Franca / ARA::COM Interoperability

Establishing interoperability of Linux-based systems and Adaptive AUTOSAR systems by model-to-model transformations

Webinar 24-25 September 2019

Giovanni VERGINE (Visteon)
Klaus BIRKEN, Torsten GÖRG, Michael LANGHAMMER (itemis AG)
Agenda

1. Introduction / Motivation
2. Conceptual mapping
3. The FARACON tool
4. Tool demo
5. Q&A
INTRODUCTION / MOTIVATION
Introduction

SOA (service oriented architecture) is a clear SW design trend in automotive.

SOA in Automotive can be represented in 3 layers:

1. Interface Description Language (IDL)
   - Define the contract between services and clients

2. Middleware
   - Generates code (from IDL) and libraries used by SOA services and clients to communicate.
   - Abstracts the underlying transport protocol

3. Transport protocol
   - Provide rules to transport the messages (message identification, serialization, …)
Problem statement

GENIVI study demonstrated that 2 SOA technologies are under the Automotive spotlights today.

- COMMONAPI + FIDL: strongly adopted in infotainment domain
- ARA::COM: AUTOSAR emerging technology

⇒ How to ensure compatibility?
Objectives

Franca Interface Definition and Specification (.fidl/.fdepl)

Client Application
- Client Implementation
- Service Proxy
- Binding

Service Application
- Service Implementation
- Service Skeleton
- Binding

Middleware Transport Layer

Service Interface Definition (ARXML)

Client Application
- Client Implementation
- Service Proxy
- Binding

Service Application
- Service Implementation
- Service Skeleton
- Binding

Middleware Transport Layer
Objectives

To ensure compatibility we have to reach **2 objectives**

1. **Talk the same language**/Use the **same** communication **concepts**
   a. Choose one IDL as reference and translate it
      i. FRANCA IDL chosen as reference for readability (ARXML is XML format)
      ii. Goal is the 2 ways translation
   b. IDL brings a set of communication concepts:
      i. Message types: Event, RPC calls, …
      ii. Data types: unitary data (UInt8, Float, …) and composed data (Struct, …)

2. **Transport the information in the same format**
   a. Uniquely identify the messages
   b. Serialize the data in the same order.
      \[\Rightarrow\] SOME/IP is the solution to align transport format.
Example of service-oriented workflow (SOA)
Technology Demonstrators in 2019/Q1

CES, Las Vegas, January 2019

European R-CAR Consortium Forum, Düsseldorf, March 2019
Model transformation tooling

Major requirement: Transform models such that the resulting code on both sides will be compatible wrt. its IPC properties.
Definition of mapping

- AUTOSAR metamodel defined by Artop metamodel (artop.org)
- Franca IDL metamodel defined by Franca Eclipse project
- Mapping between both domains is defined
  - Each language concept is implemented by a metaclass
  - Mapping is defined on metaclasses and their attributes
    - E.g., ServiceInterface $\leftrightarrow$ FInterface, ClientServerOperation $\leftrightarrow$ FMethod
- The metamodel level mapping is the starting point for tool implementation
- Detailed mapping table (GoogleDocs format)
Detailed mapping table (aka specification)

### Definition of Mapping Franca/AUTOSAR

This is the specification for the transformation AUTOSAR Adaptive <-> Franca. We disregard all concepts of AUTOSAR which purely belong to AUTOSAR Classic.

<table>
<thead>
<tr>
<th>Group</th>
<th>Franca concept or one of (IGNORE I ERROR I EMULATE) (see A10)</th>
<th>Franca metamodel export ID</th>
<th>Franca metamodel classifier</th>
<th>Franca metamodel attribute</th>
<th>AUTOSAR concept or one of (IGNORE I ERROR I EMULATE) (see A10)</th>
<th>AUTOSAR metamodel classifier</th>
<th>AUTOSAR metamodel attribute</th>
<th>Detail level (IDL, Serialization, CommonAPI, SOMEIP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>structure</td>
<td>version of type collection</td>
<td>IDL1190</td>
<td>Class FTypeCollection</td>
<td>FVersion version (optional)</td>
<td>see IDL1490</td>
<td>see IDL1490</td>
<td>see IDL1490</td>
<td>IDL</td>
</tr>
<tr>
<td>structure</td>
<td>list of types (all with visibility public)</td>
<td>IDL1200</td>
<td>Class FTypeCollection</td>
<td>List&lt;fType&gt; types</td>
<td>package contents</td>
<td>ARPpackage</td>
<td>List&lt;ListPackageableElement&gt; getElements()</td>
<td>IDL</td>
</tr>
<tr>
<td>structure</td>
<td>list of constants (all with visibility public)</td>
<td>IDL1210</td>
<td>Class FTypeCollection</td>
<td>List&lt;fConstantDef&gt; constants</td>
<td>asked MBR...</td>
<td>ServiceInterface</td>
<td></td>
<td></td>
</tr>
<tr>
<td>structure</td>
<td>interface definition</td>
<td>IDL1220</td>
<td>Class FInterface</td>
<td>n/a</td>
<td>interface definition</td>
<td>ServiceInterface</td>
<td></td>
<td>IDL</td>
</tr>
<tr>
<td>structure</td>
<td>list of attributes</td>
<td>IDL1230</td>
<td>Class FInterface</td>
<td>List&lt;fAttribute&gt; attributes</td>
<td>fields of a service</td>
<td>ServiceInterface</td>
<td>List&lt;ListField&gt; getFields()</td>
<td>IDL</td>
</tr>
<tr>
<td>structure</td>
<td>list of methods</td>
<td>IDL1240</td>
<td>Class FInterface</td>
<td>List&lt;fMethod&gt; methods</td>
<td>client server operations of a service</td>
<td>ServiceInterface</td>
<td>List&lt;IClientServerOperations&gt; on the service interface</td>
<td>IDL</td>
</tr>
<tr>
<td>structure</td>
<td>list of broadcasts</td>
<td>IDL1250</td>
<td>Class FInterface</td>
<td>List&lt;fBroadcast&gt; broadcasts</td>
<td>events of a service</td>
<td>ServiceInterface</td>
<td>List&lt;ListVariableDataPrototype&gt; getEvents()</td>
<td>IDL</td>
</tr>
<tr>
<td>structure</td>
<td>optional interface contract</td>
<td>IDL1260</td>
<td>Class FInterface</td>
<td>FContract contract (optional)</td>
<td>IGNORE</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>structure</td>
<td>inheritance for interfaces</td>
<td>IDL1270</td>
<td>Class FInterface</td>
<td>FInterface base (optional)</td>
<td>EMULATE</td>
<td>n/a</td>
<td>n/a</td>
<td>IDL</td>
</tr>
<tr>
<td>structure</td>
<td>manages-relation for interfaces</td>
<td>IDL1280</td>
<td>Class FInterface</td>
<td>List&lt;fInterface&gt; managedInterfaces</td>
<td>ERROR</td>
<td>n/a</td>
<td>n/a</td>
<td>IDL</td>
</tr>
<tr>
<td>Comm primitives</td>
<td>method</td>
<td>IDL1290</td>
<td>Class FMethod</td>
<td>n/a</td>
<td>operation</td>
<td>ClientServerOperation</td>
<td>n/a</td>
<td>IDL</td>
</tr>
<tr>
<td>Comm primitives</td>
<td>fire-and-forget flag</td>
<td>IDL1300</td>
<td>Class FMethod</td>
<td>FBoolean fireAndForget (optional)</td>
<td>fire-and-forget flag</td>
<td>ClientServerOperation</td>
<td>boolean isSetFireAndForget</td>
<td>IDL</td>
</tr>
</tbody>
</table>

- read-only link to document on GoogleDocs: [mapping table](#)
Guidelines for resolving mapping problems

- objective: generated code is compatible
- principal reasons for incompatibilities:
  - a. no corresponding concept on metamodel level (e.g., inheritance)
  - b. generated code shows different behavior (e.g., error handling)
- options for resolving incompatibilities:
  - a. check if concept can be “emulated” (e.g., flattening inheritance)
  - b. check if code generation can be fixed (either by adapting the code generator or indirectly by changing the mapping)
  - c. if all else fails: make user aware that concept cannot be mapped (e.g., by attaching warnings/errors from validation)
Main concept mappings

- interfaces, type collections
- namespaces, packages
- methods, broadcasts, attributes
- primitive types (booleans, integers, strings, ...)
- struct/union types, array types (implicit/explicit), map types
- structured comments

See detailed mapping table for a complete list.
Main concept emulation mapping

**Interface version**
New AUTOSAR package is created for each given version of the interface.

**Interface hierarchy**
Flattening of interface hierarchy, i.e. the content of the parent interface is repeated in the child interface.

**Type hierarchy**
Flattening of type hierarchy, i.e. the content of the parent type is repeated in the child type.

**Interface local types and constants**
An AUTOSAR package for the interface is created that contains all local types and constants.

**AUTOSAR package hierarchies in single file**
Map single AUTOSAR file to multiple Franca files.

See detailed mapping table for a complete list.
Main concept mapping issues

**Selective Broadcast**
- GENIVI: Allows to send broadcast to dedicated clients but against SOA paradigm where only the middleware knows the registered clients.
- AUTOSAR: Not supported.

**Polymorphic structures**
- GENIVI: CommonAPI uses a tagged-value serialization with hash value for the tag. But not defined in SOME/IP specification.
- AUTOSAR: Not supported.

**Optional fields**
- GENIVI: Introduced in AUTOSAR Adaptive 18.10. Field is present if tag is present in TLV serialization format. CommonAPI serializes structs with LV (length presence and width is configurable in FDEPL).
- AUTOSAR: Not supported.

**Data semantic**
- GENIVI: Used to define the content of an unitary data (unit, max value…)
- AUTOSAR: Not defined in FIDL.

**Method errors**
- GENIVI: Application errors are not transported on the same way. AUTOSAR uses SOME/IP error code to transport applicative errors.
- AUTOSAR: Not supported. CommonAPI generated code defines a mandatory error status.

See detailed mapping table for a complete list.
THE FARACON TOOL
The FARACON tool

- transform Franca IDL models to Artop models and vice versa
- scope for Artop models is limited to AUTOSAR adaptive
- supported versions
  - Franca 0.13.1
  - Artop 4.12 (AUTOSAR Adaptive Platform R19-03)
- no full roundtrip (information loss due to mapping limitations)
- planned: mapping of deployment data for SOME/IP
FARACON tool architecture

- based on the Eclipse ecosystem
- usable both from Eclipse IDE and command-line
- EMF as a common metamodel
- transformations implemented using Xtend language
- additional tools/frameworks: maven/Tycho, JUnit
- Artop is available only to AUTOSAR members
FARACON licensing and development

- FARACON is open-source (Eclipse Public License v2)
- FARACON development is funded by GENIVI Alliance
- public repository: https://github.com/GENIVI/franca_ara_tools
- Artop plugins have to be downloaded from artop.org, AUTOSAR membership needed!
FARACON Release 0.9

- available 2019 / week 39
- features:
  - nearly 100% of transformation logic (both directions)
  - IDE integration and command-line tool
  - maven/Tycho build for developers
- see [mapping table](#) on GoogleDocs for full details
FARACON Release 1.0

● available end of October 2019
● additional features:
  ○ prototypical mapping for SOME/IP deployment data
  ○ configurable standard AUTOSAR types
  ○ Artop-less command line tool (installation of Artop done by the user)
● plus bugfixes from beta testing feedback
Testing FARACON

- regression tests are being built together with transformation features and are part of the continuous integration build
  - for each feature in the detailed mapping table, links to the unit tests are provided
- tool is available for beta testers in order to run it on real world models and get feedback for development
  - AUTOSAR membership needed
  - please contact us if you are interested!
- integration testing (end-to-end, including runtime) is currently being planned
TOOL DEMO
Tool Demo: Command Line Tool

- command line options allow to configure tool
- important options:
  - input and output folders
  - direction (Franca-to-AUTOSAR or AUTOSAR-to-Franca)
  - control logging
Tool Demo: Command Line Tool - Example

- example: “convert all Autosar files in directory “input” to Franca files and store them in the directory “output”
Tool Demo: Eclipse-UI

- options can be specified via “Window->Preferences”
- transformation output folder is mandatory option
Tool Demo: Eclipse-UI

- select folder or file
- via the context menu the transformation can be executed
Tool Demo: Eclipse-UI - Transformation Result

- transformation result is stored in the specified output folder
- (error) messages are printed to the console
- input files can be edited within Eclipse
- results can be viewed within Eclipse
package genivi.aasr.showcase

interface IDrivingLane {
  ...
  struct LaneType {
    UInt16 frameId
    UInt32 intersectionPointX
    UInt32 intersectionPointY
    UInt32 lowerRightPointX
    UInt32 lowerRightPointY
    UInt32 lowerLeftPointX
    UInt32 lowerLeftPointY
  }

  broadcast LaneDetected {
    out {
      LaneType drivingLane
    }
  }
}
Deep Dive – Mapping of an Interface

package genivi.aasr.showcase

interface IDrivingLane {
    ...
}

* ARXML is shortened for the presentation
Deep Dive – Mapping of a Struct Type

package genivi.aasr.
showcase

interface IDrivingLane {
 ...
 struct LaneType {
   UInt16 frameId
   String frameHash
   UInt32 lowerLeftPointX
   UInt32 lowerLeftPointY
   UInt32 intersectionPointX
   UInt32 intersectionPointY
   UInt32 lowerRightPointX
   UInt32 lowerRightPointY
 }
}

* ARXML is shortened and simplified for the presentation
package genivi.aasr

interface IDrivingLane {
    ...
    struct LaneType {
        ...
    }
}

broadcast LaneDetected {
    out {
        LaneType drivingLane
    }
}
package genivi.aasr.showcase

interface IDrivingLane {
    struct LaneType {
        UInt16 frameId
    }
    method drivingLaneMethod {
        in {
            LaneType[] drivingLaneIn
        }
    }
}

<Service-Interface>
<Short-Name>IDrivingLane</Short-Name>

<METHODS>
<CLIENT-SERVER-OPERATION>
<Short-Name>drivingLaneMethod</Short-Name>
<ARGUMENT-DATA-PROTOTYPE>
<Short-Name>drivingLaneIn</Short-Name>
>Type-TRef>/genivi/aasr/showcase/LaneTypeVector</Type-TRef>
<DIRECTION>IN</DIRECTION>
</IMPLEMENTATION-DATA-TYPE>
<Short-Name>LaneType</Short-Name>
<Category>STRUCTURE</Category>
<IMPLEMENTATION-DATA-TYPE>
<Short-Name>LaneTypeVector</Short-Name>
<Category>VECTOR</Category>
</SUB-ELEMENTS>
<IMPLEMENTATION-DATA-TYPE-ELEMENT>
<Short-Name>valueType</Short-Name>
<Category>TYPE_REFERENCE</Category>
<Array-Size-Semantics>VARIABLE-SIZE</Array-Size-Semantics>
<IMPLEMENTATION-DATA-TYPE-REF>/genivi/aasr/showcase/LaneType

* ARXML is shortened and simplified for the presentation
Deep Dive – Mapping of an Interface Hierarchy

package genivi.aasr.showcase

interface IDrivingLane {
    method drivingLaneMethod {}
}

interface IDrivingLaneExt extends IDrivingLane {
    method extMethod {}
}

* ARXML is shortened and simplified for the presentation