Franca User Guide

Release 0.12.0.1

WWW.ITEMIS.DE

Contents

Part One

1. Introduction

Welcome to *Franca*! *Franca* is a framework for definition and transformation of software APIs. The core of it is *Franca IDL* (Interface Definition Language), which is a textual language for specification of APIs (§5). As Franca is based on Eclipse, there are some powerful tools (§1.3) which can be used to work with Franca. E.g., a user-friendly editor for Franca IDL files is available.

1.1 Franca framework architecture

Especially for system integrators, it is time and again necessary to combine software components or subsystems which use different kinds of inter-process-communication (IPC) and different IDLs. Possible reasons for this are:

- the need for integration of 3rd-party and legacy components
- conformance to new standards and
- non-functional system requirements like performance and footprint

This integration is often solved by ad-hoc solutions, developing wrappers or adapters manually after defining a mapping of two IDLs' concepts which should be integrated. This approach is tedious and often leads to inaccuracies and software shortcomings and bugs in the runtime system.

Franca provides a framework for bridging the gap between different IDLs by formally well-founded model transformations in an easy way. Using the Franca framework, these transformations will be encapsulated as *Franca connectors*, which are Eclipse plugins for loading/saving of IDL model instances (i.e. files which formally describe interfaces) and converting them to or from Franca IDL files. Franca (the *lingua franca*) is the pivot point for these transformations. Figure 1.1 shows how connectors (depicted by arrows) form a star-shaped network between Franca IDL and other IDLs.

Advanced features like model validation can also be included in the connector plugins. Franca provides its own extensive model validation, which can be used as an intermediate validation step while transforming models from one IDL to another.

Additionally, the Franca framework is well-suited for developing code generators. This

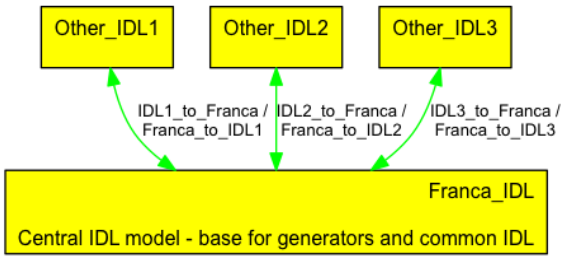


Figure 1.1: Franca transformation framework.

is shown in Figure 1.2. Using connectors (i.e., IDL model transformations) and generators together leads to powerful tool sets which can be used to convert interfaces in several IDLs to Franca and generate source code and configuration files from this. Thus, Franca really helps to solve the integration problem mentioned above in an elegant way.

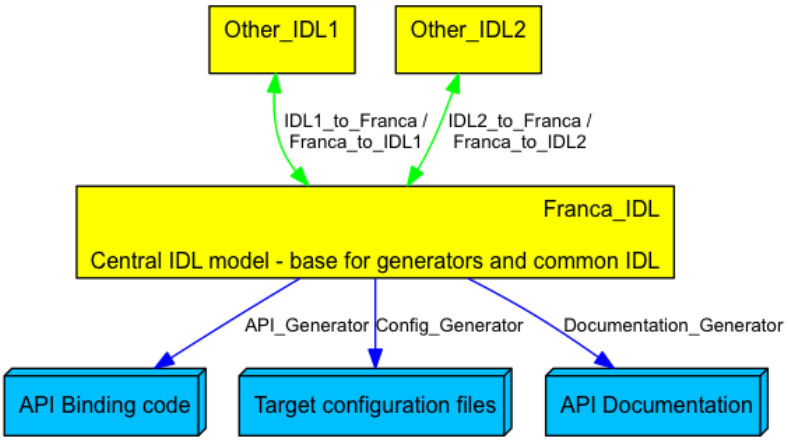


Figure 1.2: Franca transformation and generation framework.

For detail information about the architecture concepts of Franca, please refer to chapter Franca Concepts (§3).

1.2 Franca IDL

Franca IDL is language-neutral and also independent of concrete bindings. Here is a simple example with an interface which supports just one method:

```
interface CalculatorAPI {
    method add {
        in {
            Float a
            Float b
        }
        out {
            Float sum
        }
    }
}
```

APIs defined with Franca IDL may consist of attributes, methods and broadcasts. You may rely on built-in, primitive types (e.g., Int16 or String) or define your own types using

arrays, structures, enumerations, type aliases, maps and others. For some of the types, inheritance is supported.

Franca IDL also supports to optionally define the *dynamic behaviour* of an API. This is done by specifying a *contract*, which basically consists of a *protocol state machine* (PSM) which is a part of the interface. The contract defines states of the interface and transitions between those states. Each transition is triggered by a method call, a broadcast, or the change of an attribute of the interface. For client/server-like architectures, you can think of the PSM as being part of the connection.

Franca IDL specifications may be distributed to multiple files. This is especially useful if several APIs have to be defined, which share some common data types and structures.

For detail information about syntax and semantics of Franca IDL, please refer to the Franca IDL Language Reference (§5).

1.3 Franca Tooling

The core tool for Franca users is the nice textual editor which can be used to review and edit Franca IDL files. The editor is provided as Eclipse plugin and can be installed in any Eclipse environment which provides modeling support. Figure 1.3 shows a screenshot of Franca's IDL editor in action.

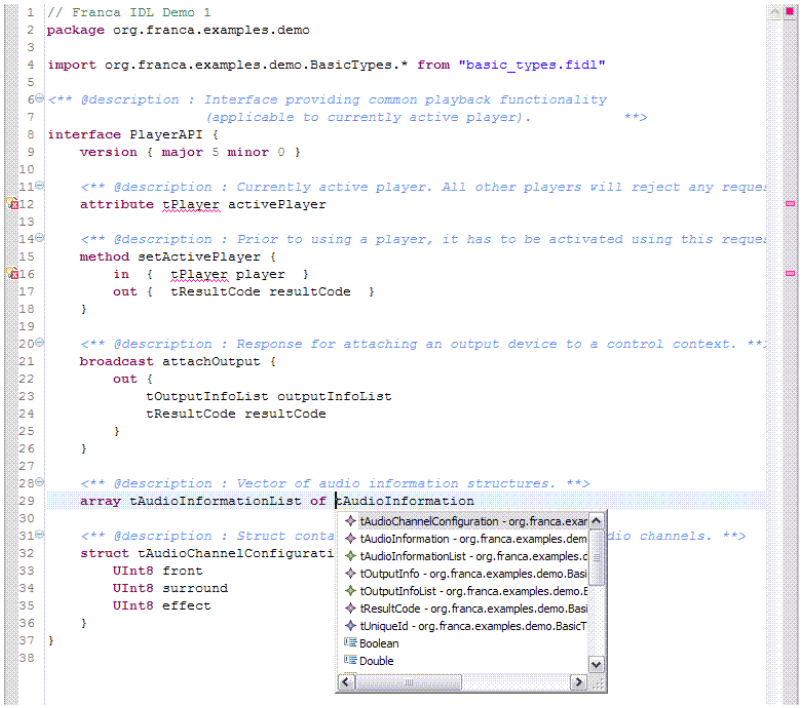


Figure 1.3: Screenshot of Franca IDL editor.

The editor provides syntax highlighting, code completion, folding, online validation, a helpful outline view, jump-to-definition and find-references with shortcuts, and many more features. You can find additional information about the editor here (§4).

The Franca distribution also contains an example generator, which produces HTML files from Franca IDL files. This can be used as is (for generating HTML documentation from interface definitions), or adapted to your needs by using the clone-and-own pattern.

For more information about the Franca IDL editor and other tooling, please refer to chapter Franca Tooling (§4).

2. Getting Started

All tooling needed for Franca is available as a set of Eclipse plug-ins. Thus, the easiest way to start with Franca is to use an off-the-shelf Eclipse distribution and add those plug-ins and their dependencies to this environment. This section will describe how to quickly install this tool platform and be able to

- define interfaces using the Franca IDL textual editor
- develop a code generator based on Franca IDL interfaces

If you haven't done so, we recommend that you read the Introduction chapter (§1) now in order to get some fundamental information about Franca.

2.1 Install Eclipse-based Franca tooling

The installation instructions for the Franca tooling are not part of this User Guide. You will find them online at the Franca EclipseLabs page (Wiki section): Franca Quick Install Guide.

2.2 Import example project

The example project *org.franca.examples.basic* is provided as part of Franca's public distribution. Put this project onto your local file system and import it to the Eclipse workspace by the following steps:

1. In Eclipse, select *File* > *Import...*.
2. In the dialog that opens up, select *General* > *Existing projects into workspace*.
3. In the *Import projects* dialog, browse the file system and locate the directory where the *org.franca.examples.basic* directory is contained.
4. Select the project in the *Projects* list.
5. Ensure that *Copy projects into workspace* is not selected.
6. Press *Finish*.

The example project will be part of your Eclipse workspace now. Open the Franca IDL example file *models/org/example/MediaPlayer.fidl* by locating it in the Package Explorer

and double-click it. The Franca IDL textual editor will open, allowing you to review the example and also do some changes.

2.3 Contents of example project

The example project provides several test cases, which are located in the package

- *org.franca.examples.basic.tests*

You will find this package in the *src* folder which is part of the *org.franca.examples.basic* example project. Each of these test cases might be started by selecting *Run As > JUnit Test* in the test's context menu.

- Test case *Franca2HtmlTest.java*: This test loads a Franca example file and generates a documentation html page from it. The html page will be located in the folder *src-gen/interfaces*. Press F5 for refreshing this folder in case it doesn't show up there after running the test case.
- Test case *HppGeneratorTest.java*: This test loads a Franca example file and generates a C++ class definition from it. The generated code will be printed on the console. It is just an example and is by far not complete.

2.4 Create new Franca interface

In order to create a new Franca interface, open the context menu of the folder *examples/franca* and select *New > File*. In the *New File* dialog, enter a file name with the extension *.fidl*. After pressing the *Finish* button the Franca IDL editor will show up with the new file, which is still empty.

Now you may use keyboard shortcuts like Ctrl-Space to use Content Assist to create the contents for your Franca file.

More information is available about Franca concepts (§3) and in the Franca language reference (§5).

2.5 Develop your own code generator

If you want to develop a new generator which can produce source code from any Franca IDL file, we recommend to use the example generator from test case *HppGeneratorTest.java*. This code generator has been developed with the Xtend language, which translates directly to Java classes. Further documentation about Xtend can be found here.

The example generator *ExampleHppGenerator.xtend* is located in folder *src*, in package *org.franca.examples.basic.generators*.

For starting quickly, clone this xtend file and also the corresponding test case and adapt the copied files for your needs. For more information on generator development, please refer to Building generators with Franca (§9).

3. Franca Concepts

The Franca framework is built on several concepts:

- the Franca core model and the Franca IDL
- the Franca transformation framework
- the Franca generator framework
- distinction of IDL model and deployment model

This chapter provides some detailed explanation for each of these concepts.

3.1 Franca Core Model and IDL

3.1.1 Franca Core Model

Franca aims at formal definitions of software interfaces. Basically, this targets information about attributes and methods provided by the interfaces which should be defined. Franca contains a *core model*, which describes these concepts (and some more, like datatypes). The core model is implemented using the Eclipse Modeling Framework (EMF). Based on the EMF technology, there is a rich set of tools available for manipulating and transforming models describing software interfaces.

The detailed API of Franca's core model is described in section Franca Model API (§8).

3.1.2 Franca IDL Model

Franca is not only able to define interfaces *somehow*, but via an IDL. Therefore, Franca IDL has been created as a textual language (a *DSL*, short for Domain-Specific Language), which can be used with the help of a nice editor to create, review and edit software interfaces. The internal representation of Franca IDL files (i.e. interface definitions) is the core model described above. The IDL is implemented with Xtext.

The detailed reference of Franca IDL is described here (§5).

3.1.3 Franca Core Eclipse Plugins

The following plugins are relevant for the core model and the IDL:

- *org.franca.core*: This plugin contains the core model based on EMF.
- *org.franca.core.dsl*: This plugin contains the IDL definition. It is based on *org.franca.core* and Xtext.
- *org.franca.core.dsl.ui*: This plugin contains the IDL textual editor. It is based on *org.franca.core*, *org.franca.core.dsl*, Xtext and the user interface parts of the Eclipse IDE.

3.2 Franca Transformation Framework

Based on the Franca core model described above, interface specifications defined in other IDLs (in many different formats, ranging from XML to textual DSLs or proprietary formats) can be transformed to/from Franca. In that context, Franca plays the role of a pivot model. Different IDLs can be transformed into each other by chaining transformations like: *IDL1 to Franca to IDL2*.

Figure 3.1 shows an example setup of Franca transformations and generators. The green arrows indicate transformation and reverse transformation between the Franca core model and another IDL's model.

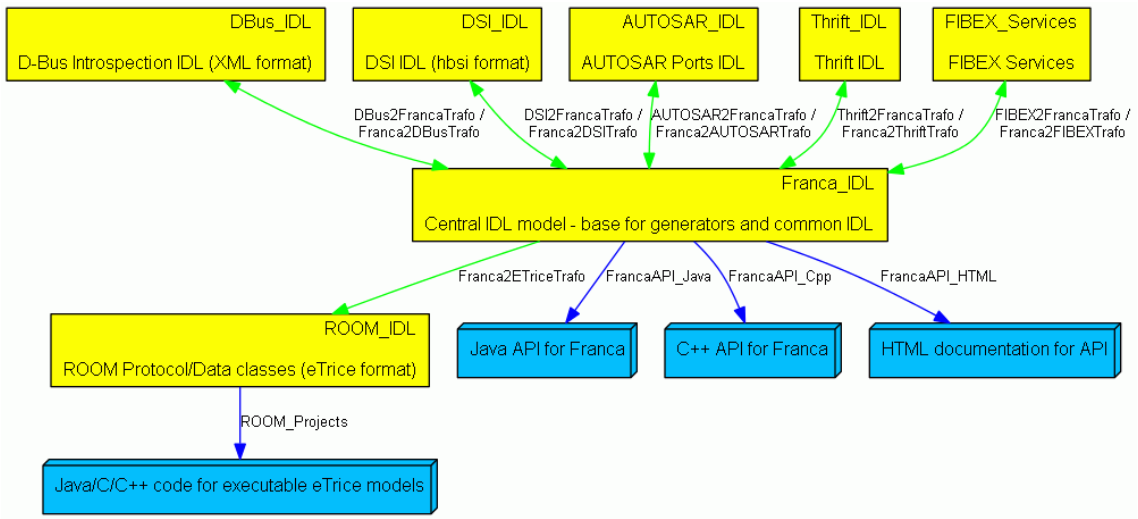


Figure 3.1: An example setup of Franca transformations and generators.

Generally, IDLs in arbitrary formats can be part of the transformation framework. However, Franca provides special support for IDLs which are implemented using EMF. In this case, the Xtend language can be used to implement transformations in an elegant way.

Franca defines a *pattern* for implementing such transformations: the *connector*. A Franca connector for a specific non-Franca IDL provides functions for loading and saving models in that IDL and also for transforming models to/from Franca. Each connector should implement Franca's *IFrancaConnector* interface located in the *org.franca.core.framework* package. Typically, each connector is provided as a separate Eclipse plug-in. E.g., Franca's *D-Bus Support* feature contains the plug-in *org.franca.connectors.dbus* (and a corresponding UI plug-in) for handling DBus Introspection XML files.

See chapter Bulding Transformations (§10) for more details on this concept. In chapter Franca Connectors (§7) the currently supported connectors are listed and explained in some more detail.

3.3 Franca Generator Framework

The Franca core model is also used as a starting point for generating code and other artifacts. As EMF provides a generated Java API (§8) out of Franca's core model, a plethora of tools can be used for building generators. We recommend to use the Xtend language for this, as it is a simple and productive way of implementing generators based on a Java API.

In Franca' source distribution you can find the HTML Generator (§4.3), which generates nice HTML pages out of Franca IDL interfaces. This generator can serve as an example of how to build generators for Franca models with Xtend.

See chapter Building Generators (§9) for more details on this concept.

3.4 Franca Deployment Models

The contents of the Franca core model and the IDL are deliberately restricted to the actual interface specification. Additionally, there is a lot of information regarding to interfaces and APIs which are related to the implementation of the interfaces on a target platform and the actual deployment of these interfaces on the platform. Some examples for this kind of data:

- How are the data types encoded on the target platform (e.g., endianness, padding)?
- Are calls blocking or non-blocking?
- How can the instances of an interface be found and addressed (i.e., service discovery)?
- Which quality-of-service promises are valid?

The next section (§3.5) provides some guidelines for deciding whether new features belong to the Franca IDL itself or to the deployment model.

The actual information in a deployment model depends heavily on the target IPC framework. Therefore, there is no generic deployment model (on a similar abstraction level as Franca IDL itself). Instead, the features of the actual deployment model for the target platform can only be defined with the target platform in mind.

As the information which must be stored in a deployment model is equally important as soon as a Franca IDL interface is incarnated on a real target, the Franca Framework provides support for defining and creating deployment models. A separate Franca Deployment Language is available, which contains of two parts:

- specification of deployment properties (done only once per deployment target platform)
- definition of actual values for deployment properties (done for every Franca interface which has to be deployed on the target platform)

As it is important that deployment properties can be accessed easily during code generation, validation, etc., Franca provides some infrastructure for retrieving the deployment data attached to various interface entities easily.

See chapter Franca Deployment Models (§6) for more details about deployment concepts, their relation to Franca IDL and some examples.

3.5 Guidelines for adding new features to Franca IDL

Although the feature set of Franca IDL can cover the needs of many today's IDLs and IPC frameworks on the market, there might be the need to extend Franca IDL. Technically, this is no big deal. However, extending the expressivity of Franca IDL will usually require adaptions on existing transformations and code generators in order to reflect the IDL changes in the dependent artifacts. Thus, every extension to Franca IDL should be well-founded and backed with a solid semantic notion.

In order to support features which are seemingly missing in Franca IDL, there are several options:

- actually extend Franca IDL to support the feature
- put the additional information in a structured comment (§5.7.2) (using an appropriate @-tag)
- put the additional information into a platform-specific deployment model (§3.4)
- deliberately ignore the feature (if there is a good reason for this)

In order to decide which of these option is appropriate consider the following criteria:

- *IDL*: information needed only for the application logic like:
  1. syntactic description of interfaces
  2. dynamic behaviour, e.g. application protocol contracts
  3. application constraints:
     - like discrete values
     - interval ranges accepted for method input parameters
     - default values for attributes
- *deployment model*: information that is used for code generation but is highly dependent on the backend IPC used
  1. call semantic: synchronous, asynchronous?
  2. communication partner addressing information, e.g. TCP/IP addresses or service discovery related information
  3. data coding: physical representation of data (e.g., alignment, memory layout)
  4. QoS information, allocation of ressources: network (bandwidth, priorities), CPU (priority, processing share), RAM (quota), etc.
- *structured comments*: information not used for productive code generation
  1. documentation
  2. meta-information used during transformations (e.g., URI or filename of source IDL
  3. meta-information regarding the interface itself (e.g. deprecation)

4. Franca Tools User's Guide

4.1 Franca IDL Editor

The core tool for Franca users is the nice textual editor which can be used to review and edit Franca IDL files. The Franca IDL Editor is a textual editor which is similar to Eclipse's text-based editors like in JDT or CDT. It is provided as Eclipse plugin and can be installed in any Eclipse environment which provides modeling support.

The following screenshot shows the Franca IDL editor in action.

The editor provides syntax highlighting, code completion, content assist, folding, online validation, a helpful outline view, jump-to-definition and find-references with shortcuts, and many more features. See Table 4.1 for a table of most important keyboard short-cuts for Franca.

You may download a nice cheat sheet with more keyboard shortcuts here.

More detail about the Franca IDL Editor will be provided here later.

Shortcut	*Function*	*Description*
Ctrl-7	Toggle comment	Toggle comment for current line or selection.
Ctrl-Space	Content Assist	Context sensitive suggestions for possible values.
F3 or Ctrl-MouseClick	Jump to definition	Jumps to the definition of the reference under cursor.

Table 4.1: Most important Eclipse keyboard short-cuts for Franca.

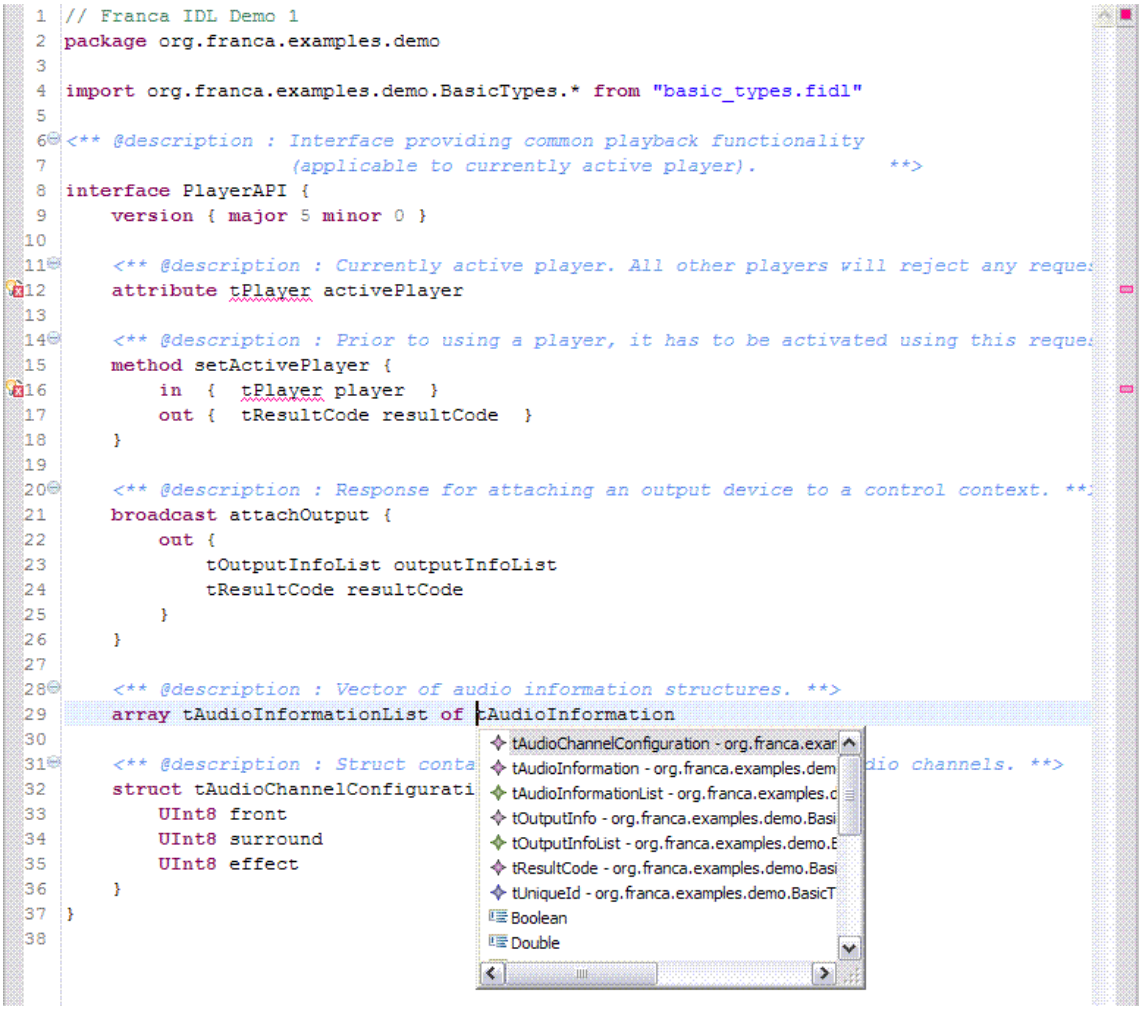


Figure 4.1: Screenshot of Franca IDL editor with content assist and validation markers.

4.2 Franca Contract Viewer

The dynamic behavior of interfaces can be modeled using protocol state machines. See section Contracts (§5.6) for a description of the correct syntax. As contracts must be an integral part of a Franca interface, the state machines are modeled in the same textual editor as the other parts of a Franca interface (as described in the previous section).

In order to understand the structure of the protocol state machine of a contract quickly, Franca comes with a *Contract Viewer*. In order to use it, the Franca *User Interface Add-Ons* feature has to be installed in the IDE.

4.2.1 Contract Viewer in Franca 0.10.0 and later

Since Franca 0.10.0, the contract viewer implementation is based on the KIELER framework. After installing KIELER and the Franca User Interface Add-Ons feature (see section Install Eclipse-based Franca tooling (§2.1) for more information on installation), the Eclipse toolbar will offer a green/yellow button which allows to open the *KIELER View Management Quick Configuration*. The following screenshots show how this toolbar button and the corresponding dialog look like.

In order to show the contract viewer, you have to ensure that the checkboxes *Enable view management* and *Show Franca contract* are selected. Now a separate contract viewer

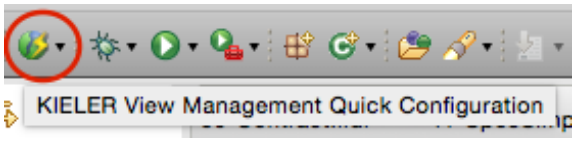


Figure 4.2: Toolbar button to launch the KIELER configuration.

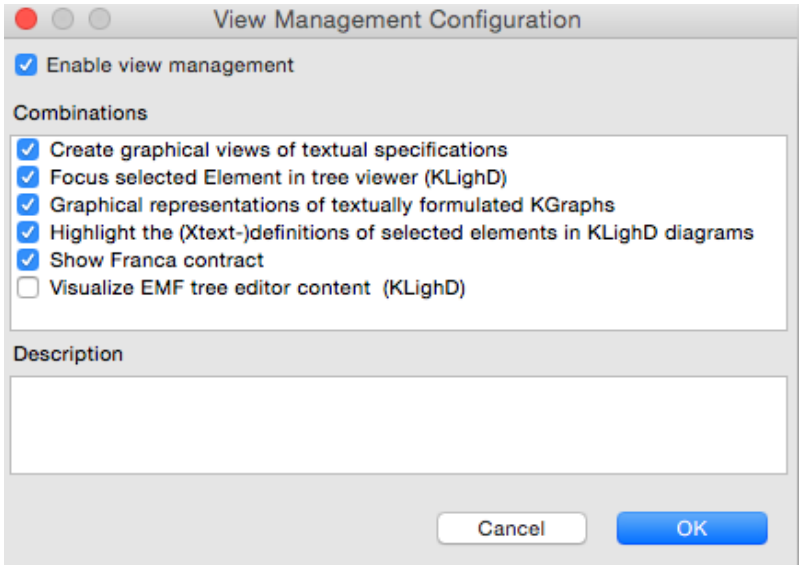


Figure 4.3: The KIELER quick configuration dialog.

will open for each opened Franca IDL file which defines an interface contract. If the viewer doesn't open automatically, select *Windows* > *Show View...* > *Other* and their *KIELER Lightweight Diagrams* > *Lighweight Diagram*.

The following screenshot shows the contract viewer with an example contract.

On the left side, the protocol state machine (PSM) for the contract is displayed, which consists of *states* (in yellow) and *transitions* (arrows). The initial state of the PSM is depicted by a black dot and an unlabeled transition pointing to it. On the right side of the viewer, an options section offers several ways to tweak the diagram. E.g., transition triggers can be hidden and the overall direction of the layout can be tuned.

By clicking on a state in the left-hand part of the view, the corresponding state definition in the Franca IDL editor is selected automatically. This allows to easily find a correspondence between the diagram and its textual definition.

4.2.2 Contract Viewer in Franca 0.9.1 and earlier

In Franca 0.9.1 or earlier, the contract viewer implementation was based on ZEST. Here, the contract viewer could be opened by selecting *Window* > *Show View...* > *Franca*. Its appearance differed a lot compared to the KIELER based viewer described above.

The "T" button in the upper right of the ZEST-based viewer window toggled the labels for all transitions of the protocol state machine. If the labels are switched off, mouse hovers are available which provide the information about transition conditions.

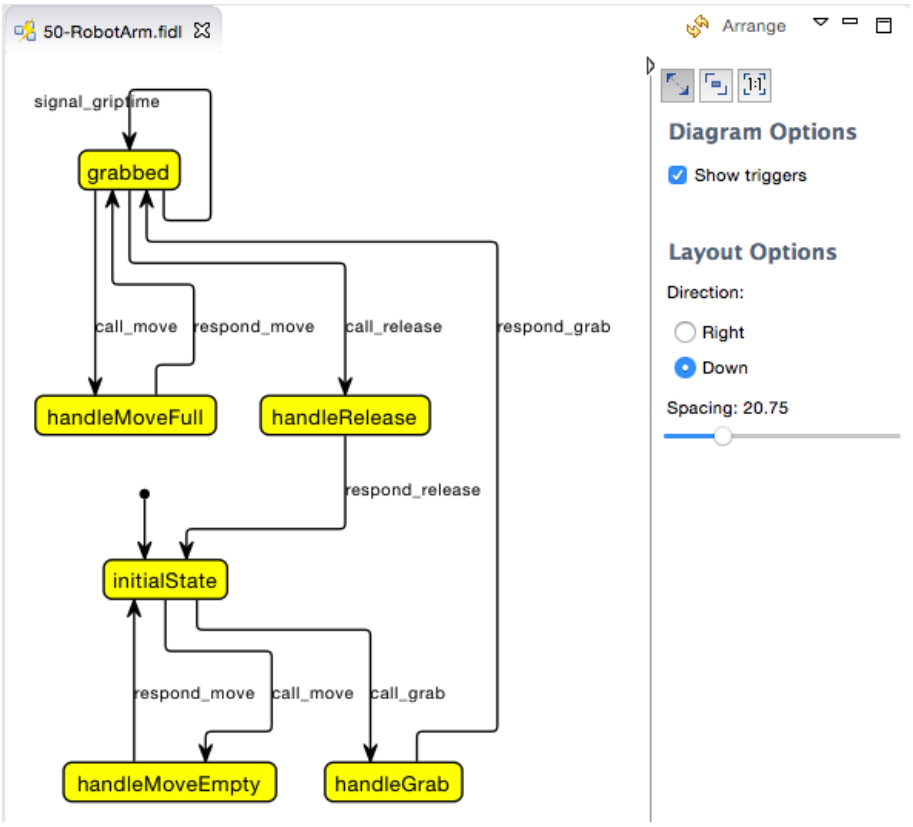


Figure 4.4: Screenshot of the Franca Contract Viewer.

4.3 Franca IDL HTML Generator

The HTML generator for Franca IDL files is provided as Java class *HTMLGenerator*. It is located in package *org.franca.generators.html* and provided by Eclipse plugin *org.franca.generators*. In this plugin, a helper Java class *FrancaGenerators* is provided which should be used to run the HTML generator:

```
FModel fmodel = FrancaIDLHelpers.instance().loadModel(inputfile);
FrancaGenerators.instance().genHTML(fmodel, "html-gen-dir");
```

The HTML generator is implemented using the Xtend language. The Xtend-builder automatically creates the Java class mentioned above from this implementation.

Part Two

5. Franca IDL Reference

The core of Franca IDL is the support for interface definitions (§5.5), consisting of attributes, methods and broadcasts. Those definitions will be based on a type system (§5.1), which provides a variety of primitive and user-defined types as well as constant definitions. Additionally, the dynamic behavior of interfaces can be specified using contracts (§5.6) consisting of protocol state machines. Types and constants can be defined either on a global scope using type collections (§5.4), or as part of an interface definition (§5.5). Finally, Franca IDL supports Java-doc like tagged comments, called structured comments (§5.7) and language features like *packages* and *imports* which can be used for defining interfaces using multiple Franca IDL files (§5.8).

5.1 Data types

Franca supports a predefined set of primitive data types and a variety of user-defined types. User-defined types may be arrays, type aliases, structures, unions, enumerations or maps. User-defined types can refer to primitive types or other, previously defined types.

5.1.1 Primitive types

The primitive types supported by Franca are listed in Table 5.1. There is a number of very specific signed and unsigned integer types with varying ranges. This allows to be very specific about the size of these types on any target platform. Additionally, Franca IDL provides a way to specify generic integer types with optional min/max range (since Franca 0.9.0, see next section).

Caveat: APIs should use arguments of type **String** or **ByteBuffer** only for transporting payload data which is not parsed by the server component which offers the interface. A useful example for **ByteBuffer** arguments is a protocol layer which examines the header of an incoming data package, but doesn't look at the payload data. If the server does *deep packet inspection* or similar on the incoming data, the argument is used as a tunnel for a protocol which is not specified by the API. This bypass of the API definition weakens the API abstraction and will inevitably lead to integration issues and runtime problems.

Type name	*Description*
UInt8	unsigned 8-bit integer (range 0..255)
Int8	signed 8-bit integer (range -128..127)
UInt16	unsigned 16-bit integer (range 0..65535)
Int16	signed 16-bit integer (range -32768..32767)
UInt32	unsigned 32-bit integer (range 0..4294967295)
Int32	signed 32-bit integer (range -2147483648..2147483647)
UInt64	unsigned 64-bit integer
Int64	signed 64-bit integer
Integer	generic integer (with optional range definition, see below)
Boolean	boolean value, which can take one of two values: false or true.
Float	floating point number (4 bytes, range +/- 3.4e +/- 38, ~7 digits)
Double	double precision floating point number (8 bytes, range +/- 1.7e +/- 308, ~15 digits)
String	character string, see caveat below
ByteBuffer	buffer of bytes (aka BLOB), see caveat below

Table 5.1: Primitive types of Franca IDL.

The actual physical encoding of the primitive types depends on the target language. A default encoding will be defined as part of Franca's deployment model. Exceptions from this default encoding can be defined in the target-language-specific deployment model.

5.1.2 Integer with optional range

The primitive integer types with fixed number of bits as defined above (e.g., **UInt32**) allow to specify interfaces with the actual implementation in mind. This is typically preferred by embedded systems developers, in order to exactly know which data is being transported. However, there are some drawbacks of using these integer types for interface definitions. The specific implementation platform might not support all of these types, e.g., Java doesn't support unsigned integers.

Since Franca 0.9.0, the IDL supports a generic integer type named *Integer*. In order to restrict the range of actual values which can be used, Franca IDL allows to specify a range of values this type can assume. Examples:

```
Integer
Integer(1,7)
Integer(−20,100)
Integer(0,maxInt)
Integer(minInt,maxInt)
```

The range is given as a minimum and a maximum value. The min/max values might be negative, but the minimum value always has to be less then or equal to the maximum value. If no range is given, the biggest available integer type for the specific platform should be used. There are special keywords **minInt** and **maxInt**, which represent the smallest and biggest integer on the specific platform, respectively.

Note that by using the **Integer** type it is possible to emulate all primitive integer types as defined above. Similarly, it is possible to replace usages of the ranged integer type by the next bigger primitive integer type given that all restrictions of the specific implementation platform are known. There is a helper model transformation in the Franca framework which supports these two type conversions, see section IntegerTypeConverter (§8.3.2).

5.1.3 Arrays

There are two ways of defining array types: explicitly named arrays or implicit array types without a name.

Explicitly named, one-dimensional arrays can be defined by **array** ExampleArray **of UInt8**

The array's element type can be any primitive or user-defined type, including another array. This can be used to define multi-dimensional arrays:

```
array ExampleArrayRow of UInt8
array ExampleArray of ExampleArrayRow
```

Implicit ("inline") array types are defined by attaching square brackets (e.g., **UInt8**[]) to any other type definition. This can be done for attributes, struct members and arguments of methods and broadcasts. See more examples below. Note: Inline arrays can only be defined one-dimensional. If multi-dimensional arrays are needed, only one dimension can be defined as unnamed array.

The syntax for defining implicit array types doesn't support specifying fixed array sizes or array size limits. If you want to define such properties for arrays, you might use Franca's deployment model (§3.4) feature.

5.1.4 Enumerations

Basic enumerations

An *enumeration* (also called *enumerated type*) is a data type consisting of a set of named values called *enumerators*. In Franca IDL, an enumeration is defined by

```
enumeration ExampleEnumeration1 {
    VALUE_1
    VALUE_2
    VALUE_3
    // ...
}
```

The enumerators are identifiers which can be used as unique constants. Values for the enumerators can be defined optionally. Each value is defined using an expression (§5.3) of type *integer*:

```
enumeration ExampleEnumeration2 {
    VALUE_1 = 100
    VALUE_2 = 100+1
    VALUE_3 = 30*30
    // ...
}
```

As the enumerator value can be any expression of type *integer*, it is also possible to specify it with a hexadecimal or binary literal:

```
enumeration ExampleEnumeration2b {
    VALUE_1 = 0xBADA
    VALUE_2 = 0X89ab
    VALUE_3 = 0b0001001
    VALUE_4 = 0B10101
    // ...
}
```

Deprecated: It is also possible to use string constants as enumerator value definitions. However, this will result in a deprecated-warning by the Franca validator. Note that in this case the string constant should be parsable as integer number:

```
enumeration ExampleEnumeration3 {
    E1 = "10"
    E2 = "20"
    E3 = "foo" // invalid, no integer
    // ...
}
```

Enumeration inheritance

Enumeration types support *inheritance* by using the keyword *extends*. This allows to derive a new enumeration type from an existing one and add further enumerators to the base enumeration. Only single inheritance is allowed for each enumeration - however, a chain or tree of enumerations can be build. Example:

```
enumeration BaseEnumeration {
    VALUE_1
    VALUE_2
}
```

```
enumeration DerivedEnumeration extends BaseEnumeration {
    VALUE_3
    VALUE_4
}
```

If the target language doesn't support inheritance for enumeration types, the *extends*-chain will be transformed into a flat enumeration consisting of all enumerators of the base type(s) and those of the derived type.

5.1.5 Structures

A *struct* (also called *record*, *tuple*, or *compound data*) is a value that contains other values, typically in fixed number and sequence and typically indexed by names. The elements of structs are usually called *fields* or *members* (source: Wikipedia).

Basic structs

User-defined structures can be defined by specifying the structure's members, with a type and name for each member:

```
struct ExampleStruct {
    UInt8 member1
    String member2
    ExampleArray member3
    UInt16[] member4
}
```

Element types might be predefined types or user-defined types. This allows e.g. nested structures or arrays of structures. Element types may also be arrays, e.g., *member4* is an unnamed array of type **UInt16**.

Note that the struct type definition also defines the order of the elements. Thus, the following two struct definitions are *different*:

```
struct ExampleStruct1 {
    UInt8 member1
    String member2
}

struct ExampleStruct1 {
    String member2
    UInt8 member1
}
```

Any communication implementation should respect the element order given by the Franca IDL type definition. This is relevant only for usecases where an implicit agreement on the element order is required. E.g., for binary serialization and deserialization the element names will not be part of the serialized message. Thus, sender and receiver of a message have to agree on the element order. In other contexts this might not be relevant, e.g., when using a JSON-style serialization or for Franca IDL struct initializer expressions (§5.2.2).

However, as it is not known at interface definition time in which context the interface will be used, the interface designer has to take the element order into account.

Struct inheritance

Struct types support *inheritance* by using the keyword *extends*. This allows to derive a new structure type from an existing one and add further structure members to the base's

members. Only single inheritance is allowed for each structure definition - however, a chain or tree of structures can be build. Example:

```
struct BaseStruct {
    UInt8 member1
    ExampleArray member2
}

struct DerivedStruct extends BaseStruct {
    ExampleArray member3
}
```

If the target language doesn't support inheritance for structure types, the *extends*-chain will be transformed into a flat structure consisting of all members of the base type(s) and those of the derived type.

Polymorphic structs

The root type in a struct inheritance hierarchy may be marked with the *polymorphic*-keyword. Consider the following example:

```
method callme {
    in {
        BaseStruct t
    }
}

struct BaseStruct polymorphic {
    Int16 a
}

struct Derived1 extends BaseStruct {
    Int16 b
}

struct Derived2 extends BaseStruct {
    String c
}
```

The caller of method *callme* might provide an argument of type *Derived1*, although the method's argument type is declared as type *BaseStruct*. In an environment where polymorphism is fully supported, the server will then be able to receive an object of the actual type, i.e., *Derived1*. It will be able to access the struct element *b*.

However, as full polymorphism support usually requires some implementation overhead in the IPC target platform, this behavior has to be explicitly switched on by flagging the root of the struct inheritance hierarchy with the keyword *polymorphic*. Without setting the root struct type to *polymorphic*, the server-side implementation of the interface will only be able to access the actual type used in the interface definition (and not derived types). This low-footprint behavior is the default.

5.1.6 Unions (aka variants)

A *union* is a value that may have any of several representations or formats; or a data structure that consists of a variable which may hold such a value (source: Wikipedia).

Basic unions

Unions can be defined by specifying its possible value types, together with a name for each representation. As a union will have only one representation at a time, the order of the union's elements is not important. Example of a union definition:

```
union ExampleUnion {
    UInt32 element1
    Float element2
}
```

Element types might be predefined types or user-defined types. This allows e.g. nested unions, or unions of structures. No two elements of an union may have the same type.

Union inheritance

Union types support *inheritance* by using the keyword *extends*. This allows to derive a new union type from an existing one and add further elements to the base union. Only single inheritance is allowed for each union definition - however, a chain or tree of unions can be build. Example:

```
union BaseUnion {
    UInt32 genericRepresentation
}

union DerivedUnion extends BaseUnion {
    Float aFloat
    String aString
}
```

If the target language doesn't support inheritance for union types, the *extends*-chain will be transformed into a flat union consisting of all elements of the base type(s) and those of the derived type.

5.1.7 Maps (aka dictionaries)

Maps represent key/value-stores and are typically implemented by B-trees or hashes. The definition for Maps with Franca IDL specifies the key type and the value type. Example:

```
map ExampleMap {
    Int16 to ExampleStruct
}
```

Key types as well as value types might be primitive or user-defined types which provide an equality-relation. NB: In some target languages, key types might be restricted to primitive types.

5.1.8 Type definitions (aka aliases)

Type definitions can be used to create new type names which are simple aliases for existing primitive or user-defined types. Example: `typedef ExampleAlias is UInt32`

5.2 Constant definitions

Franca supports the definition of constants, which are typed values identifiable by a unique name. Constants might be used as part of any expression (§5.3), given that they have a compatible type. Tools as code generators should create code which establishes these constants on the executable platform.

5.2.1 Primitive constants

Here are some examples of primitive constants:

```
const Boolean b1 = true
const UInt32 MAX_COUNT = 10000
const UInt16 SOME_ID = 0xA00B
const UInt8 BYTE_ME = 0b00110011
const String foo = "bar"
const Double pi = 3.1415d
```

The initializer expression on the right-hand side of the constant definition must evaluate to a proper type which is compatible with the constant's type. Note that complex expressions (§5.3) might be used as initializer expressions. Examples:

```
const UInt32 twentyfive = 5*5
const Boolean b2 = MAX_COUNT > 3
const Boolean b3 = (a && b) || foo=="bar"
```

5.2.2 Complex constants

Constants may also have more complex, user-defined types. This will lead to more elaborate initializer expressions. This section will show examples of how to define constants which are of array, struct, union and map type.

For array constants, the initializer expression lists the array elements in square brackets. Example:

```
array Array1 of UInt16

const Array1 empty = []
const Array1 full = [ 1, 2, 2+3, 100*100+100 ]
```

Constants of struct type will have to define an initializer expression for each struct element. Example:

```
struct Struct1 {
    Boolean e1
    UInt16 e2
    String e3
}

const Struct1 s1 = { e1: true, e2: 1, e3: "foo" }
```

If struct types are getting more complex (i.e., with nested struct types or arrays), the initializer expression is not easy to understand. Therefore, each struct element in the initializer expression has to be uniquely identified by the element name (e.g., `e1` in the example). Note that explicitly naming the elements in the struct initializer allows to initialize the elements in any order. Each element has to occur exactly once.

The syntax for union type constants is similar to struct types. However, as a union represents exactly one of its elements, the initializer expression will only refer to one element name. Some examples:

```
union Union1 {
    UInt16 e1
    Boolean e2
    String e3
}
```

```
const Union1 uni1 = { e1: 1 }
const Union1 uni2 = { e3: "foo" }
```

Finally, the initializers for a constant of type *map* contains a list of key/value pairs. Example:

```
map Map1 { UInt16 to String }

const Map1 m1 = [ 1 => "one", 2 => "two" ]
```

The initializer expressions for the keys and values of each pair might be themselves arbitrary complex expression. However, they have to be evaluated to the proper type given by the definition of the map type.

5.3 Expressions

Expressions can be used in Franca IDL in various ways:

- as default values for enumerators (§5.1.4)
- as initializer expressions for constant definitions
- as guard conditions in interface contracts (§5.6.2)
- as initializer expressions for state variables (§5.6.4) in contracts
- in the action code (§5.6.3) of contracts (e.g., in if-conditions)

5.3.1 Type system

Expressions in Franca IDL are *typed*. The type system supports the basic types *boolean*, *integer*, *float*, *double* and *string*. The type system is aware of user-defined types defined in Franca IDL. I.e., if a struct type is defined by

```
struct SomeStruct {
    UInt8 member1
    String member2
}
```

then the type *SomeStruct* is also available in expressions. E.g., if expression `s` refers to a Franca constant of type *SomeStruct*, the expression `s->member1` will be of type *integer*.

The Franca validator ensures that all expressions have a proper type which also has to be consistent with the usage of the expression. E.g., an expression used as a guard condition in a contract must be of *boolean* type.

Type system rules

Note that in order to reduce specification errors the Franca IDL type system provides only a minimum of implicit type conversions (aka typecasts). In particular, the following rules apply:

- There is no implicit conversion from integer to float or vice versa.
- There is no implicit conversion from integer to double or vice versa.
- There is an implicit conversion from float to double and vice versa.
- There is no implicit conversion from enumeration types to integer.
- For struct and union type inheritance: There is an implicit upcast for derived struct (and union) types. There is no downcast for struct and union types.
- For enumeration types: There is an implicit downcast for enumeration types (i.e., an enumerator from a base enumeration can be used as representant of any derived enumeration).

- The **typedef** mechanism is transparent for the type system (i.e., types introduced by **typedef** will be handled identically to their base types).

5.3.2 Constant values

The basic elements for building expressions are *constant values*.

```
// constant expressions
true // boolean
false // boolean
100 // integer (positive)
-273 // integer (negative)
0x0ABC // integer (hexadecimal)
0b010101 // integer (binary)
1.2f // float
6.022e23f // float
3.1415d // double
8.617e-5d // double
"foobar" // string
```

5.3.3 Comparison operators

Two expressions of equal type may be related to each other using comparison operators. The resulting expression is of type *boolean*.

Examples for all comparison operators will be given here.

5.3.4 Arithmetic operations

Expressions may be built from sub-expressions of type *integer*, *float* or *double* using arithmetic operations.

Examples for all arithmetic operators will be given here.

5.3.5 Boolean operations

Expressions may be built from sub-expressions of type *boolean* using boolean operations.

Examples for all boolean operators will be given here.

5.4 TypeCollection definition

A *type collection* is a (maybe empty) set of user-defined types and constant definitions. Previous to Franca-0.8.0, user-defined types could be located on the top-level of a Franca model. This has been banned with Franca-0.8.0 in order to provide a consistent scheme of fully qualified names and versioning for user-defined types. Type collections may also define constants.

Each type collection in Franca has some metadata: a single identifier *name* for the collection and an optional *version number* (with major/minor scheme). The name of the type collection will be scoped relative to the fully qualified name of the package defined at the beginning of the Franca file. The following example shows the basic type collection definition:

```
typeCollection ExampleTypeCollection {
    version { major 3 minor 1 }

    // put user-defined types and constant definitions here
```

```
}
```

As type collections are just a plain collection of user-defined types, inheritance is not supported for them. All user-defined types inside a type collection have global visibility. I.e., they can be used for defining interfaces as well as user-defined types in another type collection. This is different from the visibility of types that are defined as part of an interface (see below).

5.5 Interface definition

Each interface in Franca consists of some metadata (e.g., interface name and version number), the actual interface definition (consisting of attributes, methods and broadcasts), an optional contract (§5.6) specifying the dynamic behavior of the interface and, last but not least, user-defined data types (§5.1) and constants (§5.2). An interface definition might use the user-defined types of another interface definition only if it inherits (directly or indirectly) from that interface definition. Thus, the visibility of user-defined types as part of an interface definition is restricted. This is different from the global visibility of types which are part of a type collection.

5.5.1 Basic interface definition

Interface name, version and contents

The basic interface definition using Franca includes an interface name and a version number (major/minor scheme). The name of an interface has to be a single identifier. The following example gives a blueprint for the structure of an interface definition:

```
interface ExampleInterface {
    version { major 5 minor 0 }

    // put type definitions and constant definitions here

    // put attributes, methods and broadcasts here

    // put optional contract here
}
```

The name of the interface will be scoped relative to the fully qualified name of the package defined at the beginning of the Franca file.

The elements of an interface can be arranged freely. This allows to provide a logical grouping of interface elements. The interface contract (if any) should be the final element of the interface definition. In previous Franca versions, the order of the elements of an interface was fixed and thus didn't allow a logical grouping.

Interface versioning and change compatibility

The specification of a major/minor version number is optional, but is strongly recommended. Changes in the major number indicate changes which are not backward compatible. Example use cases for *incompatible* changes:

- remove attribute, method or broadcast
- remove element from struct or union
- rename an attribute
- rename a method or broadcast, or one of its arguments
- rename a user-defined data-type
- add a contract to an interface which previously didn't have any

- tighten the constraints for the dynamic behavior of an interface (i.e., its contract)

Interface designers should try hard to avoid incompatible changes, or at least provide periods during which a feature is marked as deprecated before it is removed completely.

Changes in the minor number indicate changes which are backward compatible. Example use cases for backward *compatible* changes:

- add attribute, method or broadcast
- add an argument to a method or broadcast (*)
- add a field to a struct or union (*)
- remove a contract from an interface
- ease restrictions on the dynamic behavior of an interface (i.e., its contract)

(*) In some target IDLs, adding arguments for methods or broadcast or adding members to structs might be regarded as incompatible change. It might be hard to map the version numbers between Franca and those IDLs.

Interface inheritance

Franca supports interface *inheritance* by using the keyword *extends*. This allows to derive a new interface from an existing one and add further elements to the base's members. Only single inheritance is allowed for each interface definition - however, a chain or tree of interfaces can be build. Example:

```
interface BaseInterface {
    // ...
}

interface DerivedInterface extends BaseInterface {
    // ...
}
```

Table 5.2 lists all interface members and how their semantics is handled with respect to interface inheritance.

If the target language doesn't support inheritance for interfaces, the *extends*-chain will be transformed into a flat interface definition consisting of all elements of the base interface(s) and those of the derived interface.

5.5.2 Attributes

An *attribute* is a property on the provider side, which is defined as part of the interface with its type and name:

```
interface ExampleInterface {
    attribute UInt32 someAttribute
    attribute ExampleArray otherAttribute
    attribute String[] arrayAttribute
}
```

Attributes might have primitive types or user-defined types. They also can be defined as inline, unnamed arrays (e.g., the attribute *arrayAttribute* in the example above).

The interface provider holds the data for its attributes and is able to change their values. The clients of an interface might actively read its attributes' values. The clients also might register for updates of this attribute and will get change notifications afterwards. The detailed behavior of an attribute can be specified by a combination of flags (see details below).

Interface element	*Inheritance semantics*
type definitions	All types of the base interface are inherited by the derived interface. Redefining of types is not allowed (i.e., type with same name in base and derived interface).
constant definitions	All constant definitions of the base interface are inherited by the derived interface. Redefining of constants is not allowed (i.e., constant with same name in base and derived interface).
attributes	All attributes of the base interface are inherited by the derived interface. Overloading of attributes is not allowed (i.e., attribute with same name in base and derived interface).
methods	All methods of the base interface are inherited by the derived interface. Methods can be overloaded, either in the same interface or via inheritance. I.e., methods with the same name can be defined as long as the signature of those methods is different. See section 5.5.3 for more details on overloading. In versions prior to Franca 0.7.4, overloading of methods was not allowed.
broadcasts	All broadcasts of the base interface are inherited by the derived interface. Broadcasts can be overloaded, either in the same interface or via inheritance. I.e., broadcasts with the same name can be defined as long as the signature of those methods is different. See section 5.5.4 for more details on overloading. In versions prior to Franca 0.7.4, overloading of broadcasts was not allowed.
contract	The contract of the base interface is inherited by the derived interface. It is not possible currently to redefine the inherited contract. This will change in a future release of Franca (this will allow extending the contract of its base interface by the derived interface).

Table 5.2: Franca interface members and their semantics with respect to interface inheritance.

For target IPC platforms which do not support attributes, code might be generated to add this support. Typically, a getAttribute method and a registerForUpdate method would be provided.

Readonly attributes

By default, clients can read and write the values of attributes. This access can be restricted by specifying the per-attribute flag *readonly*. When the flag is given, the clients of the corresponding interface are not allowed to change the attribute's value. However, the server might offer some methods which will change the attribute. Example for the *readonly*-flag:

```
interface ExampleInterface {
    attribute Float throttle
    attribute Float speed readonly
}
```

In the example, attribute *throttle* can be written, while attribute *speed* can only by read by the clients.

Write-only attributes

It is also possible to disallow the read access to the value of an attribute by specifying the per-attribute flag *noRead* (available since Franca 0.12.0). When the flag is given, the clients of the corresponding interface are not allowed to explicitly read the attribute's value. I.e., code generators will not provide a getter function for the generated attibute. However, the attribute's value might still be retrieved using the subscription mechanism as described below. Example for the *noRead*-flag:

```
interface ExampleInterface {
    attribute UInt32 controlRegister noRead
}
```

In the example, attribute *controlRegister* cannot be explicitly read by the clients. However, it can be written, and updates might be received using the subscription mechanism.

Client subscriptions

As mentioned above, clients of an interface might register for updates of an attribute's value. Sometimes it is necessary to specify attributes which do not support this subscription option. This can be accomplished by adding the flag *noSubscriptions* to the attribute's definition. Example for the *noSubscriptions*-flag:

```
interface ExampleInterface {
    attribute Double temperature readonly noSubscriptions
    attribute Boolean overheated
}
```

The attribute *temperature* in the example will change often, so that it is better to disallow subscriptions and use some kind of polling instead. For the boolean attribute *overheated*, subscriptions are possible (which is really what clients expect). Note that the *noSubscriptions* flag has been combined with the flag *readonly* in order to disallow changes by the clients, too.

5.5.3 Methods

Basic method syntax

A *method* in a Franca interface is called by one of the clients using the interface; the response will be sent by the server. Therefore, a method definition will contain a set of *in*-arguments and a set of *out*-arguments, each with own type and name. Example:

```
interface Calculator {
    method divide {
        in {
            UInt32 dividend
            UInt32 divisor
        }
        out {
            UInt32 quotient
            UInt32 remainder
        }
    }
}
```

Arguments might have primitive types or user-defined types. Arguments may be also specified as unnamed, inline arrays as in the following example:

```
interface LetterCount {
    method count {
        in {
            String[] words
        }
        out {
            UInt16[] counts
        }
    }
}
```

Argument names must be unique. This also applies across the *in* and *out* sections. Thus, the following example will not be valid:

```
method play {
    // validation error: "Duplicate argument name 'track' used for in and out"
    in { TrackInfo track }
    out { UInt16 track }
}
```

Runtime semantics of method calls

The runtime behavior of a method call (e.g., blocking vs. non-blocking) is subject to implementation by the underlying target IPC stack. Thus, it will not be specified as part of the interface definition in the IDL. This information will be stored in an additional deployment model (§3.4), instead.

Declaration of method errors

If a problem occurs during execution of the method on server side, the server might issue an *error* instead of sending the normal reply with its *out*-arguments. This semantics can be defined as in the following example:

```
interface Calculator {
    method divide {
        in {
            UInt32 dividend
            UInt32 divisor
        }
        out {
            UInt32 quotient
```

```
            UInt32 remainder
        }
        error {
            DIVISION_BY_ZERO
            OVERFLOW
            UNDERFLOW
        }
    }
}
```

The component providing the *Calculator* interface will either return a reply with the *out*-arguments or reply with an error code as specified by the nameless enumeration defined as part of the method specification.

In fact, the syntax above is the same as with the definition of enumerations (§5.1.4). I.e., optional values and structured comments can be added to each of the enumerators. Moreover, a reference to a base enumerator can be added using the keyword *extends* - this allows inheriting of common error codes:

```
interface Calculator {
    method divide {
        in {
            UInt32 dividend
            UInt32 divisor
        }
        out {
            UInt32 quotient
            UInt32 remainder
        }
        error extends GenericErrors {
            DIVISION_BY_ZERO
            OVERFLOW
            UNDERFLOW
        }
    }

    enumeration GenericErrors {
        INVALID_PARAMATERS
        // ...
    }
}
```

Finally, it is possible to directly reference a separately defined *enumeration*, which might also be located in a different Franca file. Example:

```
interface Calculator {
    method divide {
        in {
            UInt32 dividend
            UInt32 divisor
        }
        out {
            UInt32 quotient
            UInt32 remainder
        }
        error CalcErrors
    }
```

```
    enumeration CalcErrors {
        DIVISION_BY_ZERO
        OVERFLOW
        UNDERFLOW
    }
}
```

Fire-and-forget methods

If neither *out* arguments nor *error* return codes are specified for a method, the server will by default send an (empty) reply to its client. This behavior can be changed by specifying a *fireAndForget* flag for the method, which indicates that the server will not respond at all.

Example:

```
interface Watchdog {
    method stillAlive fireAndForget {
        in { UInt16 health }
    }
}
```

This can be used to implement lightweight communication patterns.

Method overloading

Franca IDL also allows method overloading, i.e., interfaces with two or more methods with the same name. This is allowed as long as the method signatures are different, i.e., the sets of combined *in*- and *out*-argument types must be different for each overloaded method. Note that only the types will be part of the signature, not the argument names. The following example shows a valid case of overloading:

```
interface SomeInterface {
  method callMe {
    in { Boolean b }
  }

  method callMe {
    in { Boolean b }
    out { String s }
  }
}
```

Note that although the set of *in*-arguments is identical, the second version of method callMe has an additional *out*-argument. This is a sufficient distinction to allow overloading. It is also sufficient to have a difference in

In Franca contracts and deployment models it is necessary to uniquely reference single methods. In order to tell apart several overloaded methods with the same name, the method names can be extended by *selectors*, which are not part of the method name, but which can be used to discriminate overloaded methods. This is shown in the following example.

```
interface SomeInterface {
  method callMe:a {
    in { Boolean b }
  }

  method callMe:b {
```

```
    in { Boolean b }
    out { String s }
  }
}
```

The selectors for each group of overloaded methods must be unique, which is essential for their purpose to discriminate the methods which otherwise have an identical name.

Overloading is also relevant in the context of interface inheritance. For methods of derived interfaces which overload methods from base interfaces the same rules apply as described previously. See section 5.5.1 for more information on interface inheritance.

5.5.4 Broadcasts

A *broadcast* in a Franca interface is called by the server and will be received by the clients using the interface. A broadcast definition will contain a set of *out*-arguments, each with own type and name. Example:

```
interface ExampleInterface {
    broadcast buttonClicked {
        out {
            ButtonId id
            Boolean isLongPress
        }
    }
}
```

Arguments might have primitive types or user-defined types.

Selective broadcasts

The default communication pattern for broadcasts is that each server-initiated broadcast will be send to *all* connected clients. Optionally, the keyword *selective* can be given to indicate that the server might send this broadcast to one or a subset of the connected clients, depending on application logic and/or deployment information. Example syntax:

```
interface ExampleInterface {
    broadcast stop selective {
        out { Boolean immediately }
    }
}
```

The *selective* keyword will have the following implications:

- The client must be aware that the server has explicitly chosen to send the broadcast to it.
- There might be special *register()* functions (or similar) generated on client side in order to allow clients to decide if they want to be receivers of the selective broadcast(s). However, this is not mandatory and will usually be configured by the deployment model or defined implicitly for a given target platform.
- The server will have the opportunity to select among its clients when sending the broadcast (e.g., by code especially generated for this reason).

Broadcast overloading

Also for broadcasts Franca IDL allows overloading. For more details on restrictions, selectors and inheritance see section 5.5.3. The following example shows two overloaded broadcasts with selectors.

```
broadcast tsunamiDetected:a {
    out { String location }
}
broadcast tsunamiDetected:b {
    out { Double strength }
}
```

5.5.5 Interfaces managing interfaces

In some IPC mechanisms, e.g. D-Bus, we have the concept of linked interfaces. This usually takes the form that there is a root interface, for example a bluetooth service. Then there are many interfaces which are controlled by the root interface. E.g., each physical bluetooth adapter provides some interface which has to be controlled by the root bluetooth service.

Franca IDL allows to define this association between interfaces using the keyword **manages**. A downstream tool (e.g., a code generator) could then generate appropriate code for being informed about the ”children” in the implementation of the root.

The following example shows two interfaces which specify the services SomeService and OtherService. The third interface BluetoothMaster will control all connections based on one of the two former interface types. This relationship is modeled by the keyword **manages** and a list of managed services.

```
interface SomeService {
    method getDirectory { }
    method getFile { }
}

interface OtherService {
    method playMusic { }
}

interface BluetoothMaster manages SomeService, OtherService {
    attribute UInt16 connectedClients
    method disconnectAll { }
}
```

The **manages**-keyword was introduced in Franca 0.8.8.

5.6 Contracts

5.6.1 Basic concept of contracts

For each Franca interface, a *contract* might be specified which defines the dynamic behavior of the interface. Generally, if a contract is part of the interface, all interactions of clients and providers of this interface have to obey the specification in the contract. This is different from behavior specification by sequences (e.g., MSCs in MOST), which provides a set of allowed sequences, but in general doesn't require that all legal sequences have been specified.

Note that a contract is a specification of the interaction of a pair of one client instance and one provider instance. If multiple clients are connected to one interface on provider side, one instance of the contract is (conceptually) established for each client.

In order to specify the dynamic behavior of a Franca interface, a *PSM* (short for: *Protocol State Machine*) is defined which specifies the allowed states *of the connection* and

the allowed transitions between those states. Example:

```
// specification of dynamic behavior on the interface
contract {
    PSM {
        initial idle
        state idle {
            on call setActivePlayer -> working
        }
        state working {
            on signal attachOutput -> idle
        }
    }
}
```

The client/server connection specified by this interface can have two states:

- *idle*: The client is allowed to call the *setActivePlayer* method. No other interaction via the interface is allowed. This is the initial state. If the client calls the *setActivePlayer* method, the state will change to *working*.
- *working*: The server is allowed to send the broadcast *attachOutput*. No other interaction via the interface is allowed. After this broadcast has been sent, the state will change to *idle*.

5.6.2 Protocol state machines

The *Protocol State Machines* used for specifying Franca's interface contracts use *events* to specify the transitions from one state to another. Valid events might be:

- *call*: A method call initiated by the client.
- *respond*: The server's response for a client's method call. Note that *fireAndForget* methods do not trigger this kind of event (see Fire-and-forget methods (§5.5.3)).
- *signal*: A broadcast sent by the server.
- *set*: An attribute's value is being changed by a client.
- *update*: An indication by the server that an attribute's value has been changed.

A transition will be triggered by one out of a set of events. The actual transition can be guarded by a boolean condition.

5.6.3 Transition actions

For each transition of a PSM, an optional *action* can be defined. The action consists of a set of statements which will be executed each time the transition is triggered. Note that the execution of these statements will usually not be implemented by real code on the target system. Instead, it will be interpreted during some analysis of the interaction on the interface.

All *actions* are defined in terms of an *action language*, which is part of Franca IDL. This action language is purposefully small and simple in order to allow static analysis and interpretation of the actions.

Here is an example which shows the syntax of actions:

```
contract {
    PSM {
        initial idle
        state idle {
            on call setActivePlayer -> working {
                // put action code here
            }
```

```
        }
        state working {
            on signal attachOutput -> idle {
                // put action code here
            }
        }
    }
}
```

The elements of the action language are still subject to development and will be described in a later version of this document.

5.6.4 State variables

A PSM can refer to *state variables* which have been defined as part of the contract. The state variables are statically typed. Any guard or action of a transition can refer to the state variables. The boolean guards can check the values of either message arguments, attribute values or state variable values.

The following examples enhances the contract defined above by a state variable and an action counting the number of *setActivePlayer* method calls. The two transitions of state *working* are guarded; the guards are checking the value of the state variable. After 100 interactions, the PSM will enter state *silence* and will not accept any more events. The contract specifies an interface which allows a maximum of 100 calls of method *setActicePlayer*. As the *silence* state doesn't have any outgoing transitions, the system will have to be restarted in order to allow further activities on this interface.

```
contract {
    vars {
        UInt32 count;
    }
    PSM {
        initial idle
        state idle {
            on call setActivePlayer -> working {
                count = count + 1
            }
        }
        state working {
            on signal attachOutput [count<100] -> idle
            on signal attachOutput [count>=100] -> silence
        }
        state silence { }
    }
}
```

In a future release of Franca IDL there will be mandatory extensions for defining the value domains of the state variables (e.g., ranges for integer values). This will allow model checking of PSMs.

5.6.5 Exploiting contract information

The contract information can be used in a variety of ways:

- Development tools used for the implementation of components could use the information to guide the developer through the software design process.
- Test code could be generated on client or server side which checks for correct sequences at runtime.

- Target traces can be analysed offline by validating sequences against the contract.

The *contract*-feature in Franca is still under development. However, by specifying the proper and allowed behavior of interfaces in a consistent way the value and expressiveness of interface definitions can be increased to a large extent.

5.7 Comments

Franca supports two kinds of comments: unstructured and structured comments.

5.7.1 Unstructured comments

Unstructured comments are usually one-line and multi-line comments as well known from C, C++ or Java. Examples:

```
// this is a one-line unstructured comment
typedef TypeOne is UInt8

/* this is a multi-line unstructured comment,
   it could be used also for one-liners :-) */
typedef TypeTwo is Int16
```

5.7.2 Structured comments

Structured comments consist of tagged meta-information as known for example from JavaDoc. Each tag is started with an @-sign. The set of available tags is part of the Franca specification.

```
<** @description : Currently active player. All other players will
                   reject any requests.
    @author : E. Wiggin **>
attribute tPlayer activePlayer
```

Downstream tools can access the tags and their string content and use them for analysis and code generation.

All built-in tags are listed in Table 5.3 together with descriptions.

5.8 Fully qualified names, packages, and multiple files

5.8.1 Fully qualified names

A *fully qualified name* (or short: *FQN*) is a sequence of identifiers (at least one) separated by dots.

5.8.2 Package declarations

Each Franca file starts with a *package*-declaration (similar to Java). This puts all type collections and interfaces defined in that file into the package. The absolute reference for this top-level element is computed by concatening the package's FQN and the element's FQN. Example:

```
package org.franca.examples
interface ExampleInterface {
    // this interface can be globally accessed by the FQN
    // org.franca.examples.ExampleInterface
}
```

Tag	*Description*
@description	A comment with this tag contains a description of the documented interface or datatype element. Typically, the string content of this tag will be used for generated documentation.
@author	This tag specifies the author of the documented element.
@deprecated	This tag is used to mark the documented element as deprecated. The string content should point out a better solution which can be used in order to avoid using this element.
@source−uri	This tag can be used to point to an URI which specifies some kind kind of source information for this element. E.g., if an interface has been created by transformation from a D-Bus introspection file, the source URI could refer to the original D-Bus XML file.
@source−alias	This tag can be used in addition to @source−uri for specifying source elements for the corresponding Franca element.
@see	A comment with this tag specifies a further reading or any other kind of semantic reference.
@experimental	An element marked with this tag is not a stable part of the interface definition.
@details	This tag is deprecated, it will be removed in a future version. The @description tag should be used instead.
@param	This tag is deprecated, it will be removed in a future version. Instead, the arguments of methods and broadcasts should be documented using the @description tag for the corresponding arguments.
@high−volume	This tag is deprecated, it will be removed in a future version. Specific information related to performance and QoS aspects should be modeled using Franca deployment models.
@high−frequency	This tag is deprecated, it will be removed in a future version. Specific information related to performance and QoS aspects should be modeled using Franca deployment models.

Table 5.3: List of available tags for structured comments in Franca IDL.

5.8.3 Imports and namespace resolution

If elements in one Franca file need elements from another file, they can reference the latter only if a corresponding *import* statement is provided. There are two kinds of imports. *Model imports* will import all visible top-level elements in the imported file. However, these elements have to be referenced with their absolute FQN.

```
package org.franca.examples.demo

// model import
import model "basic_types.fidl"

interface ExampleInterface {
    // ...
}
```

Namespace imports will import all visible elements in the given namespace of the imported file. These elements can be referenced with a FQN which is relative to the given namespace.

```
package org.franca.examples.demo

// namespace import
import org.franca.examples.demo.* from "basic_types.fidl"

interface ExampleInterface {
    // ...
}
```

6. Franca Deployment Models

Franca deployment models offer the possibility to enhance Franca IDL interface specifications with additional information. This is a way to ensure that Franca IDL interfaces contain the core information needed to describe the semantics of APIs. All additional information is stored in some deployment model.

Figure 6.1 shows how Franca supports the various artifacts which have to be described when defining software architectures. The base elements are *interface definitions*. Those are completely supported by the Franca IDL. All other architecture artifacts are based on these interface definitions, among them:

- interface instances or ports,
- components like servers providing interface instances, as well as clients requiring interface instances,
- architectural layers where these components live,
- and the actual transport layer properties with aspects like serialization, encoding or quality of service.

This information is typically supported by Franca's configurable deployment models. However, Franca deployment models will not completely replace architecture definition languages (ADLs) or UML models. Instead, deployment models provide a means to store all information which is directly related to Franca interfaces. The deployment information usually *refines* the interfaces defined in Franca IDL.

Contents of Franca deployment models are related to the implementation of the interfaces on a target platform and the actual deployment of these interfaces on the platform. Some examples for this kind of data:

- How are the data types encoded on the target platform (e.g., endianness, padding)?
- Are calls blocking or non-blocking?
- How can the instances of an interface be found and addressed (i.e., service discovery)?
- Which quality-of-service promises are valid?

The section Guidelines for adding new features to Franca IDL (§3.5) in the Concepts (§3) chapter provides some guidelines for deciding whether new features belong to the Franca

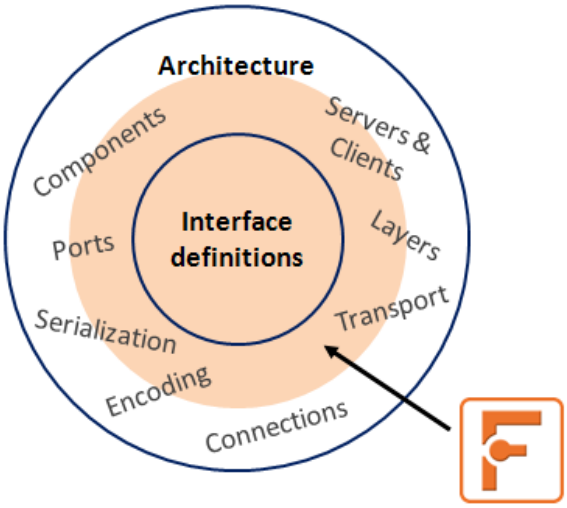


Figure 6.1: Architecture and interface artifacts and how Franca supports their definition.

IDL itself or to the deployment model.

The actual information in a deployment model depends heavily on the target IPC framework. Therefore, there is no generic deployment model (on a similar abstraction level as Franca IDL itself). Instead, the properties of the actual deployment model for the target platform can only be defined with the target platform in mind. This requires a flexible deployment modeling language which can be used to define the specification of the required deployment information as well as the actual property values for the concrete interfaces defined with Franca IDL.

6.1 Deployment model concepts

This section explains some basic concepts of Franca deployment models.

6.1.1 Deployment specifications and definitions

An interface defined with Franca IDL is independent of any target platform. Some code generator will be used to map this interface onto a real target platform, e.g., generating C++ classes and their implementations in order to implement the interface on a system were components are connected with D-Bus or another IPC mechanism.

In many cases, this generator will need additional information, which does not belong into the IDL definition. This information will be stored in an instance of Franca's deployment models. In order to do this, a separate Franca Deployment Language is available, which consists of two parts:

- *specification* of deployment properties (done only once per deployment target platform)
- *definition* of actual values for deployment properties (done for every Franca interface which has to be deployed on the target platform)

Figure 6.2shows how *deployment specifications*, *deployment definitions* and the actual Franca IDL interfaces are related. A specification is created for a given target platform (or code generator for this platform). Based on this specification, one deployment *definition* has to be provided for each actual Franca IDL interface. The specification describes which additional information has to be provided for an interface on this target platform - the definition lists this information for one concrete interface.

It is important to remember here that each Franca IDL interface can be deployed

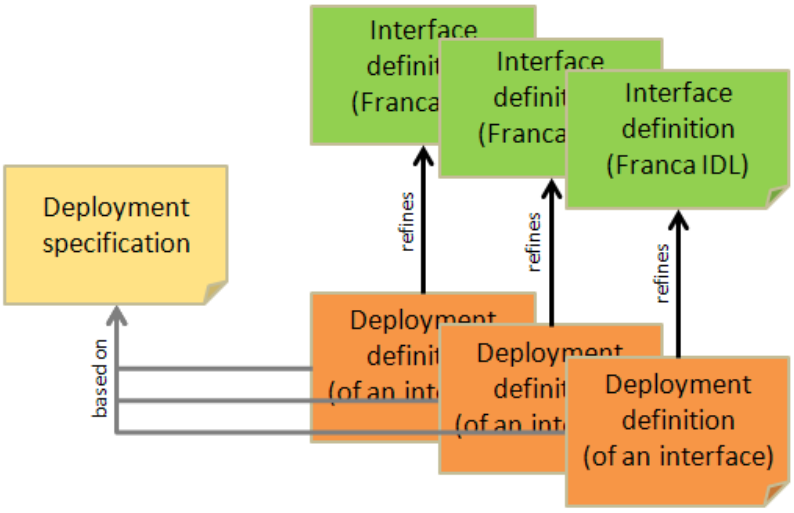


Figure 6.2: Franca deployment: specifications, definitions and actual interfaces.

on many different target platforms. Thus, there might be many deployment definitions (based on different deployment specifications) for the same interface. This is essentially the advantage of the IDL/deployment separation: The core part of each interface definition (the Franca IDL part) can be re-used in multiple environments.

6.1.2 Deployment properties

Properties are the basic units of information in deployment models. Each specification defines which properties are available by defining their *name*, their *type*, their *host* and some flags (e.g., a default value for the property, or if the property is mandatory or optional). Properties can be defined freely; there are no predefined properties. This ensures maximum flexibility when writing specifications.

The following example shows one property as part of a deployment specification. The property is named *PerformanceImpact*; it is located at the host *methods*. The property is of *enumeration* type, with enum values *none*, *medium* and *substantial*. It has the flag *optional*.

```
specification MySpec {
    for methods {
        PerformanceImpact: {none, medium, substantial} (optional);
    }
}
```

The *host* attribute of a property defines at what locations in a definition this property applies; this is a similar concept as *pointcuts* in aspect-oriented programming (but there are also some differences to that concept).

There is a variety of hosts; Table 6.1 and Table 6.2 and Table 6.2 list those related to Franca interfaces. Note that the properties for some hosts will be applied to several locations of a Franca IDL interface.

In a deployment *definition* for a given specification, concrete *values* have to be assigned to each property. Depending on the *host* definition and the optional/default flags on the specification side, properties will be mandatory at certain locations. This will be enforced by the deployment model editor and the underlying validation checks.

Host	*Description*	*Impacted Franca IDL elements*	*Example*
interfaces	Properties related to whole Franca IDL interfaces (but not its instances)	**interface**	locally or globally accessible?
type_collections	Properties related to Franca type collections	**typeCollection**	type visibility
attributes	Properties related to attributes of a Franca IDL interface	**attribute**	flag: read-only or read/write?
methods	Properties related to methods of a Franca IDL interface	**method**	calling style (blocking / non-blocking)
broadcasts	Properties related to broadcasts of a Franca IDL interface	**broadcast**	priority
arguments	Properties related to the arguments of methods and broadcasts	input and output arguments of **method**, output arguments of **broadcast**	flag: is optional?

Table 6.1: Franca deployment: Property hosts related to interface elements.

Host	*Description*	*Impacted Franca IDL elements*	*Example*
strings	Properties related to string variables	all entities with predefined type **String**: **attribute** members, arguments of **method** and **broadcast**, **struct** fields	encoding (ASCII, Unicode, ...)
integers	Properties related to all integer numbers	all entities with predefined type **Int8**, **UInt8**, ...: **attribute** members, arguments of **method** and **broadcast**, **struct** fields	allowed range, little vs. big endian
floats	Properties related to all float numbers	all entities with predefined type **Float** or **Double**: **attribute** members, arguments of **method** and **broadcast**, **struct** fields	encoding (ISO...)
numbers	Properties related to integer or float numbers	all entities of **integers** and **floats** combined	alignment
booleans	Properties related to booleans	all entities with predefined type **Boolean**	various flags (optional...)
byte_buffers	Properties related to byte buffer variables	all entities with predefined type **ByteBuffer**	maximal length of a buffer

Table 6.2: Franca deployment: Property hosts related to primitive data types.

Host	*Description*	*Impacted Franca IDL elements*	*Example*
structs	Properties related to struct definitions	**struct**	alignment / padding
struct_fields	Properties related to fields of struct definitions	fields of **struct**	alignment / padding
unions	Properties related to union definitions	**union**	flag for tag generation
union_fields	Properties related to fields of union definitions	fields of **union**	explicit tag for each field
enumerations	Properties related to enumeration definitions	**enumeration**	flag: generate enum values?
enumerators	Properties related to enumerators	values of **enumeration**	renaming, filtering
arrays	Properties related to array definitions	**array**	alignment, maximum length
typedefs	Properties related to typedefs	**typedef**	enriching type definitions

Table 6.3: Franca deployment: Property hosts related to user-defined data types.

Host	*Description*	*Impacted Franca IDL elements*	*Example*
providers	Properties of interface providers	n/a	process name, deployment node
instances	Properties related to instances of a Franca IDL interface (e.g., port)	n/a	network address of the interface instance

Table 6.4: Franca deployment: Property hosts related to providers and interface instances.

6.1.3 Providers and interface instances

Most properties will be attached to concrete locations of Franca interfaces, like methods, arguments or similar entities. However, the scope of the deployment concept usually contains components which provide interfaces (i.e., servers). Moreover, interface providers might offer several interfaces and even multiple instances of the same interface. Deployment models often have to support this, e.g., in order to define network addresses on the interface instance level.

These concepts are supported by Franca deployment models by the *host* definitions *providers* and *instances*. Table 6.4 lists the property hosts related to providers and interface instances.

6.2 Deployment specifications

This section explains how deployment specifications for a concrete target platform are defined.

6.2.1 Introductory example

Let's start with a concrete deployment example. The following deployment model contains both the specifiation part and the definition part. In the *specification*, the two properties *CallSemantics* and *IsOptional* are introduced. Both are defined for the *methods* host, which indicates that those properties will be relevant for all methods defined in a Franca IDL interface.

The property *CallSemantics* has an enumeration type with the constant values *synchronous* and *asynchronous*. As there is no further flag here, the property is mandatory and has to be set for all methods under the given specification *MySpec*. The property *IsOptional* of boolean type is also mandatory. Using this specification, it has to be specified for every method of a Franca IDL interface if the method is called synchronously or asynchronously (property *CallSemantics*) and if the method has to be implemented by the server or not (property *IsOptional*).

```
import "PlayerAPI.fidl"

specification MySpec {
    for methods {
        CallSemantics: {synchronous, asynchronous};
        IsOptional: Boolean;
    }
```

```
}

define MySpec for interface PlayerAPI {
    method setActivePlayer {
        CallSemantics = synchronous
        IsOptional = false
    }
}
```

The second part of the introductory example shows the application of the specification *MySpec* for the Franca interface *PlayerAPI*. In order to write this deployment definition, an *import* statement has to be given at the beginning which includes the *fidl*-file which contains the interface definition.

The example interface *PlayerAPI* contains exactly one method. In the deployment definition, we now have to enrich this method with the two mandatory properties defined in the specification. In the example, we define that method *setActivePlayer* will be called synchronously and has to be provided by every server offering the interface *PlayerAPI*.

Note: For typical applications of the deployment modeling concept, specifications and definitions will be stored as separate file. This is because usually the specification will be owned by a platform architect or a framework team, whereas the actual definitions will be created by the responsible architects and developers, who also define the actual interfaces with Franca IDL.

6.2.2 Deployment specification for interfaces

This section provides a more elaborate example of deployment specifications. It shows the full range of types, single types vs. arrays. It also shows how to define default values for properties. Note that the actual names of properties (and all their attributes) can be freely chosen by the specification designer. There are no predefined properties. However, the designer has to ensure that property names will be unique (the deployment editor will take care of this by validating the property names online). For the detailed restrictions regarding property names refer to section Property naming restrictions (§6.2.6).

The specification name is fully qualified in order to allow a hierarchical organization of specifications. Each property can be either defined as single value or as array. For example, property Range has been defined as array of integers, and the default value is an array which contains the values 1, 2 and 3. Property Groups is an array of enumeration values g1, g2, g3 and g4. Thus, values for this property might be an arbitrary subset of this enumeration.

```
specification org.deployspecs.SampleDeploySpec {
    for methods {
        CallSemantics: {synchronous, asynchronous} (default: asynchronous);
        Priority: {low, medium, high} (default: low);
        IsOptional: Boolean;
        Range: Integer[] (default: {1, 2, 3});
    }

    for attributes {
        IsReadOnly: Boolean (default: false);
        Groups: {g1, g2, g3, g4}[];
    }

    for strings {
        Encoding: {utf8, unicode};
```

```
    }
}
```

The example shows some properties for which reasonable default values have been defined. All properties which are neither optional nor have defaults are *mandatory*. Those properties must be defined at all impacted locations of the deployment definition (the locations are determined by the *host* of the property).

6.2.3 Providers and interface instances

This is a specification for the deployment of IP-based IPC stacks. This is just an example, an actual deployment specification for IP-based stacks will have to take into account more platform-specific details.

```
specification org.deployspecs.IPBasedIPC {
    for providers {
        ProcessName: String;
    }

    for instances {
        IPAddress: String;
        Port: Integer (optional);
        AccessControl: { local, subnet, global } (default: global);
    }
}
```

The interesting part of this example is the usage of the property *hosts* **providers** and **instances**. Those properties will not be related to actual entities of a Franca IDL interface, but to some additional architecture-related parts instead. This is depicted by the outer ring in the overview diagram at the beginning of this chapter.

6.2.4 Specification Inheritance

Specifications can inherit from base specifications. Example:

```
specification org.deployspecs.DerivedSpec extends org.deployspecs.IPBasedIPC {
    // put specification details here
}
```

The properties defined in a specification and its base specification will be aggregated. This applies recursively, such that a specification will consist of an accumulation of all its own properties and the properties of all base specifications along the chain.

This concept allows to build a hierarchy of specifications (i.e., a specification ontology).

6.2.5 The property datatype 'Interface'

Each property defined in a deployment specification is strongly typed. In the examples above, the datatypes **Boolean**, **Integer**, **String** and ad-hoc enumerations have been used.

In some use cases, it is necessary to establish relationships between Franca interface definitions. E.g., in a large-scale system each functional interface might have a corresponding diagnostics interface providing additional functionality used for testing and observation.

The following example shows how a property with the datatype **Interface** is defined. In the example, for each Franca interface the *Master* property can be set in order to define another interface which is the master of this interface. The example also defines a specific interface reference *&org.example.idl.Controller* as a default value.

```
specification org.deployspecs.Channel {
    for interfaces {
        Master: Interface (default: &org.example.idl.Controller);
    }

    // put more specification details here
}
```

Note that the **Interface** datatype allows to refer to interface *definitions*, not interface *instances*. If you are looking for a way to establish relations between interface instances, the **manages** concept might be a proper solution (see section Interfaces managing interfaces (§5.5.5)).

6.2.6 Property naming restrictions

Property are identified by their name, which is defined in the deployment specification. As each property can be used in deployment definitions specifically depending on the property host, their names are in general not required to be unique. For example, the following specification is valid, although there are properties with the same name:

```
specification SpecWithDuplicates {
    for structs {
        Property1: Integer;
    }
    for arguments {
        Property1: String;
    }
}
```

This is valid because in a deployment definition the property can be identified from the context. The property type may be different as in the example or equal (however, the property type *enumeration* is an exception from this rule, see below).

The following counterexample shows a specification which is not valid, because the two properties with the same name cannot be distinguished from the context:

```
specification InvalidSpecWithDuplicates {
    for numbers {
        Property2: Integer; // invalid
    }
    for floats {
        Property2: String; // invalid
    }
}
```

The hosts **numbers** and **floats** overlap, i.e., in a deployment definition it cannot be decided which of the two properties Property2 is referred to. Thus, validation errors would be shown for this deployment specification. In this case, a meaningful interpretation could be defined for the special case of equal property types. However, as this would increase the redundancy of the specification the validation will reject it even if the property types are equal.

The following list gives all restrictions in place for properties with same name.

- Properties with same name must not have the same property host.
- Properties with same name must not have property hosts with overlapping scope. Example: **arrays**, **numbers**, **integers floats**, **booleans** and **strings** are mutually exclusive. Another example: **struct_fields** and **union_fields** are mutually exclusive.

- A property with a property host which can be used in multiple contexts cannot be combined with another property with the same name and a different host, even if the scopes of the two hosts do not overlap. Example: A property with host **numbers** blocks defining another property with the same name and host **structs**, because the **numbers**-property can be used in multiple contexts.
- Properties with same name and ad-hoc *enumeration* types are not allowed.

The reason for the two latter restrictions is more subtle than for the other ones. Although the property might be identified uniquely by its context, the resulting Java code for the generated PropertyAccessor class would be ambiguous or not correct. See section PropertyAccessor classes (§6.4.1) for details on the generated Java classes.

6.3 Deployment definitions

This section describes how deployment definitions are created based on deployment specifications.

6.3.1 Interface deployment

This section provides a more elaborate example of deployment definitions for a Franca interface. The specification for this deployment is located in the separate file *SampleDeploySpec.fdepl*, which is imported in the first line of the example. The second line is another import statement; it imports the actual France IDL interface which should be refined by the deployment data in the remaining parts of this file.

The deployment definition for the interface *PlayerAPI* explicitly references the specification *SampleDeploySpec*. You should compare this example with the specification from section Deployment specification for interfaces (§6.2.2).

```
import "deployspecs/SampleDeploySpec.fdepl"
import "../franca/demo1.fidl"

define SampleDeploySpec for interface org.franca.examples.demo.PlayerAPI {
    attribute activePlayer {
        IsReadOnly = true
        Groups = { g1, g2 }
    }

    method setActivePlayer {
        Priority = medium
        IsOptional = false
    }

    method getPlayerInfo {
        CallSemantics = synchronous
        Priority = low
        IsOptional = true

        out {
            name {
                Encoding = utf8
            }
            description {
                Encoding = unicode
            }
        }
```

```
    }
}
```

Note that the structure of the deployment definition resembles the structure of the interface definition in Franca IDL itself.

6.3.2 Deployment of interface providers

This section provides a detailed example of deployment definitions for interface providers and interface instances. The specification for this deployment is again located in the separate file *SampleDeploySpec.fdepl*, which is imported in the first line of the example (same as in previous example). The second line is another import statement; it imports the actual France IDL interface which will be referred to by the deployment data in the remaining part of this file.

The deployment definition for the provider *ExampleServer* explicitly references the specification *SampleDeploySpec*. If you compare this with the specification from section Deployment specification for interfaces (§6.2.2), you will not find any property declarations for hosts *providers* or *instances*. These declarations are inherited from the base specification *IPBasedIPC*. See section Specification Inheritance (§6.2.4) for the inheritance feature of Franca deployment models.

```
import "deployspecs/SampleDeploySpec.fdepl"
import "../franca/demo1.fidl"

define SampleDeploySpec for provider ExampleServer {
    ProcessName = "server1.exe"

    instance org.franca.examples.demo.PlayerAPI {
        IPAddress = "192.168.1.50"
        Port = 8765
        AccessControl = subnet
    }

    instance org.franca.examples.demo.PlayerAPI {
        IPAddress = "192.168.1.50"
        Port = 7654
    }
}
```

In the example definition above, the component *ExampleServer* is defined which provides two instances of the same Franca interface *PlayerAPI*. The instances can be addressed with the same IP address, but different port numbers. The first instance of this interface can be accessed by clients in the same subnet only. The second instance can be accessed globally, because this value has been defined as default in the *IPBasedIPC*-specification and hasn't been overridden in the example above. This is just an example which shows the difference between actual interface definitions and their instances.

6.3.3 Overwriting deployment properties

In general, each user-defined datatype from a type collection or an interface has to be deployed once. I.e., a deployment definition for the type collection or interface has to contain a section for the data type, containing at least the mandatory deployment properties and (if desired) some of the optional properties.

There are various locations where a user-defined datatype might be used in a type collection or an interface definition:

- as type of an attribute
- as type of an in/out argument of a method
- as type of an out argument of a broadcast
- by another datatype (via containment or inheritance)
- as type of a constant

By default, all these usages of a deployed datatype do not require any further deployment data aside from the deployment definition mentioned above. This is usually a proper approach, as it ensures that the deployment of a user-defined type is the same for all its usages.

However, for some usecases it is needed to overwrite the initial datatype deployment and provide different deployment data depending on how the datatype is used. Currently this *overwriting* of deployment properties is supported for compound-like datatypes, i.e., for struct and union types. The remainder of this section will explain how overwriting can be used when creating deployment definitions.

The overwriting-feature is available since Franca 0.10.0.

Example using normal deployment without overwrites

This section will introduce a running example which will be used to explain all variants of the overwriting-feature. We will start with a simple deployment specification:

```
specification examples.SimpleSpec {
    for strings {
        Encoding : {utf8, utf16, unicode} (default: utf8);
    }
}
```

This specification contains only one property, which is applied to string types. It can be used to define string encodings, which can be one of utf8, utf16 and unicode. Note that a default value has been defined, which relieves us of defining a value of the Encoding property for each usage of datatype **String**.

In the remainder of this section, we will deploy different parts of the following Franca interface:

```
interface MediaPlayer {
    struct Person {
        String firstname
        String surname
    }

    struct TrackInfo {
        String title
        String album
        Person composer
        Person interpret
    }

    attribute TrackInfo currentTrack

    method play {
        in { TrackInfo track }
    }
}
```

The interface defines two struct datatypes (and one is used by the other), as well as one attribute and a method. Both the attribute and the method are using the struct type TrackInfo.

A deployment definition for this interface could look like the following example. For the struct types Person and TrackInfo, some elements of type **String** will not be deployed with the default Encoding utf8 (which is defined in the deployment specification), but with Encoding unicode instead.

```
define examples.SimpleSpec for interface examples.MediaPlayer {
    struct Person {
        firstname { Encoding = unicode }
        surname { Encoding = unicode }
    }
    struct TrackInfo {
        album { Encoding = unicode }
    }
}
```

This is the standard way of using deployment definitions. In the next section we will explain how overwrites can be used.

Example using deployment overwrites: Attribute

In some environments, it might be necessary to use a different encoding for the album element of the TrackInfo struct type, but only when used as a Franca attribute. This can be defined by overwriting the deployment properties given when the TrackInfo struct was deployed. In the example, the Encoding property of the struct element album will be overwritten with utf16.

```
define examples.SimpleSpec for interface examples.MediaPlayer {
    attribute currentTrack {
        #struct {
            album { Encoding = utf16 }
        }
    }
}
```

The syntax for overwriting a struct type deployment is the hash-character followed by the keyword *struct*, resulting in #**struct**. This and all following deployment overwrite examples will work also with **union**-types, using the keyword #**union**. The overwrite keyword always refers to the type of the containing deployment element. In the example, this is the **attribute** currentTrack.

Table 6.5 lists the different "layers" of values for property Encoding in the example above.

Example using deployment overwrites: Method

Overwriting deployment properties for elements of struct and union types is not only possible for attribute types, but also for arguments of methods and broadcasts. The following example shows how to overwrite the deployment of the album element of the TrackInfo struct type, but only when used as the type of the in argument track of method play. In the example, the Encoding property of the struct element album will be overwritten with utf16.

```
define examples.SimpleSpec for interface examples.MediaPlayer {
    method play {
        in {
            track {
```

Layer	*Source*	*Value*	
Default value	specification	utf8	overridden by...
Standard deployment	deployment for TrackInfo	unicode	overwritten by...
Deployment overwrite	deployment for currentTrack	utf16	actual value!

Table 6.5: Franca deployment overwrites: Layered values in example.

```
            #struct {
                album { Encoding = utf16 }
            }
        }
    }
}
```

Example using deployment overwrites: Nested struct

If struct and union types are built from other struct or union types, it is also possible to overwrite the deployment properties of the child structs. The following example overwrites the Encoding properties of the elements firstname and surname of the Person struct type, but only if it is used as type of element composer of the TrackInfo struct type.

```
define examples.SimpleSpec for interface examples.MediaPlayer {
    struct TrackInfo {
        title { Encoding = utf8 }
        album { Encoding = utf8 }
        composer {
            #struct {
                firstname { Encoding = unicode }
                surname { Encoding = unicode }
            }
        }
    }
}
```

Note the difference between the keywords **struct** (without hash) and **#struct** (with hash). The former starts the original deployment of struct type TrackInfo, the latter triggers the overwriting of its element composer. It is also possible to overwrite struct types which are more deeply nested in hierarchies of struct and union types.

Example using deployment overwrites: Nested struct in attribute

The following example shows overwriting of nested structs in the context of attributes. The same could be done for methods or broadcasts.

```
define examples.SimpleSpec for interface examples.MediaPlayer {
    attribute currentTrack {
        #struct {
            composer {
                #struct {
                    firstname { Encoding = unicode }
```

```
                surname { Encoding = unicode }
            }
        }
    }
}
```

In the above example, all elements of struct type TrackInfo except its element composer are deployed in a standard way. The deployment for element composer is overwritten and set to property values specifically for the context of the attribute currentTrack.

6.4 Support for accessing deployment properties

It is important that deployment properties can be accessed easily during code generation, testcase generation and other post-processing steps. Franca provides some infrastructure for retrieving the deployment data attached to various interface entities easily.

6.4.1 PropertyAccessor classes

Deployment properties will always be retrieved for a given element of a Franca IDL model. E.g., if the code generator implementation needs to access all information available for an actual FAttribute (§8.4.7) object, it shouldn't be necessary to traverse the deployment model, find all properties for this attribute object, check if there are defaults for some of these properties, type-check and cast the actual values of the properties and much more. In order to encapsulate this functionality and just offer a method to get a property value for a given Franca IDL model element, Franca provides a *PropertyAccessor* class, which does all the steps mentioned above.

The actual methods provided by a *PropertyAccessor* class will depend on the underlying deployment specification (§6.2). As this specification is part of the deployment language and will be changed by the user, the *PropertyAccessor* Java class will be generated by the Franca infrastructure while editing specifications with the Eclipse IDE.

6.4.2 PropertyAccessor example

The following simple example shows a deployment specification:

```
specification org.deployspecs.SimpleSpec
{
    for attributes {
        WillChangeOften: Boolean (optional);
    }
}
```

From this user-defined specification Franca will generate the following *PropertyAccessor* class:

```
/*******************************************************************************
* This file has been generated by Franca's FDeployGenerator.
* Source: deployment spec 'org.deployspecs.SimpleSpec'
*******************************************************************************/
package org.deployspecs;

import org.franca.core.franca.FAttribute;
import org.franca.deploymodel.core.FDeployedInterface;
```

```
/**
 * Accessor for deployment properties for 'org.deployspecs.SimpleSpec' specification
 */
public class SimpleSpecInterfacePropertyAccessor
{
    private FDeployedInterface target;

    public SimpleSpecInterfacePropertyAccessor (FDeployedInterface target) {
        this.target = target;
    }

    public Boolean getWillChangeOften (FAttribute obj) {
        return target.getBoolean(obj, "WillChangeOften");
    }
}
```

The method *getWillChangeOften()* of the generated class *SimpleSpecInterfacePropertyAccessor* should be used to get the property value *WillChangeOften* for a given attribute. The *PropertyAccessor* will retrieve the value of this property (either the actual value or the default) and will return it in a type-safe way. This is a convenient technique to get property data from deployment models.

6.4.3 Creating InterfacePropertyAccessors

In order to instantiate a specific *InterfacePropertyAccessor* object, an *FDInterface* object has to be retrieved from a deployment model. Deployment models are represented by an *FDModel* object. The helper class *FDModelExtender* provides a means to retrieve all interface deployment definitions of an *FDModel*.

The following Java snippet loads a deployment model from file system, creates a model extender in order to retrieve all *FDInterface* objects and creates the actual *InterfacePropertyAccessor* object for each of them.

```
FDModel fdmodel = FDModelHelper.instance().loadModel(inputfile);
FDModelExtender fdmodelExt = new FDModelExtender(fdmodel);
for(FDInterface fdi : fdmodelExt.getFDInterfaces()) {
    FDeployedInterface deployed = new FDeployedInterface(fdi);
    SimpleSpecInterfacePropertyAccessor accessor =
        new SimpleSpecInterfacePropertyAccessor(deployed);
    // use accessor, e.g., accessor.getWillChangeOften()

    FInterface api = fdi.getTarget();
    // call downstream tool (e.g., code generator) for FInterface 'api'
}
```

Note how the actual Franca IDL interface can be accessed from each FDInterface (i.e., the deployment specification of an interface) by using the *getTarget()* property.

6.4.4 ProviderPropertyAccessors

For properties related to providers and interface instances (§6.1.3), another *PropertyAccessor* will be generated, whose name is created as concatenation of the deployment specification's name and the suffix *ProviderPropertyAccessor*.

The creation and usage of *ProviderPropertyAccessors* is similar to the usage of *InterfacePropertyAccessors* as described above.

6.4.5 Example project

The example project *org.franca.examples.deploy* contains two example generators in the package

org.franca.examples.deploy.generators:

- *ExampleHppGeneratorWithDeployment*: This example code generator shows how deployment information can be accessed during traversal of an actual Franca IDL interface.
- *ExampleRuntimeConfigGenerator*: This example generator shows how deployment information can be accessed, which is not directly linked to an actual Franca IDL interface (e.g., interface providers and instances).

III Part Three

7. Franca connectors

The integration of Franca with other IDLs or (more generally:) models is supported by *Franca connectors*. A *connector* is yet another Eclipse plugin, which offers means to load and save models of the other IDL and transformations to and from Franca IDL models.

This chapter lists the connectors that are currently supported by Franca out of the box. The connectors can be selected during installation by choosing an additional feature provided by the Franca update site.

Note that most connectors will require additional plugins which provide the model or API (or both) for the 3rd party IDL. Thus, for installing some of those connector features, it may be necessary to install the required plugins first.

7.1 Franca support for D-Bus Introspection

D-Bus Introspection is an XML format which defines D-Bus interfaces. More details on the D-Bus IPC and D-Bus Introspection can be found at freedesktop.org.

The Franca connector plugin for D-Bus is called *org.franca.connectors.dbus*. It requires the plugin *model.emf.dbusxml*, which is available as open-source project *dbus-emf-model* on Eclipse Labs. An update site is available for *dbus-emf-model* (will not open in the browser, only in Eclipse IDE):

http://kbirken.github.io/dbus-emf-model/releases/

If you install the *Franca D-Bus support Feature* via *Help > Install New Software...*, the *model.emf.dbusxml* plugin will be installed automatically from this update site.

7.2 Franca support for OMG IDL

Describe OMG IDL connector

7.3 Franca support for Google Protobuf

Describe Google Protobuf connector

IV Part Four

8. Franca Model API

8.1 How can Franca models be accessed programmatically?

The Franca model infrastructure is based on the Eclipse Modeling Framework (EMF). The Franca core model (§3.1.1) is implemented as a EMF *ecore*-model. EMF provides a tool to generate a Java API from this *ecore*-model.

In Franca, this Java API is provided by the *org.franca.core* Eclipse plugin. The Java interfaces for the API are located in package *org.franca.core.franca*.

For accessing Franca deployment models programmatically, we recommend to use the helper classes provided by Franca and the classes automatically generated from deployment specifications. Thus, we do not include documentation for the EMF *ecore*-model API of deployment models here. See section Support for accessing deployment properties (§6.4) for more details about this topic.

8.2 Franca Model API Reference

8.2.1 General remarks

For each concept of Franca IDL, there is a Java interface in the API. As all these concepts have names which are quite common in the software architecture domain, all these API interfaces are prefixed with the letter "F" (for Franca). Examples: *FInterface*, *FAttribute*, *FMethod*.

The following properties are common for most of these entities:

- *name*: most of the Franca API entities have a name.
- *comment*: most of the Franca API entities can have a structured comment. The structured comment (of type *FAnnotationBlock*) is optional and might contain 0..* annotations.

8.2.2 FrancaFactory and FrancaPackage

The API offers a singleton class *FrancaFactory*, which should be used to create new instances of any model entities. See information about EMF to find out how the factory works. The

second singleton class of the API is the *FrancaPackage*, which does the initialization of the model and provides meta-information like ids and others.

Here is a Java code snippet which illustrates how a Franca model element can be created using the factory.

```
FInterface api = FrancaFactory.eINSTANCE.createFInterface();
api.setName("MediaPlayer");
// and so on
```

8.3 Utility classes for Franca model access

Accessing the Franca EMF model directly may be tedious, depending on the part of the model which should be traversed. Franca includes a couple of utility classes which simplify extracting data from a Franca model. This section describes those utility classes.

8.3.1 Evaluating FExpression objects with the ExpressionEvaluator

The ExpressionEvaluator will be described here.

8.3.2 Converting integer types with the IntegerTypeConverter

Some downstream tools only process primitive integer types and cannot handle ranged integer types. Franca provides the helper class IntegerTypeConverter which can do conversions from ranged integer types to predefined basic integer types and vice versa. The following code snippet shows how to convert ranged integer types to primitive integers:

```
public void generateCode(FModel model) {
    boolean haveUnsigned = true;
    IntegerTypeConverter.removeRangedIntegers(model, haveUnsigned);
    generateCode(model);
}
```

The function removeRangedIntegers will choose the smallest primitive type which can still represent the given number range. Therefore, for this conversion from ranged integers to predefined integers it can be configured if in the target type system *unsigned* types are available or not. E.g., for converting a Franca model towards a Java platform the usage of unsigned types can be disallowed.

The class IntegerTypeConverter provides a second static member function converting primitive integer types to ranged integers. There is a fixed mapping from each primitive type to the proper range of the ranged integer. This is a code example showing how to use this conversion function:

```
FModel model = ...;
IntegerTypeConverter.removePredefinedIntegers(model);
processModel(model);
```

This class can be used as a preprocessor for existing code generators or transformations. For both utility functions, the input model is transformed in-place, i.e., its FTypeRef objects are converted directly.

8.4 API for Franca models, interfaces and type collections

Some details of the ModelAPI reference section might be outdated.

This section describes the FModel class, which is the root class for each Franca model. It also describes FTypeCollection as well as FInterface and all its elements (e.g., attributes, methods and broadcasts. See the section Interface definition (§5.5) in the Franca User Guide for detailed information about interfaces.

8.4.1 Class FModel

The root class of a Franca model. It contains a list of interfaces and a set of type collections. Other Franca models might be referenced by the 'imports' attribute. The name of a Franca model is the package declaration.

- EString *name*: The package declaration for this model file.
- List<Import (§8.4.9)> *imports*: The list of import statements for this model.
- List<FInterface (§8.4.3)> *interfaces*: The list of interfaces which are defined in this model.
- List<FTypeCollection (§8.4.2)> *typeCollections*: The list of type collections in this model.

8.4.2 Class FTypeCollection

A collection of Franca type definitions. The type collection is named and should be versioned (optional). Types defined by a FTypeCollection (§8.4.2) can be referenced from other FTypeCollection (§8.4.2)s and FInterface (§8.4.3)s.

- EString *name* (optional): The name of this element. Inherited from base class FModelElement.
- FAnnotationBlock (§8.7.1) *comment* (optional): The structured comment for this element (if any). Inherited from base class FModelElement.
- FVersion (§8.4.8) *version* (optional): The version of this entity. The Apache major/minor scheme is applied.
- List<FType (§8.5.1)> *types*: The list of Franca types defined in this entity.

8.4.3 Class FInterface

This class represents a Franca interface definition. Interfaces are named and should be versioned (optional). Types defined as part of this interface can not be referenced by other FTypeCollection (§8.4.2)s and FInterface (§8.4.3)s. This type visibility differs from the types defined as part of FTypeCollection (§8.4.2)s.

- EString *name*: The name of this element. Inherited from base class FModelElement.
- FAnnotationBlock (§8.7.1) *comment* (optional): The structured comment for this element (if any). Inherited from base class FModelElement.
- FVersion (§8.4.8) *version* (optional): The version of this entity. The Apache major/minor scheme is applied. Inherited from base class FTypeCollection (§8.4.2).
- List<FType (§8.5.1)> *types*: The list of Franca types defined in this entity. Inherited from base class FTypeCollection (§8.4.2).
- List<FAttribute (§8.4.7)> *attributes*: The list of attributes defined for this interface.
- List<FMethod (§8.4.6)> *methods*: The list of methods defined by this interface.
- List<FBroadcast (§8.4.4)> *broadcasts*: The list of broadcasts defined by this interface.
- FContract (§8.6.1) *contract* (optional): The contract of this interface. A contract specifies the semantics of the interface, e.g., the order of the events across this interface.
- FInterface (§8.4.3) *base* (optional): The base interface. Franca allows single inheritance.

- List<FInterface (§8.4.3)> *managedInterfaces*: The list of interfaces managed by this interface. The interface will provide methods for discovery and handling of the runtime instances of the managed interfaces. The actual implementation depends on the specific target runtime platform.

8.4.4 Class FBroadcast

The definition of a Franca broadcast as part of an FInterface. See the Franca IDL chapter in the Franca User Guide for very detailed information on broadcast semantics.

- EString *name*: The name of this element. Inherited from base class FModelElement.
- FAnnotationBlock (§8.7.1) *comment* (optional): The structured comment for this element (if any). Inherited from base class FModelElement.
- List<FArgument (§8.4.5)> *outArgs*: The parameters of this broadcast.
- EString *selective* (optional): A flag which indicates that this broadcast will not always be send to all clients. The property will be set to a concrete string if the flag is set (e.g., the string "selective"). Do not rely on the actual value of this string. If the property is null, the flag isn't set.

8.4.5 Class FArgument

This class represents an argument (aka parameter) for a FMethod (§8.4.6) or FBroadcast. For methods, this might be an input or output argument. For FBroadcast (§8.4.4)s, there are only output arguments (i.e., from server to client).

- EString *name*: The name of this element. Inherited from base class FModelElement.
- FAnnotationBlock (§8.7.1) *comment* (optional): The structured comment for this element (if any). Inherited from base class FModelElement.
- FTypeRef (§8.5.3) *type*: The actual type of this typed element. The type might be predefined or user-defined. Inherited from base class FTypedElement (§8.6.13).
- EString *array* (optional): If the typed element is an implicit array definition, this property will be set to a string (e.g., the string '[]'). If the typed element is a single value (not an array), this property will be null. Do not rely on the actual value of the string. Inherited from base class FTypedElement (§8.6.13).

8.4.6 Class FMethod

The definition of a Franca method as part of an FInterface. Methods without out arguments might have be flagged as 'fireAndForget'. See the Franca IDL chapter in the Franca User Guide for very detailed information on method semantics.

- EString *name*: The name of this element. Inherited from base class FModelElement.
- FAnnotationBlock (§8.7.1) *comment* (optional): The structured comment for this element (if any). Inherited from base class FModelElement.
- EString *fireAndForget* (optional): A flag which indicates that this method will be just a request from client to server without response. The property will be set to a concrete string if the flag is set (e.g., the string "fireAndForget"). Do not rely on the actual value of this string. If the property is null, the flag isn't set. It can be set only for methods without output arguments.
- List<FArgument (§8.4.5)> *inArgs*: The list of input arguments for this method (aka input parameters).
- List<FArgument (§8.4.5)> *outArgs*: The list of output arguments for this methods (aka return values).
- FEnumerationType (§8.5.5) *errorEnum* (optional): If this property is not null, it represents a definition of method error codes by referencing an enumeration type

specified elsewhere. The enumerators of this enumeration type are the error codes the method might return. The enumeration type referenced here might reference a base enumeration. If this property is null, check the errors property instead.
- FEnumerationType (§8.5.5) *errors* (optional): If this property is not null, it represents a definition of method error codes by explicitly specifying a list of enumerators. These enumerators are the error codes the method might return. The enumeration type specified here might reference a base enumeration. If this property is null, check the errorEnum property instead.

8.4.7 Class FAttribute

The definition of a Franca attribute as part of an FInterface. See the Franca IDL chapter in the Franca User Guide for very detailed information on attribute semantics and the flags which can be set for attributes.
- EString *name*: The name of this element. Inherited from base class FModelElement.
- FAnnotationBlock (§8.7.1) *comment* (optional): The structured comment for this element (if any). Inherited from base class FModelElement.
- FTypeRef (§8.5.3) *type*: The actual type of this typed element. The type might be predefined or user-defined. Inherited from base class FTypedElement (§8.6.13).
- EString *array* (optional): If the typed element is an implicit array definition, this property will be set to a string (e.g., the string '[]'). If the typed element is a single value (not an array), this property will be null. Do not rely on the actual value of the string. Inherited from base class FTypedElement (§8.6.13).
- EBoolean *readonly* (optional): Indicates if this attribute is read-only. If false, clients are granted write access to the attribute.
- EBoolean *noSubscriptions* (optional): Indicates if clients can subscribe to update events for this attribute. If false, subscribing is possible. The reverse logic of this flag is to ensure a proper default in the IDL: if the keyword "noSubscriptions" is not given, subscription is possible.

8.4.8 Class FVersion

The version of this interface or type collection. It is defined according to the Apache major/minor scheme for interfaces. I.e., a change in the major number indicates a non-compatible change. It is mandatory to define both major and minor numbers.
- EInt *major*: The major number of this version specification (e.g., the "1" in "1.0").
- EInt *minor*: The minor number of this version specification (e.g., the "0" in "1.0").

8.4.9 Class Import

An import declaration. It defines a namespace from some Franca resource, which should be imported. All elements from other Franca models referenced by this model should be member of some imported Franca model.
- EString *importedNamespace* (optional): The namespace which is addressed by this import.
- EString *importURI* (optional): The URI of the imported resource.

8.5 API for Franca types

This section describes all API classes needed for Franca type definitions. See the section Data types (§5.1) in the Franca User Guide for detailed information about defining types with Franca.

8.5.1 Class FType (abstract)

This is the base class for all user-defined Franca types. It will never be instantiated directly.

- EString *name*: The name of this element. Inherited from base class FModelElement.
- FAnnotationBlock (§8.7.1) *comment* (optional): The structured comment for this element (if any). Inherited from base class FModelElement.

8.5.2 Class FMapType

The Franca map type (sometimes "map" is also called "dictionary"). This is a collection type which maps objects of a *key* type to objects of a *value* type in constant time.

- EString *name*: The name of this element. Inherited from base class FModelElement.
- FAnnotationBlock (§8.7.1) *comment* (optional): The structured comment for this element (if any). Inherited from base class FModelElement.
- FTypeRef (§8.5.3) *keyType*: The key type which is mapped to the value type. Note: Some IDLs (e.g. D-Bus Introspection) do not allow complex types as keys. Franca doesn't have this restriction.
- FTypeRef (§8.5.3) *valueType*: The value type of this map. It might be a predefined or a user-defined type.

8.5.3 Class FTypeRef

This class is a reference to some Franca type. It may be either a primitive type (§5.1.1) (property *predefined* or a derived type (e.g., struct, array, map).

- FBasicTypeId *predefined* (optional): If the referenced type is a predefined type, this property gives the actual predefined type. If the reference type is a user-defined type, this property will be null.
- FType (§8.5.1) *derived* (optional): If the referenced type is a user-defined type, this property gives the actual complex type definition. If the reference type is a predefined type, this property will be null.

8.5.4 Enum FBasicTypeId

This enum represents the basic types available in Franca IDL. See section Primitive types (§5.1.1) in the Franca User Guide for a list of available primitive types and their semantics.

This enum consists of the following literals: *UNDEFINED*, *INT8*, *UINT8*, *INT16*, *UINT16*, *INT32*, *UINT32*, *INT64*, *UINT64*, *BOOLEAN*, *STRING*, *FLOAT*, *DOUBLE*, *BYTE_BUFFER*.

8.5.5 Class FEnumerationType

This class represents an enumeration type. The enumeration will contain a list of *enumerators*. It can be derived from a base enumeration.

- EString *name*: The name of this element. Inherited from base class FModelElement.
- FAnnotationBlock (§8.7.1) *comment* (optional): The structured comment for this element (if any). Inherited from base class FModelElement.
- List<FEnumerator (§8.5.6)> *enumerators*: The list of enumerators of this enumeration.
- FEnumerationType (§8.5.5) *base* (optional): The base type of this enumeration. Will be null if this enumeration is not derived from any other enumeration.

8.5.6 Class FEnumerator

- EString *name*: The name of this element. Inherited from base class FModelElement.

- FAnnotationBlock (§8.7.1) *comment* (optional): The structured comment for this element (if any). Inherited from base class FModelElement.
- EString *value* (optional): The value of this enumerator. As the value definition for an enumerator is optional, value might be null.

8.5.7 Class FTypeDef

- EString *name*: The name of this element. Inherited from base class FModelElement.
- FAnnotationBlock (§8.7.1) *comment* (optional): The structured comment for this element (if any). Inherited from base class FModelElement.
- FTypeRef (§8.5.3) *actualType*: The actual type for this type alias.

8.5.8 Class FCompoundType (abstract)

This class represents all kinds of compound types in Franca, i.e., struct and union types. A compound type has a list of fields; each field is itself specified by its type. Thus, nested compounds can be created. The ordering of fields in the compound is relevant. E.g., serialization code generated from a Franca interface must take into account the order of the fields.

- EString *name*: The name of this element. Inherited from base class FModelElement.
- FAnnotationBlock (§8.7.1) *comment* (optional): The structured comment for this element (if any). Inherited from base class FModelElement.
- List<FField (§8.5.11)> *elements*: The elements (aka fields) of this compound type.

8.5.9 Class FUnionType

This class represents a union type in Franca. See its base class FCompoundType (§8.5.8) for a detailed description.

- EString *name*: The name of this element. Inherited from base class FModelElement.
- FAnnotationBlock (§8.7.1) *comment* (optional): The structured comment for this element (if any). Inherited from base class FModelElement.
- List<FField (§8.5.11)> *elements*: The elements (aka fields) of this compound type. Inherited from base class FCompoundType (§8.5.8).
- FUnionType (§8.5.9) *base* (optional): The base union from which this union inherits. Franca supports single inheritance for unions.

8.5.10 Class FStructType

This class represents a struct type in Franca. See its base class FCompoundType (§8.5.8) for a detailed description.

- EString *name*: The name of this element. Inherited from base class FModelElement.
- FAnnotationBlock (§8.7.1) *comment* (optional): The structured comment for this element (if any). Inherited from base class FModelElement.
- List<FField (§8.5.11)> *elements*: The elements (aka fields) of this compound type. Inherited from base class FCompoundType (§8.5.8).
- FStructType (§8.5.10) *base* (optional): The base struct from which this struct inherits. Franca supports single inheritance for structs. Structs which inherit from a base struct cannot be polymorphic.
- EBoolean *polymorphic* (optional): Indicates if struct is the root of a polymorphic type hierarchy. Structs may be either extended from a base struct or polymorphic (or none of both).

8.5.11 Class FField

- EString *name*: The name of this element. Inherited from base class FModelElement.
- FAnnotationBlock (§8.7.1) *comment* (optional): The structured comment for this element (if any). Inherited from base class FModelElement.
- FTypeRef (§8.5.3) *type*: The actual type of this typed element. The type might be predefined or user-defined. Inherited from base class FTypedElement (§8.6.13).
- EString *array* (optional): If the typed element is an implicit array definition, this property will be set to a string (e.g., the string '[]'). If the typed element is a single value (not an array), this property will be null. Do not rely on the actual value of the string. Inherited from base class FTypedElement (§8.6.13).

8.5.12 Class FArrayType

- EString *name*: The name of this element. Inherited from base class FModelElement.
- FAnnotationBlock (§8.7.1) *comment* (optional): The structured comment for this element (if any). Inherited from base class FModelElement.
- FTypeRef (§8.5.3) *elementType*: The type of this array's elements.

8.6 API for Franca contracts

This section describes all API classes needed for Franca interface contracts. The root class of the interface contract subtree is FContract. See the section Contracts (§5.6) in the Franca User Guide for detailed information about contracts.

8.6.1 Class FContract

The contract for a Franca interface. The contract specifies the dynamic behaviour of the interface. It consists of a PSM (short for: Protocol State Machine) and an optional set of state variables.

- FStateGraph (§8.6.3) *stateGraph*: The protocol state machine for this contract. It might use the state variables specified by the variables property.
- List<FDeclaration (§8.6.2)> *variables*: The declarations of all state variables used by the protocol state machine of this contract.

8.6.2 Class FDeclaration

This class represents the definition of a state variable as part of a Franca interface contract. Note that the types available for the state variable definition are all types which are accessible from this interface.

- EString *name*: The name of this element. Inherited from base class FModelElement.
- FAnnotationBlock (§8.7.1) *comment* (optional): The structured comment for this element (if any). Inherited from base class FModelElement.
- FTypeRef (§8.5.3) *type*: The actual type of this typed element. The type might be predefined or user-defined. Inherited from base class FTypedElement (§8.6.13).
- EString *array* (optional): If the typed element is an implicit array definition, this property will be set to a string (e.g., the string '[]'). If the typed element is a single value (not an array), this property will be null. Do not rely on the actual value of the string. Inherited from base class FTypedElement (§8.6.13).

8.6.3 Class FStateGraph

A state graph specifying the interface's dynamic behavior. The state graph is flat (i.e., non-hierarchical) and consists of a set of states, which are linked by transitions. One of the

states in the set is specified as *initial*. I.e., this is the initial state of the interface.

- FState (§8.6.4) *initial*: This property references the initial state for the protocol state machine.
- List<FState (§8.6.4)> *states*: The list of states comprising this protocol state machine. It contains the intial state.

8.6.4 Class FState

This class represents a single state as member of a FStateGraph (§8.6.3). The state contains a list of its outgoing transitions. The target state of each transition is contained in that transition.

- EString *name*: The name of this element. Inherited from base class FModelElement.
- FAnnotationBlock (§8.7.1) *comment* (optional): The structured comment for this element (if any). Inherited from base class FModelElement.
- List<FTransition (§8.6.5)> *transitions*: The list of outgoing transitions for this state. These transitions lead to the successor states of this state.

8.6.5 Class FTransition

This class represents a single transition as part of a FStateGraph (§8.6.3). The transition references its target state by property *to*. It doesn't reference its source state; instead, the source state contains the list of its outgoing transitions. The transition defines a *trigger*, which specifies some event which can occur during runtime. It also defines a *guard* condition. If the trigger event occurs, the guard condition will be checked (if any). Depending on this check the transition will be executed. Finally, a transition has some *action*, which will be executed each time when the transition fires.

- FTrigger (§8.6.18) *trigger*: The trigger which fires this transition.
- FGuard (§8.6.17) *guard* (optional): An optional guard for this transition.
- FState (§8.6.4) *to*: The target state of this transition.
- FExpression (§8.6.6) *action* (optional): An optional action which is executed each time this transition fires.

8.6.6 Class FExpression (abstract)

This is the common base class for all expressions of the Franca contract action language. The FExpression (§8.6.6) class will never be instantiated.

8.6.7 Class FBinaryOperation

- FExpression (§8.6.6) *left*: The left operand of the binary operation.
- EString *op*: The operator of this binary operation (e.g., '||', '+', '*', '<', and many more). Consider the FrancaIDL.xtext grammar definition for an overview of all binary operators supported by Franca contract action language.
- FExpression (§8.6.6) *right*: The right operand of the binary operation.

8.6.8 Class FConstant (abstract)

This is the common base class for all kinds of constants in the Franca contract action language. The most important types are integers, booleans and strings.

8.6.9 Class FStringConstant

This class represents strings constants in the Franca contract action language.

- EString *val*: The value of this string constant (an actual string).

8.6.10 Class FBooleanConstant

This class represents boolean constants in the Franca contract action language.

- EBoolean *val*: The value of this boolean constant (an actual boolean).

8.6.11 Class FIntegerConstant

This class represents integer constants in the Franca contract action language.

- EInt *val*: The value of this integer constant (an actual integer).

8.6.12 Class FTypedElementRef

This class is a reference to some FTypedElement (§8.6.13). If the referenced element is no compound type (i.e., struct or union), only the *element* property will be used. The *target* and *field* properties will be null. If a field of a compound type is referenced, the *field* property will point to the FField (§8.5.11) object. If the compound type is nested, the *target* property will reference the next outer level (which is itself a compound referenced by a FTypedElementRef (§8.6.12). This might be chained in order to specify a reference to a deeply nested compound element.

- FTypedElement (§8.6.13) *element* (optional): The typed element which is actually referenced by this object.
- FTypedElementRef (§8.6.12) *target* (optional): For nested compound types, this property references the next outer level. Otherwise, it is null.
- FField (§8.5.11) *field* (optional): Specifies the field of a compound type.

8.6.13 Class FTypedElement (abstract)

- EString *name*: The name of this element. Inherited from base class FModelElement.
- FAnnotationBlock (§8.7.1) *comment* (optional): The structured comment for this element (if any). Inherited from base class FModelElement.
- FTypeRef (§8.5.3) *type*: The actual type of this typed element. The type might be predefined or user-defined.
- EString *array* (optional): If the typed element is an implicit array definition, this property will be set to a string (e.g., the string '[]'). If the typed element is a single value (not an array), this property will be null. Do not rely on the actual value of the string.

8.6.14 Class FVariable

- EString *name*: The name of this element. Inherited from base class FModelElement.
- FAnnotationBlock (§8.7.1) *comment* (optional): The structured comment for this element (if any). Inherited from base class FModelElement.
- FTypeRef (§8.5.3) *type*: The actual type of this typed element. The type might be predefined or user-defined. Inherited from base class FTypedElement (§8.6.13).
- EString *array* (optional): If the typed element is an implicit array definition, this property will be set to a string (e.g., the string '[]'). If the typed element is a single value (not an array), this property will be null. Do not rely on the actual value of the string. Inherited from base class FTypedElement (§8.6.13).

8.6.15 Class FAssignment

- FDeclaration (§8.6.2) *lhs*: The left-hand side of this assignment. It references the state variable which should be set by specifying its declaration.
- FExpression (§8.6.6) *rhs*: The right-hand side of an assignment. This may be an arbitrary expression of the Franca contract action language.

8.6.16 Class FBlockExpression

- List<FExpression (§8.6.6)> *expressions*: The sequence of expressions represented by this block. This is a composite pattern used to specify a tree of expressions.

8.6.17 Class FGuard

- FExpression (§8.6.6) *condition*: The boolean condition of this transition guard.

8.6.18 Class FTrigger

- FEventOnIf (§8.6.19) *event*: The event which will trigger this transition.

8.6.19 Class FEventOnIf

This class represents a communication event, which basically corresponds to a message on the interface. Exactly one of its properties should be set, all other should be zero.

- FMethod (§8.4.6) *call* (optional): If this property is not null, this event represents the calling of a Franca method by a client.
- FMethod (§8.4.6) *respond* (optional): If this property is not null, this event represents the response of a Franca method by the server.
- FBroadcast (§8.4.4) *signal* (optional): If this property is not null, this event represents the sending of a Franca broadcast by the server.
- FAttribute (§8.4.7) *set* (optional): If this property is not null, this event represents the setting of a Franca attribute.
- FAttribute (§8.4.7) *update* (optional): If this property is not null, this event represents the update action of a Franca attribute.

8.7 API for Franca structured comments

This section describes the API classes representing structured comments. Note that it is a special feature of Franca that comments are available at all in the model. This is not the case for unstructured comments, which basically will be removed by the parser when creating the abstract syntax tree. See the section Structured comments (§5.7.2) in the Franca User Guide for detailed information about structured comments.

8.7.1 Class FAnnotationBlock

This class represents a structured comment. It is used by many of the elements of the Franca IDL model. A structured comment consists of a list of elements with one tagged comment each.

- List<FAnnotation (§8.7.2)> *elements*: The elements of this annotation block. Each element represents one tagged comment.

8.7.2 Class FAnnotation

- FAnnotationType *type* (optional): The tag of this comment (i.e., the type of this annotation).
- EString *comment* (optional): The actual comment as part of an FAnnotation.

8.7.3 Enum FAnnotationType

This enum represents the type of a structured comment (i.e., the tag starting with a '@').

This enum consists of the following literals: *DESCRIPTION*, *AUTHOR*, *EXPERIMENTAL*, *DEPRECATED*, *SEE*, *PARAM*, *HIGH_VOLUME*, *HIGH_FREQUENCY*, *SOURCE_URI*, *SOURCE_ALIAS*, *DETAILS*.

9. Building generators with Franca

When a company or R&D department decides to model its interfaces with Franca IDL then typically the interface definitions will be utilized by downstream tools, mostly by generators creating code or documentation from these interfaces. This chapter gives an introduction how generators can be implemented which produce code and other artifacts from Franca IDL models.

9.1 Introduction

This introductory section describes the basic approach and some design decisions how to build generators based on the Franca model API.

9.1.1 Basic approach

A typical approach to implement generators for Franca includes the following steps:

1. Load a Franca IDL file (with suffix *.fidl*) from file system. Result: a Franca model in main memory.
2. Generate code from the Franca model. Result: some text strings containing the generation output.
3. Save the generation output to one or more files on the file system.

These steps will be described in more detail below.

9.1.2 Which language can be used?

The loading step will result in an underlying Franca model which is the base for the generator. This model provides a Java API (§8) generated by the Eclipse Modeling Framework (EMF).

Thus, Java is a good choice for the generator implementation. However, there are other languages available which operate on the Java VM or which translate to Java: Groovy, Scala and Xtend are some examples. All those languages can be used to implement generators for Franca.

For developers already used to Java we recommend using Xtend for this task. Xtend offers a seamless integration with Java allowing the generator being implemented as a proper mix of Java (where necessary) and Xtend. We fill focus on Xtend in this and later chapters; the examples being shipped with Franca are using Xtend as well.

9.1.3 Tool integration

There are various ways to integrate a generator into your toolchain. Here are some examples:

- *Generator as JUnit test case.* Usually, this is the first step when implementing a new generator. In this case, the generator runs on a separate Java VM without an instance of the Eclipse IDE.
- *Standalone generator.* The generator is packed into a jar file and called by a Java main class which offers command-line arguments for configuring the generator. Also in this case, the generator runs on a separate Java VM without an instance of the Eclipse IDE. A standalone generator can also be integrated into some build process, e.g. to be part of a continuous integration build.
- *Generator as Eclipse action.* The generator is embedded into an *Action* class which implements *org.eclipse.ui.IActionDelegate*. Thus, the generator can be triggered from the Eclipse IDE, e.g. by selecting a proper context menu item.

As this chapter puts its focus on the actual generation step, we only describe the JUnit-based solution here.

9.1.4 Loading a Franca IDL file

Loading a Franca IDL model file (with extension *fidl*) from the file system is a standard task. It is supported by the Franca Framework, here is a Java example as part of a JUnit test:

```
import static org.junit.Assert.*;
import org.franca.core.dsl.FrancaIDLHelpers;
import org.franca.core.franca.FModel;
import org.junit.Test;

public class MyTest {
    @Test
    public void loadModel() {
        FModel fmodel = FrancaIDLHelpers.instance().loadModel("example.fidl");
        assertNotNull(fmodel);

        // ... traverse model here
    }
}
```

The *FrancaIDLHelpers* singleton class provides a convenient *loadModel()* function which initializes the infrastructure needed for Franca and loads the Franca IDL file afterwards. It might use a relative or absolute path. If the model *imports* other files, those will be loaded, too. All cross references will be resolved.

The function *loadModel()* returns an object of type FModel (§8.4.1) (follow the link to the corresponding section of the Franca Model API (§8) documentation). See the next section for how to traverse the model and generate some artifacts from it.

9.2 Traversing Franca models

9.2.1 Starting with FModel

A Franca IDL model (represented by an object of type FModel (§8.4.1)) might contain several type collections FTypeCollection (§8.4.2) and interfaces FInterface (§8.4.3). These can be accessed using the generated FModel API, the following Xtend code example shows how to access them.

```
class ExampleGenerator {
    def generateFromModel (FModel fmodel) {
        for(tc : fmodel.typeCollections)
            tc.generateTypes

        for(i : fmodel.interfaces)
            i.generateInterface
    }

    def private generateTypes (FTypeCollection types) {
        // access properties of FTypeCollection, e.g. types.name
    }

    def private generateInterface (FInterface api) {
        // access properties of FInterface, e.g. api.name or api.methods
    }
```

In the example, there is only a stub implementation for the functions *generateTypes()* and *generateInterface()*.

9.2.2 Accessing an FInterface

The following Xtend example code shows how to traverse all methods (i.e., FMethod (§8.4.6) members) of a Franca interface FInterface (§8.4.3). It generates a C++ class with member function declarations for all Franca methods.

```
def generateInterface (FInterface api) '''
    class «api.classname»
    {
    public:
        // methods
        «FOR m : api.methods»
        «m.generateMethodDecl»
        «ENDFOR»
    };
'''
```

This function uses a *template expression* (surrounded by triple single quotes), which is a very useful feature of Xtend when it comes to textual generators. Template expressions can span multiple lines and contain expressions which are evaluated and their string representation being inserted at that position.

9.2.3 Benefits due to Xtend features

Template expressions are one useful feature of Xtend which supports the implementation of generators very nicely. Here are some more Xtend features which are very handy for generator developers:

- The implicit *it*-variable allows to reduce syntactic sugar, especially in the case of lots of small methods as usually found in generator code.

- The *switch-statement for classes* allows to implement type-safe dispatching without *instanceof*-cascades and downcasts.
- *Lambda expressions* are very useful for the traversal and filtering of object graphs (as in the Franca model).
- *Extension methods* allow to split the generator functionality into a set of loosely coupled classes. This is even more valuable if used in conjunction with *dependency injection* based on *com.google.inject.Inject*.
- *Dispatch methods* introduce a polymorphic behaviour, which is handy for dispatching *generate()*-calls to various subclasses in a class hierarchy (e.g., look at FType (§8.5.1) and its subclasses.

There are some more nice features of Xtend which help in implementing generators. See the documentation on Xtend for more details on this. It is available online and as printable pdf.

9.2.4 Next steps

In the following, the Franca IDL model has to be traversed further, extracting information from the model and generating code from it. The typical way of implementing this is top-down, starting with FModel (§8.4.1), FTypeCollection (§8.4.2) and FInterface (§8.4.3), and continuing with type definitions from type collections and interfaces (e.g., FType (§8.5.1) / FTypeRef (§8.5.3), FStructType (§8.5.10), FArrayType (§8.5.12) and many more). Finally output is generated for the elements of interfaces (e.g., FAttribute (§8.4.7), FMethod (§8.4.6), FBroadcast (§8.4.4) and FArgument (§8.4.5)).

There are some special features in the Franca model API which should taken care of:

- inheritance for FInterface (§8.4.3), FStructType (§8.5.10), FUnionType (§8.5.9) and FEnumerationType (§8.5.5) (see *base*-property for these classes)
- implicit arrays, which is relevant for all subclasses of FTypedElement (§8.6.13) (via its *array*-property)
- error enumerators for methods, available in two variants (see FMethod (§8.4.6))

We will not go into more detail here, as there is a detailed chapter on the Franca Model API (§8) in this User Guide.

9.2.5 Contract section of FInterface

Each Franca interface might contain a definition of a *contract*, specifying the dynamic behavior of the interface. The corresponding classes of the Franca model API are described in the User Guide section API for Franca contracts (§8.6).

As the contract is specifying the behavior of the interface, it should not contain any application logic. Thus, it should not be necessary to access this part of the model when generating target code. However, in order to use the specified contract as part of a *runtime verification monitor* or for an offline trace analyser, it will be needed to traverse this part of the model and generate some artifacts from it.

Section Contracts (§5.6) describes the notion of contracts and their syntax and usage in more detail.

9.3 Accessing Franca deployment models

For the Franca IDL model there is the Franca Model API (§8) which allows programmatic access to IDL models. This is also the recommend way for building generators from Franca IDL. For Franca deployment models (§6), there is also an API generated by the EMF infrastructure. However, we discourage using this API because deployment models are

far more generic than IDL models and contain many implicit assumptions and logical constraints. In order to work with deployment models programmatically, *PropertyAccessor* classes should be used.

As this is not specific to building generators, but will be needed also for model-to-model transformations and other Franca IDL downstream processing tasks, please refer to common section Support for accessing deployment properties (§6.4) to get the details of accessing Franca deployment models from your generator implementation.

10. Building transformations to/from Franca

One particular sweet spot for Franca is its ability to integrate software systems by supporting transformations from other interface definition languages to Franca IDL. This chapter describes some concepts around this topic and details how those transformations can be designed and implemented.

10.1 Introduction

Software interfaces which are modeled using some other technology than Franca can be mapped to Franca by providing a model-to-model transformation. This can be done in two directions:

- from Franca IDL to another IDL
- from another IDL to Franca IDL

A *Franca connector* is a Java class which contains both transformations for a given IDL. Usually, the other IDL will be modeled based on some meta-model. In many environments an *ecore*-model of the IDL is available (which is a meta-model in the format given by the Eclipse Modeling Framework, short: EMF). If the other IDL is modeled using some XML-based format, an XML schema or a DTD is available, which can be converted automatically to an *ecore*-model.

Based on the *ecore*-model, the transformations can be implemented using any technology which can access either the models directly or uses the Java API of the *ecore*-model as generated by EMF. Options are:

- use plain Java or another JVM-based language
- use Xtend (maybe mixed with Java), which will be transformed into Java automatically
- use a declarative M2M framework, e.g. ATL

In the following we recommend using Xtend for this task. It has several nice features which especially support the implementation of imperative model-to-model transformations.

10.2 Transforming Franca to other models

10.2.1 Plain transformations of Franca IDL

Implementing a plain transformation from Franca IDL models to other models is very similar to the task of building a generator for Franca IDL. In both cases, a Franca model has to be traversed; the extracted information is used to produce a resulting artifact (either generated code or another model). See the chapter on Building generators with Franca (§9) for more insights about this topic.

Xtend has proven to be a powerful and expressive language for traversing a model and constructing another model from it. There are a couple of features supporting model traversal (§9.2.3). Another Xtend feature especially for constructing models as graphs of objects are *create-functions*. These provide a cache mechanism to avoid constructing the identical object twice. See the Xtend reference documentation for details. An example on how to use *create-functions* can be found below (§10.3.1).

Example: The plugin *org.franca.connectors.dbus* is part of the Franca release. It contains the transformation from Franca to DBus introspection files implemented in Xtend. This can serve as an example for implementing this kind of transformation. You will find it in Xtend file *Franca2DBusTransformation.xtend* in package *org.franca.connectors.dbus*.

10.2.2 Transforming Franca deployment models

If a model has to be created from Franca deployment models (including a reference to an Franca IDL model), two models have to be traversed simultaneously. Again, this is a topic which is very similar to generating code from Franca deployment models.

For the Franca IDL model there is the Franca Model API (§8) which allows programmatic access to IDL models. This is also the recommend way for building transformations from Franca IDL. For Franca deployment models (§6), there is also an API generated by the EMF infrastructure. However, we discourage using this API because deployment models are far more generic than IDL models and contain many implicit assumptions and logical constraints. In order to work with deployment models programmatically, *PropertyAccessor* classes should be used.

As this is not specific to building model-to-model transformations, but will be needed also for building generators and other Franca IDL downstream processing tasks, please refer to common section Support for accessing deployment properties (§6.4) to get the details of accessing Franca deployment models from your transformation implementation.

10.3 Transforming other models to Franca

10.3.1 Plain transformation to Franca IDL

The typical structure of a transformation of another model into Franca IDL looks like the following:

1. load the source model from file system (or a different source)
2. traverse the source model starting from the root (this is typically a top-down traversal)
3. extract information from the source model elements and create a Franca IDL model
4. save the resulting Franca IDL model or use it for further processing

During the third step elements of the Franca IDL model have to be created and filled with data. The Java class *FrancaFactory* has to be used for element creation. See section FrancaFactory and FrancaPackage (§8.2.2) for more information on how to do this.

In the context of an Xtend implementation of a transformation, *create-functions* can be used to create the proper objects and initialize them. Here is an example showing a create function for FInterface (§8.4.3).

```
// this function will return an object of type FInterface
def create FrancaFactory::eINSTANCE.createFInterface transform(InterfaceType src) {
    // interface name is just a string and doesn't have to be transformed
    name = src.name

    // transform interface methods one-by-one
    methods.addAll(src.method.map [transformMethod])

    // transform further elements of 'src' and store them in the new FInterface object
    // ...

    // implicit return parameter 'it' (the new FInterface object)
}
```

InterfaceType is the class corresponding to *FInterface* from the source model of the transformation (most likely another IDL). When the *transform()*-method is being called, it will first check if the source element has been transformed before. Only if it hasn't been transformed yet, a new FInterface (§8.4.3) object will be created. The implicit variable *it* has to be used to refer to this object. In the assignment *name = src.name* the left-hand side is a property of FInterface (§8.4.3), this is a short-hand for *it.name*.

The second line calls uses *lambda functions* (another handy Xtend feature) in order to call *transformMethod()* for each member of the collection *src.method* and store the resulting objects (of type FMethod (§8.4.6)) into the list *it.methods*. This is another pattern typically used in model transformations implemented with Xtend.

Detailed example: The plugin *org.franca.connectors.dbus* is part of the Franca release. It contains the transformation from DBus introspection files to Franca IDL implemented in Xtend. This can serve as an example for implementing this kind of transformation. You will find it in Xtend file *DBus2FrancaTransformation.xtend* in package *org.franca.connectors.dbus*.

10.3.2 Creating additional deployment models

It sometimes is necessary to transform a third-party model into two resulting models: a Franca IDL model and a corresponding Franca deployment model. The first part of this transformation has been covered by the previous section. In order to create the deployment model, the helper class *DeployModelBuilder* should be used. It provides helper functions for setting deployment properties according to a given deployment specification.

In Xtend code, the *DeployModelBuilder* can be imported as a static extension:

```
import static extension org.franca.deploymodel.core.DeployModelBuilder.*
```

Here is an Xtend code example showing the basic idea of creating IDL model and deployment model at the same time (the types of the variables could be omitted and are here for clarity only):

```
// load deployment model containing specification(s) and get the first one
var fdSpecifications = FDModelHelper::instance.loadModel("deployspec.fdepl")
var FDSpecification fdSpec = fdSpecifications.specifications.get(0)

// create Franca IDL model and deployment model
var FModel fModel = FrancaFactory::eINSTANCE.createFModel
var FDModel fdModel = FDeployFactory::eINSTANCE.createFDModel

// prepare DeployModelBuilder
```

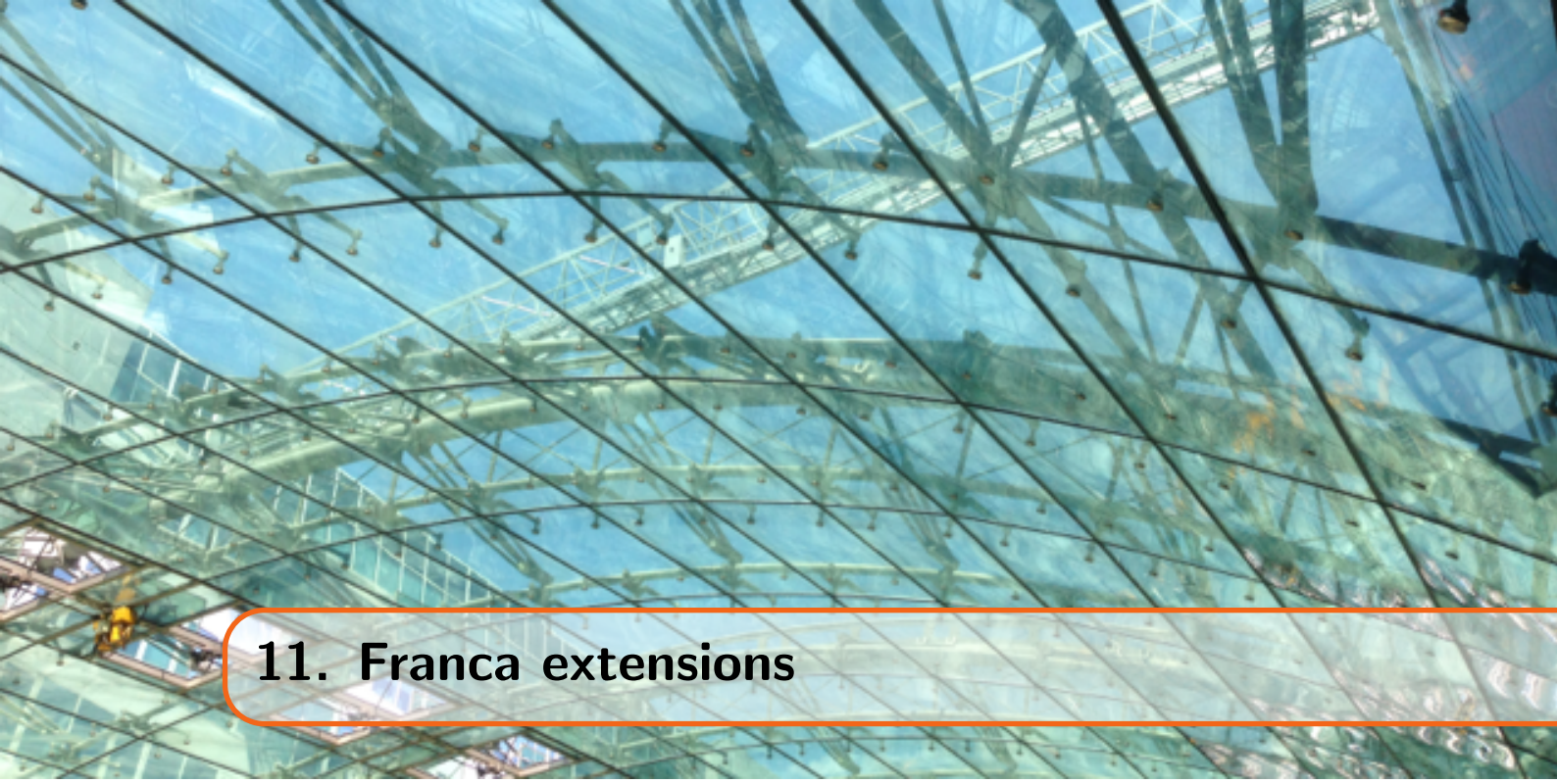
```
// transform all interfaces
for(src : sourceModel.interfaces) {
    var FInterface result = src.transform

    // create deployment model element and add it to deployment model
    var FDInterface fdInterface = FDeployFactory::eINSTANCE.createFDInterface
    fdModel.deployments.add(fdInterface)

    // link deployment interface to IDL interface
    fdInterface.target = result

    // check properties in the source model and add them to deployment model
    if (src.checkSomeProperty()) {
        // set a deployment property (this uses DeployModelBuilder)
        fdInterface.setProperty(fdSpec, "SomeProperty", true)
    }
    // ... check more properties and set them, if necessary
}
```

Note that the *setProperty()* function is defined as part of *DeployModelBuilder* and is called for a deployment model element (here: *fdInterface*). The following parameters are: the underlying deployment specification (here: *fdSpec*), the actual name of the property (here: "SomeProperty"), and the new value for the property. If the new value for the property is equal to the default as defined in the deployment specification, *DeployModelBuilder* will not set the property.

See the Javadoc documentation of org.franca.deploymodel.core.DeployModelBuilder to understand all kinds of functions it offers.

Depending on the complexity of the source model, it might be tedious to create the Franca IDL model and the deployment model at the same time. The difficult detail is to set the *target* attribute for all elements of the deployment model (see example above). Xtend *extension methods* and Google's dependency injection framework might help here by allowing to separate the Franca IDL model construction from the deployment model construction (see *com.google.inject* and the *@Inject* annotation for Java and Xtend).

There is no example for this kind of transformation in the Franca release yet. This will be added in a future version.



11. Franca extensions

11.1 Additional validators

In some environments using Franca, it is required to impose additional restrictions on the contents of IDL or deployment files. E.g., there might be company-specific naming conventions which should also be applied to names of Franca data types, interface names or deployment properties. This can be enforced by implementing an *external validator* which checks the additional specific restrictions.

For each additional validation rule in the validator, it can be chosen if a rule violation should be classified as info, warning or error.

The following sections describe how to implement validators and register them. In the Franca repository, there is an example project which provides running examples of IDL and deployment validators (project *org.franca.examples.validators* in the *examples folder*).

11.1.1 Adding a validator for Franca IDL

A specific validator for Franca IDL models is an implementation of the Java interface *IFrancaExternalValidator*. The actual validation rules are implemented by overriding the method *validateModel*. The method arguments provide access to the model which should be validated and to a message acceptor which can be used for issuing warnings and errors.

The following example shows an external validator which ensures that Franca method names do not contain underscores.

```
public class MethodNameValidator implements IFrancaExternalValidator {
    @Override
    public void validateModel(FModel model, ValidationMessageAcceptor messageAcceptor) {
        for(FInterface _interface : model.getInterfaces()) {
            for(FMethod method : _interface.getMethods()) {
                if (method.getName().contains("_")) {
                    messageAcceptor.acceptError(
                            "Method names must not contain underscores!", method,
                            FrancaPackage.Literals.FMODEL_ELEMENT__NAME,
                            ValidationMessageAcceptor.INSIGNIFICANT_INDEX,
```

```
                                null);
                        }
                    }
                }
            }
        }
```

The API of *ValidationMessageAcceptor* provides methods for issuing info, warning and error messages, depending on the severity of the violation.

In order to register the new IDL validator class, it has to be configured as extension for the extension point *org.franca.core.dsl.francaValidator*. This can be done either by using the *Extensions* dialog in the Eclipse IDE or by manually editing the *plugin.xml* file of the project where the *IFrancaExternalValidator* implementation is located. The following excerpt from *plugin.xml* shows how to register the *MethodNameValidator* from the previous example.

```
<plugin>
   <extension point="org.franca.core.dsl.francaValidator">
      <validator
            class="my.specific.validators.fidl.MethodNameValidator"
            mode="FAST"
            name="Method name validator">
      </validator>
   </extension>
</plugin>
```

The property *class* has to define the fully qualified class name of the validator class. This is a usual pattern for registration of Eclipse extensions. The *name* property defines some descriptive text for the extension. Finally, the *mode* property specifies the check mode:

- FAST or NORMAL: check will be executed each time the Franca IDL file is saved
- EXPENSIVE: check is triggered manually from the context menu of Franca files

The new validation checks will then be executed automatically by the Eclipse IDE. However, if Franca models are handled by standalone tools (without the Eclipse IDE), there is no extension point infrastructure and the additional validator will not be registered. Hence, for standalone tools, the registration of additional validators has to be done programmatically. The class *ExternalValidatorRegistry* in the package *org.franca.core.dsl.validation* provides static methods for doing this, e.g. the method *ExternalValidatorRegistry.addValidator()*.

Note that the *ExternalValidatorRegistry* approach should only be used in standalone mode. In the Eclipse IDE the extension point approach has to be used, in order to avoid coupling and prevent initialization problems due to the on-demand loading of plug-ins.

11.1.2 Adding a validator for deployment models

Adding a validator for Franca deployment models is similar as for IDL model validators. The following example shows a validator which ensures that the names of deployment specifications are not shorter than a hard-coded minimal length. Note that the interface which has to be implemented for new deployment validators is a different interface than in the IDL case.

```
public class SpecNameValidator implements IFDeployExternalValidator {

    static final int SPEC_NAME_MINIMUM_LENGTH = 5;
```

```
    @Override
    public void validateModel(FDModel model, ValidationMessageAcceptor messageAcceptor) {

        for(FDSpecification spec : model.getSpecifications()) {
            String name = spec.getName();
            if (name.length() < SPEC_NAME_MINIMUM_LENGTH) {
                messageAcceptor.acceptError(
                        "The name of the specification is too short (minimum is " +
                                SPEC_NAME_MINIMUM_LENGTH + " characters)!", spec,
                        FDeployPackage.Literals.FD_SPECIFICATION__NAME,
                        ValidationMessageAcceptor.INSIGNIFICANT_INDEX,
                        null);
            }
        }
    }
}
```

The details of implementing validation rules are quite similar. Again, different kinds of messages can be issued based on the severity of the violation. As models can contain both deployment specifications and deployment definitions, specific deployment validation rules might check properties of both specifications and definitions as well.

In order to register the new deployment validator class, it has to be configured as extension for the extension point *org.franca.deploymodel.dsl.deploymentValidator*. This can be done either by using the *Extensions* dialog in the Eclipse IDE or by manually editing the *plugin.xml* file of the project where the *IFDeployExternalValidator* implementation is located. The following excerpt from *plugin.xml* shows how to register the *SpecNameValidator* from the previous example.

```
<plugin>
   <extension point="org.franca.deploymodel.dsl.deploymentValidator">
      <validator
            class="org.franca.examples.validators.fdepl.SpecNameValidator"
            mode="FAST"
            name="Specification name validator">
      </validator>
   </extension>
</plugin>
```

The registration in the standalone case (without the Eclipse IDE) is done using *ExternalValidatorRegistry* in the package *org.franca.deploymodel.dsl.validation*. It provides static methods for registering external validators, e.g., the method *ExternalValidatorRegistry.addValidator()*. Note that two different registration classes for IDL and deployment validators are used. They share a common name and common method names, but are located in different packages.

Again, note that the *ExternalValidatorRegistry* approach should only be used in standalone mode. In the Eclipse IDE the extension point approach has to be used, in order to avoid coupling and prevent initialization problems due to the on-demand loading of plug-ins.

11.2 Providing deployment specifications

Some downstream tools (e.g., code generators for Franca models) might use Franca deployment models as additional input format. These tools will usually bring their own

deployment specification which defines the deployment properties needed as additional input by the tool.

Usually these tool are packaged as Eclipse plug-ins. The deployment specification file then is just another resource in the plug-in jar. In order to allow users to refer to the deployment specification without knowing its exact URI, Franca provides an extension point which can be used by the tool's plug-in to register the deployment specification with an abstract (logical) name. Franca will then take care via scoping to allow importing this deployment specification.

Here is an example of using the *deploySpecProvider* extension point in order to register the deployment specification from file *PlatformDeploySpec.fdepl* with the logical name *CoolSpec*:

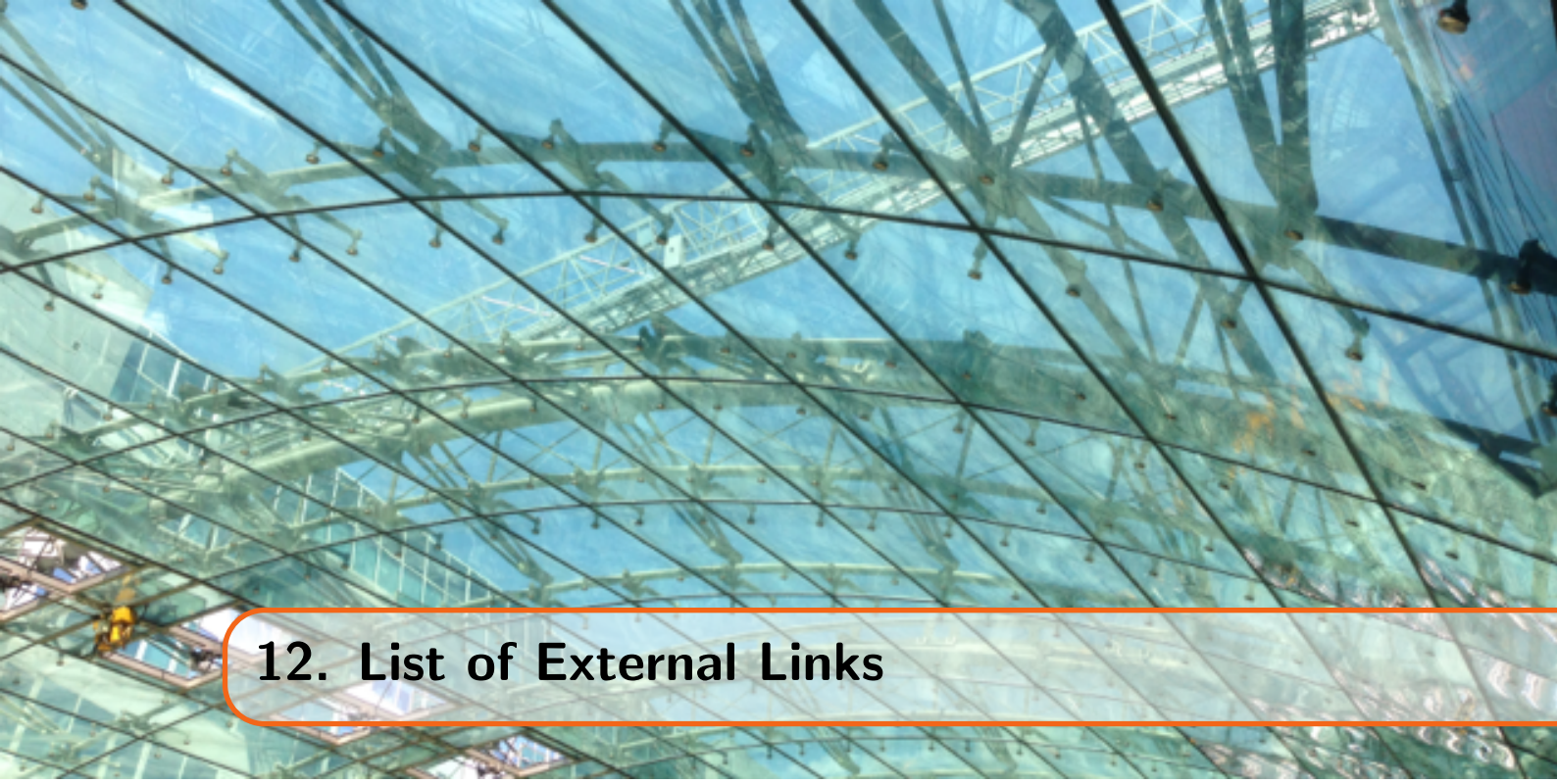
```
<plugin>
   <extension point="org.franca.deploymodel.dsl.deploySpecProvider">
      <model
            FDSpecification="org.example.spec.PlatformDeploySpec"
            alias="CoolSpec"
            resource="model/platform/PlatformDeploySpec.fdepl">
      </model>
   </extension>
</plugin>
```

The registration of the extension can be done either by using the *Extensions* dialog in the Eclipse IDE or by manually editing the *plugin.xml* file of the plug-in which wants to register the deployment specification.

This deployment specification can then be imported in any deployment model by the following import statement:

```
import CoolSpec

specification MySpec extends org.example.spec.PlatformDeploySpec {
    // ...
}

define org.example.spec.PlatformDeploySpec for interface SomeAPI {
    // ...
}
```

In the example, the deployment specification imported by its alias is used as a base specification for MySpec and for a deployment definition.



12. List of External Links

https://github.com/franca/franca/wiki/Franca-Quick-Install-Guide
http://www.freedesktop.org/wiki/Software/dbus
http://www.eclipse.org/modeling/emf/
http://www.eclipse.org/xtext
http://kbirken.github.io/dbus-emf-model/releases/
http://www.xtend-lang.org
http://5ise.quanxinquanyi.de/2012/01/13/xtext-end-user-domain-experts-cheat-sheet/
https://github.com/kbirken/dbus-emf-model