WHITEPAPER

JUNE 2019

Category: Generic Communication Protocols Evaluation

Generic Protocols Evaluation Results

Introduction

When designing the current generation of connected vehicles, the automotive industry has to cope with too many choices, too much diversity, too much boiler-plate code, adaption layers and incompatibility. The initial objective of the Generic Protocol Evaluation Project (GPRO) team was to survey and compare available protocols supporting in-vehicle communications and communications with the cloud and to identify industry-preferred options.

Reducing the choices from over 15 to 3-5 preferred protocols reduces complexity and, ultimately, will save time and money in development and validation. The project's first phase is now completed, and the results are published. It involved the preparation of a poll to get the feedback from the automotive industry on the preferred communication protocols. The poll measured familiarity with 23 protocols, importance of those protocols for in-vehicle and outside of vehicle communication, current protocol usage and recommendations to the automotive industry for protocol usage.

An initial snapshot of the poll was taken reporting results from the first 22 participants. Proceeding from the results of the poll, it was built and demonstrated during CES2019 the GENIVI-Adaptive AUTOSAR interoperability demonstrator that highlights the usage of SOME/IP as a standard communication mechanism across the car network between two systems, an IVI system and an AUTOSAR Adaptive system. The interaction between the two systems is enabled by a new tool that allows for transformation of an Adaptive AUTOSAR model (arxml) and a GENIVI model (Franca IDL), resulting in code that enables interaction between GENIVI IVI and Adaptive AUTOSAR APIs.

List of Relevant Technologies

During the project's first milestone, called the "knowledge building and sharing", a large group of communication technologies was investigated, prior to the preparation of the poll. We give in this section a short overview of the most notable technologies.

REST, Request/Response and Serializations

REST is an architectural style that is not dependent on any other protocol. For the web case HTTP/HTTPS is a common choice. REST doesn't depend on a specific language. Each call is independent from the other and contains the necessary information to complete successfully, making REST a successful choice for designing stateless services. Because of the above, REST can be quite flexible on term of the underlying protocol/technology choice: is used in multiple places and is very common in IoT or when interacting with cloud-based services. An alternative to HTTP for REST is CoAP, a very lightweight protocol that supports the HTTP methods and error codes represented in a binary form instead of the HTTP textual representation. Unlike SOAP, REST supports multiple format including JSON, XML and YAML. However, JSON seems to be the preferred choice when implementing REST via HTTP(S).

Indeed, JSON is much simpler than XML. JSON has a much smaller grammar and maps more directly onto the data structures used in modern programming languages. Compared to XML, JSON is more compact and less convoluted which makes it more readable for a human. It needs less resources to be processed/transferred because of the smaller size and its simplicity compared to XML. JSON Integrates well with most programming languages due to the mapping between JSON data types and programming languages data types. Efficient libraries exist to deal with JSON (json-c for C as an example). JSON schema standard was developed to allow the validation of JSON instances.

In an IoT environment, or in general when the bandwidth is limited, an alternative to JSON is offered by Google Protocol Buffers. It is a binary serialization toolkit that lets the user to describe a message structure in a .proto file and provides extensible generators for different languages that produce serializers and deserializers for the message described. The focus of Google Protocol Buffers is to increase the bandwidth available reducing the size of a message.

Broker-based and Publish/Subscribe Communications

Aside from the client-server approach of HTTP and CoAP solutions, D-Bus and MQTT rely on a different kind of architecture, in which a central broker is used to gather and distribute all the data.

MQTT is a protocol that relies on a publish/subscribe message exchange pattern: clients are peers that connect to the broker and subscribe and publish data to topics, basically implementing the observer pattern on the network. When a client publishes a message to a topic, the broker forwards the message to all the other clients that are subscribed to the same topic.

MQTT typically need a small code footprint, so it is ideal if processor or memory resources are limited. Having a very small packet overhead, it is a good choice when the bandwidth is low. It uses three different QOS levels (at most once, at least once, exactly once), making it useful also on unreliable networks.

Typical MQTT implementations work on top of TCP/IP, optionally using SSL/TLS. MQTT over WebSocket is possible (making the browser to be an MQTT client). Authentication is based on username and password. In the IT world, MQTT is a very wide spread protocol. Client libraries are available publicly in most of the programming languages as well as many kinds of brokers. On non-TCP/IP networks (e.g., Zigbee) is used MQTT-SN (protocol for sensor network).

MQTT is publish/subscribe, but request/response message exchange pattern are possible. MQTT5 adds in the protocol the possibility to send a response topic when publishing a message, while in MQTT3 this logic is demanded to the user.

D-Bus is a message bus system. Like MQTT, relies on a broker (called usually D-Bus daemon) gathering and dispatching messages. The messages are described in D-Bus interface files (XML format). The D-Bus code generator helps to generate marshallers and unmarshallers for such messages that can then be used for RPC over D-Bus. The broker has access to the interface files, making it possible to configure the access and the usage of specific interfaces.

The message bus is built on top of a general one-to-one message passing framework, which can be used by any two apps to communicate directly (without going through the message bus daemon). Typically, UNIX domain sockets are used for communicating applications within the same host, while TCP/IP is also suitable.

A similar more powerful protocol is WAMP, which has been proposed as an IETF standard. It provides a combination of features found in many of the popular protocols. While a bit less focused on efficiency and resource-constrained environments compared to MQTT, it is friendly to both native and web implementations by its defined JSON data exchange (MsgPack as a supported alternative for a smaller encoding). WAMP provides basically a superset of features of other protocols, including publish/subscribe behaving similar to MQTT, but also remote procedure call, and furthermore including abstract addressing of providers of services (i.e. not only “direct”, but also *routed* remote procedure calls).

Autonomous Driving: ROS and Apollo's CyberRT

A trend emerging in the last years in the automotive industry are the Advanced Driver-Assistance Systems (ADAS) that should help the driver during the driving at different levels, up to the complete automated driving. Such systems today rely on protocols developed on purpose or borrowed from the robotics industry. The most notable example is ROS.

ROS (Robotics Operating System) is an open source initiative that provides a developer environment for creating robot applications. The ROS architecture is based on nodes (processes) that perform local or distributed computation. Nodes exchange messages each other in a publish/subscribe fashion. The protocols used for such communication is commonly referred in the community as ROS1 and - its successor - ROS2.

In ROS1, the broker is substituted by the concept of Master. The Master a process that provided service registration and lookup (discovery service), however does not dispatch the messages between the clients, that communicate directly after having found each other via the Master. A node that provides a service interacts with the Master via http/xml (xmlrpc) advertising the topics on which he would receive messages, as well as the specific data format. Other clients can query via xmlrpc the master for finding such client based on the topic and the data format used. Then, a binary protocol is used between the two clients with a publish/subscribe paradigm like in MQTT, with the difference that the underlying communication is direct and is not routed through the Master.

Protocols other than the binary ones are available via rosbridge. It provides a JSON interface to ROS and adds support for more transport layers, such as WebSockets and TCP. Modifications by LGE to rosbridge make possible to use Google Protocol Buffers instead of JSON.

In ROS2, the system is completely decentralized and there is no Master. Typically, it is built on top of DDS/RTPS middleware, which handles not only discovery, but also serialization and transportation: each node advertises itself to other nodes on the network, that respond with the information about themselves.

Unlike ROS, Apollo Cyber RT is an open source framework developed purposely for ADAS. Apollo Cyber RT is compatible with ROS up to version 3.0. The subproject apollo-ros adds shared memory support, RTPS, Google Protocol Buffers and decentralization to ROS.

Starting with version 3.5 Apollo is not compatible with ROS. Its main benefits seem to be high performance and the user level scheduler that let the developer to tune the system according to the application and hardware resources.

Franca Interface Description Language and Deployment Model

Franca is a powerful framework for definition and transformation of software interfaces used in the automotive industry, but not limited to it. It is used for integrating software components from different suppliers, which are built based on various runtime frameworks, platforms and IPC mechanisms. The core of it is Franca IDL (Interface Definition Language), which is a textual language for specification of APIs. C++ code can be generated directly from Franca interface specifications with CommonAPI C++. CommonAPI C++ is a standardized C++ API specification for the development of distributed applications which communicate via a middleware for inter-process communication. The main intention is to make the C++ interface for applications independent from the underlying IPC stack.

The Unique Flexibility and Power of Franca

Franca IDL is designed to write interface descriptions for all types of internal and external communication and programming interfaces, which could also mean to reuse of the same interface description in different contexts. When translating an interface description to a particular context it usually needs some additional information. Other IDLs often add such things to its main interface language itself, which is initially convenient, but ultimately limits the applicability of the IDL. Franca IDL instead places these in a separate *deployment model.* Since each environment needs different things defined, even the deployment *model* content is not constrained by a fixed Franca specification but instead another level of abstraction is provided: The deployment model is constrained by a *deployment specification* that can be written uniquely for each case. Typically, a tool implementor for Franca translation (e.g. a code generator) will write such a specification. Like Franca IDL, the deployment specification is written in plain text with a formal syntax. Already existing tools (Eclipse environment) will provide syntax guidance, not only for writing interfaces in Franca IDL (controlled by fixed rules defined by the Franca specification) but also for writing the deployment model. The rules for what shall be written in the model are *dynamically* defined by the *specification* written by each implementor. Note that for this there is no additional plugin or programming step needed – to see this, simply open the Eclipse editor and edit a deployment specification, then switch to editing the deployment model and the editor will *immediately* provide syntax guidance according to the rules of the newly edited specification.

Deployment details may thus control *anything* that the target environment needs, but some common examples are:

- How each part of an interface is translated to target programming language features
- Datatype encoding and binary representations
- Specify for methods if they shall be implemented with asynchronous or synchronous calling semantics
- How to address interacting components.

Clearly these needs differ in different contexts. For example, looking at addressing the provider of a defined interface: IP-addresses might be applicable in a network deployment of the interface, but for a local programming API, component would just be linked by a compiler and found in a global namespace or in namespaces derived from the namespaces used in the interface description file and may therefore not need additional information. In some other environment services might be addressed using another abstract identifier that needs to be specified. If the IDL lacked such details it would not be clear how to overcome this lack of information and ad-hoc solutions might be applied. Conversely, if the IDL (like many failed attempts do) included such details then its specification would grow indefinitely until reaching the limits of what this IDL is tailored for.

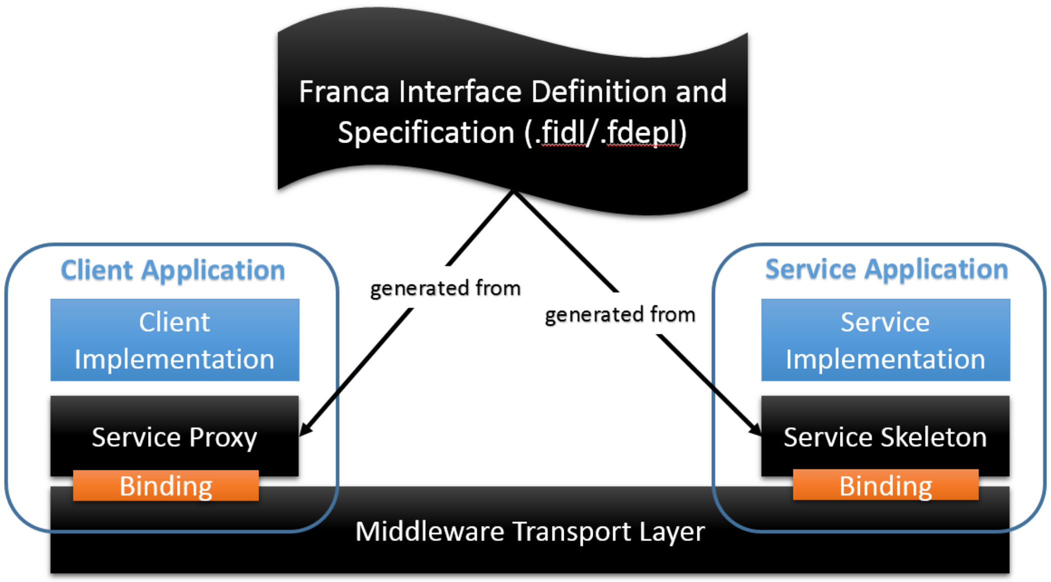
Franca Plus

To address the next possible step in a model-based development, the interconnection of communicating components could be defined. Traditionally UML has been used for such purposes, but an alternative text-based approach is investigated in a project called Franca Plus (Franca+). This makes the component/system design approach familiar to users of Franca IDL (similar syntax and workflow) and using a text-based modeling language makes it more attractive to programmers and also enables the use of typical source code versioning tools (git) for managing the system model.

Deployment Model of CommonAPI C++

The code generator for CommonAPI C++ bindings uses a few types of *deployment models*, each providing the details needed for the communication binding (e.g. D-Bus, SOME/IP, ...)

CommonAPI is in turn, designed to decouple generated API from the actual IPC stack and cooperates seamlessly with Franca. It does so by providing bindings that allow to use any IPC mechanism semantically compatible to Franca; out-of-the-box are available D-Bus, WAMP and SOME/IP bindings.



Franca with CommonAPI, SOME/IP and ARA::COM

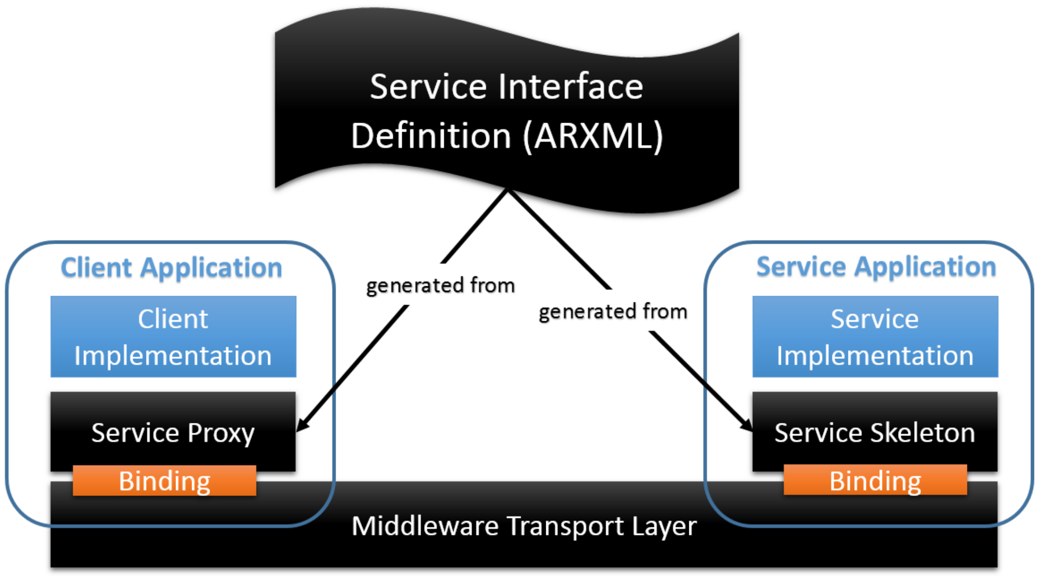
Given the centrality of AUTOSAR$^{(R)}$ in the ECUs today, the usage of CommonAPI SOME/IP bindings have been very interesting for the GPRO team.

SOME/IP is an AUTOSAR automotive/embedded communication protocol which supports remote procedure calls, event notifications and the underlying serialization/wire format. It was designed to fit devices of different sizes and different operating systems. This includes small devices like cameras, AUTOSAR devices, and up to head units or telematics devices. It was also made sure that SOME/IP supports features of the Infotainment domain as well as that of other domains in the vehicle, allowing SOME/IP to be used for MOST replacement scenarios as well as more traditional CAN scenarios.

While IT solutions often only support single middleware features (e.g. RPC or Publish/Subscribe), SOME/IP supports a wide range of middleware features:

- Serialization – transforming into and from on-wire representation.
- Remote Procedure Call (RPC) – implementing remote invocation of functions.
- Service Discovery (SD) – dynamically finding and functionality and configuring its access.
- Publish/Subscribe (Pub/Sub) – dynamically configuring which data is needed and shall be sent to the client.
- Segmentation of UDP messages – allowing the transport of large SOME/IP messages over UDP without the need of fragmentation.

CommonAPI is not the only framework that can use SOME/IP as underlying communication mechanism. Indeed, AUTOSAR Adaptive Platform offers a runtime for Adaptive Application. Such runtime, called ARA, contains its own communication middleware specification known as ARA::COM.



ARA::COM offers bindings to SOME/IP, but bindings can be implemented also for any other technology that supports publish/subscribe/event patterns. SOME/IP, however, remains the default transport layer available on the shelf for ARA::COM. It is worth to mention that SOME/IP libraries could include the serialization of the primitive datatypes, but CommonAPI and ARA::COM implementations include SOME/IP-compliant serialization on their own, in their SOME/IP bindings. Both implementations are implementing the serialization defined in the AUTOSAR SOME/IP specification.

While CommonAPI uses Franca for the definition of the software interfaces, ARA::COM uses the Service Interface Definition in an XML format (ARXML).

Poll

The participants have been asked about their familiarity with the technologies taken in consideration and their importance for onboard (between ECUs) and offboard (to the Cloud) usage. Also, the participants were asked to point out with technologies they would recommend and which they are currently using.

As mentioned before, part of the activity of the GPRO team was dedicated to the preparation of a poll to get the feedback from the automotive industry. We asked the participants to evaluate the familiarity with each of the 23 technologies, as well as the importance of its usage for in-vehicle communication and communication to the outer world. For each technology, we also asked if the participant is using it during development and if he would recommend its usage.

We were then able to understand with which technologies the respondents were most familiar with, which are most important for them and which they would recommend, making sure to weight the importance and recommendation with the familiarity. We were also able to find out the most used technologies today. Comparing the times a technology is recommended but not used, and the times a technology is used but not recommended, we can estimate which technologies the participants are asking to be introduced or used more in the automotive industry and which ones they are not happy with.

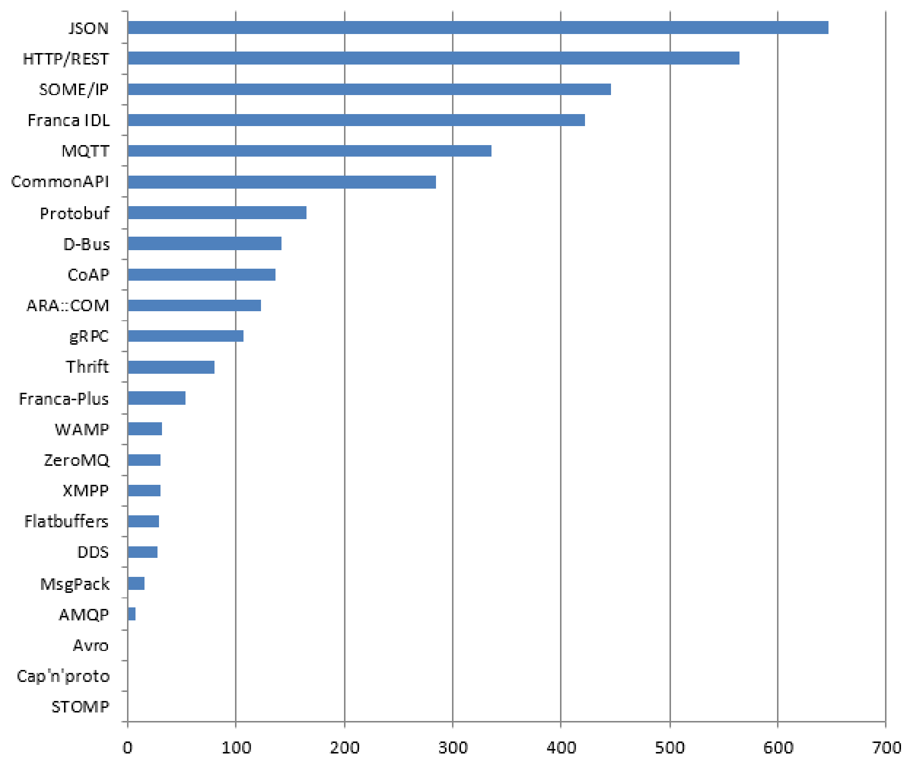
According to the results, the technologies people are most familiar with are JSON, D-Bus, and HTTP/REST. Among the respondents of this survey, SOME/IP, Franca IDL, MQTT and CommonAPI also are well known.

When asked about the importance of a given technology for communication between ECUs in the vehicle, SOME/IP and Franca IDL got the highest scores, followed by JSON and CommonAPI. The scenario changes for communication outside of the vehicle (e.g. to the Cloud), where JSON and HTTP/REST are the clear winners, followed by MQTT. The most used ones today seem to be also the most recommended: JSON, HTTP/REST and Franca IDL. D-Bus is more used than recommended, while gRPC is more recommended than used.

In the end, we were able to assign a popularity score to each technology with the following formula:

Popularity = (Inside-vehicle importance + Outside-vehicle importance) x familiarity x (1 + recommended)

The popularity score is shown in the following diagram:



In general, the trend is quite good for the couple JSON and HTTP/REST, that have the highest scores in our overall popularity score, followed by SOME/IP and Franca IDL. A little far away, also MQTT and CommonAPI got a fairly good result.

When asked about the importance of a given technology for communication outside of the vehicle (e.g., to the Cloud), JSON and HTTP/REST are the clear winners, followed by MQTT.

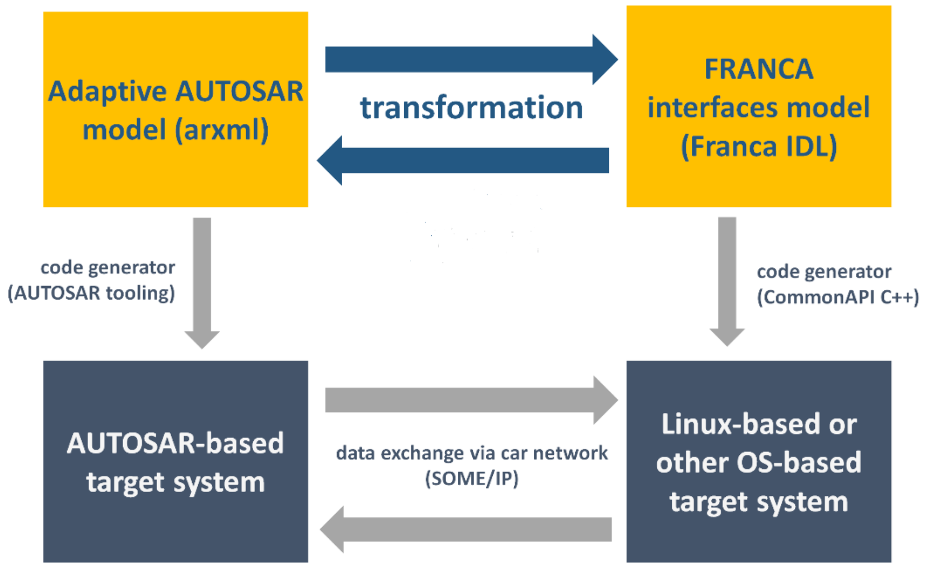
SOME/IP, Franca and CommonAPI are the preferred option according to our poll when it comes to communication between ECUs. Indeed, it is already available a product that uses the three of them: the CommonAPI Franca-based generators and runtime with SOME/IP bindings. It was then of great interest in the GPRO team to understand at which level such toolkit could interoperate with an important ecosystem in the Automotive Industry represented by AUTOSAR ECUs. Our focus went in the direction of the AUTOSAR Adaptive Platform because it represents a big trend among OEMs.

Franca/ARA::COM Interoperability

When it comes to inter-domain communication between ECUs, messages must be translated between CommonAPI and ARA::COM. This is not only tedious, but prone to human errors. However, both communication technologies are based on model definitions (.arxml and .fidl/.fdepl files). This opens the possibility to translate using model-to-model transformation methods that can improve software quality and reduce development time and engineering costs. In this section, we present a tool that uses model-to-model transformation to achieve a compatible code generation on both sides of ARA::COM and Franca IDL. When combined with CommonAPI bindings and ARA::COM runtime, this achieves a runtime translation between the systems. Using such a tool makes it possible to have a specification in a single format (we propose

Franca IDL for this), and yet to use the full advantage of both technologies on both sides of the communication.

The Franca project offers not only an interface definition language (Franca IDL), but also a framework for building model-to-model transformations. This framework is being used here to implement transformations from Adaptive AUTOSAR models to Franca IDL and vice versa. These transformations can be used as part of any current Eclipse IDE. For build automation and Continuous Integration (CI) it also useful to deploy the transformations as a command-line tool. The goal of the automatic transformations is to apply code generation by AUTOSAR-compatible code generators as well as Franca-compatible generators (e.g., CommonAPI C++) in a way that leads to transparent communication between both systems at runtime. Therefore, the tooling is based on a proper mapping between Adaptive AUTOSAR concepts and Franca IDL concepts. For example, each operation on an AUTOSAR service interface is mapped to a method in Franca IDL. The following diagram shows how the transformation tooling interacts with the code generators. The generated code on the AUTOSAR and GENIVI subsystems is using SOME/IP protocol for communication. As the subsystems are integrated on model level, the communication is automatically compatible.


Adaptive AUTOSAR model (arxml)
transformation
FRANCA interfaces model (Franca IDL)
code generator (AUTOSAR tooling)
code generator (CommonAPI C++)
AUTOSAR-based target system
data exchange via car network (SOME/IP)
Linux-based or other OS-based target system

The ARA-to-Franca tool is implemented using the Xtend language, which is an extension of Java providing language features which ease the implementation of model-to-model transformations. It uses Artop (release 4.10) as an implementation of Adaptive AUTOSAR and Franca 0.13.0. Both are using the Eclipse Modeling Framework, which provides a common way of working with models.

The Franca ARA Demonstrator was built to provide proof of interoperability for a wide range of Franca IDL artifacts like message and data types and its mapping to Adaptive AUTOSAR. The setup consists of four Ethernet connected ECUs running the GENIVI, the Adaptive AUTOSAR and the Classic AUTOSAR platforms. Together, those ECUs form an Emergency Brake Assistant with visualizations on the GENIVI IVI system.

The common basis for communication between the ECUs is established using SOME/IP. However, on top of the protocol stack, different middleware such as CommonAPI and ARA::COM is used as binding to the applications. The following steps were executed to establish the communication between the applications:

- Interface definition using Franca IDL
- Transformation of the Franca IDL interface definition to an equivalent ARXML definition using the Franca model-to-model transformations
- Creation of the corresponding SOME/IP deployments for CommonAPI and ARA::COM
- Generation of the proxies and skeletons for both platforms
- Integration of the generated code into the demo applications.

A clear advantage of following this approach is that there is no need to define the same interface twice in Franca IDL and AUTOSAR ARXML. This reduces errors coming from manual maintaining the service interfaces by having only one source of origin.

The results of this project consolidate the use of Franca-based descriptions of interfaces when designing complex systems by enabling a seamless integration of GENIVI APIs and components into the automotive software engineering processes used for autonomous vehicle functions (a.k.a. Adaptive AUTOSAR). The current implementation of the model transformations from Adaptive AUTOSAR to Franca IDL and vice versa is prototypical in several aspects. It is sufficient to support the requirements of the Franca ARA Demonstrator, but must be enhanced significantly to be used for production development in actual automotive projects.

The project demonstrates the value and relevance of the GENIVI vehicle domain interaction strategy through the delivery of tangible and useful technology (e.g. code generators) to implement the interfaces of automotive complex systems. Overall, the next step is to develop a (near) production-ready tool, which has the goal of reducing the cognitive load of developers as much as possible. This will lead to less errors and a more reliable communication between AUTOSAR and IVI systems.

Conclusions

The team researched the current technologies used inside and outside the Automotive Industry for model-based development and communication. We asked the opinion of some representative of the Industry, that confirmed the importance of Franca+CommonAPI+SOME/IP and AUTOSAR Adaptive software stacks in their current and future plans. Therefore, we demonstrated possible way to interconnect both environments introducing a new tool.

During CES2019 and internally in OEMs and Tier1s, there has been good feedback for such tool also in accordance of the future plans of development. AUTOSAR Adaptive platform and SOME/IP non-AUTOSAR ECUs are clearly having a trend in the near future, so would make sense to bring the tool to a production-ready level. Thanks to model-to-model transformations we could indeed provide a set of tools for easy and fast connection of domains (IVI, ASIL, DI, ADAS, Cloud, ...), either via new bindings for the Franca-based tools like CommonAPI, either as separated gateways generated by model definitions. Please notice that upcoming changes in the AUTOSAR specifications would need continued development of the tooling and framework to ensure compatibility.

References

- https://github.com/GENIVI/franca_ara_integration
- https://github.com/GENIVI/franca_ara_tools

Authors

- Giovanni Vergine (Visteon), gvergine@visteon.com
- Christopher Schwager (ITK), Christopher.Schwager@itk-engineering.de
- Klaus Birken (Itemis), kbirken@itemis.de
- Gunnar Andersson (GENIVI), gandersson@genivi.org
- Participants of the Generic Communication Protocols Evaluation working group