Principles of meta/delta-specification (formal specification with reference/reuse)

In a situation where good quality specifications already exist it is essential to reuse them in a well planned manner. This is the case for the **Automotive Virtual Platform** definition developed within the HyperVisor project, which can build on the VIRTIO specification and potentially others.

This is often the case for an industry like automotive, in which OEMs want to leverage existing standards so that not every requirement needs to be written uniquely. (As suppliers know, there are more than enough unique requirements ;-)

Obviously this is done all the time - consider for example requirements on the basic bolts and nuts of a physical assembly. It would most often be done by referencing a **standard**, which in turn includes the requirements for the structural integrity and strength of the steel they are made of.

Developing industry-wide **Open Standards** and open source-code is what GENIVI is all about.

We see this pattern also in software/functional requirements:

for: OEM requirements (referencing) Standards (referencing) other standards

This makes up a kind of hierarchy of documents similar to an *upstream/downstream* situation in open-source software projects. The documents will have a relationship. Often we will find generic (upstream, open standards) applied in a more specific document (downstream / OEM requirement specification).

If we can't use a specification to 100% without changes, then the difference needs to be handled. There are several options:

- A) Prepare and suggest all modifications to the original specification as early as possible ("upstream first" principle in FOSS)
- B) Forking the original specification into a new variant
- C) Writing a new specification from scratch
- D) Writing a meta-specification (delta-specification) clarifying the applicability of the main specification.
    - If this is more than an applicability matrix then, we might rather call it a "delta-specification".

Considerations

Upstream Modification

The theoretically ideal approach is to affect the "upstream" to include our requirements representing the automotive industry. There may be trade-offs however. For OEMs in particular, unfortunately many refer to the time it takes to convince other stakeholders and to get a new specification/code released to be prohibitive. A specification might of course become a "kitchen sink" if it tries to be everything for everyone.

Secondly, the focus of the stakeholders of the adopted (upstream) specification might be slightly different, or a fruitful discussion ends with the "agree to disagree" outcome. Applying embedded requirements onto a specification developed primarily by cloud server engineers might not always be a good fit.

For these reasons it might not always be possible to modify the original specification into exactly what is desired.

Forking

Forking is usually not a great approach because it causes a disconnect, despite that much of the information is and should be the same. This disconnect causes the projects to drift apart and the effort of "keeping up to date" increases.

- Several aspects to consider:
    - License conditions of course must allow the content to be copied & modified. (F)(L)OSS licenses are the most common example.
    - Good citizen / good behavior. (In open source, there is the concept of a *hostile fork).*
    - Avoiding confusion between versions/variants.
- It is rarely a good approach but is explicitly allowed by licenses and it might be right to apply when it is appropriate.

If a fork is done, using best-practices for software is recommended. We should aim to keep the specification in plain text-format and use a git repository, that allows efficient diffing and merging. This is to make maintenance and interaction with the upstream even manageable. Even with these tools there is a lot of work to keep a combination of "patches" to a specification from becoming contradictory or illogical. There are no compilers and automated unit-tests to check your modified/combined work, as is done with software.

Applicability matrix / meta- / delta-specification

Very often a good compromise is the applicability matrix approach - or if more differences are expected, writing a delta-specification.

It has the potential to create a short, clear, and unambiguous specification that still fits exactly to ones needs, while avoiding proliferation of multiple overlapping documents. It is a kind of "meta-specification" because is one level above, and refers down to the others. But perhaps a "meta-specification" strictly means a specification that defines how specifications should be written, so we can adopt the name delta (difference) -specification here.

Delta Specification format

When creating a specification by referencing existing ones we should consider the following:

- Decide which parts of the specification to reuse (by reference) as a whole, and which ones will be clearer and better by rewriting them.

- When only some parts of the general specification is applicable we must modify referenced specifications by an applicability list or matrix, still meeting these goals:
    - Completeness
    - Avoiding ambiguity
    - Clarify how to handle conflicts and inconsistencies. (What takes precedence)
- Finally, we shall add unique requirements in a way that is not too difficult to manage.
- The tradeoffs here are between reducing repetition and divergence, versus complexity for both the reader and writer. When referring to a combination of multiple documents, it could lead to ambiguity or make the consumption of the specification difficult and impractical.

Principles:

1. Each chapter should define the requirements by minimal repetition, by referring to already written text
2. Sometimes, some limited copying/repetition is desired for clarity, overriding rule 1. Consider both the convenience of specification writers /maintainers and that of the consumers of the specification.
3. Modifications can be done by adding *additional text* as a modifier to what the original spec states. This still avoids repeating a lot of the major work that went into the original.
4. In particular remember that in the applicability definitions, things like "optional", "conditional" and "mandatory" can be modified compared to the original specification. For example, something that is well defined but stated as optional in a referred specification can be made mandatory according to *our* requirements.

Example:

Automotive Virtual Platform Specification, version 0.9

1. Introduction

Purpose, Scope, goals, context, applicability...

2. Architecture

Assumptions made about the architecture, use-cases...

Limits to applicability, etc...

3. General requirements

Automotive requirements to be met (general)...

3. Virtual Device Requirements

3.1 Serial Device

3.1.1 Standard Serial Device

- 💡 REQ-1: Requirement according to chapter x.y in [VIRTIO].
- 💡 REQ-1.2: Exception: Requirements 4 and 6 in that chapter shall not be implemented.

3.1.2 Network device exposed as serial device

*For this special type there are some **additional requirements**.*

- 💡 REQ-2: Requirement according to chapter 3 in [VIRTIO]
- 💡 REQ-3: The virtual device MUST provide an auxiliary programming interface (e.g. ioctl on POSIX) with the following features:
    - *feature 1*
    - *feature 2*

3.2 Block Device

💡 REQ-4: According to chapter 4.5.6 in [VIRTIO] **and** chapter 5.2.1 in [Other spec]

💡 REQ-4.1: When applying REQ-3, if there are conflicting requirements (note: disk performance, chapter 4.9.9), the requirement in [Other spec] shall take precedence.

💡 REQ-4.2: Device type Foo, listed as optional, shall here be mandatory.

💡 REQ-5: Device type Bar described in chapter 1.2.3 in [Other spec], shall here be conditionally implemented:

a. When implementing a system of type A, Bar is optional.

c. When implementing a system of type B, Bar shall be mandatory.

b. When implementing a system of type C, Bar shall be mandatory with the following modification of chapter 1.2.3

....

3. References

[VIRTIO] Virtual I/O Device (VIRTIO) Version 1.0, Committee Specification 04, release 03 March 2016.
