# An approach to model complex systems

## based on two principles being successful thousands of years by now

Carsten Pitz

carsten.pitz@gmx.de

## Abstract

The principle divide et conquere is even much older than its adaption by the Romans, but is still the key to handle complex systems. Another old principle is what has remained since a long time, will most likely remain a long time.

The approach beside being based on these two very basic principles respects cognitive limits and common habits of humans, the needs of different stakeholders and the fact a system architecture is something living that evolves over the time. Furthermore the approach is designed to optimize collaboration, because collaboration is viewed as the key to master highly complex systems.

But the paper is not about nomenclature, it is about principles. How the elements the paper deals with get named is up to your organization. That is the reason sometimes several names separated by slashes occur. In this cases I simply wrote the most common terms used.

## Levels of Structure

### On the Macro-Structure

The scope of the macro-structure are the dependencies between configuration items. The structure of configuration items is 2-dimensional. The two dimensions are spawn up by capabilities and the three levels of abstraction each of these capabilities is specified at.

#### 1st Dimension - Capabilities

A natural metaphor common to most people is capability. A capability is the ability to do something. A capability is sometimes referred to as feature. A feature, I was told some years ago is something a car dealer writes on the windscreen of a car to sell. I really like that definition.

## 2nd Dimension - Levels of Abstraction

Different stakeholders of a system architecture require different information. Product Owners (POs) are mainly interested in which capabilities (= user stories) get covered. Also a RAMS[i] (Reliability, Availability, Maintainability & Safety) engineer requires the capability specifications to perform the hazard analysis. But an RAMS engineer requires implementation details too, to approve the design. Project leads (PLs) require information enabling them to define and plan work packages. A software developer ...

Given that

- a PO requires to know the operational architecture only
- a PL can perform first planning on logical architecture level
- a RAMS engineer is damned to think on all levels of abstraction
- a software developer requires the physical architecture only
- ...

## Operational Architecture

On the highest level of abstraction, referred to as operational architecture (OA) a capability is described by an actor capable to fulfill a set of use cases.

A capability shall have a well-defined level or several well defined levels of quality. Beside the criteria given by the ISO/IEC 25000:2011 [ii] also requirements on throughput and latency (timing constraints) can and shall be specified on that level.

Also requirements on both safety and security can and shall be specified on that level.

An operational architecture mainly specifies what shall be implemented. The specification of how to implement it, is limited to operational procedures.

Both the common understanding what a capability is about and operational procedures are mostly extremely long-living. As a small anecdote: I was recently working as a systems architect in the railways domain, stating to a team member being train operator for more than 30 years that we currently model operational procedures defined some 120 years ago I got the answer: “Well, that is simply the way to operate trains.”. Operational procedures tend to be established to the level of common sense. Think of operational procedures defined for road traffic. How to proceed in front of stop signs or traffic lights is established to the level of common sense or “that is simply the way to ...”.

Operational architecture is a term used by the OMG UAF[iii], the Arcadia method[vi] and others. Using the terms defined by MDA[iv] the operational architecture is referred to as Computation Independent Model (CIM).

## Logical Architecture

The questions “Which tasks of an operational procedure is performed by whom?” and/or “How the use cases get implemented?” is answered by the logical architecture (LA).

The question “Which tasks of an operational procedure is performed by whom?” is merely the question “Which tasks shall be automated and which tasks shall be performed by humans?”. As a result the actors defined by the operational architecture either get realized by a system providing services or become operational roles.

In general the logical architecture maps aspects of a capability on logical entities (systems) performing services. The logical entities (systems) can be either technical systems or humans.

The logical architecture describes how capabilities get implemented. To clarify on that, the operational architecture may specify safety goals on a capability. The logical architecture shall specify mechanisms required to fulfill the safety goals, but shall not parameterize these mechanisms. As an example, if it is on logical architecture level obvious that redundancy is required to fulfill specific safety goals the logical architecture shall enable redundancy. But the logical architecture shall not specify if 2 or 3 redundant instances shall be present.

Even if logical architectures do not tend to be as long-living as operational architectures, logical architectures tend to stay for decades.

Multiple logical architectures may exist for a single operational architecture. Either because technology evolved enabling a more advanced logical architecture to be designed. Or because different levels of quality as specified by the operational architecture are mapped to different logical architectures.

Logical architecture is a term used by the SEBok[v], the Arcadia method[vi] and others. The OMG MDA method[iv] calls the logical architecture Platform Independent Model (PIM).

### *Physical Architecture*

Specifications like whether 3 redundant instances shall ensure the safety goals to be met are part of the physical architecture. The physical architecture is to answer the question “With which means the technical systems specified by the logical architecture get realized?”.

As the physical architecture depends on technical items being available, on physical architecture is obsoleted if a technical item required by the physical architecture reaches EOL[vii]. As an example a physical architecture of an electronic control unit (ECU) becomes obsolete if the used micro-controller (µC) gets discontinued. As a result physical architectures tend to life more short-term.

Multiple physical architectures may exist for a single logical architecture. Either because, as discussed before, an item used gets discontinued. Or because of rising volumes a different physical architecture can be produced cheaper (i.e. using an ASIC[viii] instead of a software based solution). Or because an altered physical architecture reduces production costs for other reasons. Or ...

Physical architecture is a term used by the SEBok[ix], the Arcadia method[vi] and others. The OMG MDA method[iv] calls the physical architecture Platform Specific Model (PSM).

### *Cross-Level Elements*

Some elements cover several levels of abstraction. Such elements might be

- a library model containing the dimensions and units to be used
- a library model containing an overarching taxonomy of terms
- a library requirements model containing requirements imposed by an overarching specification/standard

beside others.

When migrating legacy architectures, these legacy architectures referenced are most likely cross-level elements.

Cross-level elements can be manifold. The cross-level elements reside as a fourth category separated and might be referenced by all three architecture levels.

### *Meta-configuration items*

The approach results in a large number of individual, self-contained , versioned architectures. Each of these architectures is a configuration item. Meta-configuration items bundle all architectures belonging to a capability specification of a given version.
The meta-configuration items are a mere means to ease navigation.

## On the Meso-Structure

Scope of the meso-structure are the set of services a capability provides. Each service a capability provides is provided via well defined interfaces. Every single interface specification is treated as an own self-contained configuration item. This approach has been chosen, because every single interface has its own set of stakeholders and its own life-cycle.

On operational architecture level the stakeholders of an interface and their requirements on that interface get documented for each interface. On logical architecture level the information to be interchanged is specified. On logical architecture level only the dimension and the type boolean, enumeration, number or string of the information items get specified. Beside this the interchange protocol and the means to ensure integrity and confidentiality are specified on logical architecture level. The protocol shall also specify failure handling. The unit and exact physical representation of each information item is specified on the physical architecture level. Furthermore the physical architecture specifies with what values the means get parameterized to ensure the required degree of integrity and confidentiality.

The meso-structure provides nothing more than the black-box view on the services the capabilities provide. Based on the black-box view both integration tests and functional tests of the services can be specified.
There are several reasons for the strict separation of concerns, which gets realized by modeling for each atomic concern a dedicated, self-contained interface as a separated, self-contained model.

First of all an interface is a contract to be negotiated by all stakeholders. I have worked in companies that print out the interface specifications and let these print-outs be signed by all stakeholders. The smaller the group of stakeholders is the less effort is required to get the stakeholders commit on an interface. Furthermore the less complex an interface is the less effort is required to get the stakeholders commit on that interface. The negotiation process for each interface is divided into three steps. First each stakeholder can issue its requirements. These stakeholder requirements get consolidated and published as operational architectures. The operational architectures get committed on and released. In a second step for each concern a logical model based on the released operational architectures is created. As soon as a logical model is committed on and released the physical model based on that logical model is created. The detailed physical model needs only to be committed on by system and/or software architects. As a result each interface specification on logical architecture level references the released operational interface specification it realizes.
Another reason is that developers tend to highly underestimate the effort required o create something from scratch and to highly overestimate the effort required to re-use a

given item. Making individual interfaces as less complex as possible increases the motivation to re-use the interfaced item.

## On the Micro-Structure

The micro-structure contains the white box view of the capabilities. For each capability the white box view gets modeled as logical architecture first. As soon as the logical architecture is reviewed, committed on and released the physical model based on that logical model is created. The physical model then gets reviewed, committed on and finally released.

The logical architecture of a capability references the operational architecture it is based on and the logical architectures of its interfaces. In analogy the physical architecture references the logical architecture it is based on and the physical architectures of its interfaces.

As soon as the physical architecture of a capability is released a meta-configuration item referencing all architectures created for that capability gets created and released. It has been proven to be a good idea to also attach PDF documents generated based on the operational architectures. Just because a PDF document is much more likely to be opened and read than a model.

# The Meta-Model

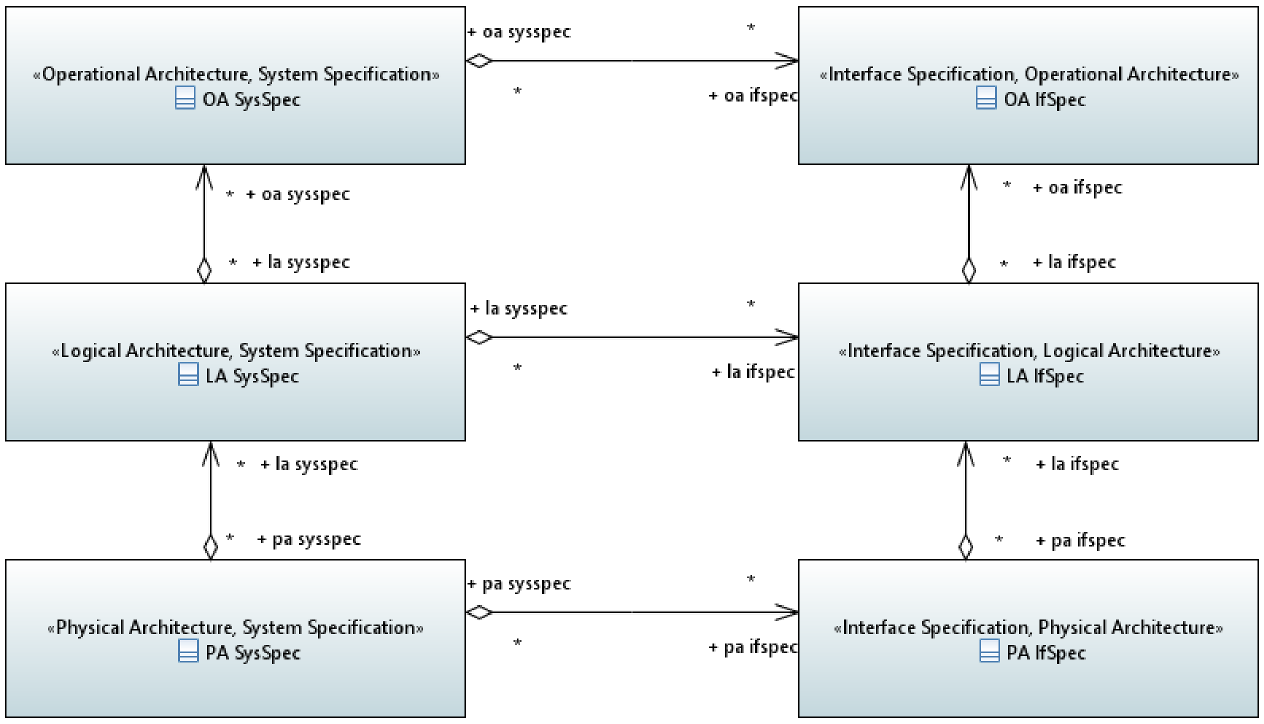

«Operational Architecture, System Specification»
OA SysSpec
«Interface Specification, Operational Architecture»
OA IfSpec
«Logical Architecture, System Specification»
LA SysSpec
«Interface Specification, Logical Architecture»
LA IfSpec
«Physical Architecture, System Specification»
PA SysSpec
«Interface Specification, Physical Architecture»
PA IfSpec
+ oa sysspec
+ oa ifspec
+ la sysspec
+ la ifspec
+ pa sysspec
+ pa ifspec
*

*Illustration 1: The Meta-Model*

The meta-model is as follows

On all three abstraction levels both system specifications as well as interface specifications exist.

Each system specification can reference any number of interface specifications of the same abstraction level. In the following examples a provided interface is referenced by an UML::Realization whereas a required interface is referenced by an UML::Usage.

Given that, the only way for a system to interact with another system is to use interfaces provided by the other system. As mentioned before these interfaces have to be committed on by the affected stakeholders and be in the state released.

Each specification can reference any number of specifications of the next higher abstraction level. These references are shown as UML::Abstraction in the following examples.

## The Configuration Item Specification

All specifications are specified to be configuration items. Each configuration item has its own lifecycle. Its current state is defined by its version, given by major, minor and patchLevel and its maturity state referred to as state.

### On the version numbering scheme

I strongly advise to use following semantics

major change - interface are either deleted or altered incompatibly

minor change - interface got extended only

patchLevel change - interface remain unchanged (bug fixing only)

The reason is, this allows a system to remain unchanged if its required interfaces change in their patchLevel or even in their minor version number.

## The Configuration Item Meta-Model

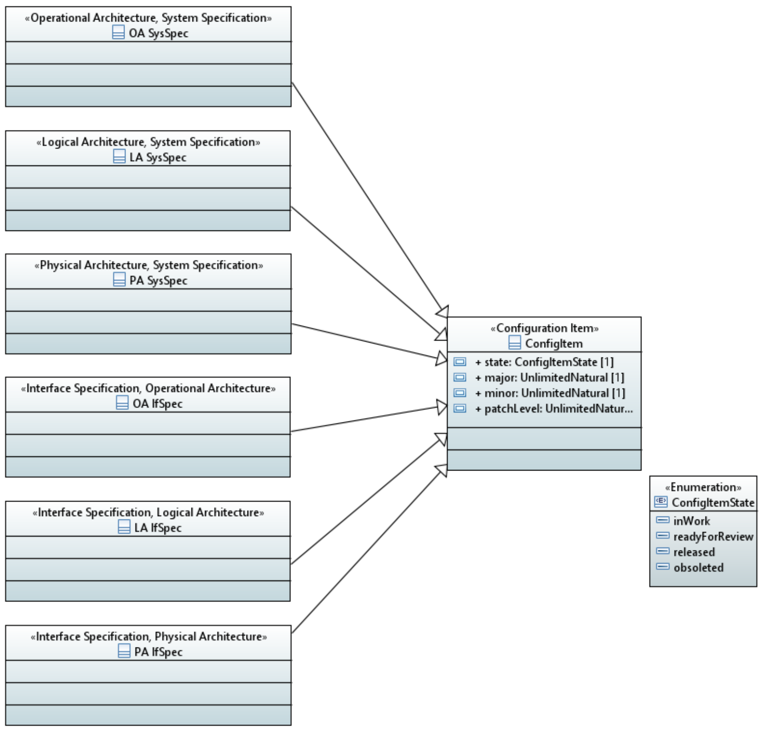

«Operational Architecture, System Specification»
OA SysSpec
«Logical Architecture, System Specification»
LA SysSpec
«Physical Architecture, System Specification»
PA SysSpec
«Interface Specification, Operational Architecture»
OA IfSpec
«Interface Specification, Logical Architecture»
LA IfSpec
«Interface Specification, Physical Architecture»
PA IfSpec
«Configuration Item»
ConfigItem
+ state: ConfigItemState [1]
+ major: UnlimitedNatural [1]
+ minor: UnlimitedNatural [1]
+ patchLevel: UnlimitedNatur...
«Enumeration»
ConfigItemState
inWork
readyForReview
released
obsoleted

*Illustration 2: The Configuration Item Meta-Model*

## Top-Down & Bottom-Up

The approach supports both the top-down as well as the bottom-up design approach. Interestingly the mechanism, interfaces is same for both approaches. It is a matter of the timely order of the specifications being created.

Both approaches result in the same principle structure being given by the meta-model ( Figure 1: The Meta-Model). As a consequence you can mix both approaches. Feel free to use whatever approach is better fitted in the very situation.

### On functional break-down - the top-down approach

A functional break-down is performed by specifying required interfaces on operational architecture level and realizing the required interfaces afterwards. That is the way to perform a FAST analysis or alike.

### On assembling compounds - the bottom-up approach

Compounds are assembled by specifying a system using interfaces provided by other interfaces to assemble a more complex compound system. This can be done on all levels of abstraction.

## Operational Architecture Example

This example shows only an incomplete excerpt (as the following two examples as well)

The operational architecture is the highest abstraction level existing. Consequently no UML::Abstraction relations start from the operational architecture level.

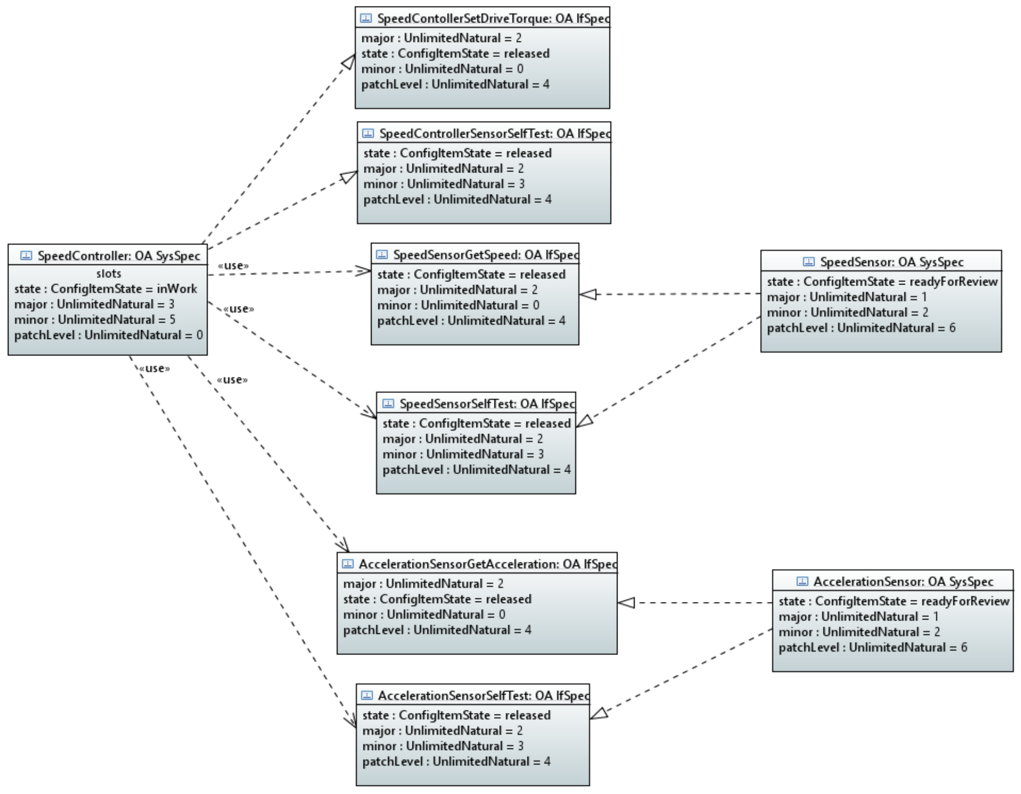

SpeedContollerSetDriveTorque: OA IfSpec
major : UnlimitedNatural = 2
state : ConfigItemState = released
minor : UnlimitedNatural = 0
patchLevel : UnlimitedNatural = 4
SpeedControllerSensorSelfTest: OA IfSpec
state : ConfigItemState = released
major : UnlimitedNatural = 2
minor : UnlimitedNatural = 3
patchLevel : UnlimitedNatural = 4
SpeedController: OA SysSpec
slots
state : ConfigItemState = inWork
major : UnlimitedNatural = 3
minor : UnlimitedNatural = 5
patchLevel : UnlimitedNatural = 0
«use»
«use»
«use»
«use»
SpeedSensorGetSpeed: OA IfSpec
state : ConfigItemState = released
major : UnlimitedNatural = 2
minor : UnlimitedNatural = 0
patchLevel : UnlimitedNatural = 4
SpeedSensor: OA SysSpec
state : ConfigItemState = readyForReview
major : UnlimitedNatural = 1
minor : UnlimitedNatural = 2
patchLevel : UnlimitedNatural = 6
SpeedSensorSelfTest: OA IfSpec
state : ConfigItemState = released
major : UnlimitedNatural = 2
minor : UnlimitedNatural = 3
patchLevel : UnlimitedNatural = 4
AccelerationSensorGetAcceleration: OA IfSpec
major : UnlimitedNatural = 2
state : ConfigItemState = released
minor : UnlimitedNatural = 0
patchLevel : UnlimitedNatural = 4
AccelerationSensor: OA SysSpec
state : ConfigItemState = readyForReview
major : UnlimitedNatural = 1
minor : UnlimitedNatural = 2
patchLevel : UnlimitedNatural = 6
AccelerationSensorSelfTest: OA IfSpec
state : ConfigItemState = released
major : UnlimitedNatural = 2
minor : UnlimitedNatural = 3
patchLevel : UnlimitedNatural = 4

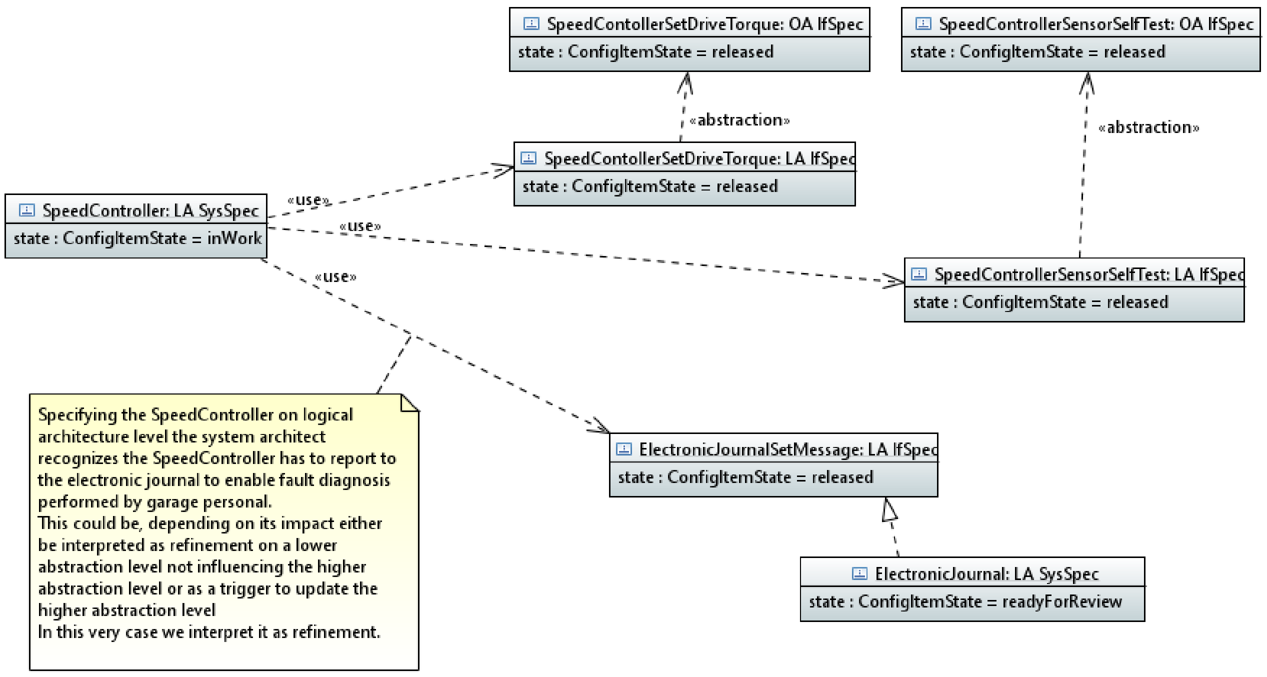
*Illustration 3: Operational Architecture Example*

The interface SpeedSensorGetSpeed is realized (UML::Realize) by the SpeedSensor and used by the SpeedController (UML::Usage).

## Logical Architecture Level Example

On the Logical Level everything is pretty much the same, interfaces get realized and are used. But both system as well as interface specifications shall reference the system specifications respective interface specifications these are based on.

In the example this is only shown for the interface specifications SpeedControllerSetDriveTorque and SpeedControllerSelfTest. Nevertheless all system and interface specifications reference their bases.


SpeedContollerSetDriveTorque: OA IfSpec
state : ConfigItemState = released
SpeedControllerSensorSelfTest: OA IfSpec
state : ConfigItemState = released
«abstraction»
«abstraction»
SpeedContollerSetDriveTorque: LA IfSpec
state : ConfigItemState = released
SpeedController: LA SysSpec
state : ConfigItemState = inWork
«use»
«use»
«use»
SpeedControllerSensorSelfTest: LA IfSpec
state : ConfigItemState = released
ElectronicJournalSetMessage: LA IfSpec
state : ConfigItemState = released
ElectronicJournal: LA SysSpec
state : ConfigItemState = readyForReview

Specifying the SpeedController on logical architecture level the system architect recognizes the SpeedController has to report to the electronic journal to enable fault diagnosis performed by garage personal.
This could be, depending on its impact either be interpreted as refinement on a lower abstraction level not influencing the higher abstraction level or as a trigger to update the higher abstraction level
In this very case we interpret it as refinement.

*Illustration 4: Logical Architecture Example*

As a convention all specifications being based on shall be released. The state of each specification is specified by state:ConfigItemState. The type ConfigItemState has, as mentioned before, been named intentionally to express each specification is a configuration item. Each configuration item has its own lifecycle representated by its version (major, minor, patchLevel) and by its current state (state).

As shown in the example, on logical architecture level interfaces might get realized or used by a system not being referenced by the system's specification on operational architecture level. It is allowed to specify details on logical architecture level that are not of interest on operational architecture level. As an example the SpeedController uses the Electronic Journal via the ElectronicJournalSetMessage interface. This usage has no counterpart on operational architecture level.

## Physical Architecture Example

Everything specified for the logical architecture level also applies for the physical architecture level. On the physical architecture level it is also allowed to introduce details not of interest on the logical architecture level.

A special case of a detail likely to be introduced on physical architecture level are the use of third party libraries. It absolutely does not make sense to mention third party libraries on logical architecture level or even on operational architecture level. That is why these are only represented by an interface specification on physical architecture level. In the example the ThirdPartyNumericalDifferentiationLibrary represents such a third party library.

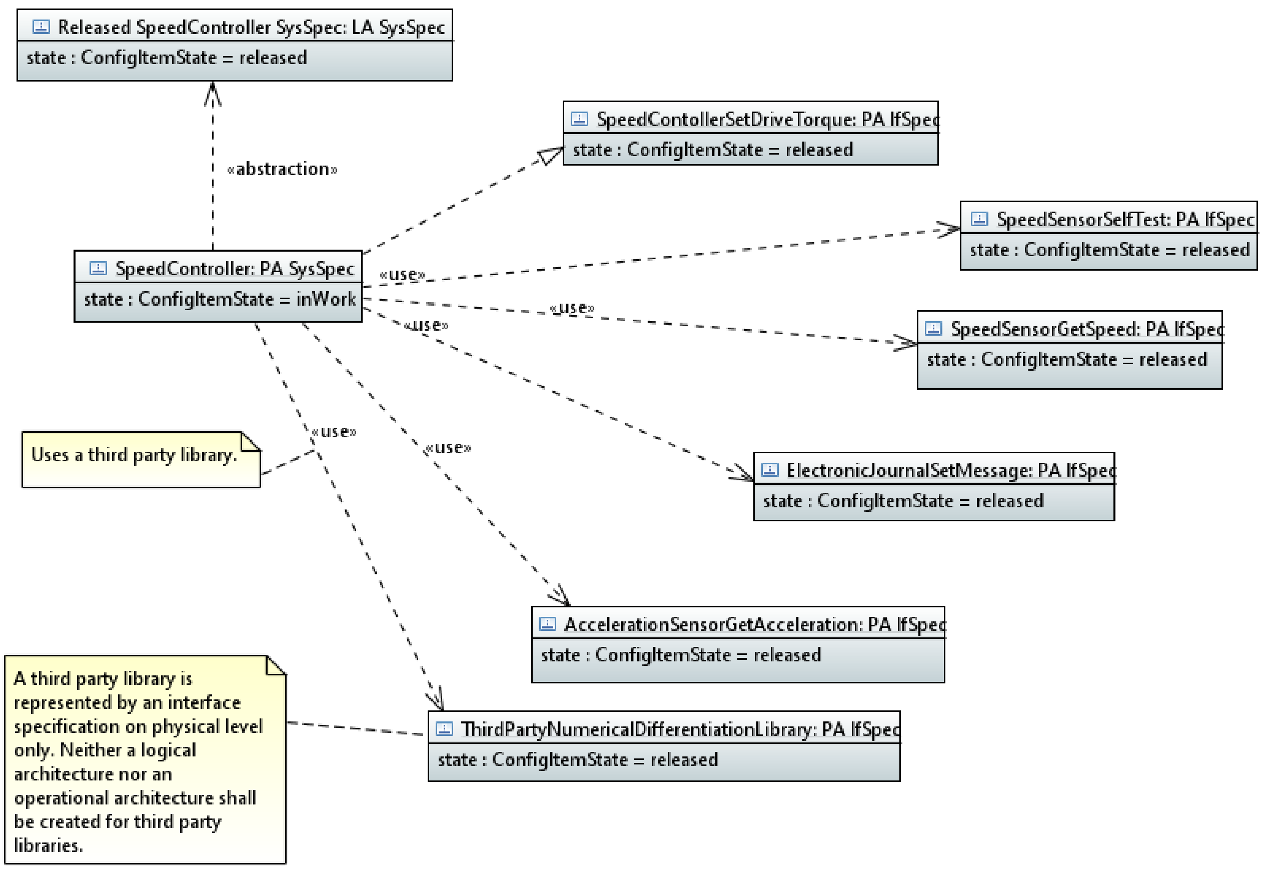

Released SpeedController SysSpec: LA SysSpec
state : ConfigItemState = released
«abstraction»
SpeedContollerSetDriveTorque: PA IfSpec
state : ConfigItemState = released
SpeedSensorSelfTest: PA IfSpec
state : ConfigItemState = released
SpeedController: PA SysSpec
state : ConfigItemState = inWork
«use»
«use»
«use»
SpeedSensorGetSpeed: PA IfSpec
state : ConfigItemState = released
Uses a third party library.
«use»
«use»
ElectronicJournalSetMessage: PA IfSpec
state : ConfigItemState = released
AccelerationSensorGetAcceleration: PA IfSpec
state : ConfigItemState = released
A third party library is represented by an interface specification on physical level only. Neither a logical architecture nor an operational architecture shall be created for third party libraries.
ThirdPartyNumericalDifferentiationLibrary: PA IfSpec
state : ConfigItemState = released

*Illustration 5: Physical Architecture Example*

## Augmenting Specifications

The mechanisms used by the meta-model also allow specifications to be augmented.

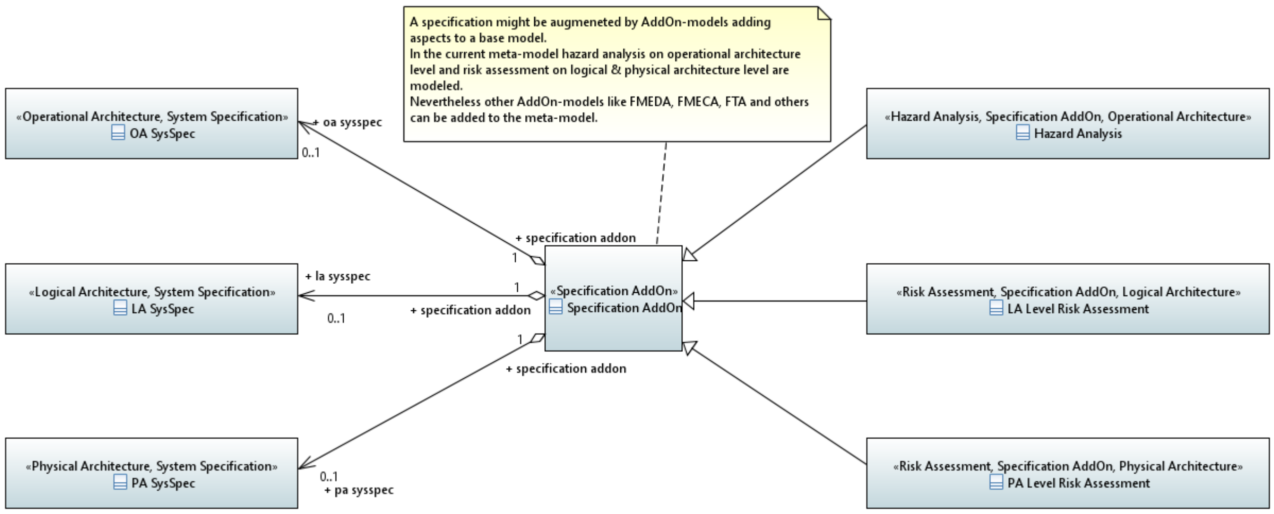
### The AddOn Meta Model

In the meta-model shown below only augmentation of system specifications is allowed. This has been done to simplify the meta-model to better express the mechanism.

An specification add-on references the specification it augments. The meta-model shows a hazard analysis on operational architecture as well as risk assessments on logical and physical architecture level as examples. This can be extended by FMEDA, FMECA, FTA or other models that augment a specification.

If interface specifications shall be included, the dependencies between the system and the interfaces that system provides have to be modeled too.

The rules for an add-on model to reference a specification model do not need to be as strict as the rules required for a system specification to reference an interface specification. The reference may already exist if the specification to be augmented is in work. But an add-on model shall only be released if the augmented model is released.

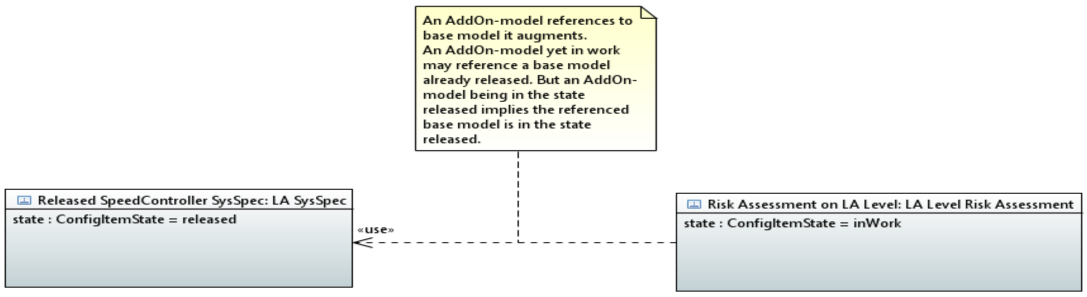

A specification might be augmeneted by AddOn-models adding aspects to a base model.
In the current meta-model hazard analysis on operational architecture level and risk assessment on logical & physical architecture level are modeled.
Nevertheless other AddOn-models like FMEDA, FMECA, FTA and others can be added to the meta-model.

«Operational Architecture, System Specification»
OA SysSpec
+ oa sysspec
0..1
«Logical Architecture, System Specification»
LA SysSpec
+ la sysspec
0..1
«Physical Architecture, System Specification»
PA SysSpec
0..1
+ pa sysspec
+ specification addon
1
+ specification addon
1
+ specification addon
1
«Specification AddOn»
Specification AddOn
«Hazard Analysis, Specification AddOn, Operational Architecture»
Hazard Analysis
«Risk Assessment, Specification AddOn, Logical Architecture»
LA Level Risk Assessment
«Risk Assessment, Specification AddOn, Physical Architecture»
PA Level Risk Assessment

*Illustration 6: The AddOn Meta Model*

## Operational Architecture Level Augmentation Example

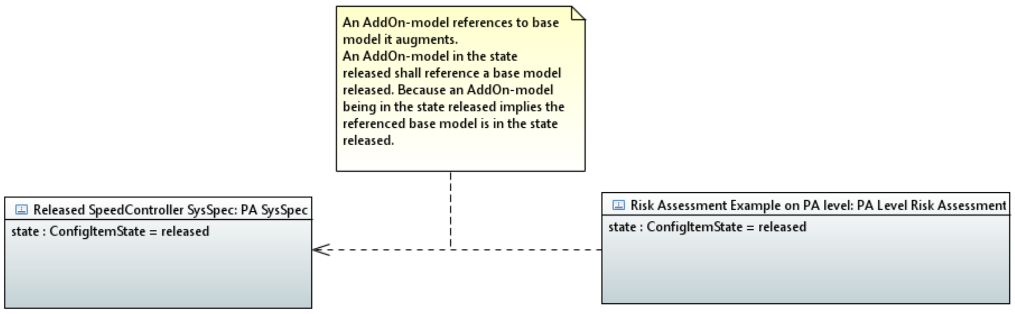
## Logical Architecture Level Augmentation Example


An AddOn-model references to base model it augments.
An AddOn-model yet in work may reference a base model already released. But an AddOn-model being in the state released implies the referenced base model is in the state released.

Released SpeedController SysSpec: LA SysSpec
state : ConfigItemState = released
«use»
Risk Assessment on LA Level: LA Level Risk Assessment
state : ConfigItemState = inWork

*Illustration 7: Logical Architecture Level Augmentation Example*

## Physical Architecture Level Augmentation Example


An AddOn-model references to base model it augments.
An AddOn-model in the state released shall reference a base model released. Because an AddOn-model being in the state released implies the referenced base model is in the state released.

Released SpeedController SysSpec: PA SysSpec
state : ConfigItemState = released
Risk Assessment Example on PA level: PA Level Risk Assessment
state : ConfigItemState = released

*Illustration 8: Physical Architecture Level Augmentation Example*

## Agile or V-Model or ...

Despite the fact the approach presented is a native agile approach it can also be used in environments using the V-Model.

We had this before: It is simply a matter of the timely order of the specifications being created. Both approaches result in the same principal structure being given by the meta-model ( Figure 1: The Meta-Model). As a consequence you can mix both approaches. Feel free to use whatever approach is better fitted in the very situation.

### Going Agile

Going agile you define epics and refine each epic with a set of user stories. Each user story gets fully implemented individually.

Given that an epic is a compound being divided into parts, called user stories using the top-down approach discussed before. Both an epic as well as an user story semantically perfectly fit into the operational architecture level. I suggest to describe an epic textually and its user stories as an use case diagram within a single operational architecture level specification. Furthermore I highly advise you to specify the actors (UML::Actor) in a separate operational architecture level specification referenced by the epic specifications.

To facilitate that, simply extend the meta-model (Figure 1: The Meta-Model) to support a system specification to reference system specifications on the same level of abstraction. I intentionally did not mention this before, because this extension will be most likely misused to couple systems directly and not via interface specifications. As a result I strongly advise you only to introduce this extension if you have a well-established governance. An IVVQ (Integration, Validation, Verification & Quality) department, to be mentioned later in this article, is an organization well suited to realize governance.

Each use case then gets realized by an interface (UML:Interface) specified by a logical architecture level interface specification which itself gets realized by an interface (UML:Interface) specified by a physical architecture level interface specification. The interfaces then get realized by elements (UML::BehavioredClassifier) specified by systems specifications on the respective abstraction level.

As a hint the UML specification explicitly allows an actor (UML::Actor) to realize (UML::Realize) an interface (UML::Interface). Given that on physical architecture level an (human) actor can be defined that realizes an interface. This does not make any sense on logical architecture level because that level is about “How?” and not about “With what means?”.

Even going agile a strongly advise you to go through all three levels of abstraction, because this facilitates the highest possible re-use.

### Following the V

As mentioned before following the V - or to be accurate, using the big-bang approach - all operational architecture level specifications are completed first.

The released operational architecture, meaning all contained operational architecture level specifications are released then gets realized by the logical architecture.

The released logical architecture, meaning all contained logical architecture level specifications are released as a third step then gets realized by the physical architecture.

### The capability based approach

The NAF v3, the DODAF as well as the Arcadia method are based on a capability based approach. A capability is defined as the ability to perform some task. A capability is semantically identical to an use case. It is just expressed in another way. Instead of stating for example “As a train company we want to be able to inform our customers about delays” one can also state “I as a train company want to inform my customers about delays”.

Given that equivalence the capability based approach is equivalent to going agile.

## On Interfaces

I recognized writing this paper, reflecting what I did the last 20+ years altered my view on interfaces. My view did not get completely altered, but the priorities of the aspects shifted. Before, the aspect that an interface abstracts from implementation had the top priority. Now the aspect that interfaces significantly reduce the required design and implementation efforts by decoupling users from providers is on the top.

### On the Rationale

As shown in the three examples (Figure 3: An Operational Architecture Example, Figure 4: An Logical Architecture Example and Figure 5: An Physical Architecture Example) users access provided entities via interfaces. Well, the meta-model (Figure 1: The Meta-Model) only allows users to access provided entities via interfaces and that is by very intention.

The intention, or to be more precise the rationale leading to this decision is: An interface, merely presenting the black-box of a specific aspect of an entity is in general significantly less complex than the white-box view of that entity given by its system specification. As an interface is significantly less complex, it is significantly less error prone or to express it the other way round much more stable in time.

Given that simple fact, forcing systems to interact via interfaces eliminates the need ,to perform an impact analyses on each change of the system specifications of the used entities. That very need for frequent impact analyses was a really serious problem for a team designing a system aggregating data delivered by 47 different systems connected. Introducing aspect specific interfaces as self-contained, versioned configuration items reduced the effort required to perform this impact analyses by more than a magnitude. Instead of being in permanent “battle mode” requiring, as time sheets proved, approx. 87% of the resources to react on the change reports issued by the authors of the connected systems, after forcing the other teams to design, deliver and maintain aspect specific interfaces as self-contained, versioned configuration items that number reduced to a tolerable 4%. In that very case it was needed to stop the payments to the other teams to make them design, deliver and maintain aspect specific interfaces as self-contained, versioned configuration items. That is why I wrote “forcing”.

I - consulting the team responsible for the data aggregator - did a surprisingly precise a priori estimate of the effort simply by stating the effort reduction factor expected is the number of impact analyses performed divided by the number of impact analyses requiring the system to be changed. Now, almost 11 years after I consulted on that issue, writing this paper I state: The a priori estimate was that exact, simply because only interface changes require an user to adapt.

## On Abstraction Levels, Contracts and the like

I assume how to specify the information to be interchanged via an interface on logical and physical architecture level does not require any further explanation. But what about interfaces on operational architecture level?

### On OA Level Interfaces

It was stated before that a system specification on operational architecture level describes a capability or an epic, depending on which term is preferred. Nice, but interfaces are about information interchange, about interaction. Do capabilities/epics interact?

Well, not directly but a capability/epic is always bound to operational roles represented by the actors (UML::Actor) involved. These Actors either use or provide parts of that capability/epic. Actors being users in the context of a capability/epic have operational information needs. Actors being providers in the context of a capability/epic provide operational information. The actors interact both within the context of a specific capability/epic and/or combine capabilities/epics to a compound capability/epic.

Given the above it gets obvious, an interface on operational architecture level describes an operational information provided by an actor.

I used the term "describes" by intention, because on that level of abstraction in general we have to deal with rather informal prose descriptions. These can be represented as Requirements (SysML::Requirement). E.g. in the case of the capability/epic "approving a business trip" a stakeholder might state "The employee shall state the reason for traveling.".

That level of detail and formalism proved to be sufficient on operational architecture level. Remember it has been stated before: The operational architecture level is about the question "What?". The "How?" is later specified on the logical architecture level.

### On Contracts aka the Liskov Substitution Principle

Interfaces are contracts in the sense of the Design by Contract Principle[x]. Given that both pre- and post-conditions have to be specified.

A pre-condition specifies what the user has to assure. E.g. a technical component using an electronic journal to persist issue messages shall only use message type IDs defined. Even though it is the duty of the user to assure each pre-condition, I highly advise to specify the providers to check if the pre-conditions are fulfilled and perform an error reaction in the case the pre-condition is not fulfilled. In my experience this leads to more robust, easier to fix and maintain systems. Simply because the explicit error reaction of the provider discloses the faulty implementation of the user. This discloses errors early and significantly reduces the effort required for debugging.

A post-condition specifies what the provider has to assure. E.g. a sensor is required to have a tolerance of at most 10mm in the range from 0mm to 1000mm. Post-conditions often cannot be checked by the users. In the rare cases this is possible (e.g. a disposition system shall ensure that tasks shall not have a due date before start date.) this check shall be specified for the same reasons as pre-conditions shall be checked.

One benefit of following the Design by Contract Principle is described by the Liskov Substitution Principle[xi]. Any system requiring pre-conditions at most as strong as specified and providing post-conditions as least as strong as specified fulfills the contract. This eases to introduce drop-in replacements significantly.

On operational architecture level the pre-conditions specify what can be presumed when using the capability. These can be skill levels of human actors, weather conditions and the like. For example it can be stated the capability presumes wind speed is at most 100kn.

Post-condition on operational architecture level is sometimes referred to as effect level. Effect Level is a term used in capability management[xii]. The effect level required specifies how complete and how mature the implementation of a capability shall be. In my experience it is in most cases sufficient on the completeness dimension to differentiate whether only the necessary requirements shall be implemented or all requirements regarded to be sufficient to gain broad user acceptance shall be implemented. The necessary case is sometimes also referred to as MVP[xiii]. The dimension maturity level is more complex. This dimension may include beside others all aspects of RAMS (Reliability, Availability, Maintainability and Safety) and Security. A MVP only to be used in a piloting phase mostly requires a lower maturity level than a product to be rolled-out for production.

As illustrated by the MVP only to be used in a piloting phase vs. product to be rolled-out for production comparison, different sets of interface specifications requiring different levels of pre- and post-conditions may exist for a single system specification.

### Other aspects

Beside various different sets of interfaces that may exist for a single system specification. Each set may have various access protocols. For example for a software system an interface for local access specifying a local API[xiv] may exist along-side with a remote access via remote API realized as a set of web services[xv] may exist.

Interfaces are provided and used. Because the number of users is in most cases higher than but at least the number of providers by convention the provider-side port gets conjugated[xvi]. This convention is established among ROOM[xvii] und RT-UML[xviii] users for decades. That eases users to access the interface and further improves acceptance of re-use.

## On Requirements

In most existing projects the requirements sets reflect neither the abstraction level nor the breakdown structure. Requirements of all abstraction levels and of different scope are mixed together in a huge single requirement-set. Sometimes attributes exist to specify the scope. Such requirement-sets are managed by storing them apart from the models in dedicated requirement management systems. This is the reason, why these are referred to as external requirements. This approach requires a bridge connecting the modeling tool with the requirement management systems.

Such requirement-sets define a single specific product and cannot be re-used. For that reason I strongly plead to handle such requirements sets as legacy. The order of the following three sub-chapters describes the suggested procedure to iteratively migrate to scoped re-usable requirement-sets.

## Map external requirements on model elements

This is the commonly present legacy state. Requirements contained in a single requirement-set get referenced by architecture elements fulfilling these requirements.

## Map external requirements on local requirements

In my experience the approach described in this paper results in at most a few hundred requirements needed for a system or interface specification. And even that figure is a rare worst case. This can be managed using a SysML requirements model eliminating the need to additionally use a dedicated requirements management system and as a consequence a bridge. As a result in the following target state instead of three only one single application has to be mastered by the architects and maintained by the operators.

### Use only local requirements

This is the suggested target state. Nevertheless beside local requirements also over-arching requirements exist. Such over-arching requirements are requirements implied by domain or organization specific specifications or standards (i.e. the VDA LV124/LV148 for the automotive domain). These specifications or standards in my experience result in specification-sets that can be managed using a SysML requirements model.

Traceability can be, depending on the required conformance level either be achieved by the navigation features present in modeling tools or by scripts analyzing the model. One might say dedicated requirement management tools deliver traceability out of the box. But in practice these are on the same level as navigation features present in modeling tools. In practice traceability is mostly verified by DXL or equivalent scripts.

### Requirements vs. Design Decisions vs. Constraints

In most if not all projects I was involved the last 20+ years design decisions like “Buffer size shall be 10” have been treated as requirements. Even though an IREB[xix] template has been used for the sentence, it is no requirement as no stakeholder requires the buffer size to be 10. It is a design decision based on a, hopefully educated guess by an architect.

In SysML this is handled by a parameterization specified by a set of parametric diagrams. UML does not feature a dedicated means to parameterize, but this can be achieved by setting cardinalities to a literal, by specifying constraints and other means.

We just mentioned the term constraint. A constraint constrains something. Both UML and SysML feature formal (i.e. OCL) constraints that can be validated automatically. Consequently using formal constraints instead of requirements not only puts the information in the right place, the use of formal constraints also enables automated validation.

### On Traceability

A reference shall always start at the user and reference a provider. Given that, an item fulfilling a requirement shall reference its base requirement. But why?

The rationale is simple: A provider provides its capability independently by whom the capability is used. Consequently a user depends on a provider, but not vice versa.

Besides that a provider may be used by more than one single user. Even worse, further users may be added later. Linking from a provider to a user would enforce the provider to be altered each time a new user is added. This not only introduces additional effort to unfreeze the provider, add the links and re-release the provider, but furthermore triggers the need for impact analyses.

A side note: As IREB requires requirements to be atomic, a requirement shall not be refined. The rationale is simply atomic (ancient greek ἄτομος) means cannot be divided. Consequently either the requirement is not atomic or cannot be refined.

## On Variant Handling

The only and hence suggested way to specify a variant supported by this method is to extend a base (compound) giving a compound specifying the variant. There is no way to express a variant that shall exclude some capabilities of some base (compound). Nevertheless a strip-down variant can be specified, with some little more effort. The strip-down variant itself becomes the new base (compound) and all other variants, including the former base (compound) get re-based on the new base (compound).

The re-base does not change the interfaces provided by the existing variants and is, consequently a local operation not affecting the existing users of the existing variants. That way of specifying variants extends the semantic of compounds beside being a mere container, compounds are also the mean to specify variants.

In my experience specifying variants by extending base (compounds) only, simplifies the analysis what a specific variant provides significantly.

## On IVVQ, RAMS and Security

Both the aerospace domain (DO 178 and DO278) as well as railways domain (CENELEC) explicitly require IVVQ and RAMS. The automotive domain (IEC/ISO 26262) is not that strict, but nevertheless IVVQ and RAMS are also of interest in that domain.

Security is of growing interest, not only in the domains mentioned above but even in commercial applications. Furthermore security issues allowing man-in-the-middle attacks corrupt safety.

### IVVQ - Integration, Verification, Validation & Quality

Even though IVVQ terminology is based on the V-Model, IVVQ fits perfectly into this agile approach. I think that is not at all surprising, hence, as mentioned before, big bang and agile just differ in the order in which tasks are performed.

#### Integration

Integration is performed by specifying compounds. Compounds aggregate capabilities, features or functions to provide higher level capabilities, features or functions. The integration check is done implicitly. Integration is done on operational architecture level first. Then it is performed on logical architecture level and finally on physical architecture level.

Furthermore integration is typically done incrementally. Compounds get integrated into compounds of compounds and so on. Each integration shall integrate at most 7[xx] items into a compound. That number represents a cognitive limit that shall be respected.

## Verification

As defined by the V-Model verification is the process to check whether the specifications on the next higher abstraction level are sufficiently concise to enable specifications on the current level of abstraction. Verification is a process being performed on the left hand side of the V. A verification results in a feed-back to the authors of the specifications on the next higher abstraction level. A verification either states a specification on the next higher abstraction level is accepted or issues a set of change requests. A specification being realized by several lower abstraction level specifications might get accepted by some of them while others issue change requests. This enables the work on the lower level specification accepting the base specification to start. Well, the others are still pending. But nevertheless the approach enables significant concurrency specifying system of systems.

In this approach not only lower abstraction level specifications being based on a specification verify this specification but also any specification on the same abstraction level using this specification. As a result verification in this approach is even more comprehensive than in the V-Model.

## Validation

As defined by the V-Model validation is the process to test products. Validation is a process performed on the right-hand side of the V. Test in this context means, run test cases on a product. Test cases shall be created for each specification.

On physical and also on logical architecture level test cases specify unit and integration tests. These can be both functional tests as well as non-functional tests. Each test case being specified on logical architecture level applies for each realization. Test cases on logical architecture level are abstract and consequently need to be adapted to reflect the concrete realization. Nevertheless concrete input values to be tested can be defined on logical architecture level.

An acceptance test is located on the operational architecture layer. Hence the test cases shall reflect operational procedures. The test cases for an acceptance test are also referred to as acceptance criteria and might be part of a commercial contract.

Despite the fact that test cases can only be executed on a realized product, the test cases shall be specified as soon as a first draft of a specification exists. Architects and test engineers shall collaborate. This is why I suggest creating test specifications as add-on models being bi-directional linked to the specifications the test cases apply to.

Beside of a test specification being a means of validation creating a test specification also verifies the affected specification and assures the specification to be testable.

Validation is a further means to give a feed-back to the authors of specifications. In the case executing test cases discloses specification issues, these have to be solved by correcting the specifications.

### Quality

The described approach assures items to be based on quality assured items only. This reduces GIGO[xxi] significantly. Furthermore verification is even more comprehensive than with the V-Model. Validation is supported on all test levels. As a result the approach enables quality targets to be defined and reached.

I highly recommend using the criteria defined by the ISO/IEC 25000:2011 to specify quality requirements.

The approach is quality target specification method agnostic. Feel free to use kaizen[xxii], six sigma[xxiii] or any other method to define or refine quality targets.

This qualifies the approach to be used to build critical systems[xxiv].

## RAMS - Reliability, Availability, Maintainability & Safety

RAMS is a term originated in building safety-critical systems[xxv] also in some domains referred to as life-critical systems. Safety-critical systems are required to conform to RAMS metrics. For example a realization of a SIL 4 rated capability must assure the mean time between hazardous events to be at least 100000 years. This implies mathematical models (i.e. FMEA) have to be set-up and calculated to calculate this figures. In some cases formal proves are required to formally prove system's conformance.

RAMS also implies feedback loops. In the case a capability / epic / feature cannot be implemented conforming to RAMS requirements this capability / epic / feature has to be re-thought. In the case a logical architecture does not enable the RAMS requirements to get fulfilled, the logical architecture has to be re-worked.

Each aspect shall be handled by augmenting the affected specifications using add-on models.

### Reliability

Reliability is defined as the probability of components, parts and systems to perform their required functions for a desired period of time without failure in specified environments with a desired confidence. Given that reliability is related to operation time.

The expected reliability is mainly calculated performing a FMEDA. The reliability required is defined on operational architecture level. It is based on the capabilities specified. On logical architecture level the FMEDA can be set-up. The logical architecture mainly is about allocating behavior to elements or in brief about structure. Hence the elements to be considered and how these elements relate are given by the logical architecture. Consequently the template for the FMEDA can be created on logical architecture level. On physical architecture level the concrete values get available. So on physical architecture level these values are filled into the FMEDA template and the calculation can get performed.

- Mean Time Between Failure (MTBF), which is defined as: total operation time / #failures
- Failure Rate (**λ**), which is defined as: #failures / total operation time

What are these values? These values may be the number of redundant instances used or the reliability of an off-the-shelf product used or others.

## Availability

Availability is defined as the probability that the system is operating properly when it is requested for use. As example a system can be requested to be available 98% each working day between 06:00 and 22:00. Given that availability is mainly related to wall clock time.

As side effect availability requirements indirectly specify possible maintenance intervals.

As with reliability also availability gets required on operational architecture level, enabled on logical architecture level and implemented on physical architecture level.

## Maintainability

Maintainability is highly related with diagnostics. A maintainer needs to know what happened with an item to maintain that item. Diagnostic figures may include operation time since last maintenance, but also issues detected (i.e. ECC errors on memory reads).

Consequently also maintainability mainly gets required on operational architecture level, enabled on logical architecture level and implemented on physical architecture level. Why "mainly gets required on operational architecture level"? Simply because some requirements for diagnostics first become obvious at logical architecture level or even at physical architecture level.

## Safety

Safety is about livings not to get harmed. Harm can only happen, in the case something happens. Given that harms are bound to events or actions, or to generalize to behavior. The desired behavior of an item is specified via a capability / epic / feature specification on operational architecture level. Consequently the HARA[xxvi] (hazard analysis & risk assessment) shall augment a capability / epic / feature specification on operational architecture level.

A HARA results in a classification for each behavior to what degree shall be either assured or prevented. Such classifications mainly use SIL[xxvii], ASIL[xxviii] or similar classification schemes. As example commonly the event an airbag opens without need shall be prevented with ASIL D. On the other side an airbag shall assure to open when needed with ASIL B.

Beside others (i.e. reliability figures) a SIL / ASIL / or similar classification typically implies limits for

- $\lambda_{SD}$ = rate a Fail Safe gets detected,
- $\lambda_{SU}$ = rate a Fail Safe gets not detected,
- $\lambda_{DD}$ = rate a Fail Dangerous gets detected,
- $\lambda_{DU}$ = rate a Fail Dangerous gets not detected.

Furthermore the decision whether a system shall be fail safe or fail operable shall be made.

Consequently safety gets required on operational architecture level, enabled on logical architecture level and implemented on physical architecture level.

As with reliability a FMEA template shall be created on logical architecture level and calculated on physical level.

### Security

Security is about protecting information. Nevertheless security can be handled analogous to safety.

On operational architecture level, based on a risk assessment the decision is to be made to what degree each information item shall be protected. The logical architecture shall enable the level of protection and the physical architecture finally shall realize the level of protection.

As with RAMS the security aspect shall be handled by augmenting the affected specifications using add-on models.

Even though handling security and safety aspects have a lot in common, I strongly advise both aspects to be handled by different persons.

## Complexity, the main Rationale of the approach presented

The approach cannot make complexity vanish by magic. But the approach provides a structure to master complexity by distributing it. The overall complexity is system inherent and as a consequence remains the same whatever structure is used.

### On Complexity Metric Values

#### *The current status quo*

Currently the most complex systems are designed by the automotive domain.

Audi states[xxix] currently it requires

- ~60 million requirements
- ~1 billion lines of code

to build their cars.

#### *How many specifications are required for that?*

I intensely discussed that question first at a project at a German car OEM and a second time 2 years later at a project at a German 1st level automotive ECU supplier. In both cases the result was approx. 150.000 for all Level[xxx] 3 automated cars of that OEM.

At the first glance that figure is rather scaring. But dividing the 60 million requirements by the 150.000 specifications the result is an average of 400 requirements per specification. That is a sane figure, respecting the fact most requirements are design results or constrains.

A single high end ECU, the system under construction at the German 1st level automotive ECU supplier consists of approx. 1000 specifications. But a mere 150 of them were considered product specific. The rest of 850 were considered shared specifications. Given that ratio 85% of the specifications can be re-used.

*What to expect in near future?*

And these are Level 3 automated cars. Level 5 automated cars will be magnitudes more complex.

## On how to master complexity -- On Collaboration

Highly complex system cannot be specified by a single person in a reasonable time. Specifying and implementing highly complex systems respecting all constraints imposed by RAMS and Security requires collaboration, large scale collaboration. In my actual project in the railways domain the core design team consists of currently 20 persons and the complexity to be mastered is mediocre. In the case designing a Level 3 automated car this figure is more like a 1000 persons, excluding suppliers. In the case of a Level 3 automated car the design team is distributed. While overarching aspects are done by the car company itself, the component level gets specified by various 1st tier suppliers.

Even today's figures stress V-Model based approaches to their extreme limits and are likely to fail even more often in future.

Facing this I designed this approach with optimized collaboration as highest priority goal. It is all about enabling feedback and reducing feedback latency in the collaboration process as much as possible. It is not about correspondence, meaning to respond to each other. Collaboration, meaning to work together is about communication, meaning to get a common understanding. To get an understanding not just common to a handful persons collaborating on a single specification, not just common on a single level of abstraction, but being common on all levels of abstraction.

Validation and Verification are about communication, about getting a common understanding. That is why I elaborated on these terms before.

### On coherence

The above leads to an even more fundamental aspect, the coherence[xxxi]. The whole system of specifications has to be coherent. Validation and Verification are, beside being about communication also about coherence.

Coherence, or to be more accurate coherentism is about first-order logic[xxxii] also referred to as predicate logic and about proof theory[xxxiii]. Both first-order logic and proof theory work is better to be performed by finite state machines than by humans. Doesn't that contradict with the statement Validation and Verification is about communication.

I clearly state no, just because these are two completely different aspects. Neither first-order logic nor proof theory is able to check semantic, this part is up to humans. Even better finite state machines can free humans from the burden to do first-order logic and proof theory work themselves enabling them to focus on semantic, on understanding.

# Conclusion

The approach summarizes the author's 20+ year experience building highly complex systems. All mechanisms required by the approach can be achieved using Eclipse Papyrus or some commercial UML/SysML modeling tools.
The author also realized mental barriers. An example for such a mental barrier was the statement: "If you require more than a single model to model whatever system, you are doing wrong." issued by a well-known UML/SysML Trainer / Coach. Well, I am quite

sure today no complex software system is programmed using just a single source file. Interestingly the approach is well-established in programming software for some 60 years. The same is true for digital circuit design, PCB design and other domains. The approach represents nothing really new. It is just an adoption of a well-known, well-established approaches followed by different domains for decades.
Nevertheless mental barriers are the problem most difficult to solve.

i https://en.wikipedia.org/wiki/RAMS

ii https://de.wikipedia.org/wiki/ISO/IEC_25000

iii https://www.omg.org/spec/UAF/About-UAF/

iv https://www.omg.org/mda/

v https://www.sebokwiki.org/wiki/Logical_Architecture_(glossary)

vi https://en.wikipedia.org/wiki/Arcadia_(engineering)

vii https://en.wikipedia.org/wiki/End-of-life_(product)

viii https://en.wikipedia.org/wiki/Application-specific_integrated_circuit

ix https://www.sebokwiki.org/wiki/Physical_Architecture_(glossary)

x https://en.wikipedia.org/wiki/Design_by_contract

xi https://en.wikipedia.org/wiki/Liskov_substitution_principle

xii https://en.wikipedia.org/wiki/Capability_management_in_business

xiii https://en.wikipedia.org/wiki/Minimum_viable_product

xiv https://en.wikipedia.org/wiki/Application_programming_interface

xv https://en.wikipedia.org/wiki/Web_service

xvi https://www.omg.org/spec/UML/2.5.1/PDF (chapter 11.8.14 Port [Class])

xvii https://en.wikipedia.org/wiki/Real-Time_Object-Oriented_Modeling

xviii https://www.omg.org/news/meetings/workshops/presentations/embedded-rt2002/04-1_Selic-Watson_RT-UML.tutorial

xix https://www.ireb.org

xx https://en.wikipedia.org/wiki/The_Magical_Number_Seven,_Plus_or_Minus_Two

xxi https://en.wikipedia.org/wiki/Garbage_in,_garbage_out

xxii https://en.wikipedia.org/wiki/Kaizen

xxiii https://en.wikipedia.org/wiki/Six_Sigma

xxiv https://en.wikipedia.org/wiki/Critical_system

xxv https://en.wikipedia.org/wiki/Safety-critical_system

xxvi https://uspas.fnal.gov/materials/12UTA/08_hazard_assessment.pdf

xxvii https://en.wikipedia.org/wiki/Safety_integrity_level

xxviii https://en.wikipedia.org/wiki/Automotive_Safety_Integrity_Level]

xxix https://se-trends.de/tdse-2017-menschen-modellierung/

xxx https://www.sae.org/news/2019/01/sae-updates-j3016-automated-driving-graphic

xxxi https://en.wikipedia.org/wiki/Coherentism

xxxii https://en.wikipedia.org/wiki/First-order_logic

xxxiii https://en.wikipedia.org/wiki/Proof_theory