Component Definition Language

Franca+

User Guide

**Draft for V0.13.0**

Copyright

© 2018 BMW Group

www.bmw.com

This document and the accompanying software/materials are made available under the terms of the Eclipse Public License v1.0 which accompanies this distribution, and is available at http://www.eclipse.org/legal/epl-v10.html.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing. This software is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications which may create a risk of personal injury. If you use this software in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure the safe use of this software. BMW Group disclaims any liability for any damages caused by use of this software in dangerous applications.

Inhalt

[1 Introduction 5](#_Toc513480680)

[2 Getting Started 5](#_Toc513480681)

[2.1 Source Code and Building 5](#_Toc513480682)

[2.2 Editor 5](#_Toc513480683)

[2.2.1 Examples and Hello World 6](#_Toc513480684)

[3 Concepts 6](#_Toc513480685)

[4 Component Model 7](#_Toc513480686)

[4.1 Package 7](#_Toc513480687)

[4.2 Naming 7](#_Toc513480688)

[4.3 Component 8](#_Toc513480689)

[4.3.1 Version 8](#_Toc513480690)

[4.3.2 Composition 8](#_Toc513480691)

[4.3.3 Port 8](#_Toc513480692)

[4.3.4 Connectors 9](#_Toc513480693)

[4.3.5 Inheritance 10](#_Toc513480694)

[4.3.6 Attributes 10](#_Toc513480695)

[4.4 Device 11](#_Toc513480696)

[4.4.1 Adapter 11](#_Toc513480697)

[4.5 Comments 12](#_Toc513480698)

[4.5.1 Tags 12](#_Toc513480699)

[5 Deployment 12](#_Toc513480700)

[5.1 Package 13](#_Toc513480701)

[5.2 Naming 13](#_Toc513480702)

[5.3 Use 13](#_Toc513480703)

[5.4 Component 14](#_Toc513480704)

[5.5 Service 14](#_Toc513480705)

[5.5.1 Port Deployment 15](#_Toc513480706)

[5.6 Device 15](#_Toc513480707)

[5.6.1 Adapter 15](#_Toc513480708)

[5.7 Variant 16](#_Toc513480709)

[6 FRANCA+ IDE 16](#_Toc513480710)

[6.1 Import declarations 16](#_Toc513480711)

[6.2 Outline 17](#_Toc513480712)

[6.3 Reference Proposals 17](#_Toc513480713)

[7 Command line options for Franca\_SDK 17](#_Toc513480714)

[8 References 17](#_Toc513480715)

# Introduction

Architecture deals to a significant extent with interfaces. Interfaces are an essential building block for designing a system. With architecture framework Franca (<https://github.com/franca>) we can define interfaces as text files in a simple but formal way. The interface definition serves as input for code generation and is the base for a contract between consumers and providers.

The consumer and provider as distinct entities are unfortunately not part of the logical model of Franca. With the introduction of a component model Franca is extended by the ability to express services with roles as provider and consumer of interfaces.

With the Franca Component Definition Language (FCDL) a software architect is enabled to describe a system by modeling components with ports providing or requiring interfaces. He can compose components to new more complex components. The relations between components can be expressed by connectors for assembly and delegate.

With compositions a complex model can be summed up to simple-to-understand architecture.

In order to deploy the software model to a hardware model, FCDL introduces the device. A device can be seen as very generic place holder for a ECU, core, machine, whatever the target shall be. Device can also be composed to complex devices. Each device has adapters to setup communication between each other.

In the Franca deployment the architect partitions the instances of components and ports to devices and adapters.

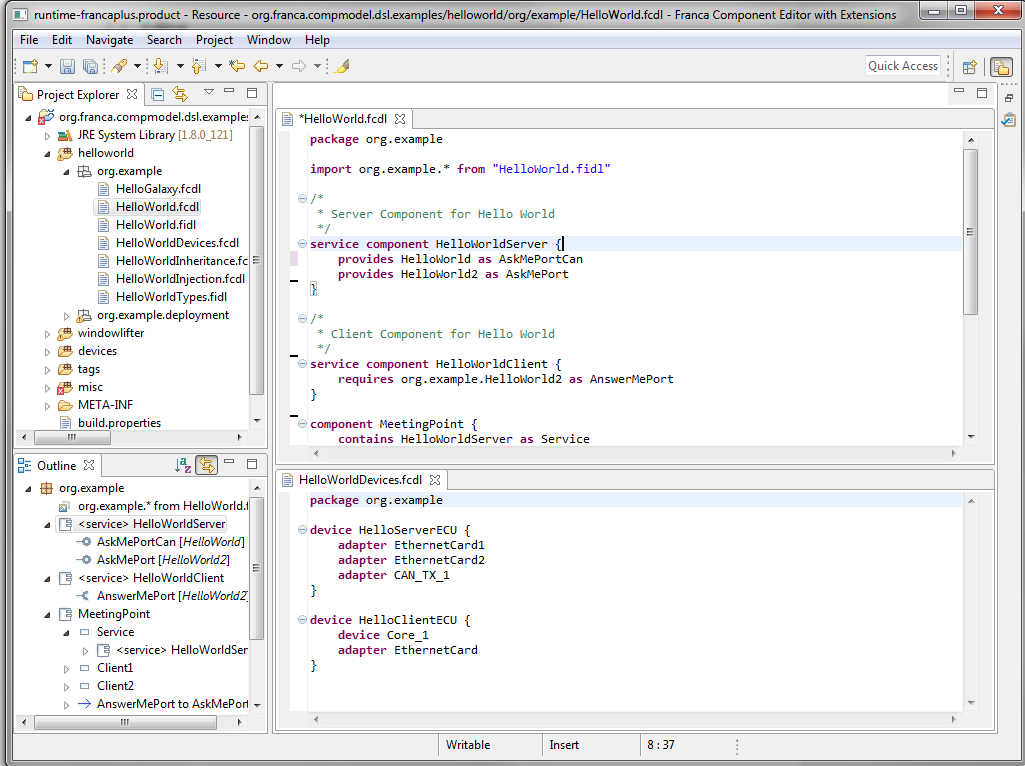
# Getting Started

## Source Code and Building

Sources for the editor and generator framework for FCDL (called FRANCA\_SDK) can be obtained from <https://github.com/GENIVI/franca_plus> together with the necessary build commands.

## Editor

The editor is Eclipse based. It can be found in the built product under *releng\org.francaplus.product\target\products\org.francaplus.product.*



Picture 1 FCDL Editor

### Examples and Hello World

FCDL is stored in files with suffix *fcdl*. The deployment for components is stored in Franca deployment files *cdepl*.

With the sources comes the example package *org.franca.examples.components.* Import it into the editor by *File -> Import -> General -> Existing Projects into Workspace*.

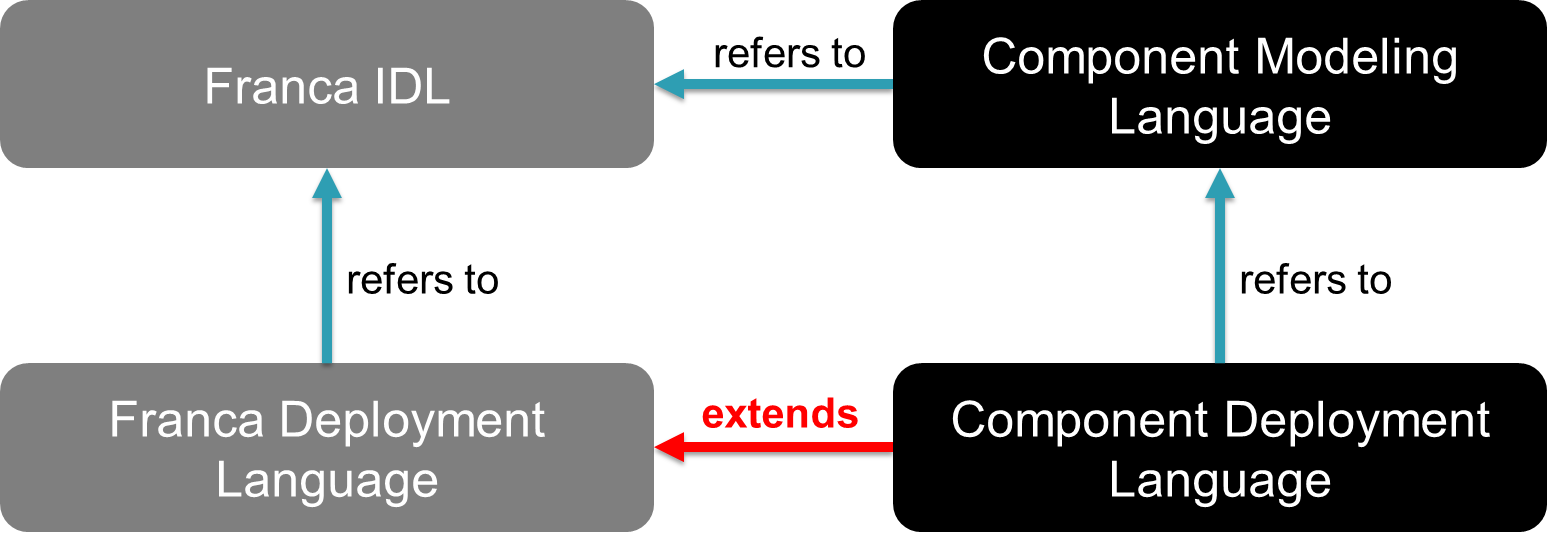
The *helloworld* package contains in the file *HelloWorld.fcdl* a simple client server example with a *HelloWorldServer* and *HelloWorldClient* components and their deployment to devices in different variant.

# Concepts

FCDL is a textual component modeling framework for software and hardware components. It provides entities to specify components with distinct responsibilities in an architecture model.

FCDL is an extension to the architecture framework Franca and uses the same language style as Franca IDL. FCDL is stored in text files with suffix *fcdl*. The Franca interfaces are used for typing the ports of components in the FCDL. For software components it supports components, composition of components, typed ports and connectors between ports. Components can be composed and inherited. Certain attributes allow to classify components, e.g. as service, root, singleton etc.

For hardware components it supports device, composition of devices and adapters. Different variations of an architecture can be expressed by variant.



Picture 2 Franca Language Relations

The deployment of components is done in a language derived from the Franca “core” deployment. The files end with *cdepl*. The complete Franca IDL deployment is part of FCDL. Fcdl files can import fdepl files. Architects define their own deployment specifications with additional property hosts for components, services, provided and required ports, devices and adapters.

The link between software and hardware components occurs in the deployment. The architect partitions instances of components (aka services) to deployed devices.

With the FCDL based deployment it is possible to define ports for different technologies in one service. For each port an optional distinct deployment specification can be given. By using different deployment specification within one service, it is possible to model services, which act as gateway between communication technologies.

The textual FCDL is designed to be can be easily transformed to a representation in, for more information see <https://at.projects.genivi.org/wiki/display/PROJ/YAMAICA+Toolset>.

# Component Model

## Package

Every Franca file belongs to a package. FCDL and Franca component deployment files start with a package statement.

## Naming

Every entity in Franca has a unique name. An entity name is always a simple name, following the expression [A-z\_]+[A-z0-9\_]\* with no dots in it. A unique name is built by embedding the entity in a namespace. A namespace is a concatenated string of segments, each pointing to a single entity. The namespace string is separated by dots.

All references in Franca are described by fully qualified names. Depending on the entity the FQN is build slighting different. IDL and FCDL entities are referenced via FQN including the package.

In the deployment component instances have a FQN starting with the system root component.

## Component

The main building block of FCDL is the component. It provides and requires interfaces as ports. A component is generally interpreted as a piece of software, which communicates with other components via ports.

A component name shall be unique in a model.

**package** org.example

/\*

\* Server Component for Hello World

\*/

**component** HelloWorldServer

### Version

A component has a version like a Franca interface. The version is split into major and minor version. Versions *can* be used to validate the compatibility of components. The version is optional.

**component** HelloWorldServer {

**version** { **major** 1 **minor** 0 }

}

### Composition

Components contain other components as composition.

A component contained in a composition is called Component Prototype. The component prototype has an own name, by default the component name.

**component** PartialService

// a composition

**component** HelloWorldServer {

**contains** PartialService **as** InnerStructure

}

### Port

A port is an owned feature of a component. A component communicates via ports with other components. The port is either of provided or required kind. Each Port is typed by exactly one interface[[1]](#footnote-1).

In the sense of Service-Orientation a provided port characterizes a component as server for the interface. A required port characterizes a component as client for an interface.

A component has zero, one or more ports. There may be more than one port of the same type in a component.

The port is distinguishable by his name. The default name of a port is the name of the interface. A distinct name can be given - especially necessary, if a component has more than one port with the same interface.

**import** org.example.\* **from** "HelloWorld.fidl"

**component** HelloWorldServer {

**provides** HelloWorld

**provides** HelloWorld **as** AskMePort

}

**component** HelloWorldClient {

**requires** org.example.HelloWorld **as** AnswerMePort

}

### Connectors

Component ports are used describe “use” relations between components. Connects can be established in two ways: either between two ports of two component prototypes contained in a composition or from a component prototype’s port to composition port.

The connectors are seen as static description of inter component usage. They do not imply any specific semantic for the runtime existence of a connection between two component instances.

#### Assembly

An assembly connector is used between component prototypes residing in the same composition. It links the required port of a component with a provided port of another component. The connection is established from the required to the provided port.

This kind of connector complies with the UML assembly connector. There may exist multiple connectors from different required ports to one provided port. For a required port only one outgoing connector possible.

The interface types of connected provided and required ports must match. It is supported that a required port connects to a port typed with an interface which is derived from the required port’s interface.

**component** MeetingPoint {

**contains** HelloWorldServer **as** Service

**contains** HelloWorldClient **as** Client1

**contains** HelloWorldClient **as** Client2

**connect** Client1.AnswerMePort **to** Service.AskMePort

**connect** Client2.AnswerMePort **to** Service.AskMePort

}

#### Delegate

A delegate connector routes a port from the outer hull of a component to a ports of a contained component prototype. Delegate always links ports of same direction, e.g. provided ports with provided or required port with required ports. The interface types of delegated ports must match.

**component** MeetingPoint {

**requires** HelloWorld **as** Outline

**provides** HelloWorld **as** Inline

**contains** HelloWorldServer **as** Service

**contains** HelloWorldClient **as** Client

**delegate** **required** Outline **to** Client.AnswerMePort

**delegate** **provided** Inline **to** Service.AskMePort

}

A required delegate can link multiple contained ports with one outer port.

A provided delegate always links one outer port with exactly one contained port (1:1).

The required and provided keywords after the delegate connector help to scope easy to the appropriate ports and prototypes.

#### Optional Connects

To guarantee the validity of the model all required ports must be connected with provided ports, by default. As an exception of this rule a required port can be denoted as optional. Optional port can remain unconnected to any providing port.

**component** HelloWorldClient {

**optional** **requires** HelloWorld **as** AnswerMePort

}

### Inheritance

A component can inherit all features (ports, connectors, delegates, component prototypes) from another component. Only single inheritance is supported.

**component** ServerExtension **extends** HelloWorldServer {

**requires** PhoneBook

}

### Attributes

Components can be declared with a couple of attributes, which modify the intended use.

#### Service

The *service* attribute marks a component as service running on the target platform. A service offers port for communication with other services. In contrast as “normal” component has only structuring character. A component marked with service is. In terms of IPC (Interprocess Communication) a service component is the executable for a process.

**service component** HelloWorldServer {

**provides** HelloWorld

**provides** HelloWorld **as** AskMePort

}

Only components marked with service are available for instantiation as service in the component deployment.

#### Root

The *root* attribute marks a component as the root component of a component hierarchy. Usually a root component is a composition of other components.

**root** **component** World {

**contains** MeetingPoint **as** Room1

**contains** MeetingPoint **as** Room2

}

For the deployment of services it is necessary to have a root component in the model. The root component’s name is the first segment of the fully qualified name of the service.

#### Abstract (experimental)

The *abstract* attribute marks a component as abstract. An abstract component and prototypes of it are not deployable.

An abstract component can be inherited from in order to get a deployable component or service.

**abstract** **component** Abstraction

#### Singleton

The *singleton* attribute marks a component as singleton. Singleton components shall be instantiated in a variant of the deployment only once.

**abstract** **component** ExistsOnlyOnce

#### Implement (experimental)

The *implement* attribute is used in conjunction with component prototypes. It allows to override the default implementation of a component prototype in a derived component.

**component** DerivedHelloWorldClient **extends** HelloWorldClient {

**implement** Chatter **as** InjectedChatter **by** DerivedChatter **finally**

}

The optional modifier *finally* shall prohibit overwrite by implement in derived class.

## Device

In order to have complete software model, it might be necessary to map the instances of services to some hardware. For this purpose FCDL has the *device* component and the *adapters*.

A device component has a name and might contain other devices. Each device can have zero, one or more adapters. E.g. an ordinary CPU is modeled as device.

**device** HelloECU {

**device** Core\_1

**device** Core\_2

}

In the deployment the architect partitions a service component instance to a device. The service component is hereby meant to run on that device.

### Adapter

The adapter of a device is a communication endpoint. It represent for example an Ethernet card or a CAN transceiver. Special properties of the endpoint are defined by the deployment.

**device** HelloECU {

**device** Core\_1 {

**adapter** EthernetCard

}

**device** Core\_2 {

**adapter** EthernetCard

}

}

In the deployment the architect assigns the ports of the partitioned services to the adapters of the device.

## Comments

FCDL supports the unstructured and structured comments of Franca.

### Tags

FCDL extends the structured comments of Franca by adding the possibility to define custom tags.

**package** architecture

// for structuring

**tag** **String** *@system*

**tag** **String** *@domain*

**tag** **String** *@application*

// for tooling

**tag** **String** *@guid*

The custom tags can be used like the built-in tags in Franca structured comments. They are made available like ordinary imports.

**import** **model** "platform:plugin/architecture/Tags.fcdl"

<\*\*

*@guid: AX12-WKK6-0FOD-KOWL*

*@domain: infrastructure*

\*\*>

**component** Logger {

**provides** log

**provides** admin

}

# Deployment

For deployment of FCDL is a new component deployment language – CDEPL – is used. It is derived from Franca deployment, so all feature of Franca deployment are included.

A typical deployment definition is:

**package** org.example.deployment

**import** "platform:/plugin/org.franca.someip/network\_SOMEIP\_spec.cdepl"

**import** "platform:/plugin/org.franca.architecture/ /architecture \_spec.cdepl"

**import** "../HelloWorld.fcdl"

**import** "Interface.cdepl"

**define** system.architecture **for** **service** World.Room1.Service **as** HelloService {

**provide** AskMePort **on** network.someip {

**use** org.example.deployment.SomeIpDeployment4HelloWorld

SomeIpInstanceID = 1

SomeIpReliableUnicastPort = 30506

SomeIpUnreliableUnicastPort = 0

}

}

It defines a deployment for a service with the specification system.architecture giving it the name HelloService and the component port AskeMePort with a SOME/IP binding.

The term “element X is deployed” means there exists a deployment definition for element X.

## Package

Each deployment file belongs to package and starts with a package declaration.

## Naming

All deployment definitions have a unique name. The deployment definition name is prefixed with the package name of the CDEPL file.

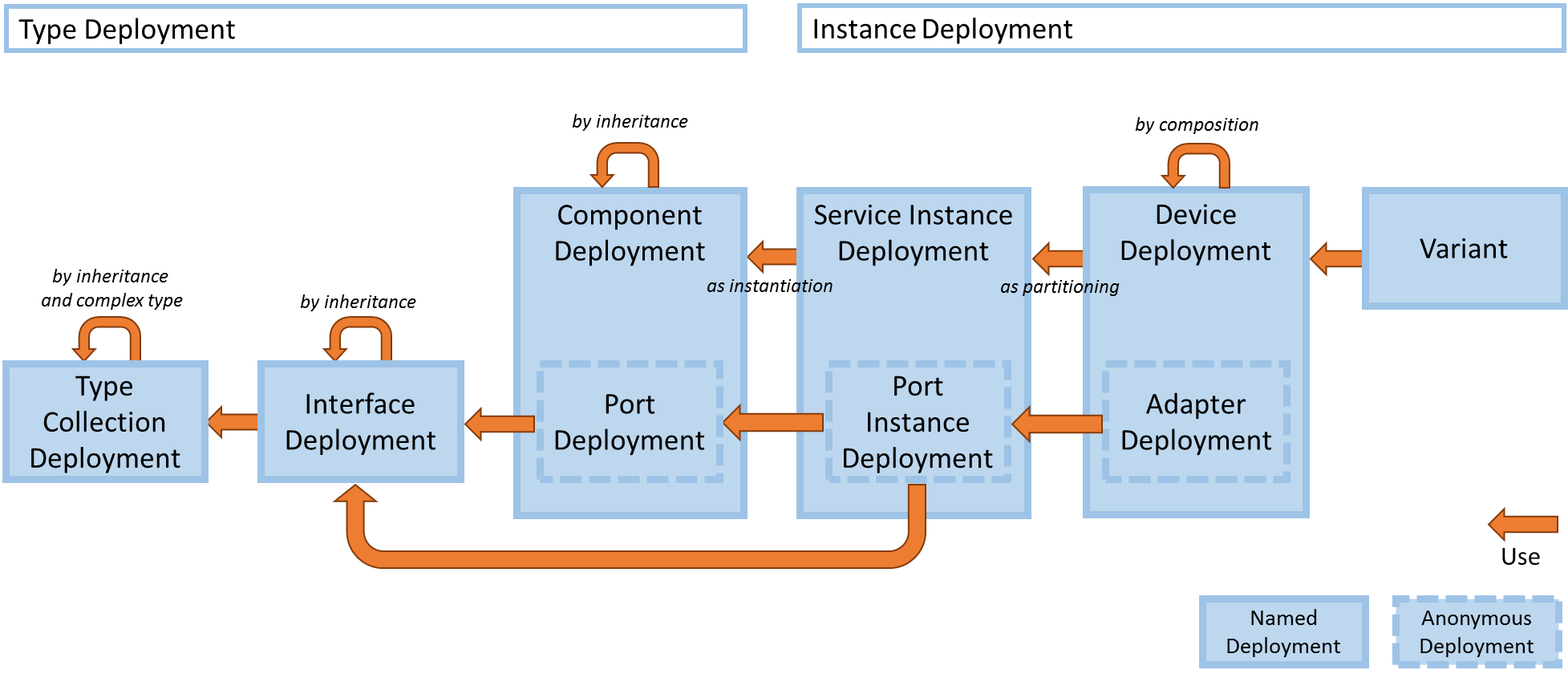
If the user omits the explicit naming of a deployment definition, the Franca\_SDK generates an implicit name by suffixing the deployed element with \_depl.

In the example above the service and its deployment definition have the same fully qualified name org.example.deployment.HelloService.

## Use

Each deployment definition can be *used* by reference with name. E.g. an interface deployment definition is used either by a derived interface deployment or by a service port deployment definition.

Again in the example above, the deployment definition of the interface org.example.deployment.SomeIpDeployment4HelloWorld is used to add the properties of the type deployment to the instance of the service port.



Picture 3 Use of Deployment Definitions

The content assistant of the editor offers the semantically correct deployment definition for use.

## Component

A component can be deployed as type deployment in the following way.

**package** org.example.deployment

**import** "platform:/plugin/org.franca.someip//network\_SOMEIP\_ spec.cdepl"

**import** "../HelloWorld.fcdl"

// deployment for the component type

**define** network.someip **for** **component** org.example.HelloWorldClient {

**require** AnswerMePort **on** network.someip

}

Most time we will directly deploy an instances of a component aka service.

## Service

The most important deployment definition is the service. A service is an instance of a component marked with attribute service in the context of root container component.

A service definition and each of the port definitions may have different deployment specifications. If For a port with no deployment specification given, the specification of the service is used.

The path through the hierarchy of the logical model identifies the instance of the service in the example as World.Room1.Service. The instance name always starts with root, then followed by (non-abstract) prototypes and must end with a prototype of a component with attribute service.

**package** org.example.deployment

**import** "platform:/plugin/org.franca.someip/network\_SOMEIP\_spec.cdepl"

**import** "platform:/plugin/org.franca.architecture/architecture\_spec.cdepl"

**import** "../HelloWorld.fcdl"

**import** "Interface.cdepl"

**define** system.architecture **for** **service** World.Room1.Service **as** Server {

StartupAtSystemInit = **true**

**provide** HelloWorld **on** network.someip {

**use** org.example.deployment.SomeIpDeployment4HelloWorld

SomeIpInstanceID = 1

SomeIpReliableUnicastPort = 30506

SomeIpUnreliableUnicastPort = 0

}

}

By naming the instance as Server the fully qualified name is org.example.deployment.Server. The name without extra name Server would be org.example.deployment.Service\_depl.

### Port Deployment

Ports are either deployed as *provide* or *require* depending on the definition in the logical model.

The deployment example of the port HelloWorld is explicitly noted to be for network.someip. It *uses* properties from the interface type deployment.

If the interface would be inherited the architect would have to denote multiple use statements. For each definition of an inherited interface an own use is necessary.

The specification system.architecture has no knowledge about the network.someip properties. The property SomeIpInstanceID from the network.someip specification is a mandatory property. The service component has a second port AskMePort. If the service would be deployed for network.someip, then the missing deployment leads to an error, because the specification for the port is derived from the service and the property SomeIpInstanceID is not set. But with the specification of system.architecture the deployment is valid.

## Device

By deploying devices the architect can declare, which services run on which hardware. The deployment simply uses a deployment spec and partitions services onto it by use statement.

**define** system.architecture **for** **device** org.example.HelloECU.Core\_1 {

**use** org.example.deployment.Service

}

### Adapter

The adapter binds the deployed ports of a service to a physical communication endpoint. Again the use statement does the work.

**define** system.architecture **for** **device** org.example.HelloECU.Core\_1 {

**use** org.example.deployment.Service

**adapter** EthernetCard1 **on** network.someip {

**use** org.example.deployment.Service.HelloWorld

SomeIpUnicastAddress = "127.0.0.1"

}

}

It might be helpful to plan in the specification for a communication binding a property like IsDefaultAdapter. It can be used to denote for all not explicitly bound ports to bind implicitly to the adapter with that property.

**adapter** EthernetCard1 **on** network.someip {

IsDefaultAdapter =true

SomeIpUnicastAddress = "127.0.0.1"

}

## Variant

An architecture may exists in various variants. These variants can be declared and assembled with the variant deployment. This deployment takes no specification. It is just an assembly of devices.

The statement of root describes for which logical model the variant is built. Only services originating from this or a derived root are allowed in a variant.

**package** org.example.deployment

**import** "HelloECU.cdepl"

**import** "ClientECU.cdepl"

**import** "../HelloWorld.fcdl"

**define** **variant** Standard **for** **root** World {

**use** org.example.deployment.HelloECU

**use** org.example.deployment.ClientECU

}

**define** **variant** Multi **for** **root** World {

**use** org.example.deployment.HelloECU

**use** org.example.deployment.ClientECU

**use** org.example.deployment.ClientECU2

}

# FRANCA+ IDE

Here are only additions to standard Franca editor noted.

The assistant works case insensitive. The inserted names are either fully qualified names or short names. This depends on the imported namespaces.

## Import declarations

For import name proposals the content assistant works on the last segment of imported filenames. Eg. for getting a proposal for "or.example.infrastructure.vector.fidl" start typing vec... - without "

For getting proposals for import in classpath style prefix the input with classpath:, e.g. classpath:vec

For getting proposals for import deployment specifications in platform style start your input with platform:, e.g. platform:network

## Outline

The outline of FCDL and CDEPL uses distinct icons for each element. In the deployment it is possible to drill down from a deployment definition to the target element in the component model.

## Reference Proposals

In Component and Deployment models the naming of references is supported by a context aware content assistant. The assistant is triggered for all references by <strg>+<space> after the keyword. The assistant also pops up, if you enter the separating dot of fully qualified names.

# Command line options for Franca\_SDK

The SDK supports the following command line options. They are set as JVM options.

-DnoDeployGenerator

If set no JAVA classes for deployment accessor will be generated upon the call of the Franca project builder.

-DdeploymentBundles=<plugin>[,<plugin>]\*

Denotes a comma separated list of plugins. These plugins are searched for fdepl and cdepl files. The found files are interpreted as platform specifications. These specifications are made offered by the import proposal provider in the deployment editors.

-DcomponentBundles=<plugin>[,<plugin>]\*

A similar feature is available to provide standard tags in the component model.

# References

[1] Franca User Guide: [**https://drive.google.com/drive/folders/0B7JseVbR6jvhUnhLOUM5ZGxOOG8**](https://drive.google.com/drive/folders/0B7JseVbR6jvhUnhLOUM5ZGxOOG8)

1. In contrast UML standard allows to type a port with more than one interface. [↑](#footnote-ref-1)