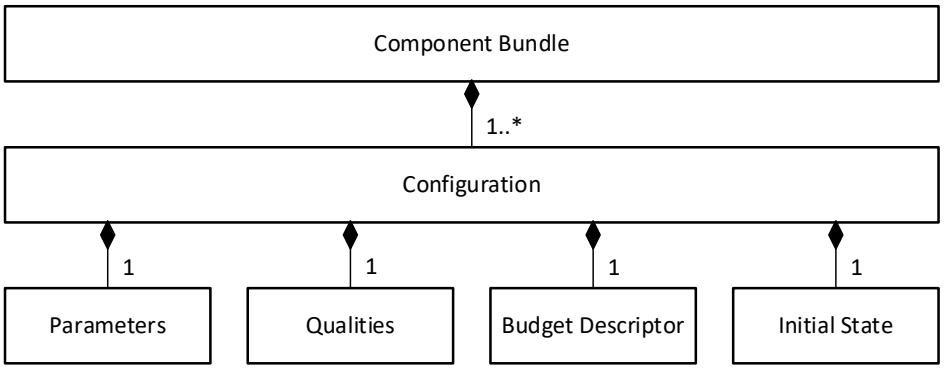


Figure 21: Structure of Component Bundle.

Virtual Execution Platforms are composite components whose models can be stored in the aforementioned structure. Given the fact that a Virtual Execution Platform is composed of one or more virtual resources, the Budget Descriptor for such composite components has a hierarchical structure containing the budgets of all their virtual resources. In CompSOC, the deployment of applications in Virtual Execution Platforms is done by *Virtual Execution Platform Managers (VEPMs)*. They create virtual resources (and eventually VEPs) according to the Budget Descriptors that describe VEPs that

applications require for deployment. These bundles are stored in certain sections in the ELF (Executable Linkable Format) binary which contains the compiled application.

Example. For an application that runs on two processors in a parallel fashion, the Budget Descriptor contains two *Tile Budget Descriptors* which describe the required virtual resources on each platform tile such as processors, instruction/data memories, DMAs, etc. Based on this bundle, a VEPM creates virtual resources (e.g., virtual processors) on two platform tiles separately to create the required VEP that can execute the parallelized application.

6.3.2 Example Instance: Virtual Platform Models in PREESM/ SPIDER

In PREESM[8] rapid prototyping tool for heterogeneous multi/many-core systems, as well as in its run-time version SPIDER [HPD+2014], the Linear System-Level Architecture Model (LSLA) [PMD+2017] is adopted for virtualization. An example of LSLA has been shown in Figure 17. Basically, it is composed by processing elements, communication nodes and links. Each LSLA is a functional model of the underlying architecture in terms of a specific metric, e.g. energy or latency, making it possible to estimate the value of such metric for the whole architecture according to the current application being executed and its mapping among available resources. The corresponding estimated metric value, namely the cost, can be exploited for decision making purposes.

The LSLA is linear since the architecture cost is obtained through the linear combination of the costs of its components: processing elements, communication nodes and links. LSLA is represented with an undirected graph $G = (P, C, L, cost)$, where P is the set of processing elements (PEs), C is the set of communication nodes (CNs), L is the set of arcs between two CNs or between one CN and one PE, while $cost$ is a function that associates a cost to the different components of the model. To execute a certain application, each processing token (atomic part of application processing) must be mapped onto a PE, while each communication token (atomic part of application communication) must be mapped onto a CN.

Example: According to the FitOpTiVis approach, to make LSLA models fit with the proposed reference architecture and domain specific language, constraints related to processing and communication tokens mapping can be expressed as budgets and, in particular, “processing” budgets could be provided only by PEs and “communication” budgets instead could be given by CNs. On the other hand, the cost modelled by the LSLA can be expressed as a quality of the virtual resource, so that it can be exploited to take reconfiguration decisions and to manage resources.

6.4 Quality and Resource Management Conceptual Architecture

The quality and resource management architecture is shown in the reference architecture in Figure 1. In the reference architecture, platform and application

[8] https://preesm.github.io/

components have been optimized and characterized at design-time and may in general support multiple configurations, while the platform resources that are actually available will be known only at run-time. The quality and resource management in FitOpTiVis reference architecture works with the component abstractions of the platform resources and the applications. For the platform, these abstractions form the virtual reconfigurable platform shown in the same Figure 1.

Also applications are characterized by their own components abstractions, and expose their configurations with corresponding trade-offs between quality and resource budget requirements. The compositions of platform components and application components defines which combinations are feasible and which are not, and which combinations are optimal by means of constraints on the configuration parameters.

The quality and resource manager is the entity in charge of identifying possible solutions and selecting which solution(s) will be realized.

The envisioned FitOpTiVis multi-objective optimization aims at ordering possible solutions in terms of better and worse. The FitOpTiVis reference architecture does not prescribe how optimal choices are selected, whether this is done in a centralized or a distributed manner, or whether users are involved in the decision making or not. So, the generic optimization problem is likely to be complex and domain-specific methods and heuristics should be used. The quality and resource manager only ensures that Virtual Platforms are created or modified for the applications to run on and that the applications are started or reconfigured. In the following, few examples of quality and resource management strategies that could be adopted in FitOpTiVis will be described, underlining how they fit with the FitOpTiVis reference architecture.

6.4.1 Example Instance: Quality and Resource Management in CompSOC

The hardware layer in the CompSOC platform [GAC+2013] contains multiple tiles (including processor tiles, memory tiles, peripheral tiles, etc.) interconnected by a NoC. In CompSOC, virtualization is used to consolidate multiple applications in a composable manner on the same platform. Accordingly, resources are partitioned into multiple *virtual resources* which are further composed together to form *Virtual Execution Platforms (VEPs)* on which applications are deployed.

To realize the composable virtualization, physical resources are either exclusively dedicated to a unique VEP or composably shared among multiple VEPs. Resources such as DMAs and local memories, which are relatively cheap in area, are exclusively allocated to a VEP. Other resources such as processors and global memories are temporally or spatially shared among VEPs. In CompSOC, resource-specific entities called *resource managers* are used to virtualize resources. A virtual resource is the result of *programming a required budget* into a resource, which leads to the reservation of a part of the resource for a VEP. Abstracting physical qualities and other parameters of resources away, budgets are used to model virtual resources. The virtual resources are the abstractions of the resources that include all information that is relevant to quality and resource management. And the budgets are abstract models that describe precisely what is necessary to verify that the needs of an application are met.

In CompSOC, processors and the NoC are *partitioned in time* by Time-Division Multiplexing (TDM) arbitration, which realizes composable resource sharing. A TDM-arbitrated resource provides a periodic budget. Its provided budget can be expressed by a tuple (S, I) where S is the *service* that is provided in every I units of time. Section 7.1 illustrates how for a particular application model (static dataflow) performance predictions can be derived from such budget descriptions.

Service describes the type and size of a budget. For processors and the NoC, services are expressed in cycles and bytes respectively. Therefore, virtual processors (virtual NoCs) are modeled by their budgets which are represented by the number of cycles (Bytes) that are provided/required in certain intervals.

Example. Suppose we have a processor which is partitioned by TDM arbitration, and its TDM wheel is of length $1ms$ is partitioned into 4 TDM slots, each of which provides 100 kcycles. The budget of a virtual processor that has been allocated one TDM slot of this processor can this be abstractly modelled as $(100k, 4ms)$. Such a budget description is rich enough for an application component to perform response time analysis. Imagine a task that takes $550k$ cycles to complete. It can be assigned a response time of $24ms$ for performance or schedulability analysis purposes. For a virtual processor that has been allocated two non-consecutive TDM slots, the budget can be captured as either $(200k, 4ms)$ or $(100k, 2ms)$. Note that the latter provides a strictly stronger guarantee than the former, i.e., response times computed from the former budget are never longer than response times computed from the latter.

Memories in the CompSOC platform (it distinguishes instruction, data, and global shared memories) are *spatially partitioned* by memory controllers such that the partitions do not have overlaps in space. The required/provided memory capacity expressed in Bytes are used to describe spatial memory budgets in CompSOC. Next to spatial budgets applications also need budgets to access memories that characterize latency and bandwidth of such accesses. The memory controllers in the platform are specifically designed to be composable to eliminate interference between applications and to be able to provide budget guarantees to its virtual resources.

6.4.2 Example Instance: Quality and Resource Management in SPIDER

The Synchronous and Interfaced Dataflow Embedded Runtime (SPIDER) [HPD+2014] is a Real-Time Operating System (RTOS) for the efficient scheduling of applications on multi-core architectures. It adopts Parameterized and Interfaced Synchronous DataFlows (PiSDFs) [DPN+2013] to describe applications. PiSDF is a good trade-off between flexibility, offering parameterization and hierarchy possibilities, and predictability of the behaviour. In SPIDER, this model is translated at run-time in order to have a global view of dependencies between tasks. In terms of parameters and hierarchy, it is needed to compute parameters before scheduling and this can be done only sequentially among different hierarchy levels, one level at a time. The scheduling consists of two phases, namely *task ordering* and *mapping*: the former sorts the non-executed tasks, while the latter assigns them to each processing element in the architecture. SPIDER can schedule tasks in order to optimize different metrics, like latency, throughput, memory utilization or energy.

Once defined the optimization goal, the scheduling may still vary according to the evolution of the parameters of the PiSDF application and to the execution constraints. A SPIDER RTOS is based on a master/slave execution scheme, as depicted in Figure 19. Local RunTimes (LRTs) are slave lighweight operating systems capable of processing PiSDF tasks, while the Global Run Time (GRT) is the master, being aware of the whole PiSDF application topology and taking decision upon scheduling strategies. These latter are communicated through Jobs, that embed data needed to execute PiSDF tasks and are sent to LRTs by means of job queues. Different kinds of data can be sent back from LRTs to GRT: they can be output parameters, timing, or any other quality provided directly by the LRT underlying physical resource. These data are the core of the SPIDER quality and resource management approach, since they may change the way tasks are scheduled and executed on the underlying physical architecture.

Example. In FitOpTiVis the quality and resource management is the core of the reference architecture, connecting applications abstraction, quality requirements and virtualized platform resources In the same way, SPIDER is the conjunction between PiSDF applications, embedding qualities in parameters or providing them through dedicated queues (e.g. timing ones in Figure 19), and the virtualized platform, for instance modelled by means of the LSLA model of architecture (see Section 6.3.2).

A possible example of SPIDER quality and resource management could be a video encoder (described as a PiSDF application) where quality of the encoding can be tuned by means of a PiSDF parameter. The video encoder is implemented on an embedded multi-core system, so that an energy constraint is present, given by the remaining battery level. By default, the application is executed at the maximum quality and the scheduling effort is put on throughput. When the remaining battery level goes above a certain threshold, SPIDER can decide to optimize energy, to lower the quality of the encoding or to lower the throughput, if there are not higher priority constraints on these metrics. Here, it is possible to define different set points, each providing a different tuple of encoding quality/consumed energy or throughput/consumed energy, to increment the number of possible solutions and take more effective decisions.

7. Instances of the Reference Architecture

7.1 Component Abstractions for Multi-Source Streaming

This section shows an example of an instantiation of the reference architecture in the specific domain of dataflow-based modelling for mapping of dataflow applications onto a predictable multi-processor architecture such as CompSOC [GAC+2013, Deliverable 4.1]. It introduces the component abstraction for an example similar in spirit to the multi-source streaming use-case.

Let S_1, S_2 and S_3 be three streams, which come from three different sources and encoded as M-JPEG. We assume that sources can be configured to send streams either with 720p or with 360p resolution, which we refer to as *full scale* and *half scale*, respectively. There is a full HD display, on which a user may desire to visualize one or a combination of the streams either at half scale or full scale. Figure 22 depicts this scenario. We assume that the required frame rate for both half and full scale streams is 30 fps. Network and screen can be modelled as platform resources in the component abstraction, but we focus here on modelling of the processing part.

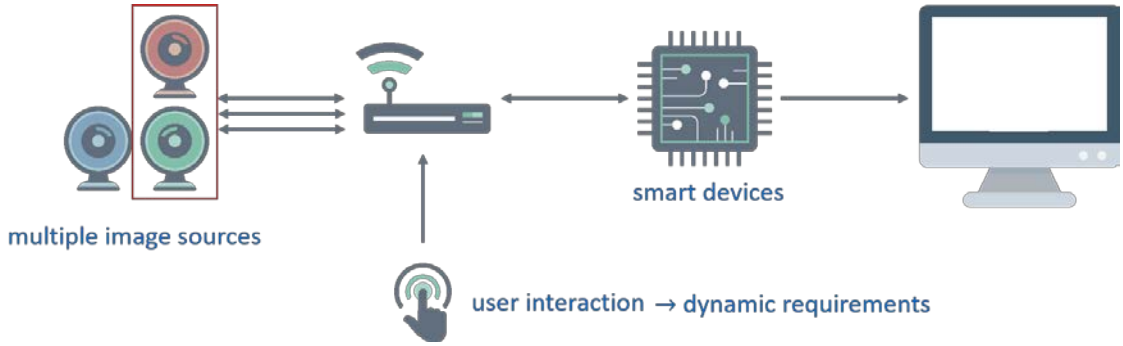


Figure 22: multi-source streaming example

To provide streaming applications with guaranteed performance properties, such as minimum throughput and maximum latency, we model streaming applications with the Synchronous Data Flow (SDF) model of computation [LM87,SB09, SGTB2011]. SDF enables us to conservatively capture the timing behaviour of the system. We would like to embed the SDF model into the reference architecture as a component abstraction.

A dataflow graph describes repetitive tasks and their dependencies in an application. Figure 23 depicts an example SDF graph, where the nodes are the set of actors $A = \{a_0, a_1, a_2, a_3\}$ representing individual tasks of an application. Edges represent the set of channels $C \subseteq A \times A$ modelling dependencies, which can be data dependencies or control dependencies. An actor can be executed once its input channels have at least the number of tokens that are denoted by rates on input channels. Once the actor firing is completed, it consumes the tokens on its input channels and produces tokens on its output channels. For the sake of simplicity, rates of 1 for production and consumption are not shown. Different configurations of an application can be represented as separate SDF graphs and the integrated application including reconfiguration can be modelled in a model called *Scenario-Aware Dataflow* (SADF). The details of this model and performance properties analysis are explained in more detail in [SGTB2011].

A dataflow application can be captured as a component abstraction in the reference architecture. Half scale vs. full scale is a *parameter* that determines the *configurations* of the streaming application. We see later how resource budgets are also included in the configurations. The inputs and outputs of the component abstraction correspond to unconnected ports of the SDF graph model. Half scale or full scale is also considered as one of the *qualities* of the component. Note that it may happen that a certain aspect occurs in multiple roles in the component abstraction. The scale is, on the one hand, a configuration that impacts the type of input and output data and the required computation budget, but on the other hand, it can also assume the role of a quality aspect, full scale being better to look at than half scale.

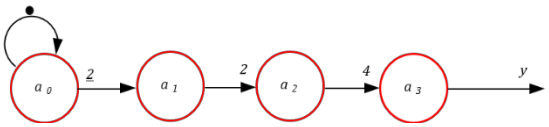
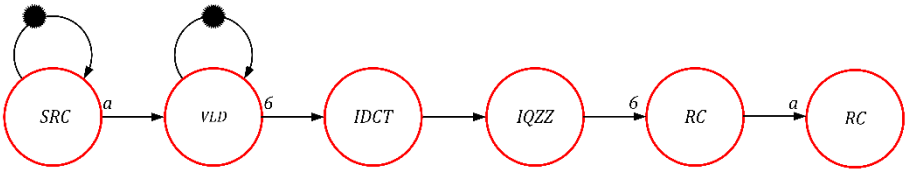


Figure 23: A Synchronous Data Flow graph

Figure 24 depicts the SADF model of applications S_1, S_2 *and* S_3, described in the introduction part of this subsection. In this graph, VLD is the variable length decoder function, whose input is a compressed stream and output are macro-blocks constructed of six blocks. IDCT and IQZZ stand for inverse discrete cosine transform and inverse quantizer and zig-zag ordering, respectively, both operate at the level of blocks. RC is the reconstruction block producing a bitmap digital image of a macro block. SRC and DP are the source and display blocks. Their functionalities operate at the video frame level.



Scenario	a
Half scale	900
Full scale	3600

Figure 24: SADF model of S_1

There are mathematical performance analysis techniques that can compute tight conservative bounds on *throughput* and *latency* of dataflow graphs, if we have worst-case execution times or response times of the functional components of the application [SGTB2011]. Network and memory resources can also be modelled and accounted for [BDLT2019], but we do not include them in this example.

In the FitOpTiVis architecture, we cannot assume that we know a priori on which processor(s) the application is mapped and what budget it gets from the processing resource. Instead, we intend to calculate conservative bounds on throughput and latency, only using the *required budget* information. The computed throughput and latency metrics then serve as additional *qualities* of the streaming application component.

The analysis and design-space exploration can be performed at design-time and leads to a number of alternative processor budgets with different, Pareto optimal combinations of the scale, and latency metrics under the throughput constraint derived from the video

frame rate of 30fps. To determine latency and throughput, a mapping and scheduling of the dataflow graph need to be decided and they are recorded as part of the component configuration.

In the FitOpTiVis reference architecture, we introduce budget abstractions of the virtual resources that the dataflow applications execute on. We consider, as an example, the following budget model for a virtual processing resource. We assume that processing budgets are abstractly characterized for a single processor by a pair of two numbers as follows.

$$B = (C, I)$$

I is a positive real-valued number and C is an integer that denotes a lower bound on the number of processor cycles that the processing resource provides in any time interval of length I. This abstraction is suitable for budget schedulers, such as round robin or TDMA based pre-emptive schedulers [MO2014]. More refined models such a (some finite representation of) service curves of Real-Time Calculus [TCN2000] may provide tighter bounds but are also more complex to handle by design-time or resource optimization and management techniques. From the budget abstraction we can determine a lower bound on the number of cycles for any given interval, as well as a minimal interval for any required number of cycles. These relations are visualized in the graph in Figure 25 with the time interval Δ on the horizontal axis and the corresponding bound on the number of cycles on the vertical axis.

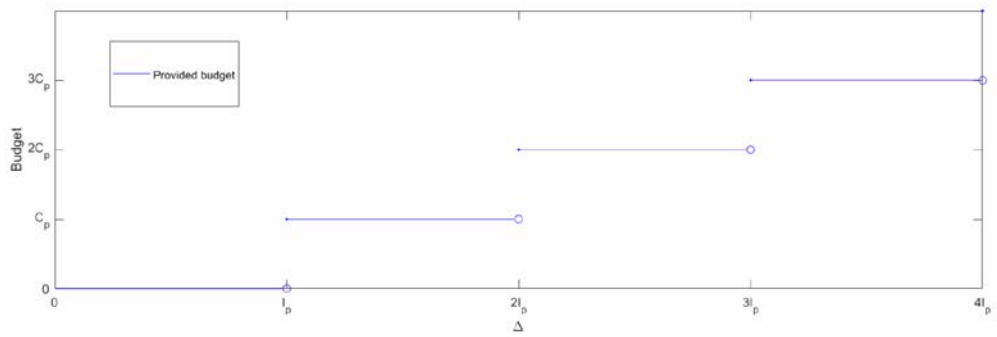


Figure 25: An abstract budget relating time intervals and cycles

One of the research questions addressed in the FitOpTiVis project is whether existing execution or response-time based performance analysis techniques for dataflow can be generalized to provide performance bounds based on allocated budgets from virtual resources for a dataflow graph with a given binding and scheduling to virtual processors. Another question is how to find good mappings and schedules on virtual processors.

A known scheduling order can be incorporated in the dataflow model in the form of additional dependencies [DSG+2012]. The configurations of the application can be represented with models such as shown in Figure 26. The four actors of the model of Figure 23 are mapped onto two virtual processors P_1 and P_2. The number of kcycles required per actor firing on the specific processor is annotated inside the actor. A schedule $a_0 a_1 a_1$is enforced on the first processor by adding extra dependencies and initial tokens. Similarly, schedules are enforced on the second processor and between the processors.

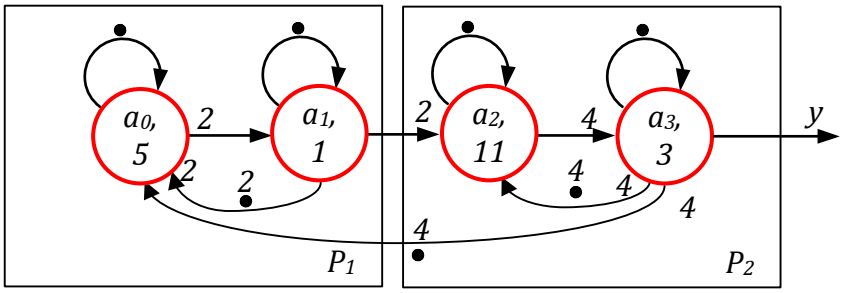


Figure 26: A dataflow graph with virtual processor binding and scheduling

The response time of a sequence of actor executions mapped onto a processor budget can be calculated. For instance, assume that the provided budget by virtual processor P_1 is $(3\ kcycles, 2\ ms)$. To calculate the worst-case response time of the sequence of actor firings $a_0 a_1 a_1$, first, the (worst-case) required number of cycles for the execution of this sequence $(5 + 2 \times 1 = 7\ kcycles)$ is computed. This number, divided by a lower bound on the number of cycles provided per each interval is computed, which provides the worst-case number of time intervals required. In the example, this is $7\ kcycles\ /\ 3\ kcycles$, rounded to a whole number of intervals, which equals 3 intervals. Hence, we can conclude that it takes at most three times the interval length, which means it takes at most 3 times $2\ ms$. Therefore, it is shown that the execution time of this sequence cannot be more than $6\ ms$ when executed with the given budget.

The component is finally captured with a finite number of configurations that give it a trade-off between the allocated budget and the quality that is provided. The resulting component abstraction of the dataflow application component is something like Table 2, where each row represents a different configuration with a particular processing *budget*, throughput, latency and scale *qualities*. Configurations also include the actor mapping and scheduling, which are not shown in the table. For the half scale case, the channel (a_2, a_3) has rate 2 on the side of a_3, and this is the only difference between this and the full scale configuration.

Table 2: Component abstraction of a decoder application.

# of Proc.	Pbudget (kcycles, ms)	Throughput (θ)	Latency (ms)	Scale
1	(3,2)	20	50	full
2	(3,2)	27.7	38	full
4	(3,2)	27.7	38	full
1	(4,2)	26.6	38	full
2	(2,2)	18.5	56	full
4	(1,2)	9.25	108	full
1	(3,2)	38.5	26	half
2	(3,2)	46.8	24	half
4	(3,2)	46.8	24	half
1	(4,2)	50	20	half
2	(2,2)	31.25	34	half
4	(1,2)	15.6	64	half

Assume that we have a number of identical virtual processors with the same provided budget of $(3\ kcycles, 2\ ms)$. The corresponding latency and throughput of the application mapped onto one processor is shown in the first row of the table. The second and third rows of the table represent the quality of the scheduled SDF graph mapped onto 2 and 4 identical processors with the budget of $(3\ kcycles, 2\ ms)$, respectively. The third row shows that although the resource allocated for application is doubled compared to the second row, the throughput and latency are not improved. Thus, this is not a Pareto optimal configuration and is best removed from the component model.

The next three rows of the table compare the quality of different numbers of processors with the overall provided budget. This reveals that distributing a constant budget deployed for running application tasks in parallel would decrease the quality of this application. The remaining rows similarly represent the half scale configurations.

This table enables the system to select an optimal configuration, as long as it realizes the constraints on either quality properties and the required budget at run-time. For instance, to run a half scale with a throughput at least 30 and the latency not bigger than 30ms, the system considers one CPU with $(3\ kcycles, 2\ ms)$ as a feasible option.

7.2 Component Abstractions for an Industrial Inspection System

UC4, the Industrial Inspection system use case, defines multiple components in the model/architecture (for the details please see D1.1, D4.1 and D5.1). This section describes the main components that take part in such use case. The ZG3D Industrial Inspection system captures objects in free fall using 16 cameras. To analyse an object the system performs multiple complex and computationally costly operations. The main objective is to decrease the required bandwidth and increase the throughput using a distributed system of IoT low-power devices (edge capturers) to pre-segment and transfer images to the following components of the ZG3D system.

The system we are proposing is composed of network resources, edge capturers and camera devices, etc. All of them require different configurations and settings, and quality features like resolution, latency, workload, availability of edge devices and workers, among others.

An abstract application of the system has been derived and divided in multiple components as shown in Figure 27.

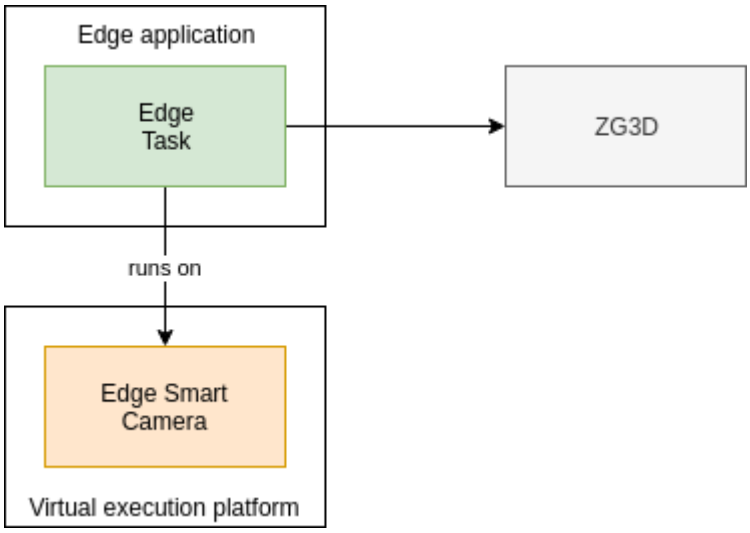


Figure 27: An Industrial Inspection System

The main component that is added in FitOpTiVis is the Edge component with the following internal architecture described in Figure 28.

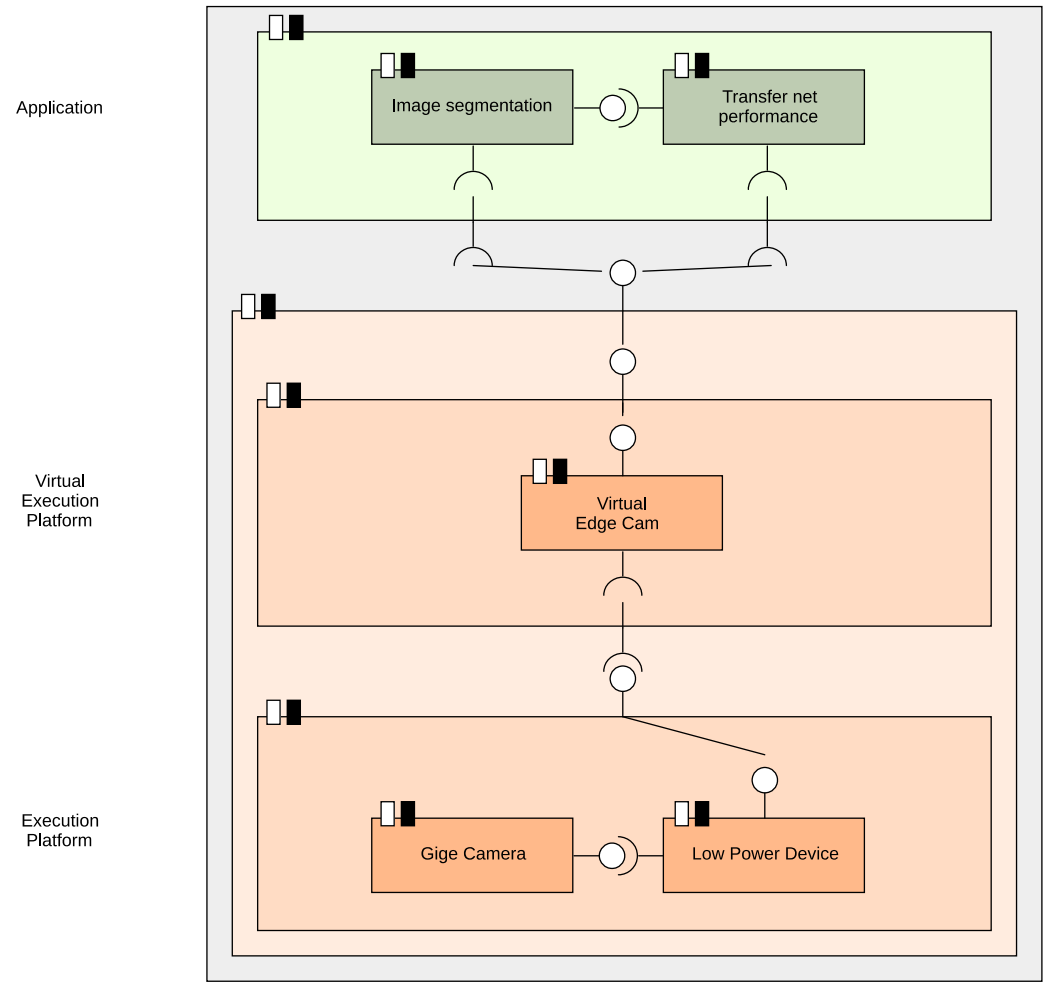


Figure 28: Component Abstraction of an Industrial Inspection System

This Edge component has the following component, budgets and channel definitions that describe it at its first stages:

```
budget BoardExecution {
  property bandwidth;
  property throughput;
  property consumption;
  property resolution;
}

budget VirtualEdgeExecution {
  property bandwidth;
  property throughput;
  property consumption;
  property resolution;
}
```

```
channel ImageStream {
  property resolution;
  property encoding;
}

component Camera {
  configuration capturer {
    outputs ImageStream out {
      resolution = { width: 2280, height: 2048 };
      encoding = "raw";
    }
  }
}

component EdgeBoard {
  quality memory;
  quality bandwidth;
  quality throughput;
  parameter resolution;
  supports BoardExecution board {
    bandwidth = this.bandwidth;
    throughput = this.throughput;
  };
  requires ImageStream cam {
    resolution = this.resolution
  };
}

component VirtualEdge {
  quality memory;
  quality bandwidth;
  quality throughput;
  requires BoardExecution board;
  supports VirtualEdgeExecution vep {
    bandwidth = this.bandwidth;
    throughput = this.throughput;
  };
}

component ImageSegmentation {
  inputs ImageStream in;
  outputs ImageStream out;
  equires VirtualEdgeExecution vep;
}
```

```
component TransferEncoding {
  inputs ImageStream raw;
  outputs ImageStream raw;
  requires VirtualEdgeExecution vep;
}
```

7.3 Model-based component abstraction

The project intends to develop also a methodology that provides UML-MARTE based component abstractions that comply with the reference architecture developed in WP2.

The work in WP2, includes the definition of abstract components that can be easily integrated in a UML-MARTE design flow. The methodology is based on work in other projects (e.g. MegaMaRt2) in which the basic component modelling methodology and tool framework have been developed [MV2017]. The FitOpTiVis innovations in this part are focused in five areas: efficient specification of dataflow models, non-functional parameter specification and verification, single-source based virtual platforms for real-time video systems, modelling of multiple component configurations (set points) and efficient run-time reconfiguration.

The FitOpTiVis abstractions and component templates have been integrated in a UML-MARTE based design flow. Additionally, a C++ implementation methodology that supports multiple implementations (or set points) and run-time re-configuration has been developed and it is being integrated in the UML-MARTE based framework that is presented in Deliverable 3.1. The main steps of the methodology are presented in Figure 29.

The FitOpTiVis abstractions and DSL descriptions will be captured with a wizard that transforms the FitOpTiVis concepts into UML-MARTE elements. The models will be captured with the UML-MARTE tool ecosystem that has been described in deliverable D3.1 and they could use the available platform/application component libraries. The UML-MARTE ecosystem provides several types of tools such as code generators and virtual platforms. The software synthesis tool generates base component implementations from the UML-MARTE models. In FitOpTiVis, a new C++ implementation methodology has been defined. The new approach defines the way to codify UML/MARTE components in C++. These components support several implementations (e.g. sequential/concurrent implementation, GPU implementation, FPGA implementation, …) that can be selected on run-time. This allows adapting the system to particular system situations (system resilience). From a UML-MARTE model, the methodology defines a base component implementation. Typically, this base component is a C++ class that defines the component interfaces. These classes derive from a parent class that provides run-time reconfiguration capabilities. From the base component, different target specific implementation classes are derived.

Additionally, the UML-MARTE framework generates all the infrastructure that is required to simulate the application in a virtual platform and analyses its performances. The host-compiled virtual platform takes into account the platform model (that include the hardware and operating system models) and the application source code. The virtual platform verifies the system functionality and estimates basic performance (execution

time and power consumptions). Other system qualities are estimated with simulation traces and simulation monitors.

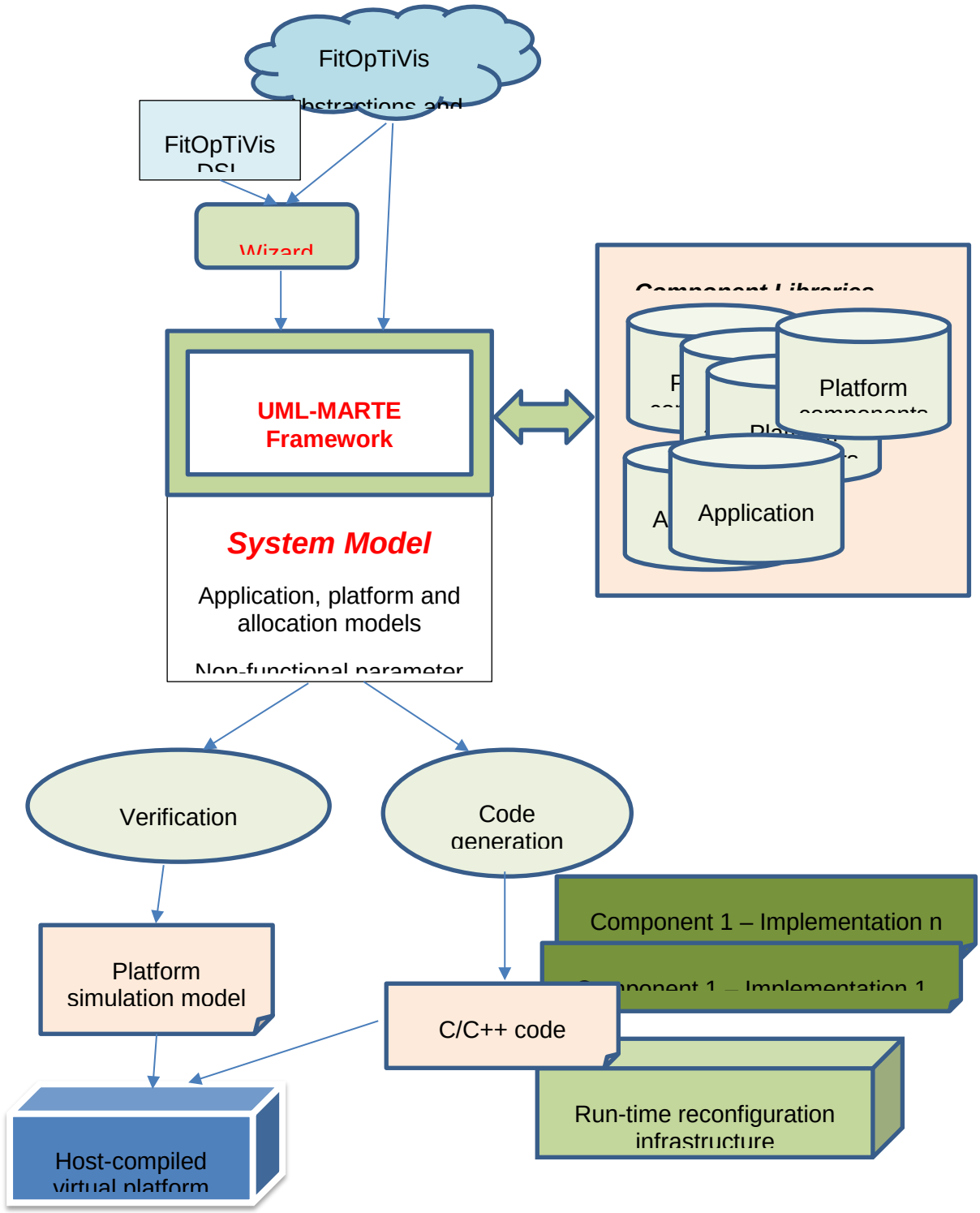


Figure 29: UML-MARTE based design methodology

7.3.1 Component modelling in UML-MARTE

This section presents an extension of an existing methodology that has been developed in the MegaMaRt2 project. The extension supports the FitOpTiVis concepts and abstractions. A classical UML-MARTE methodology defines 3 system views:

- PIM (Platform Independent Model). This view captures information related with the application and it describes the platform-independent functionality of the system. PIM exhibits an enough degree of independence so as to enable its mapping to different platforms. To develop this view in the proposed methodology, two types of diagrams are included: application view and verification view.
- PDM (Platform Dependent Model) that describes the computing resources in which the application will be implemented. This view models the hardware (e.g. CPU cores, GPUs, memories, buses, …) and software resources (e.g. RTOS, memory spaces) of the execution platform. In the proposed methodology, the PDM includes 3 views: SwPlatformView, HwResourceView and MemorySpaceView.
- PSM (Platform Specific Model) that defines the relation between the PIM and PDM views. This model specifies the mapping of the application threads/tasks to the platform computing resources. So, it is captured all the implementation decisions taken during the design process. In this case only the architectural view is included in the proposed methodology.

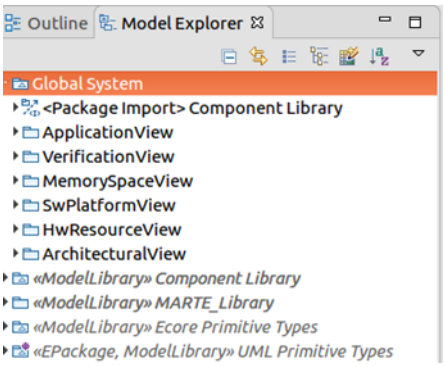


Figure 30: Proposed methodology views

These views are captured with and Eclipse-based framework (Papyrus) that has been improved with several specific plugins that provide requirement capture, performance analysis as well as automatic generation of software and verification code. Figure 30 presents the list of the PIM, PSM and PDM views of the proposed methodology in the UML-MARTE framework (Papyrus).

In FitOpTiVis, application and platform components are used in the same view. UML-MARTE provides different views for these components in order to facilitate PIM and PDM independent specification. This approach improves physical platform reuse and application porting to different physical platforms.

In the proposed approach, the application is described as a network of components. The basic abstraction that models these components is the “UML-MARTE Generic

Component Model (GCM)". A generic component only requires a description of the external component interfaces or ports. The original methodology only supports service interfaces (Client-Server Ports in UML-MARTE). A service is a functionality that the component requires or provides. These services can be modelled as functions with input, output and input-output parameters that are equivalent to the "input and output and parameter ports" of the FitOpTiVis abstract model. The methodology has been extended in FitOpTiVis to support dataflow ports (FlowPorts in UML-MARTE) that provide access to objects that are not included in the component. The components are included on packages that facilitates their reuse.

The generic component can provide services and/or dataflow ports to other components. Additionally, they can require services or data from other components. Next figure presents the structure of a system with 3 components whose ports provide/require services to other components.

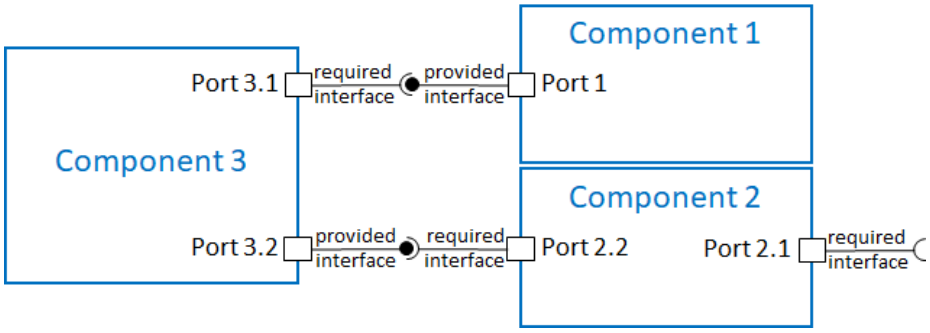


Figure 31. Model-based application example

Currently, different component implementations are specified with UML attributes. These attributes specify the variant identification, the required computation resources and several performance-related parameters (e.g. execution time, required memory, latency …). In the original methodology [MV2017], only 2 types of components are supported: passive and active components. An active component is a concurrent computing unit with real-time features (a real time unit, RTUnit, in UML-MARTE). For example, a sensor that provides periodic samples (periodic task) is typically modelled with an active component. The passive components (Protected Passive Unit, PPUnit, in UML-MARTE) do not own schedulable resources. A function that transform an image is a typical implementation of a passive component. In FitOpTiVis, the original methodology has been extended to support OpenMP based implementations. The new approach defines active components that include OpenMP-based code with passive components.

In the proposed approach, the PIM is captured with the application view. The PIM also includes the test plan that is modelled in the verification view. The application view models the relations between components as well as the component hierarchy or system structure. Figure 32 presents an example of an application view in the Eclipse framework. The components are interconnected through ports that use required/provided interfaces.

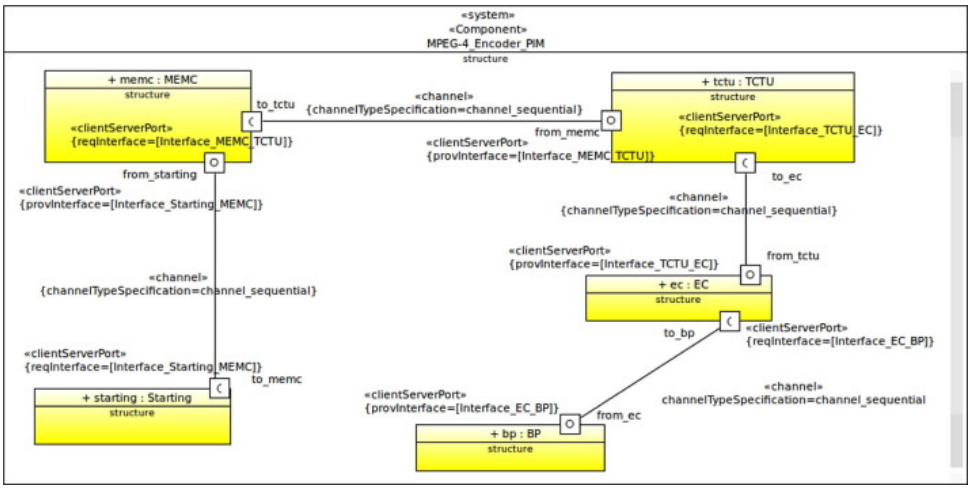


Figure 32: Application view example

In order to model the platform component, the methodology defines 3 views: memory space, software platform and hardware resource views. The memory view identifies the application processes that normally require a specific memory space (MemoryPartition in UML-MARTE). The software view models the RTOS (real-time operating system) and peripheral/system drivers. The hardware resource view defines all the hardware elements of the platforms: sensor, processors, buses, memories, … Figure 33 presents an example of a hardware resource view.

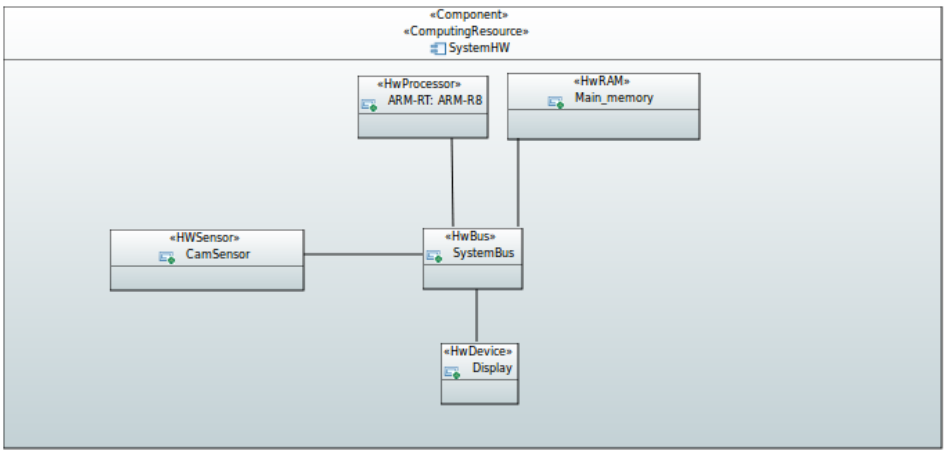


Figure 33. Hardware Resource View

The mapping between the application and platform components is modelled in the architectural view. For every component different mapping or set points can be specified. At run-time, the previously commented reconfiguration infrastructure allows selecting the specific component implementation that has to be used.

From these views, the UML-MARTE framework generates the implementation code as well as all the information required to estimate system behaviour and performance [HMV2017].

7.4 Component Abstractions for Time Sensitive Networks

This section introduces an abstraction of the Time Sensitive Network (TSN= station component). A TSN station can be classified as TSN bridge if it is able to forward TSN streams with required real-time Quality-of-Service (RT-QoS) (bounded latency, guaranteed bandwidth). Otherwise, it is designed as TSN end-station. Consequently, TSN end-stations can be assimilated as TSN bridges with no forwarding capability.

The user should specify RT-QoS traffic objects through the configuration API. Traffics are recognized by a combination of protocol fields and encapsulated into multicast VLAN frames, conforming TSN streams.

A TSN station should support the following capabilities:

1) Identification and prioritization of entering user traffics
2) Forwarding policy to entering TSN streams. TSN streams can be forwarded to other ports or VLAN-stripped to be handed to the user application layer.
3) Execution of the generalized Precision Time Protocol (gPTP, [IEEE802.1AS-2011]). This includes the capability of
    a) collaborating in the election of the network time reference (grandMaster),
    b) self-synchronization to the grandMaster
    c) Time synchronization event message forwarding, with corresponding correction to the timestamp generated at the grandMaster.

These functionalities are implemented through three major functional subsystems, which require specific configuration.

First, the VLAN sub-module should identify entering traffics, i.e. untagged user traffics or forwarded TSN streams. On the one hand, untagged user traffics should match any of the user-provided combination of protocol fields to apply specific RT-QoS or routing policy. On the other hand, forwarded TSN streams should also be identified to apply the corresponding forwarding rule. The configuration of this submodule is specified on the `vlan_entry` budget.

Second, the TAS sub-module should perform strict time-and-priority-driven cyclic scheduling of the output bandwidth. Egressing TSN streams are queued according to their priority and released following a strict time-driven cyclic schedule. Therefore, a scheduling table consist of a time interval and a list with the opened and closed gates, as exposed on the `sched_config` budget.

Third, the timing sub-module should execute gPTP and eventually forward time synchronization information to attached stations. Furthermore, it is responsible to spread network timing to local entities. It requires configuration to send protocol messages at a certain frequency besides information to be elected as grandMaster (priorities).

Hence, a VLAN budget can be captured as follows.

```
budget vlan_config {
  property vlan_tag;
  property traffic_protocol_fields;
  property forwarding_policy;
}

budget sched_config {
  property gatelist_cfg;
  property interval_time;
}

budget traffic_cfg {
  property vlan_config[N];
  property sched_config[M];
}

budget gPTP_cfg {
  property protocol_message_periodicity;
  property priorities;
}

channel TSN_stream {
  property RT-QoS;
  property vlan_tag;
}

channel user_traffic {
  property traffic_protocol_fields;
}

channel gPTP_sync_info {
  property origin_timestamp;
  property correction_field;
}

component TSN_station {
  requires traffic_cfg cfg_config;
  requires gPTP_configuration gptp_config;

  inputs TSN_streams TSN_in “or” user_traffics ustrf_in;
  inputs gPTP_sync_info sync_in;

  outputs TSN_streams TSN_out “or” user_traffics ustrf_out;
  outputs gPTP_sync_info sync_out;

  property synchronization_accuracy;
  property traffic_differentiation_and_prioritization;
  property output_bandwidth_time_driven_scheduling;
}
```

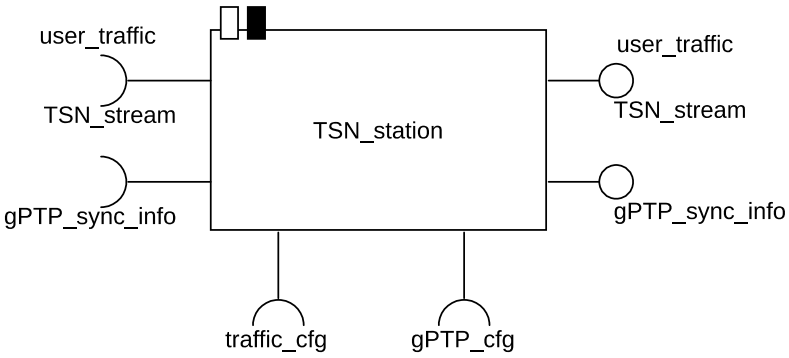


Figure 34. Model of a TSN station

7.5 Component Abstractions for High-availability Seamless Redundancy in Remote Terminal Units

The Surveillance of Smart grid critical infrastructure use case (UC9) defines a High-availability Seamless Redundancy (HSR) network component to ensure the exchange of information between different Remote Terminal Units (RTU) that allow the communications with the other components of the use case.

An abstract definition of this component is shown below:

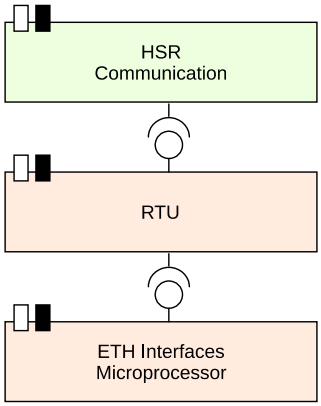


Figure 35 HSR Abstract definition

The main objective of this component is to avoid the interruption in the communications, even for minimal times, because in critical infrastructure that is unacceptable. To deal with this need, redundant communication must be included. In an abstract way the requirement to implement redundancy is to have additional links for communication and a redundancy control protocol to administrate the links.

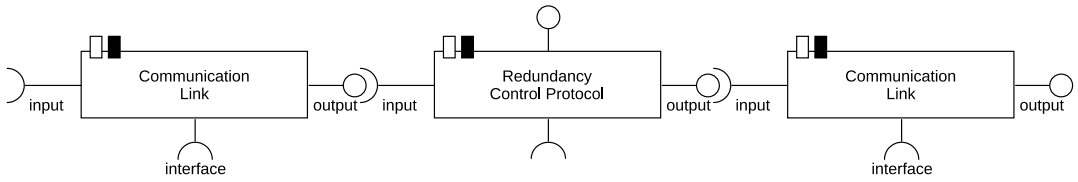


Figure 36 Components for HSR implementation in RTU

```
budget interface {
  property transmission delay
  property availability
  property jitter
  property packet loss
  property bit rate
  property latency
}
```

Figure 37: Budget for HSR implementation in RTU

```
component communication link {
  requires interface
  inputs communication packet
  outputs communication packet
}
```

Figure 38: Communication link component for HSR implementation in RTU

```
component redundancy control protocol {
  requires microprocessor
  inputs communication packet
  outputs communication packet
  provides 0 time recovery in a single fail
}
```

Figure 39: Redundancy control protocol component for HSR implementation in RTU

7.6 Component Abstractions for People Tracking System

This section includes a basic overview of the components of the people tracking system. The detection and tracking of people have always been an important area of study in the field of computer vision. However, computer limitations derived from algorithms with high computational requirements have been a limit to real-time performance. Today, the new approaches proposed have led to significant improvements in this field and allow us to develop more reliable and efficient systems.

The main objective of this system is both to detect the different human subjects that may appear on the scene during video recording given by n cameras in situations likely to

include people and to monitor and track their situation within the area of interest of the recording. Thus, this system includes two fundamental tasks that have been called "*Human detection*" and "*People tracking*" according to their purpose within the system. Thus, these tasks, in turn, determine the two fundamental abstract components of the system, as shown in Figure 40.

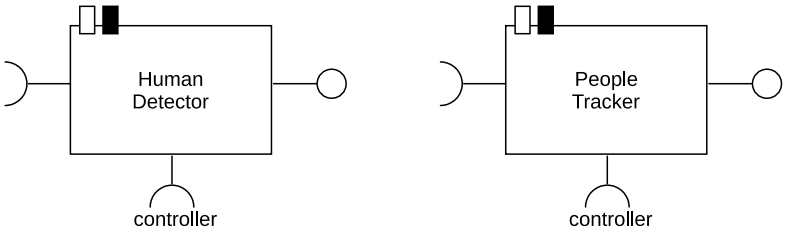


Figure 40: Components for Person Tracking System

The human detector component (Figure 42) is responsible for carrying out a detection of the different people within each of the frames of the video stream that receives as input. Thus, the output of this component will be the detections made on each of the analysed frames.

The people tracker component (Figure 40) is responsible for monitoring the people detected by the human detector. Depending on the amount of camera perspectives available for a particular scene, this component can track people in the video (a single camera perspective) and can track people in the 2D plane of the ground (more than one camera perspective for the same scene). We call this last feature World Tracking. Moreover, the existence of more than one camera perspective allows the execution of the occlusion handler, since different camera views allow to better detect targets that may be partially overlaid.

```
budget controller {
  property memory;
  property core_count;
  property gpu_core_count;
}

channel video_stream {
  property resolution;
  property fps;
}

channel detection_frame {}

channel person_position {
  property video_position;
  property world_position;
}
```

Figure 41: Components for Person Tracking System

```
component human_detector {
  requires controller cnt {}

  inputs video_stream raw;
  outputs detection_frame df, video_stream raw;
}
```

Figure 42: Human detector component for people tracking system

7.7 Component Abstractions for Action Recognition

This section shows the main components that take part in the use case of Habit Tracking (UC3). The habit tracking will be performed using techniques of action recognition. The issue of solving the problem of identifying different human actions through video analysis has gained importance in recent years. So, one of these techniques will be part of our main component in this regard.

The problem of recognizing the different actions that a person can perform at their own homes could be very interesting because if we focus mainly on using this for elderly people, the system would be able to record the activities they carry out and, for example, detect those actions that may have harmful consequences for them.

Despite the different technologies available that can be helpful for monitoring people, using just video analysis for detecting actions can provide a successful approximation considering works published in the state of the art within other fields of application.

In relation to the main components that take part in this system, we consider one main component related to the task of action recognition, which the main objective will be to describe and detect the action performed in a video stream provided by the camera component. Thus, Figure 43 shows a graphical representation of the components mentioned before. We should consider that inside the action recognizer controller, depending on the resources that we have and the performance that we want to get, there are several set points that offer different *accuracy*, *precision*, and *recall* considering more or less resource consumption. The qualities are better or worse depending on the number assigned in set point from 1 to 5, number 5 being the best that can be reachable, and 1 the worst.

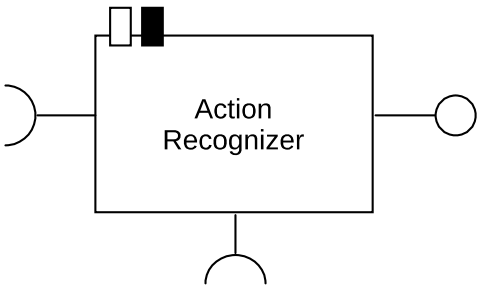


Figure 43: Action recognizer component

```
budget controller {
  property memory;
  property core_count;
  property gpu_core_count;
}

channel video_stream {
  property resolution;
  property fps;
}

channel classification_output {}
```

```
component action_recognizer {
  requires controller cnt { }
  quality performance;
  quality accuracy;
  quality precision;
  quality recall;

  inputs video_stream raw;
  outputs classification_output out, video_stream raw;

  any [ // Definition of set points
    all [performance=5, accuracy=3, precision=2, recall=4],
    all [performance=3, accuracy=4, precision=3, recall=4],
    all [performance=1, accuracy=5, precision=5, recall=5]
  ]
}
```

8. Conclusions

In this deliverable we have laid down the initial reference architecture for FitOpTiVis. It acts as a common reference model for the diverse activities in the project to allow for general solutions to emerge from those activities.

The ingredients of the architecture and the structure of the architecture have been introduced, namely, the component abstraction, a DSL to describe systems and components, a semantics to the model in terms of compositions, parameters, constraints and multi-objective optimization criteria. The architecture and the various aspects of the models are illustrated with examples related to the use cases. The architecture defines a structured outline for resource and quality management and the virtualization techniques required to realize it.

Next steps are to consolidate the DSL with tool support and to describe use cases, design-time methods, run-time methods and components with it in collaboration with other work packages. These activities lead to a second iteration of the reference architecture that will be presented in Deliverable D2.2.

9. References

[AGB+2016] H. A. Ara, M. Geilen, T. Basten, A. Behrouzian, M. Hendriks, and D. Goswami: Tight temporal bounds for dataflow applications mapped onto shared resources. In Proceedings of SIES 2016, Krakow, Poland, 2016

[BBB+2018] R. Ballouli, S. Bensalem, M. Bozga, J. Sifakis: Programming Dynamic Reconfigurable Systems. In Proceedings of FACS 2018, Pohang, South Korea, 2018

[BBS2006] A. Basu, M. Bozga, J. Sifakis: Modeling heterogeneous real-time systems in BIP. In Proceedings of SEFM 2006, Pune, India, 2006

[BCL+2006] Bruneton, E., Coupaye, T., Leclercq, M., Quema, V., Stefani, J.-B.: The Fractal component model and its support in Java. Software: Practice & Experience. 36, 1257–1284, 2006

[BDLT2019] S. S. Bhattacharyya, E. F. Deprettere, R. Leupers, J. Takala: Handbook of Signal Processing Systems, Springer, Cham, 2019

[BGH+2013] T. Bureš, I. Gerostathopoulos, P. Hnětynka, J. Keznikl, M. Kit, F. Plášil: DEECo – an Ensemble-Based Component System, in Proceedings of CBSE'13, 2013

[BHP2006] T. Bures, P. Hnetynka, and F. Plasil. SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model. In Proceedings of SERA 2006, Seattle, USA, 2006

[BJMS2012] M. Bozga, M. Jaber, N. Maris, J. Sifakis: Modeling dynamic architectures using Dy-BIP. In Proceedings of SC 2012, Prague, Czech Republic, 2012

[BS2008] S. Bliudze, J. Sifakis: The algebra of connectors structuring interaction in BIP. IEEE Transactions on Computers 57(10):1315–1330, 2008

[CDM+2012] E. Cannella, O. Derin, P. Meloni, G. Tuveri and T. Stefanov, “Adaptivity Support for MPSoCs Based on Process Migration in Polyhedral Process Networks,” Hindawi VLSI Design, 2012.

[CGK+2018] E. M. Clarke, O. Grumberg, D. Kroening, D. Peled and H. Veith: Model Checking, 2nd edition, The MIT Press, 2018

[CL2014] J. Castrillon and R. Leupers: “Programming Heterogeneous MPSoCs,” Springer, Cham, 2014

[CSC+2009] J. Ceng, W. Sheng, J. Castrillon, A. Stulova, R. Leupers, G. Ascheid and H. Meyr: “A high-level virtual platform for early MPSoC software development,” in Proceedings of CODES+ISSS '09, Grenoble, France, 2009

[DPN+2013] K. Desnos, M. Pelcat, J.-F. Nezan, S. S. Bhattacharyya and S. Aridhi: “PiMM: Parameterized and Interfaced dataflow Meta-Model for MPSoCs runtime reconfiguration,” in Proceedings of SAMOS 2013, Agios Kostantinos, Greece, 2013

[DS2016] A. Diaz and P. Sanchez, “Simulation of attacks for security in wireless sensor network”. Sensors, 16(11):1932, 2016

[DSG+2012] M. Damavandpeyma, S. Stuijk, M. Geilen, T. Basten, H. Corporaal, H., Parametric throughput analysis of scenario-aware dataflow graphs, Computer Design (ICCD), 2012 IEEE 30th International Conference on, 2012, pp.219-226

[FGH2006] Feiler, P.H., Gluch, D.P. and Hudak, J.J.: The Architecture Analysis & Design Language (AADL): An Introduction. Technical Report, CMU/SEI-2006-TN-011, 2006

[FJE+2017] F Herrera, J Medina, E Villar, Modeling Hardware/Software Embedded Systems with UML/MARTE: A Single-Source Design Approach. Handbook of Hardware/Software Codesign, 141-185. 2017.

[FSL+2002] J.-P. Fassino, J.-B. Stefani, J. Lawall, and G. Muller. Think: A Software Framework for Component-based Operating System Kernels. In Proceedings of the 2002 USENIX Annual Technical Conference, Monterey, California, USA, June 2002.

[GAC+2013] K. Goossens, A. Aevedo, K. Chandrasekar, M. Dev Gomony, S. Goossens, M. Koedam, Y. Li, D. Mirzoyan, A. Molnos, A. Beyranvand Nejad, A. Nelson and S. Sinha: “Virtual execution platforms for mixed-time-criticality systems: the CompSOC architecture and design flow,” ACM SIGBED Review 10(3):23-34, 2013

[GBTO2007] M. Geilen, T. Basten, B. Theelen, and R. Otten: An algebra of Pareto points. Fundamenta Informaticae 78(1):35-74, 2007

[GHP+2009] A. Gerstlauer, C. Haubelt, A. D. Pimentel, T. P. Stefanov, D. D. Gajski and J. Teich: “Electronic system-level synthesis methodologies,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 28(10), 2009

[GS2003] T. Grandpierre and Y. Sorel, “From Algorithm and Architecture Specifications to Automatic Generation of Distributed Real-Time Executives: a Seamless Flow of Graphs Transformations,” in Proceedings of MEMOCODE 2003, Mont Saint Michel, France, 2003

[JSS+15] P. Jääskeläinen, C. Sánchez de La Lama, E. Schnetter, K. Raiskila, J. Takala, H. Berg: pocl: A Performance-Portable OpenCL Implementation, International Journal of Parallel Programming 43(5): 752–785, 2015

[HAG+2019] M. Hendriks, H. A. Ara, M. Geilen, et al. Monotonic optimization of dataflow buffer sizes. Journal of Signal Processing Systems 91(1):21-32, 2019

[HBP+2009] P. Hnětynka, T. Bureš, M. Prochazka, R. Ward, Z. Hanzálek: SOFA High Integrity: Our Approach to SAVOIR, in Proceedings of DASIA 2009 - DAta Systems in Aerospace, 2009

[HBV+2016] M. Hendriks, T. Basten, J. Verriet, M. Brassé, L. Somers. A Blueprint for System-Level Performance Modeling of Software-Intensive Embedded Systems. International Journal on Software Tools for Technology Transfer, STTT. 18(1):21-40, February 2016

[HMV2017] F. Herrera, J. Medina, E. Villar: Modeling Hardware/Software Embedded Systems with UML/MARTE: A Single-Source Design Approach. In Handbook of Hardware/Software Codesign, Springer, 2017

[HPD+2014] J. Heulot, M. Pelcat, K. Desnos, J.-F. Nezan and S. Aridhi: “SPIDER: A Synchronous Parameterized and Interfaced Dataflow-Based RTOS for Multicore DSPs,” in Proceedings of EDERC 2014, Milan, Italy, 2014

[IEEE802.1AS-2011] IEEE Standard for Local and Metropolitan Area Networks - Timing and Synchronization for Time-Sensitive Applications in Bridged Local Area Networks," in IEEE Std 802.1AS-2011 , pp.1-292, 30 March 2011

[KDW+2002] B. Kienhuis, E. F. Deprettere, P. van der Wolf and K. Vissers: “A methodology to design programmable embedded systems,” in Embedded processor design challenges, SAMOS 2001, LNCS 2268, 2002

[KRS+2009] Ji Eun Kim, O. Rogalla, S. Kramer, and A. Hamann. Extracting, specifying and predicting software system properties in component based real-time embedded software development. In Proceedings of ICSE 2009, Vancouver, Canada, 2009

[KS2004] V. Kianzad and S. S. Bhattacharyya, “CHARMED: A multi-objective cosynthesis framework for multi-mode embedded systems,” in Proceedings of ASAP2004, Galveston, TX, USA, 2004

[LM87] E. Lee, D. Messerschmitt: Synchronous Data Flow, IEEE Proceedings 75(9): 1235-1245, 1987

[MK1996] J. Magee and J. Kramer. Dynamic structure in software architectures. In Proceedings of SIGSOFT FSE’96, San Francisco, CA, USA, pages 3–14. ACM, 1996.

[MMT1995] B. M. Maggs, L. R. Matheson, and R. E. Tarjan: “Models of parallel computation: A survey and synthesis,” in Proceedings of HICSS 1995, Wailea, HI, USA, 1995

[MO2014] O. Moreira, H. Corporaal: Scheduling Real-Time Streaming Applications onto an Embedded Multiprocessor, Springer, Cham, 2014

[MV2017] J. L. Medina and E. Villar, Towards MARTE++: an enhanced UML-based language to Model and Analyse Real-Time and Embedded Systems for the IoT age, FDL 2017

[NLFS2018] P. Nuzzo, M. Lora, Y. A. Feldman, and A. Sangiovanni-Vincentelli: “CHASE: Contract-based requirement engineering for cyber-physical system design,” in Proceedings of DATE, Dresden, Germany, 2018

[NSSP2012] P. Nuzzo, A. Sangiovanni-Vincentelli, X. Sun, A. Puggelli: Methodology for the design of analog integrated interfaces using contracts. IEEE Sensors Journal 12(12):3329–3345, 2012

[NS2018] P. Nuzzo, A. Sangiovanni-Vincentelli: Hierarchical System Design with Vertical Contracts. In Principles of Modeling, LNCS 10760, Springer, Cham, 2018

[OLK+2002] R. Ommering, F. Linden, J. Kramer, and J. Magee. The Koala Component Model for Consumer Electronics Software. Computer, 33(3):78–85, 2000

[OMG2017a] OMG: SysML – System Modeling Language, OMG document formal/17-05-01, 2017

[OMG2017b] OMG: UML – Unified Modeling Language, v 2.5.1, OMG document formal/17-12-05, 2017

[OMG2018] OMG: UML Profile for MARTE: Modelling and Analysis of Real-Time Embedded Systems, v. 1.2 beta, OMG document ptc/18-07-03, 2018

[P1971] V. Pareto. Manuale di Economia Politica. Piccola Biblioteca Scientifica, Milan, 1906. Translated into English by Ann S. Schwier, Manual of Political Economy, MacMillan, London, UK, 1971

[PDH+2014] M. Pelcat, K. Desnos, J. Heulot, C. Guy, J.-F. Nezan and S. Aridhi: “PREESM: A Dataflow-Based Rapid Prototyping Framework for Simplifying Multicore DSP Programming,” in Proceedings of EDERC, Milan, Italy, 2014

[PMD+2017] M. Pelcat, A. Mercat, K. Desnos, L. Maggiani, Y. Liu, J. Heulot, J.-F. Nezan, W. Hamidouche, D. Menard, S. Bhattacharyya: “Reproducible Evaluation of System Efficiency with a Model of Architecture: From Theory to Practice,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 37(10), 2018

[SB200] S. Sriram, Shuvra S. Bhattacharyya, Embedded Multiprocessors: Scheduling and Synchronization, Second Edition, Marcel Dekker, Inc., 2009

[SGTB2011] S. Stuijk, M.C.W. Geilen, B.D. Theelen, and T. Basten, Scenario-Aware Dataflow: Modeling, Analysis and Implementation of Dynamic Applications, In: Proc. of the International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (IC-SAMOS), pp. 404-411, 2011

[SVB+2008] S. Sentilles, A. Vulgarakis, T. Bureš, J. Carlson, I. Crnković: A Component Model for Control-Intensive Distributed Embedded Systems, in Proceedings of CBSE 2008, Karlsruhe, Germany, Springer, 2008

[TCN2000] L. Thiele, S. Chakraborty and M. Naedele, "Real-time calculus for scheduling hard real-time systems," 2000 IEEE International Symposium on Circuits and Systems. Emerging Technologies for the 21st Century. Proceedings, Geneva, Switzerland, 2000, pp. 101-104 vol.4.

10. Appendix A Grammar of the DSL

Here follows the grammar of the proposed DSL in Extended Backus-Naur Form (EBNF). The version written here is optimized for reader clarity and can be used as reference point when using the language.

```
<Model>: { <Element> }

<Element>: <Import>

      | <BudgetDefinition>

      | <ChannelDefinition>

      | <ComponentDefinition>

      | <SystemDefinition>

<Import>: “import” “(“ <StringLiteral> “)” “;”

<BudgetDefinition>: “budget” <ID>

      “{“ { <QualityDefinition> } “}”

<ChannelDefinition>: “channel” <ID>

      “{“ { <QualityDefinition> } “}”

<QualityDefinition>: “quality” <ID> “;”

<ComponentDefinition>: “component” <ID>

      “{“ ( <DefaultConfiguration> | <Configurations> ) “}”

<DefaultConfigruation>: <ConfigurationBody>

<Configurations>: { <Configuration> }

<Configuration>: “configuration” <ID>

      “{“ <ConfigurationBody> “}”

<ConfigurationBody>: { <ComponentRule> “;” }

<ComponentRule>: <SupportsPredicate>

      | <RequiresPredicate>

      | <InputsPredicate>
```

```
      | <OutputsPredicate>
      | <PropertyPredicate>
      | <SubcomponentPredicate>
      | <ConstraintPredicate>
<System>: “system” <ID> “{“
      {( <SubcomponentPredicate> | <ConstraintPredicate> )} “}”
<SupportsPredicate>: “supports” <InterfaceUsagePredicate>
<RequiresPredicate>: “requires” <InterfaceUsagePredicate>
<InputsPredicate>: “inputs” <InterfaceUsagePredicate>
<OutputsPredicate>: “outputs” <InterfaceUsagePredicate>
<SubcomponentPredicate>: “component” <ID> <ID>
                        [ <ArrayIndex> ]
<ArrayIndex>: “[“ <Expression> “]”
<InterfaceUsagePredicate>: <ID> [ <ID> ] [ <ArrayIndex> ]
      [ <InterfaceUsageConstraints> ]
<InterfaceUsageConstraints>:
      “{“ { <ConstraintPredicate> “;” } “}”
<PropertyPredicate> :
      ( “quality” | “property” | “parameter” ) <ID>
      [ “=” <Expression> ]
<ConstraintPredicate>: <AndPredicate>
                  | <OrPredicate>
                  | <ImplicationPredicate>
                  | <RunsOnPredicate>
                  | <OutputsToPredicate>
                  | <BooleanExpression>
<AndPredicate>: “all” “[“ { <ConstraintPredicate> “,” }
```

```
                [ “,” ] “]”
<OrPredicate>: “any” “[“ { <ConstraintPredicate> “,” }
                [ “,” ] “]”
<ImplicationPredicate>: <BooleanExpression> “=>"
                            <ConstraintPredicate>
<RunsOnPredicate>: <QualityExpression> “runs” “on”
                        <QualityExpression>
<OutputsToPredicate>: <QualityExpression> “outputs” “to”
                        <QualityExpression>
<BooleanExpression>: <InExpression>
           | <UnaryBooleanOperator> <BooleanExpression>
           | <ComparisonExpression>
            { <BinaryBooleanOperator> <ComparisonExpression> }
<InExpression>: <Expression> “in” <InlineArrayExpression>
<ComparisonExpression>:
     <Expression> <ComparisonOperator> <Expression>
<Expression>: <AdditiveExpression>
           | <InlineArrayExpression>
           | <InlineObjectExpression>
<InlineArrayExpression>: “[“ <Expression>
           { “,” <Expression> } “]”
<InlineObjectExpression>: “{“ <InlineObjectMemberExpression>
           { “,” <InlineObjectMemberExpression> } “}”
<InlineObjectMemberExpression>: <ID> “=” <Expression>
<AdditiveExpression>: <MultiplicativeExpression>
           { <AdditiveOperator> <MultiplicativeExpression> }
<MultiplicativeExpression>: <Term>
```

```
          { <MultiplicativeOperator> <Term> }

<Term>: <BracketedExpression>

      | <UnaryExpression>

      | <QualityExpression>

      | <CallExpression>

      | <Literal>

<BracketedExpression>: “(“ <AdditiveExpression> “)”

<UnaryExpression>: <UnaryOperator> <Term>

<QualityExpression>: <ArrayAccessExpression>

      | <SubQualityAccessExpression>

      | <ID>

<ArrayAccessExpression>: <QualityExpression>

      “[“ <Expression> “]”

<SubQualityAccessExpression>: <QualityExpression> “.” <ID>

<CallExpression>: <ID>

      “(“ [ <Expression> { “,” <Expression> } ] “)”

<Literal>: <IntLiteral>

      | <StringLiteral>

<UnaryBooleanOperator>: <LogicalNot>

<LogicalNot>: “!”

<BinaryBooleanOperator>: “&&” | “||”

<ComparisonOperator>: “==” | “<” | “>” | “<=” | “>=” | “!=”

<AdditiveOperator>:             “+”             |             “-”
<MultiplicativeOperator>: “*” | “/”

<UnaryOperator>: “+” | “-”

<StringLiteral>: ‘”’ { <Character> } ‘”’

<ID>: ( <Letter> | “_” ) { <Letter> | “_” | <Digit> }
```

```
<IntLiteral>: <Digit> { <Digit> }

<Digit>: “0” | “1” | “2” | “3” | “4” | “5” | “6” | “7” | “8”
      | “9”
```

<Letter> is any letter of the Latin alphabet (in regular expression notation [a-zA-Z]), <Character> is any valid printable character except for quotation mark and backslash, which must be escaped with backslash.