Minutes from 2015-04-23, AMM in Stuttgart

Introduction

See GENIVI_CommonAPI-C_20150423_AMM_v1.0.pptx for an introduction. The source code reviewed during the session will be soon available in a public repository.

Participants contributed to the discussion:

- Pavel Konopelko, Manfred Bathelt, ~klaus.uhl, Klaus Birken, ~sriram.g, ~luiz.vondentz

Review comments for the 'reference' code

- It should be possible for the client application to connect with instances that realize different versions of the same interface. Specifically, the code generated for different versions of the same interface should not result into collisions during linking.
  - Integrating interface version numbers into the header file and/or function names is one way to handle this.
- It should not be possible for the client application to invoke methods from an interface that a particular instance does not realize.
  - Introducing unique instance types for each interface would allow using type checking by the compiler.
  - Currently, all such types have the same implementation (i.e., `struct cmc_client_instance`). What is the simplest and efficient way to express this in C?
- Consider combining class and instance creation into a single call (e.g., creating the class instance in `cmc_client_instance_new()`).
  - Removing the need for one extra call would make the client code simpler.
  - Having just one class per interface is the most frequent case.
- Consider the necessity to expose pointers to class type instances to the server application code.
  - In the current implementation the application can only create and delete them.
  - Reference counting could be used to keep track of class type instances used by the individual instances.
- Consider generating default method implementations for the server.
  - One implementation per class is the most frequent case.
  - The values of `struct cmc_Service_impl` can be then pre-populated by the generator. The application can then override the defaults with its own implementations.
  - The initialization of the `struct cmc_Service_impl` values can be done in a dedicated function to be implemented/overridden by the application.
  - Verify if such default implementation could make adding alternative interface implementations more difficult.
  - Clarify if this approach would create additional drawbacks.
    - This might require to manually modify a generated module. Where should such module be put in the file system? Keeping it with the generated code would overwrite it during the next generation. In case the manually modified code was moved to other application modules, the subsequent generations would pollute the generate code with duplicate stubs.
    - Leaving the default method implementations with the generated code and overriding them in the application code would result into dead code.
  - Alternatively, the thunk implementations serve as de-facto stubs. Currently the gracefully fail when a particular method is not implemented. This behavior could be changed into something else.
- Initialize the values of `struct cmc_Service_impl` using the designation initializers introduced in C99 (i.e., `.field = value`).
  - The would simplify matching functions to methods they implement and also reduce the likelihood of mistakes to use a wrong pointer.
  - Existing implementations would not affected by Franca interface modifications that only change the sequence in that methods are introduced by the interface.
  - This would require using C99 as the minimum language standard.
- The server applications should be able to provide a pointer to the function to be called to finalize the processing of a method invocation.
  - This would require removing the out arguments from the generated function signatures.
  - This is required to implement asynchronous method handlers.
  - This would allow executing code after a reply to the method invocation has been sent.
    - Unclear how to do this with just one function, though.
- The method handlers implemented by server applications must receive the instance they were invoked through as a parameter.
  - This is required to implement attributes
  - This would allow attaching private data to instance implementations.

Miscellaneous topics

- CommonAPI-C++ 3.0 allows customizing the serialization of all data types. This is required for SOME/IP support.
- The portability to non-Linux environments would require CommonAPI-C to define its own abstract data types for the types missing in C and its standard libraries
  - 'Mapping' such ADTs to a platform-specific implementation through, for example, macros should be relatively easy and should not add considerable overhead.
- The requirement to optionally avoid dynamic memory allocation could be addressed by introducing allocators.
  - The runtime implementation would invoke them whenever new objects are created. The implementation can be customized for a particular environment to either use malloc/free or statically allocated object instances.

- To ensure correctness of the generated code, two kinds of tests are typically used. Unit tests in the target language are used to exercise the code generated for a set of examples. Unit tests in the generator language (i.e., Xtext/Xtend) are used compare the code generated for a set of examples with with the 'reference' code.
    - Comparison to the 'reference' code is implemented through a special class that, for example, ignores comments and whitespace in the 'reference' code.
    - 'Reference' code can be embedded directly into the unit test implementation (rather than stored e.g. in a repository).
    - OTOH, keeping it in the repository and fetching it from there to implement the generator unit tests would allow using the 'reference' as a tool for developing new features and extensions as well as maintaining the unit tests for the generated code.
    - cmc should use a third kind of test that exercises client/server pairs to test the end-to-end functionality.