ARCADIA
QUESTIONS & ANSWERS

Real life answers to real projects questions

Jean-Luc Voirin

Table of Contents

1 Scope of this document

ARCADIA is a tooled method devoted to systems & architecture engineering, supported by Capella modelling tool.

It describes the detailed reasoning to

- *understand the real customer need,*
- *define and share the product architecture among all engineering stakeholders,*
- *early validate its design and justify it,*
- *ease and master Integration, Validation, Verification, Qualification (IVVQ).*

It can be applied to complex systems, equipment, software or hardware architecture definition, especially those dealing with strong constraints to be reconciled (cost, performance, safety, security, reuse, consumption, weight…).

It is intended to be used by most stakeholders in system/product/software or hardware definition and IVVQ as their common engineering reference and collaboration support.

ARCADIA stands for ARChitecture Analysis and Design Integrated Approach.

This document collects questions raised by engineering teams of various domains and organisations, about to deploy Arcadia, or already deploying it and needing some support or clarification on the method and its use.
The answers given in the document are those that were proposed to engineering teams and applied by them in a coaching context, with little or no filtering. So they may not be relevant to any context or domain, neither are they accessible to any engineer, but at least they faithfully reflect real life concerns and the way they were addressed.

Warning:
These questions and answers are by no means self-sufficient to understand Arcadia and master its deployment. The reader is strongly recommended to read Arcadia reference documents before entering this one.

2 Arcadia Reference Documents

An in-depth introduction and description of Arcadia, with explanations on the method, on the language, illustrated by detailed examples of application, can be found in the Arcadia reference book:

> *Jean-Luc Voirin, 'Model-based System and Architecture Engineering with the Arcadia Method', ISTE Press, London & Elsevier, Oxford, 2017*

A presentation of Arcadia main principles and concepts can be found in the following online documents, including this one:

- Arcadia Engineering Landscape: an introduction to Engineering as supported by Arcadia
- Arcadia User Guide: a first level description of Arcadia approach and main engineering Tasks
- Arcadia Reference - Activities: an in-depth description of Arcadia tasks and activities
- Arcadia Reference - Data Model: data created and exploited by these activities
- Arcadia Reference - Capabilities: main processes supporting engineering
- Arcadia Language - MetaModel: a more formal description of Arcadia language concepts
- Arcadia Q&A: real life questions and answers on deploying Arcadia

See table 'Summary of reference Documents Contents' next page.

For easier navigation capabilities (including in diagrams, between activities and data, etc.), a web version can be browsed here.

Advanced practitioners in modelling and Arcadia can also access the Arcadia-compliant Capella model of Arcadia, from which this material is automatically extracted, here.

Summary of reference Documents Contents		Book	Landscape	User Guide	Reference - Activities	Reference - DataModel	Reference - Capabilities	Language - MetaModel	Q&A
History	Why was the method created and tooled? For which purpose? With which benefits?	✓							
Philosophy	What are its objectives and expected scope? What are its specificities?	✓	(✓)						
	How does it address Engineering Issues and Challenges?	✓		(✓)			(✓)		
	What kind of major levers does it use to address them?	✓		(✓)			(✓)		
Principles and approach	What are the drivers of each core perspective ....? How to build each of them?	✓		(✓)	(✓)				
	How to address Major engineering Issues using Arcadia and these perspectives?	✓					✓		
Details for implementation	What are the detailed processes to build each of the core perspectives?	(✓)			✓				
	How and where are engineering data elaborated and used to address major engineering challenges?	✓			(✓)	(✓)	✓		
	What is the formal definition of the Arcadia language & concepts?	✓				(✓)		✓	
	Examples and samples of models?	✓							
Hints for Deployment	Which major questions arise in projects applying Arcadia?								✓

✓ : fully detailled (✓) : simplified or partial

3 Arcadia core Perspectives: Why and How

3.1 What are the objectives of the Operational Analysis?

Operational analysis is an essential contribution to the development of input data for defining the solution. It requires temporarily setting aside expectations about the solution itself, in favor of understanding users, their goals, and their needs, free from preconceptions about the solution.

However, its most important use is to stimulate a deeper understanding and definition of the need beyond the customer's requirements alone, achieved through a critical analysis of the formalization represented by the operational analysis.

Much of the added value of operational analysis lies in the "analysis" aspect rather than just capture or formalization, as it stimulates different perspectives, opportunities, constraints, and risks that might not otherwise have been taken into account. It can lead to proposing new capabilities that the system could offer, new services expected from it, as well as operational contexts and previously unforeseen risk situations.

For example, in many operational analyses, the first-level description is often not very different across missions. This is normal, but to be useful, it is necessary either to refine it to reveal differences at a finer level, which will feed the reflection (not recommended a priori), or to ask the question: "where are the differences, on which entities and activities do they occur, what constraints do they impose?" The difference may lie in constraints, non-functional properties, quantitative elements of scenarios, nature of information and time scales, etc., which will then be essential for the performance of the solution in real operational context.

Furthermore, the behavior of cooperative entities is often well described in operational analysis (although often generic, see above), but the behavior of other actors (threats, objects of interest, or non-cooperative actors in particular) is often not. However, there is a good chance that many of the constraints and expectations on performance come from the specificities of their behavior, their temporal evolution, their own goals and capabilities, etc. All of this must be captured and analyzed within the framework of operational analysis.

It is then essential, of course, to compare the analysis of the system's needs (and beyond) with this analysis and to evolve one or the other if necessary.

3.2 What approach to follow for an Operational Analysis?

A good way to start is to forget about modeling and instead focus on "telling a story" that describes the needs, expectations, and daily lives of end users and stakeholders.
This can be done using existing documents, observing stakeholders using current systems and means, or conducting interviews ("what is your job? Tell me about a typical day or mission. What makes your activities easier? What complicates or jeopardizes them?..."). At this stage, the formalization is limited to writing free text and checking that they are representative of the need.

- Establish the dimensioning situations and scenarios, the required capabilities to detect/face them and their dependencies, the induced constraints that will later influence the system
- Put these capabilities into a time perspective (capability roadmap)
- Define the operational doctrines, concepts, and roles of the actors, associated operational procedures
- Evaluate new situations, along with gaps and discrepancies in the capabilities of the existing system relatively to the goals to be achieved
- Identify opportunities that will then feed the definition of solution alternatives in response to this need
- Define the conditions for the future operational evaluation of the solution to be developed: expected properties, constraints, operational scenarios, etc.

Secondly, identify the key words in the previous material and determine for each how it could be represented in the operational analysis model (if relevant):

- missions and capabilities,
- operational entities/actors and their links,
- activities carried out by these entities and interactions between them,
- operational information and domain model,
- operational processes, temporal scenarios,
- mission phases, operational modes,
- engagement rules, etc.,
- programmatic vision (capability increments...).

Finally, build an initial model on this basis and validate/extend it with stakeholders.

Here are the types of questions that can be asked during the development of the OA, which can either lead to completing it or deriving elements for the rest of the architectural definition (not exhaustive!):

- For each major activity and stakeholder (entity or actor), what makes it different from others? What makes it effective, what hinders it? What does it need to improve its effectiveness? What are the opportunities to provide other outputs, to enrich the service rendered - possibly with additional inputs?
- For each activity, who else is likely to carry it out (entity, actor, other activity)?
- For each interaction between activities (and entities), what could disrupt it? Under what conditions does it occur? Who else besides the identified destinations could benefit from it? Who else besides the identified sources could provide it, or provide complementary elements? And would that be desirable?
- What disruptions are likely to occur in the activity or its inputs? What unintended uses could be made of its outputs?
- What are the conditions of overlap or exclusion with other desired, imposed, and suffered activities? What is the link with any operational modes and states of entities?
- What data or information, activities, actors or entities, interactions, etc., are most operationally important, according to the main points of view (value for the operational mission, criticality from a safety and security perspective...)?
- What representative operational scenarios and processes describe the desired use of operational activities, interactions, etc., to contribute to associated missions and capabilities ("sunny days scenarios"), and how do they fit into time (timing of the mission, operations, expected system lifecycle, etc.)?
- What unwanted operational scenarios and processes should be avoided ("rainy days scenarios")?
- What constraints apply on each element (especially non-functional ones: performance, security, safety, operational importance, value...)?
- What uncertainties, sequence disruptions, contextual changes, undesired states, degraded modes are likely to occur, and what consequences will they have on the previous elements?
- How will the needs, constraints, contexts, situations, processes, etc., evolve over time?

Important note: there are many other analysis and creativity approaches, such as CD&E, design thinking, approaches like C-K, and methodological approaches supporting enterprise architectures such as NAF V4, which would be useful and which the analysis of OA does not replace (and which I also recommend considering). But on the one hand, they are complementary and do not cover the same needs or the same scope of reflection as Arcadia, and on the other hand, their data should mutually nourish each other with that of OA.

3.3 How to use the Operational Analysis for system needs analysis?

Based on operational analysis (OA), system needs analysis (SA) will define the contribution of the solution (referred to as the "system" hereafter) in terms of scope and expectations, particularly functional expectations. In other words, what services should the system provide to users to contribute to operational capabilities?

When conducting system needs analysis from operational analysis, the following questions should be asked:

- "Among the operational capabilities, which ones concern the system and/or its operators/users?"
- "How can the system contribute to each operational activity (from OA)? What services (or functions) can be derived from it, whether they are expected from the system, users or operators, or entrusted to other external systems?" This may also generate new features to offer or different system contribution alternatives.
- "For each interaction between operational activities, should/can the system be involved? In what way (displayed to the operator or treated internally...)?" This can also generate new services or expected functions.
- "For each function that is expected from the system, how can each operational activity or interaction benefit from it? How can it constrain or influence this function?"
- The same applies to operational processes and scenarios, states and modes... "How should the system assist users in each operational process, scenario, in each of their states and modes of operation? What constraints coming from these elements apply to the services it must provide?"
  Study the conditions of overlap of several scenarios, states and modes, and how this overlap can alter them: new activities, need for parallel activities, separation of an activity into several... and associated functions.

The separation between OA and SA, in addition to the benefits mentioned above, also allows the first major choices of engineering to be formalized and analyzed, particularly in the case of a new product or product line.

In these situations, several ways of meeting the same operational need may appear: different distributions of contributions between the system and its environment (use of other systems, for example), more or less advanced automation entrusted to the system, etc.

Engineering is therefore required to make an initial choice, not of design, but of delimiting the need allocated to the system.

It may then be wise to explicitly capture the different alternative terms, either through variability in the same SA, or in several SA candidates to be evaluated to select the most appropriate one for the context.

3.4 How to verify the validity of your system/user allocation?

A good definition of the human/system interaction is important, particularly regarding the users or operators of the system, both for communicating with the client and for preparing for integration, verification, and validation (IVV). To achieve this, you can ask yourself the following questions:

- Do the exchanges between operator/user and system accurately translate all the actions the user must take towards the system to carry out their operational activities?
- For each user activity that uses expected system elements, does it provide the feedback the user needs?
- Does the naming of system and user functions accurately reflect the part that each of them takes on?
- Are there multiple system implementation contexts (e.g. novice or expert user; or multiple levels of automation...)?
- In which cases does the user initiate the interaction, and in which does the system initiate it (e.g. alert, end-of-task notification...)? What information should the system present in the latter case?
- Between two interactions with the system, does the user need to make a decision that will condition the rest (in which case, it may be good to display it as an exchange between two operator functions, to make the Functional Chains or scenarios more meaningful)?
- Can the user be interrupted and resume their activities later, and if so, are there any constraints? In this kind of case, it would be preferable to divide the operator activity into several functions...
- Is the level of description of user activities and interactions necessary and sufficient to define and execute the verification/validation procedures?

3.5 Logical Architecture or Physical Architecture, do you need both ?

In some (mainly software oriented) modelling approaches, what is called a "logical" architecture is a definition of (software) components and assembly rules, while "physical" architecture is a description of one or more deployments of instances of these components, on execution nodes and communication means.

This is not the case in Arcadia: the major reason for logical architecture (LA) is to manage complexity, by defining first a notional view of components, without taking care of design details, implementation constraints, and technological concerns - provided that these issues do not influence architectural breakdown at this level of detail.

By this way, major orientations of architecture can be defined and shared, while hiding part of the final complexity of the design, and without dependency on technologies. As an example, some domains or product lines have one single, common logical architecture for several projects or product variations.

In fact, logical architecture should have been named 'notional architecture', or 'conceptual architecture', but we had to meet existing legacy denominations.

Physical architecture (PA) describes the final solution extensively: functional behavior, behavioral components (incl. software) realizing this functional contents, and resource implementation components (incl. execution nodes).

Physical Architecture provides details what have not been taken into account at LA level, in order to give a description of the solution ready to sub-contract, purchase or develop, and to integrate. So all configuration items, software components, hardware devices... should be defined here (or later), but not before.

For this reason, maybe physical architecture would be better named as ‘finalized architecture’…

As you can imagine, then, for one logical component, we can often find several physical components, relation between logical and physical levels being one to many. Similarly, functional description of component is notional in LA, and detailled enough in PA so as to sub-contract it.

Consequently, for example, it is much easier to start defining an IVVQ strategy or a product variability definition, on the limited number of functions and components mentioned in LA, to quickly evaluate alternatives and select best ones. Once this is done, the former definition can be checked, refined and applied to the fully-detailed description of PA.

For the same reason, once they have finalized their physical architecture, many people feel the need to synthetize it in a more manageable and shareable representation, grouping some components and functions, hiding others, etc. Which is simply building a posteriori their initially lacking logical architecture!

However, if really the level of complexity does not require two functional levels for the description of the solution, then the following recommendations should be followed:

- If an implementation view is not needed, then an LA is sufficient without a PA.
- If an implementation view is needed, then it would be better to put the detailed functional description in PA to have a regular model (and in Capella, the viewpoints, query engines, and other semantic browsers will thank you).

4 Arcadia language specifics

4.1 What is a Capability?

A **capability** is the ability of an organization (incl. Actors & Entities) or a system to provide a service that supports the achievement of high-level operational goals. These goals are called **missions**.

A mission uses different capabilities to be performed. A capability is described by **a set of scenarios, and functional chains** (or **operational processes** in operational analysis), each describing a use case for the capability.

In Arcadia, there are operational missions and capabilities in OA (Operational Analysis), as well as system missions and capabilities (including users) in SA, LA, and PA (System Need Analysis, Logical, Physical Architecture). An expected operational capability of an actor should, if the system contributes to it, be derived (via traceability links) to one or more system capabilities in SA.
In Capella, there should also be missions in both OA and SA, as provided for by Arcadia.

To understand what the concept of mission and capability covers, we can use the analogy of an employee in a company:

- The company assigns a **mission** to the employee (often summarized in the job title).
- The company ensures that the employee has the required **capabilities** to fulfill this mission (their skills, as listed on their CV).
- The employee performs daily **activities** (analogous to functions or operational activities), which are carried out by applying **processes** and procedures (functional chains or scenarios).
- Just as a (hiring) job interview validates the required skills, IVV will verify the capabilities of the solution.
- We can see that capabilities do not represent everything an engineer can do in their daily work; this falls under operational activities or functions, depending on whether we model in OA or SA, LA, PA.
- Therefore, Capabilities have more "added value" than activities or functions, and are directly related to a goal. This is why there are generally a limited number of capabilities in the model, and why they structure IVV, i.e. the adequacy of the system to the client/user's needs.

4.2 Leaf functions, parent functions?

The expected behavior of the system or a component is described in Arcadia by the functional definition constituted by the dataflow (functions, functional exchanges, functional chains and scenarios, modes and states, exchanged data).
A function is fully described by its textual definition (and/or the requirements associated with it), its possible properties or attributes and constraints, the data it needs as inputs and those it is responsible for providing as outputs, as well as its implications in the functional chains and scenarios that involve it: functional exchanges involved, definition (textual) of the function's contribution in the chain or scenario ('involvements' in Capella).

This functional definition must be made at the finest level of detail necessary for understanding and verifying the expected behavior. It is the functions at this level of detail (called "leaf functions") that are allocated on the components and that define their behavior.

For convenience, both to offer a more accessible level of synthesis and representation, and to structure the presentation of the functional analysis, these leaf functions can be grouped into more synthetic parent functions. But these parent functions have only a documentary role (by categorizing the leaf functions): the reference remains the leaf functions. For Arcadia, they alone carry the definition and behavioral analysis of the architecture.
For example, a functional chain that would pass through a parent function itself instead of its leaf functions would leave a major ambiguity in the model regarding the role of each leaf function and its expectations (data to be produced, contribution to the chain, etc.).

Therefore, in a finalized state of the modeling, all leaf functions should carry the functional exchanges involved and ideally be each involved (as well as its exchanges) in at least one functional chain or scenario.

4.3 Functional exchanges only on leaf functions?

Most modeling tools offer (mainly) a top-down construction approach based on decomposition (of components, functions), and the delegation of links connecting the previous elements at each level of decomposition. For example, first-level functions will be defined, connected by exchanges, then sub-functions will be inserted "into" each function, and a delegation link will be placed between a sub-function and the end of each exchange connecting the mother function to its sister functions.

In addition to the added complexity if a sub-function needs to be moved from one level to another, or if exchanges need to be added from the sub-functions, experience shows that this approach is neither the most natural nor the most common. For example, when modeling textual requirements, each of these requirements will result in the creation or enrichment of leaf functions, and exchanges will naturally occur between these leaf functions. One may then want to group these leaf functions into parent functions, but without having to go back and recreate all the exchanges to link them to the first-level functions with delegation links at each level. This is rather a bottom-up approach, and most often a combination of both (middle-out) is used.

This is why Arcadia recommends that, in a finalized state of modeling, only leaf functions carry exchanges, directly between them and without delegation. Thus, in a bottom-up approach, it will suffice to group leaf functions into a parent function at as many levels as desired; in a top-down approach, it will suffice to create sub-functions and then move each exchange from the mother to the sub-function that will take care of it.

Arcadia and Capella then propose automatic synthesis functions, only in the representation, to display diagrams in which the exchanges of the sub-functions are reported on the parent function if it is the only one visible. Similarly, exchanges can be grouped into hierarchical categories, which allow a simplified representation of several exchanges of the same category into a synthetic pseudo-exchange*.

* *Capella supports simple (and multiple) exchange categories, but does not yet support hierarchical categories.*

4.4 How to distinguish Dataflow from its usage contexts?

The dataflow describes all the inputs required by each function and its potential outputs. However, these outputs and inputs may not be constantly solicited or used in all the system usage contexts, as defined in particular by its capabilities.

Similarly, the functional dependencies in the dataflow (functional exchanges) indicate that a function may receive its inputs from several other functions that are capable of producing them, but not necessarily all at the same time.

Therefore, the dataflow describes all the possible exchanges between functions, which must not be questioned in the system operation (that's why it's immutable and cannot be modified).

Furthermore, the system is subject to variable usage conditions (or use cases), not all of which may require the full range of production capabilities and exchanges between all functions.

Thus, in one of these use cases (referred to as a context, often described in a capability), only a subset of the dataflow is valid and used. This is what is described by the functional chains and scenarios associated with the capability: which part of the dataflow is used in a given context, and how this subset of the dataflow is implemented. And that's why functional chains and scenarios are described separately from the dataflow itself, and depend on the context (capability) in which they are exercised.

4.5 How to use Functional Chains?

In Arcadia, initially, a functional chain is intended to carry non-functional constraints, typically end-to-end, in a given context (capability):

A functional chain is a means to describe one specific path among all possible paths traversing the dataflow,

- either to describe an expected behavior of the system in a given context,
- or in order to express some non-functional properties along this functional path (latency, criticality, confidentiality, redundancy…).

In order to avoid any ambiguity, especially when allocating non-functional properties to a functional chain (latency, criticality, confidentiality level…), some rules should be defined and respected to precise the meaning of a functional chain contents.

As an example, let's consider the case of a latency that the functional chain must respect end-to-end:

- If it is a question of expressing a need (specification) for a delay between a source event and a consequence, then a '1 start - 1 end' rule is sufficient and should be applied, otherwise there is ambiguity.
- If it is a question of expressing the need for multiple consequences or multiple source events, then the interpretation rule of first or last arrival, etc. must be added (e.g. latency is specified between the first of the inputs, and the output, or the last of the outputs, or the first…).
- If we want to express a latency holding property in these situations as a result of the design, then it must be true regardless of the intermediate inputs;
  otherwise, if the actual latency depends on additional inputs, this must be specified, but FCs do not have the required expressiveness for the temporal axis, loops, etc., so in this case a scenario, a constraint, text, or a dedicated viewpoint should be used for this expression.
- If we want to calculate latency by analyzing the model, either we use the simple cases mentioned above ('1I/1O' or 'nI/1O' or '1I/nO' + conventions), or we must define the behavior of each function in addition (output/input dependencies, synchronizations between them, etc.), for example in the form of an extension – and the associated conventions and interpretation rules (model of computation?) must be defined.

Obviously, I would recommend the simplest solution whenever there is no counter-indication to do so.

Note that this may require characterizing/classifying functional chains according to their uses and the associated conventions for each.

4.6 Should I use a Scenario or a Functional Chain?

Functional Chains Vs Scenarios:

Both can and probably should be used. Functional Chains (FC) are more appropriate to describing a "path" in the dataflow, either to better understand the dataflow itself, or to specify (often non-functional) constraints such as latency expectations, criticality for safety reasons (e.g. associated to a feared event)…

For time-related details, scenarios (SC) are better adapted, but they hide the overall context that dataflow describes.

When FC become complex (especially with several entries and outputs), then understanding their behavior may become tricky, and scenarios could be easier to understand: a rule of thumb could be "if you need to show how the FC behaves, by using your mouse or hands on a diagram, then consider using scenarios."

Although it is true that scenarios and functional chains are two facets of the same concept, each represents it for a different purpose:

- FC: we see its context and the other available paths within a dataflow; it is rather intended to carry non-functional constraints of the "end-to-end" type;
- SC: highlights the chronology, which is sometimes implicit or ambiguous in FC; can be defined at any level and be partial (i.e. "omit" parts of the FC)."

4.7 Adding Sequence Flows in Data Flows or not?

Data flows are there to show dependencies between functions (data required for operation, data produced), especially in order to build and justify the resulting interfaces. So it is indeed "real" exchanges that must be considered and not just pure control without any underlying "concrete" exchange.

I think that trying to transform Arcadia data flows into activity diagrams (AcD) or EFFBD would be a mistake: they do reflect the essential initial objective of expressing the need and orienting the solution (architecture):

- express the "contract" or "operating instructions" of each function (what it needs to operate and what it can provide),

- express the dependency constraints between functions (what F2 needs can be provided by F1, or F3),
- and do so globally (not locally to a diagram, as in an AcD or EFFBD): the function is characterized by the sum of all its dependencies expressed in all the diagrams, and these must be coherent (which is visible and done through the notion of function ports).

This contract or "user manual" of a function may include, as input dependencies, "activation conditions" indicating that the function can only be executed upon explicit demand: the characterization of exchange items as events can, for example, fulfill this purpose. Therefore, a functional exchange can also be activating, for example.

But this is absolutely not a *sequencing* link in the EFFBD or AcD meaning, a link that simply translates a precedence or execution anteriority; by the way, this often appears unnecessary or even harmful to impose while describing the need or orienting a solution, as it would be an over-specification issue, risking freezing constraints that do not need to be.

Let's take an example:

I can chain the oil change of Mr. Dupont's car after the changing of the coolant when it arrives at my garage because his engine is hot. As a result, I tend to put sequence flows between these two functions in this order.

But for Mr. Durand, who dropped off his car at the same time, his engine is cold when I take care of it after Mr. Dupont's, so I will first change the coolant before doing the oil change. So, pure sequence flows are "contextual" and do not reflect an "intangible" dependence between functions.

Moreover, if we look at the change of the coolant alone, we can hardly see what it would need as input from the oil change. On the other hand, we have overlooked (perhaps by focusing a little too much on control) an essential input, which is the engine temperature, and also the need for a function to control it, or even to cool down/heat up the engine beforehand…

Let us remember that Arcadia was designed for the description and verification of architectures, and not for directly executable processes. As a result, there are different needs and also limits to be put in place: for example, if we were to put a pure sequence link (without any data exchange) between two functions, and each one was allocated to a component located on a different processor, we would introduce an inconsistency in the definition of the architecture and interfaces: without "physical" communication between the two components, there is no way to impose the order of precedence described in this way.

However, it may be necessary to express - separately - the aspects of 'sequence in a given context'; the Arcadia functional chains and scenarios are there to express this aspect, without polluting the dataflow, by separating the concepts well. Pure sequencing information can be added here if needed.

A loop with multiple iterations can be represented, for example, in a scenario (sequence diagram), and even a pure sequence link can be used in a functional chain or operational process (being aware of the risk of inconsistency between sequence links and real interfaces, mentioned above).

Similarly, if necessary, a sequencing function that commands those to be chained (involving an exchange with each of them, of the activation item type) and one or more scenarios that express the behavior of this function in different contexts can be added, thus reflecting the required activation order in each context.

4.8 How to specify values or conditions for exchanged data?

Reminder: Arcadia separately defines the nature (or type) of data that can be exchanged (between functions or between behavioral components) and the use that is made of them.

The data model describes a piece of data independently of any use: its composition (its attributes), its relationships, possibly non-functional properties such as the level of confidentiality for example (via associated properties or constraints).

The content of each exchange is described by the concept of *Exchange Item*, which groups (and references) several data to be exchanged as a whole, simultaneously and coherently with each other (for example, the three geographical coordinates and the velocity vector of a mobile).

Each exchange carrying an exchange item references it in turn in the functional dataflow (several exchanges carrying the same EI may reference it). But so far, only the nature of the exchanged data has been described, which details the dependency between the functions of the dataflow.

Most often, the need to specify that a data must take a particular value arises to describe the expected behavior in a given context, i.e. in a reference to an exchange of a scenario or a functional chain. In this case, it is the *reference* to the exchange mentioned by the functional chain or the scenario that will specify the value taken by each data mentioned in the EI*.

Another need also arises when it is necessary to describe a conditional behavior.
This is the case for a 'control node' in a functional chain: the condition on the data is expressed on each 'sequence link' leaving the control node.

This is also the case for a 'control functional sequence' in a scenario: the condition on the data is expressed on each control functional sequence**.

* *This reference is implemented, in Capella, in the form of a 'functional exchange involvement' for a functional chain, or a 'sequence message' for a scenario; the description of the value on the data can be defined in the 'exchange context'.*

*** In Capella, the description of the value on the data can be defined in the 'guard' of the 'operand' grouping the exchanges concerned.*

4.9 How to represent a request with reply in the functional flow?

Let's imagine that we have one or more clients requesting data (or other information) from a server. At the functional level, we will have a "provide..." function allocated to the server component and a "request..." function allocated to the client component.

At the SA level, we can often limit ourselves to defining a single functional exchange from "provide" to "request" to represent the dependency between the functions, and nothing more.

In LA (and possibly in PA), the natural and most logical rule would be to create two functional exchanges:

- One with the request (e.g., the reference to the data) from the "request" function to the "provide" function
- The other with the requested data from "provide" to "request"

This ensures the readability of scenarios and functional chains, and characterizes the definition of inputs and outputs of each function.

These two exchanges and the associated exchange items (EI) could then be translated into a single behavioral exchange between the client and server (with a convention to be defined, such as the request being directed from client to server, which is consistent with scenario sequence messages and returns). The EI of the component would have an IN parameter, the reference to the data, and an OUT parameter, the data itself, and would ideally be connected to the EI of the functional exchanges.

If we want to save modeling effort and use only one functional exchange, we can use the IN and OUT of the EI to represent exchanges in both directions, with the following precautions (in this case, the convention must be described precisely as it is subject to interpretation):

- Either the functional exchange starts from the "request" function to "provide" to represent a query, and the EI has an IN "request parameters" and an OUT "request result"; this is consistent with the design & development view, as a message or request sent from the requester to the provider; this is what we would like to see on a scenario, with a sequence message and a return.

- Or, in the same situation, the exchange starts from "provide" because it is the direction of data flow, but the convention needs to be defined and seems less natural.
- Or we can use two exchanges to ensure the consistency of functional chains.

4.10 How many Input and Output ports for exchanges?

In the Arcadia method, not only is it allowed to have multiple exchanges exiting from the same port, but it is even recommended, and even essential, in certain cases, and not just to simplify the work of the modeler (although it does contribute to this):

By definition of a function, output ports express what it is capable of producing, independently of its uses (and thus of the number of uses).

For example, a GPS localization function is capable of providing a position. This position can, depending on the case, be used by one, two, 10... other functions, and this is not the concerns of the localization function. It is out of the question to define as many output ports as there are potential users, because that would mean that its definition would depend on the number of its users and would have to be modified every time a new one arrives...

Therefore, in this case, it has one output port, and only one. Each use of this position will result in a functional exchange between this output port and a port of the function that needs it.

Note that the remark also applies to input ports: they indicate what inputs the function needs to work; even if a function can receive data from multiple sources, if it processes them indiscriminately, then only one input port should be defined, which receives exchanges from various sources.

Linking several exchanges to a port means that they have no a priori relationship with each other: they are likely to occur at any time, independently of each other (such as requests from multiple clients to a server, for example, or sending emails by a server to multiple recipients), and the data transmitted a priori only share a common type (defined in the data model and the relevant exchange item), but they may be different instances.

On the other hand, to indicate that a position calculation function must combine a position from a GPS and another position received from an inertial unit, for example, this function will have two input ports (one for each source) carrying the same exchange item, which will group and describe the coordinates of the position.

If additional constraints are to be specified, such as simultaneous distribution of the same data to multiple recipients, or selective sending..., then dedicated functions for this purpose (split, duplicate...) are used. And if the same output is to be provided with different quality of service (e.g., a position accurate to within 10 meters every 10 seconds and another accurate to within 100 meters every second), two distinct ports will be defined, each with a constraint or a requirement on the quality of service.

4.11 Can we allocate Functions to Implementation Components?

Arcadia does not provide for the allocation of functions to implementation components (IC), but only to behavioral components (BC).

It's not so much the allocation of functions to ICs that is problematic, but rather its consequences for exchanges: what happens, for example, if two functions communicate, one on an IC and the other on a BC? Through which ports? What is the concrete physical meaning of this?

Likewise, this makes the meta-model more complex (should functional exchanges be allocated to physical links?), as well as the exploitation of the model by analysis views (safety, security, RAMT, etc.).

In fact, in a number of cases, it may be more appropriate to ask why: for example, if power supply functions are placed in an IC for an electronic system-level model, is it because they need to interact with processing functions? If so, there is probably a behavioral component to define to manage them, interaction "protocols" to define (at the level of exchanges and behavioral components), and routing of these exchanges through physical links, etc.; placing the 'power supply' function on an IC would not solve all of this. And is it really useful *at this engineering level*? It all depends on how it is used; if the main reason is "documentary" (to remember the need for such a function in later development), attaching a constraint or requirement on the component concerned, qualifying them appropriately (voltages, currents, loads, etc.), would probably be simpler and more useful.

4.12 Can Behavioral Components carry Physical Links?

« In physical architecture, a Behavioral Component (BC) could be a hardware component. In this case, why can't we create Physical Links between BCs? With Capella, I'm forced to artificially create 2 BCs. »

« A Behavioral Component could be a hardware component. » Yes, in the sense that a functionality can be realized by a hardware component (e.g. generating a 28V power supply, managing memory – by a MMU...). This component is therefore a behavioral component.

However, even in this case, it is necessary to distinguish between the outputs of the functionality (electrical energy, memory segment addresses...) and the means of conveying them (cables or backplane, address bus...). It is therefore important to distinguish behavioral exchanges from physical links, and to be able to allocate the former to the latter.

Similarly, for components, the interest of distinguishing behavioral components from implementation components lies notably in different non-functional properties: for example, if I want to allocate software components to execution resources, I need to separately specify the resources required by each software component (BC), and the resources offered by each implementation component (IC): computing power, memory, etc., in order to compare them. And I may also want to define different ways of allocating the same BC to different ICs. To achieve all this, it is therefore necessary to have the two concepts separated.

Of course, simplifications are always possible, but it is necessary to make sure that this will not cause other problems, especially for the exploitation of the model by analysis viewpoints (safety, security, RAMT, etc.). It was therefore preferred to systematically create an IC in these situations, which sometimes does require "putting an IC around a BC," but makes the model more regular and easier to exploit.

« If my job is 'power supplies', then a transformer that takes 220V and transforms it into 12V is behavioral, and the physical links are on these components. I don't want to create an Implementation Component (IC or Node) for nothing and have a Behavior Component inside. What would you do for my example? »

I would distinguish:

- a "voltage converter" BC that receives and provides an electrical voltage (AC or DC, to be specified) through behavioral exchanges and carries out the transfer function,
- and a "transformer" IC to which the electrical wires carrying the electrical voltage are connected as physical links.

That way, I could distinguish between a case where the transformer is used as a voltage converter, and another case where the same transformer (IC) is used as a low-pass filter, for example (the distinction being made by the BC).

(By the way, we could also imagine mechanisms in Capella that would assist in this creation of ICs transparently, as is the case for ports today, created automatically as soon as an exchange is created).

4.13 What benefits does the Physical Architecture Representation bring?

The expressiveness of languages such as basic UML or SysML (as available by default in COTS tools), which generally use 'Blocks' for all types of components and functions, is often insufficient, as is the tool support. For example:

- How to differentiate and represent software or firmware components from the boards, processors, FPGAs, etc. that support them, and visualize this allocation in a natural way? How to distinguish the same electrical transformer used as a voltage reducer or as a low-pass filter?
- How to separately specify messages and physical, material links that carry them, while allocating them to each other?
- How to model the routing of messages if they travel through multiple successive or parallel links, or a complex path?
- How to visualize the functional content of components at the same time as the architecture?
- How to represent the functional chains running through the system, and allocate properties to them (latency, criticality...)?
- How to define the modes and states of the system? How to specify the behaviors associated with each state or mode and what they apply to? How to show mode changes in scenarios?
- How to separate and represent the physical data formats coherently? For example, a position from an inertial unit via an ARINC bus, processed in software, then transmitted to the human-machine interface in Java and on an Ethernet bus via XDR... How to show the transport conditions of data sets?
- How to separately represent the need and solution (as is done for requirements in SSS and SSDD/PIDS) and manage their configurations and different lifecycles?
- The same goes for interfaces, preliminary ICDs, ICDs, and IRSs...
- How to automatically generate documentation, taking these different lifecycle stages into account?
- How to exploit models in order to articulate several levels of engineering?
- How to practically perform impact analysis on all this, especially through automatic analysis of the model according to specialized viewpoints such as safety, RAMT, security, etc., if we do not distinguish the different concepts to consider and confront?

For these various points (and many others), Arcadia and Capella, while not perfect or claiming to be exhaustive, offer proven solutions; among others:

- Separation of functional, behavioral structural, and implementation structural levels
- Explicit allocation of one on the other

- Separation of dataflow from functional dependencies and their uses in a given context (via functional chains and scenarios)
- Addition of dedicated concepts for the articulation and consistency between these different views (functional paths and physical paths, configurations and situations of modes and states, etc.)
- Separation and articulation between data (classes), transport units (exchange items), interfaces, exchanges, and ports

4.14 What is an Interface? How to use it?

- Interfaces in the context of Arcadia and Capella are not classes, and the two concepts are clearly distinct:
- Classes ('Data' in Arcadia) are only used to define the data manipulated in exchanges between functions or components. Defining a method for a class in Capella would have no use in the model; it will never be explicitly called in exchanges between components, but only potentially in the internal software code of the component (outside of the Capella model).
- Interfaces structure the services and means of interaction offered by the components:
    - An interface is composed of exchange items (EI), each of these EIs defining an elementary service (or event, data flow...) that can be provided by one or more components and used (required) by other components.
    - An interface usually has a functional or semantic coherence: for example, for a multifunction printer, different interfaces can be used for the scanning function, the printing function, the photocopy function, printer management, etc. And the printing function interface will offer EIs (services) for choosing print parameters (paper size, orientation, color or B&W), printing the document, but also alerting in case of lack of paper or ink.
    - An EI "carries" a group of data (or parameters), each of which is characterized by its membership class and its name in the EI.
    - An interface is attached to a component port and contributes to defining its "user manual."

All of these concepts are not necessarily to be used in a given context and may - must - be adjusted to strict needs.

5 Model building hints

5.1 What are the frequent mistakes? How to avoid them?

First, is Arcadia adapted to your scope and goals? Do you really target solution building, architecture definition & design and engineering mastering? Or are you mainly addressing better customer need understanding, focusing on customer side-capability operational deployment, programmatic issues… that he wants you to address (possibly together)?

- If stakeholders need mastering is the major expectation of your work, then an "architecture framework" based approach might fit better
- If your purpose is to explore the problem space or the solution space at high level, for orientation or concept definition and experimentation, here again, Arcadia does not fully address this: it will possibly enter the game later
- On the other side, if (architecture-driven) engineering is your focus, Arcadia targets engineering and architecture design, including operational and capability-related considerations that feed engineering – and here it is much better than architecture frameworks for that.

Before starting, it would be useful to list main challenges and expectations of engineering, along with building a 'maturity map' of the subjects that your engineering will have to face, so as to orient modelling towards addressing low maturity subjects firstly, when and if appropriate. Driving modelling by its major uses and challenges is key for stop criteria and contents definition.

Some of the most frequent misunderstandings in applying the method lie in:

- Not clearly separating need (OA, SA) Vs solution (LA, PA) ; this leads to cluttering the need description with solution considerations, so it is costly to maintain, difficult to read and approve by the customer; it constrains the solution (because approved by the customer) and prevents from considering other alternatives;
    - Therefore, take care to capture only need elements in SA especially, and keep it as small as possible.
- Considering that the functions describing the solution (e.g. in LA) are just a refinement of the need functions (in SA); this is usually not the case, especially if you reuse existing parts of the system, and might "corrupt" your SA or skew your LA ;

  - So check consistency between need and solution by traceability links (e.g. between system functions and logical functions, between functional chains and scenarios on both sides…) but don't try to strictly align both sides.

- Reducing the physical architecture to an allocation work of logical functions and components to hardware implementation components; if so, then your LA will probably be too detailed, thus costly to maintain and unnecessarily cluttered for most users;

  - Instead, consider the LA as the introductory high level description that will help presenting and discussing on the major features and alternatives of the architecture, sufficient to reason on major architecture design choices, but not detailing the behavior too much, this being done in physical architecture (where interface definition could be finalized as well).
    Of course, this can be adapted to your own context, but beware the temptation to over-detail too early.

Other typical pitfalls follow:

- Wrong level of focus on concerns and parts of the system and model: most engineers focus on what they know, and detail this a lot, while neglecting new or low maturity parts.

  - You should of course do the other way around, in order to raise possible problems and manage risk as early as possible.

- Addressing Reuse of existing components too early or too late: building a solution from parts without correctly mastering the real need, or at the opposite, designing a solution architecture from scratch, and then trying to insert existing stuff.

  - To manage confrontation between existing systems contents and new expectations, when modelling practices are established, existing contents should be described in an initial PA, while new needs should be captured in OA and SA; so confrontation takes place in the LA: LA describes expected new architecture, and compares to physical existing assets; gaps are detected between LA and PA, and the physical architecture is modified accordingly to specify required evolutions.

- Considering the IVV issues lately, waiting for the definition of the solution to be complete.

  - Capabilities are often a good way to drive and structure IVV and delivery strategy, while scenarios and functional chains will give you a prefiguration of test campaigns and test suites; the model will then help you identifying the required order of components deliveries, and test means contents, for example. This can be useful during bid phase as well, to size IVV activities and means, and also shape IVV strategy. Your design might also be positively influenced by taking IVV issues into consideration early, to make capabilities easier to verify, to split functional chains according to progressivity of integration, etc.

- Modelling without a clear vision of the kind of engineering problems that you want to solve with this modelling.
  The criteria for stopping the modelling and its orientation, as well as the guidelines given to each, depend on this answer. For example:
  - If it is a question of justifying a development cost, a rough model is more than enough.
  - If the interfaces need to be justified, then the functional analysis and data must be detailed, but especially between the components (or actors).
  - If you need to secure the reuse of existing assets, the functional analysis must be fine-tuned because it is in the details that reuse incompatibilities are hidden.
  - For performance analysis, it is towards the functional chains and the superposition scenarios that we must look first, then the definition of the processing and communication resources, with a functional detail limited to the dimensioning.
  - For IVVQ, a homogeneous definition guided by scenarios and functional chains is required; the architecture must be broken down according to integration constraints, dependencies between components and separation of test chains (functions crossed by the tests).
- Regarding roles and responsibilities, just a strong warning: the risk exists that modelling be run beside “the real engineering stuff” rather than at the heart of it; this would result in possible mistakes in model and poor representativeness, but also in architecture and design decisions not relying on the model, thus weaker engineering and poor value for modelling.
  - I would recommend that the architect, design authority and major authorities in system design be fully aware of model contents, and justifying their decisions based on it. This means not only monthly reviews, but day-to-day validation and appropriation of the model with the modelling team – or the architect could also be the lead modeler.

5.2 Why and how to use the Functional Analysis?

Arcadia, based on a component-based approach to architecture construction, relies on functional definition (dataflows + capabilities, functional chains, scenarios) to define and justify components, their interfaces, their expectations, and especially transverse behaviors that involve multiple components (functional chains, states, modes...).

- Components are created by grouping or segregating constrained functions
    - segregation/grouping constraints can both be functional and non-functional, but are all expressed through functional definition: safety-related feared events & hazards, reuse, product line, etc.
- Interactions between components are automatically deduced from functional allocation
    - via associated data flows that cross component boundaries
- Refinement is applied first to the functional analysis and the exchanges, and therefore automatically applies to components to which functions are allocated. Justification is done through functional traceability with the previous requirement model and requirements
    - via choice of precise behavioral design for each expected function, overall optimization between functions, etc.
- The behavior of each component is clearly defined by the scenarios that are allocated to it, as well as the functional content that justifies and specifies its interfaces and scenarios
- Each component is described as a full-fledged subsystem,
    - with its scenarios, its interfaces,
    - as well as its functional content,
    - the functional chains that cross it,
    - the process for obtaining the data it provides and the use of the data it requires...
- In a product line approach, variabilities are defined on the functional level (optional functions or functional chains, for example), and their impact on components is visible, traceable, providing an architecture that is easier to design for the product line.
- The overall behavior of the system is always visible, as are transverse functional chains
    - since they are based on functional definition, which remains visible and transverse to the definition of the components.

5.3 How to define and justify Interfaces between Components?

The basic idea of Arcadia for modeling interfaces between behavioral components is to rely on functional definition:

- precisely describe the expected behavior using functional analysis, through functional exchanges (FE) between functions that will be allocated to behavioral components
- group the data to be exchanged together into an exchange item (EI), which reflects the need to exchange them in a coherent and simultaneous manner
- reference the EI in the FE(s) that carry it between functions
- describe the dynamic behavior using functional chains and scenarios using these functions and exchanges

Once the functions are allocated to behavioral components:

- group and allocate the FE into behavioral component exchanges (CE) between behavioral components
- do the same for functional and behavioral ports if necessary (especially to be able to reuse a "stand-alone" component in a library, for example, without external exchanges)
- reorganize the exchanged data and EIs on the FE, if necessary, to construct the elements that will be exchanged between components (messages, commands, requests, or services, etc.) and allocate them to the CE
- group and structure the EIs exchanged by the components, referencing them in interfaces that characterize the conditions of use of the component; allocate these interfaces to the ports of components exchanging these EIs
- describe the dynamic behavior between components using scenarios that involve the aforementioned CE(s)
- describe the communication steps, if applicable, through mode and state machines whose changes are mentioned in the scenarios; each machine being allocated to a component involved in the communication

In summary, we synthesize and group the functional EIs into an interface (by simple reference), and similarly group the FE into a CE.

Finally, when behavioral components are allocated to implementation components:

- group and allocate the CE to physical links (PL) which will carry them between implementation components

As a result, it is possible to have several EIs on a component exchange, originating from one or several FEs. However, if we want to indicate that sometimes we exchange one part, and sometimes another, and we want to show it in the CE, it is preferable to create several behavioral exchanges. In this case, it may be necessary to create categories for grouping the behavioral exchanges for documentary purposes.

Note: The software approach known as "component-based development" (CBD) favors encapsulation; that is, we try whenever possible to use a component by only looking at it from the outside, without having to know how it is made inside, and therefore without having to "open the box" to connect to internal sockets.

The ports and interfaces are there to group, abstract, and provide a view of the "user manual" of the component (at least for behavioral components); they should therefore provide the appropriate "natural" level of detail in the component behavioral exchanges. The detailed interactions should be seen at the functional level (via functional exchanges).

Similarly, according to the principles of CBD, we should only connect high-level components (not sub-components) to preserve the abstraction that high-level components constitute (being able to consider them as a black box). The ports of sub-components would therefore only be connected to the ports of the parent component through delegation links. However, experience shows that there may be exceptions, particularly in more physical or material domains. In these cases, direct links between sub-components are necessary.

5.4 Should we use textual requirements or models?

Textual requirements are, for most customers, the traditional and most used way of specifying their needs (note that more and more customers use and require models as well). So they cannot and should not be discarded in our process, at least as an input for our engineering and a source of justification links towards IVV for Customer.

However, textual requirements suffer from weaknesses that may impact engineering and product quality:

- They are possibly ambiguous, and contradictory or incoherent with each others, with no formalized language to reduce these ambiguities
- Their relationships and dependencies are not expressed, and being implicit and informal, they may be wrong or contradictory
- They cannot be checked or verified by digital means, in most cases

- They are poorly adapted to collaborative building and reviewing, for the former reasons
- The process of creating links to each requirement (for design, IVV, sub-contracting…) is unclear, without a defined method; the way to verify these links is undefined; the quality of these links proves to be poor in many cases (as discovered lately in IVV)
- Textual requirements (alone) are not adapted to describing an expected end to end system solution (each of them only expresses a limited and focused expectation); they are not adapted to describe a solution
- They alone can hardly be sufficient to describe subsystems need, including usage scenarios, detailed interfaces, performances, dynamic resource consumption, and more
- …

Models tend to solve these weaknesses, thanks to:

- Defining a formalized language, less ambiguous and shareable, digitally analyzable to detect inconsistencies
- Bringing internal consistency thanks to the language properties and concepts (e.g. linking functions by data dependency, making them coherent with functional chains, allocating them to components, linking a required function to design architecture behavior implementing it, adding non-functional (NF) requirements such as performance or safety expectations on functional chains, etc.)
- Explicitly describing and tooling the process to build, link, analyze model elements above
- Relating all modeled elements to each other into one global view, thus providing means to check their coherency and consistency
- Favoring collaboration by natural, functional structure, and means to confront different views (e.g. capabilities/ functional chains/scenarios, functions and dataflows, data and interfaces definition, modes & states, components, and more)
- Describing need and solution separately, while providing justification and traceability links that can be semantically checked
- Describing solution based on both functional/NF and structural descriptions, functional/NF one justifying structural definition (interfaces, performances, resources usage…)
- Constituting a detailed components development contract (or subsystem specification), including all the former elements

- Easing IVV strategy and justification by directly feeding test campaigns with capabilities, scenarios and functional chains, thus improving the quality and efficiency of IVV
- ...

This leads, internally in our engineering practices, to using models as much as possible to describe both need and solution at each level, while complementing them with textual requirements, either to detail expectations (e.g. to describe what is intended from a leaf function behavior), or to express requirements that are not efficiently expressible in the model (e.g. environmental or regulation constraints).

However, it should be noted that:

- Not all requirements can be represented in the model.
- Those that can do not exempt a textual description of these elements themselves, which becomes more expressive and flexible than the model's formalism. This description may or may not be formalized as a 'textual requirement' object in the traditional meaning.
- Requirements that can’t be represented in the model must still be managed, allocated, and traced between systems and subsystems/SW/HW, AND linked to the engineering elements on which they relate (other requirements of different levels, tests, etc.).

But the key point is to treat these requirements according to their real usage in engineering, and governed by the model whenever possible (for structuring, navigation, justification, etc.).

Therefore, internally, requirements will be mostly model elements, and complementary textual ones. Both will be related to customer requirements by traceability links, allowing justification from test results up to customer requirements, through model elements verification.

5.5 Which requirements can be model-based?

Requirements that can be formalized as model elements (hereafter referred to as model requirements) include requirements of the following types:

- Functional, interfaces, performance: the most common, allocatable to elements of architecture model, verifiable by IVV scenarios (demonstration and tests).
- Structural (SWaP (size weight & power), distribution on geographical sites, recurring cost, number of copies, etc.): allocatable to elements of architecture model, often verifiable by inspection.

- Non-functional but related to architecture (safety, security, availability, etc.): allocatable to architecture model elements, in the expression of need (feared events, threats, essential goods, critical functions, etc.) and solution (safety and security barriers, redundancies, degraded modes, etc.), and verifiable by model analysis.
- Non-functional but verifiable by analysis or demonstration (simulation, formal proof, analysis by specialty tools including safety, security, availability, etc.): allocatable to specialized models supporting analysis or demonstration.

Requirements that can hardly be modeled are for example:

- Regulatory requirements: compliance with standards, etc. They can be transmitted to subsystems, but most of the time verified by inspection, therefore linked to little or no engineering assets.
- Contractual requirements: supply deadlines, maintenance duration, repair deadlines, etc. ditto
- Environnement-related requirements : temperature range in operation or storage, resistance to salted spray, etc. ditto
- Requirements directly allocatable to subsystems and without impact on the system: to be dealt with at a lower engineering level
- [Requirements that are modelable by nature (performance, safety/security, etc.), but whose current MBSE practices maturity does not yet allow to do so]: temporarily, they can have the same uses as modelable requirements.

5.6 Why use model requirements?

There are at least three major uses for these model-based requirements, as well as complementary associated textual requirements:

- To build a solution that takes into account User Requirements (URs). Arcadia proposes to formalize/confront/connect them in OA/SA, and then to construct a solution traced with respect to this formalization; thus, everything happens in the model if they have been captured and formalized there.
  Note: it is necessary to keep the possibility of directly linking model elements located in LA and PA to these URs: to have more accurate traceability, and thus to avoid too much detail in the SA if it was the only perspective that accounts for these URs, while increasing the number of URs that can be modeled. This also applies to so-called requirements which are more likely premature design elements.
- To define a Verification and Validation (V&V) process that demonstrates the adequacy of the produced solution with these requirements.
  What is actually verified is not these descriptions in the form of textual requirements,

but the satisfaction of the expected capabilities, through scenarios and functional chains representing this need, in accordance with the entire model description (including these textual description elements).
The Arcadia approach consists of constructing test campaigns based on the capabilities/functional chains/scenarios, while explicitly specifying and ensuring the link between need and solution (which is not done in the traditional approach), which allows for checking the model elements and associated URs. Thus, we always remain in the model, and there are model/tests links, from which we deduce the UR-Tests links indirectly.
The justification of the solution with respect to URs is done, for model-driven requirements, by "indirection": we check the model elements (via tests based on the capabilities, functional chains and scenarios of the model); once these are verified, we can verify the associated user requirements, and we can generate justification tables, for example, UR-test results.

- To define the realization contracts of subsystems. These are constructed from the physical architecture, and any impact and traceability analysis is also done in the model, which is specifying (the model elements are subsystem requirements (SSR)). If the subsystem is also in a model-driven approach, then we remain in this context, and the SYS/SS traceability is done between SYS and SS model elements. Otherwise, we will generate documents or even requirements exports, under the same conditions as for the client SSS, as described above.
  Note: the case of URs that can be directly allocated to subsystems and have no impact on the system should be considered. In this case, the simplest solution is to directly link these URs to the associated SS components in the PA.

In these three uses, I do not see any reason to "take the requirements out of the model" for the engineering level concerned. So for me, the simplest and most productive solution is to manage these requirements (system requirements, sub-system requirements) in the model only, with the appropriate links to URs, which are themselves by default in the model.

This also has the merit of greatly simplifying everything related to reuse, variability in product line, versioning, branch reporting, configuration management, feeding inputs and models of specialty engineering, etc. (all sources of complexity that are still ahead of us...).

The essential point here is to consider and treat these requirements according to their real use in engineering, and to govern them by the model whenever possible (for structuring, navigation, justification...).

To summarize the global approach:

- When analyzing need (SA, maybe OA): “translating” **customer/user requirements** (UR) to model elements, you link UR with SA model elements when appropriate; these model elements constitute the **system requirements**, along with complementary textual requirements if needed (linked to model as well)

- When defining the solution architecture in LA and PA, you create links between LA/PA functions and SA functions' (and functional chains, and other SA elements). You can consider that this brings indirect links between UR and LA PA components (component to LA PA function, to SA function, to UR)
- In some cases, you may directly link LA PA model elements to system or user reqs, if those reqs deal with physical considerations (eg the kind of operating system required by the customer, or environmental conditions…); it is better in this case not to artificially links these textual reqs to SA functions, which would create meaningless need functions, but rather to only link them to appropriate PA objects
- PA model elements, allocated to sub-systems or components, constitute most **sub-system requirements**; you can add some complementary sub-systems textual requirements to your PA (linked to model elements),
    - either to detail PA elements expectations (e.g. to describe expectations on a physical function)
    - or to "split" a system req to allocate parts of it to different subsystems
    - or to add reqs associated to system design choices.

5.7 Where to link textual requirements with the model?

Here, we mainly consider the requirements received from the client (user requirements, UR), assuming that the main uses of the model are:

1) defining the interfaces between subsystems,
2) defining the functional expectations of each subsystem,
3) making overall design choices,
4) driving the system IVV.

- If a requirement concerns the architecture (e.g. realization technology, constraints on a specific component...), then link it to the LA or PA, not necessarily to the SA.
- If a requirement impacts only one subsystem, it is preferable to link it only in the subsystem model (or documents), unless it obviously impacts the system-level SA functional analysis.
- If a requirement can be fully verified at the IVV level of a subsystem, same as above; unless it obviously impacts the system-level SA analysis.

- If deemed really necessary, these requirements can be linked not only to the subsystem model but also allocated to the relevant component representing the subsystem in the system model (in LA or PA, but not in SA).
- If the requirement concerns a functional analysis expectation, several cases are possible:
  - some requirements can only be expressed on a functional chain (e.g. end-to-end latency constraint, cyber-security...), or on a scenario
  - some requirements are specific to a function independently of its uses (e.g. criticality level), or common to all its uses (explaining and detailing the function behavior, thus detailing the function name)
  - some requirements are specific to the use of a function in a given context - capacity, functional chain, scenario, or even mode or state - (e.g. function behavior different depending on whether the system is in manual or automatic mode), and thus should be attached to the use of the function in the functional chain or scenario (the "involvement" in Capella).
- If the requirement concerns data to be exchanged, or a condition of interaction between two functions..., it would be more accurate to link it to the exchange, since it is likely to carry the definition of the associated data, rather than to link it to the source or destination function of the exchange (which allows in particular to find the requirements mentioning a data by going back from it to the EI and FE that carry it and from there to the associated requirements).

What is important is that in a finalized engineering, every requirement is allocated either to a model element, to another engineering element (simulation, study...), or to a subsystem. It would be useless and artificial to constrain or overload the SA to account for requirements that do not constrain either the functional aspects or the system's interactions with the outside world (this may be better understood with an example like "the processors used must be octo-core Core I7 ").

5.8 Does the system appear in the operational analysis or not?

It is important to understand the purpose of the operational analysis and when it comes into play in engineering. The operational analysis is particularly useful when seeking to best satisfy a client's need, without having an imposed system scope, or by seeking innovative ways to satisfy this client's need.

The OA should not mention the system, as it aims to understand the client's need without any assumptions about how the system will contribute to it; this is to avoid too quickly restricting the field of possibilities (which will only be done in System Analysis, by deciding at this level what will be requested from the system and what will remain with the operators and external systems).

The observation is that when the system is mentioned in operational analysis, potentially interesting alternatives in system definition are already excluded.

Let's consider this example: the client's need is to have means to hang a painting on the wall, the trend is to formulate the need as "how to use my drill system, what to do with it". As a result, we find the system (drill) in the operational analysis, so the game is over, no smarter alternative is possible. Operational analysis is not very useful, as it duplicates the role of System Analysis.

What Arcadia (and Architecture Frameworks such as NAF) suggests is to restrict Operational Analysis to what the client and stakeholders need to do: the (operational) capacity to have a localized fixation point in a specific location. Therefore, I am not yet talking about a drill in the operational analysis, but I am trying to analyze the need well: should the fixation be reversible? can the wall be damaged? should the position of the painting be adjustable? what will be the maximum weight of the painting? what will be its size? will it have to be placed and removed frequently? who will ensure the fixation, with what qualification?

From this Operational Analysis, Arcadia recommends a capability analysis, i.e., finding the different possible alternatives and comparing them: here,

- a hole + dowel + drill,
- but also a self-drilling point and a hammer,
- or an adhesive hook or a contact glue…

We analyze the various solutions (compromise facilitated, price, user training...) in subsequent candidate System need Analysis and early Architecture perspectives (SA/LA/PA), and choose the best (best compromise). Suppose it is the drill and dowel; we can now define the system, its system capabilities (fix a dowel) and its functions in final System Analysis (drilling, choosing the diameter, controlling the depth...).

Revisiting the OA with each evolution of technologies can then allow us to propose new products or solutions.

Once again, focusing on the system in Operational Analysis can bias the analysis. Two examples (a bit caricatural):

- If the system under engineering is an execution platform (multi-processors or private cloud, for example): if we focus the Operational Analysis on the platform, we naturally define how each actor interacts with it - at the risk of forgetting that the software applications hosted by the platform also communicate (first) with each other, and that the platform has a role to play in this communication.
  The approach proposed by Arcadia or the Architecture Frameworks would consist, in Operational Analysis, of analyzing for each actor with whom and why they

communicate - and thus, the "direct" communication between applications actors would appear. By building the System Analysis afterward, we should wonder how the platform system can support this inter-application communication, and we would thus reveal communication services provided by the platform - which would also allow us to properly characterize and size the needs of applications (e.g., broadcasting, events, microservices, etc.).

- Another example: a communication management system in an airliner: if the initial OA had been done by integrating the system (wrongly, as you understood ;-)), it would describe the links between the system and the actors (airlines, air traffic control, airport, weather services, etc.), and these links would be characterized by frequencies, protocols, etc.
  In this case, when moving to the SA, the added value of the SA compared to the OA is often not seen - rightly so. Furthermore, there are still no means to dimension the complexity of the operator load (e.g. during final approach) or data exchanges.
  The OA must therefore be done, temporarily forgetting the radio management system and focusing on the activities of the actors (airlines, ATM, airport, etc., but also crew) and the associated exchanges; in doing so, it highlights the nature of the exchanges (flight plans, weather files, etc.), the flight phases in which these exchanges occur... and the workload induced for the pilots.
  When moving to the SA and introducing the communication management system, its contribution to these exchanges is defined, which this time are dimensioned and time related; this makes it possible to establish a profile of system performance. And one can also see what is traditionally outside the system domain (e.g. voice communication), and which could inspire functional enrichment of the system (e.g. digital messaging replacing or complementing voice communication).

Another question and scenario: how to talk about maintenance in OA without taking into account the thing being maintained?

Taking into account the thing being maintained does not mean representing it as an actor: one can perfectly well describe, at the OA level, the maintenance processes of the system under engineering, describing only the maintenance team activities: dismantling, bench testing, fault localization, software and firmware updating, configuration verification, etc.
Then, in the SA, we will try to deduce the requirements on the system, by formulating questions derived from the OA: "how should the system be designed to be easily dismantled? What enabling systems are needed to test it, and what functionalities should they have? What observability capabilities must the system have to assist in fault localization? What software updating means? etc.
Some of this will translate into SA as requirements allocated to the system, and some as required functions of the system (e.g. download, version supply, observability or alerting reporting functions...).

5.9 Operational Analysis: Where to stop?

DO NOT include the system in the OA, as indicated above. The OA focuses on the users; the scope and expectations of the system will be defined and analyzed in System Needs Analysis (SA).

Set the boundaries of the coverage area: this is not easy, but otherwise you will end up modeling the entire organization of the client or users. The question to ask is like, "will this concern the users of my system, the systems interacting with them and with it, etc.?"

Do not forget to look for discriminators, priorities, criticality, etc., which will guide the system's choices and compromises in the following perspectives – along with value-driven engineering.

It is impossible and useless to formalize the *whole* operational material. It is sometimes better to capture it with organized or structured text and extract from it, to populate the OA model:

- Typical situations that will also be used for IVV, value analysis, cyber, etc.
- Illustrative cases of opportunities, constraints, or threats
- References to textual descriptions that provide all the detail and diversity of situations.

The model is there to organize and help analyze, detail and illustrate the raw textual data, provide a representative and synthetic view, that can be manipulated and analyzed in a systematic way, and then be related to the rest of the analysis and definition (SA, LA PA, simulations, IVV, etc.). But not exhaustively.

Obviously, the consequences of such an analysis can be scary, fearing an inflation in the size of the OA model, on the one hand, and the prospect of having to translate all this into SA afterwards. Several suggestions or recommendations in this case:

- In many of the above cases, most operational situations fall into a limited number of generic patterns; the diversity of situations is reflected in particular expectations for each in a given pattern, via constraints, non-functional properties, feared or expected scenarios, on existing elements (functional chains, scenarios...), therefore not an inflation of the elements already present.
- In other cases (as much as possible!), this will result in new needs or service opportunities on the system (and therefore to be captured in SA, and traced with respect to the elements of the OA that illustrate it, even if they are commented on by appropriate requirements or constraints)
- And regarding the OA → SA transition work, I think it also needs to be managed "economically": for example, an OA scenario representative of a situation that will need to be confronted in the design does not necessarily need to be translated into SA; it may suffice for future scenarios in PA illustrating the realization of the expected

capacity to reference the OA scenario and the associated services in SA, linked to each other as indicated above.

5.10 What's the difference between Modes and States?

In Arcadia, Modes and States are similar in concepts and description, but bear different meanings and uses.

- A Mode is a behavior expected of the system or a component, in some chosen conditions.
  Examples of modes for a drone: Autonomous flight, waypoint-driven path, remote-controlled flight; Collision avoidance active or not; take-off, en route, stationary flight (mission phases).
- A State is a behavior undergone by the system or a component, in some conditions imposed by the environment.
  Examples of states for a drone: Starting or On or Shutdown or Off; Available, or Degraded, or Failed/out of order; Day or Night; windy or still weather.

The first important issues to address in architecture modeling for M&S, are mainly:

- Identifying system modes and states, their conditions of realization and the functional behavior that they allow or need (e.g. functional chains)
- Identifying sources and nature of conditions for transitions
- Defining contribution of each subsystem or component to these system M&S, and complementary "local" M&S to be supported by the component according to its internal behavior.
- Checking that sources and contents of transitions are properly defined in interfaces (external and between subsystems or components) , and that appropriate functions support generating these triggers and supporting expected actions
- Checking that M&S state machines of various subsystems or components are coherent with each other (e.g. no deadlock or starvation in triggers…) – this is still to be tooled.

Theoretically speaking, any trigger should be produced by a well-identified function somewhere in the system, and be carried by functional / components exchanges.

By the way, remember that Arcadia and Capella target *architecture* engineering, and not detailed [software] design: if complexity of M&S raises, this might (or not) mean that they describe detailed design rather than expected general behavior.

For a more advanced "Modes & States engineering", Arcadia and Capella introduce new concepts:

- A Situation is a combination of states and modes representing the conditions of superposition of these states and modes simultaneously at a given instant.
  Example of a situation: (Autonomous or remote-controlled flight) AND Collision avoidance active AND Night AND windy AND Available.

- A Configuration is a set of System or model items that are globally available or unavailable in a given mode or state. Each mode or state is described by an atomic configuration.
  Example of configuration: collision avoidance->available, autonomous flight ->available, remote control link->available, localization sensor->failed

A recommended approach to verify that architecture design is valid concerning expected behavior, taking benefit from these concepts, would be (simplified):

- Define expected configurations that should be valid in different conditions in operation (mainly defining capabilities, functional chains or scenarios that should be supported by the system in these contexts)
- Define typical situations, representative of what could be expected or could happen in real operation (e.g. main kinds of component failures, main mode changes)
- Compute the global configuration resulting of superposing the different atomic configurations brought by modes and states in each situation
- Compare this result to each expected configuration

5.11 Should operators be in the system?

Operators can be modeled outside the system (as actors) or inside the system (in which case they are represented by components of the system*). In both cases, we can and should distribute the required functions to support the need between the system itself and its operators.

One criterion for differentiating the cases could be:

- If no constraint or need is expressed about operators in OA and SA (by the client in particular), then operators are considered as internal components of the system, some of which will be human; they will therefore only appear in the logical architecture, since this will result from a design choice (e.g. choice of a crew concept distributing roles to operators among several possibilities).

- If roles or characteristics (such as skill, availability, responsibilities, etc.) are assigned to operators in the expression of need, then it is preferable to make them appear from the SA (or even the OA), as full-fledged actors.

* *Capella allows a behavioral or implementation component to be characterized as human.*

5.12 What similarities and differences between System and Actors?

From the theoretical point of view, System and Actors definitions should have the same kind of capabilities, both being built of implementation components (IC), hosting behavioral components (BC).

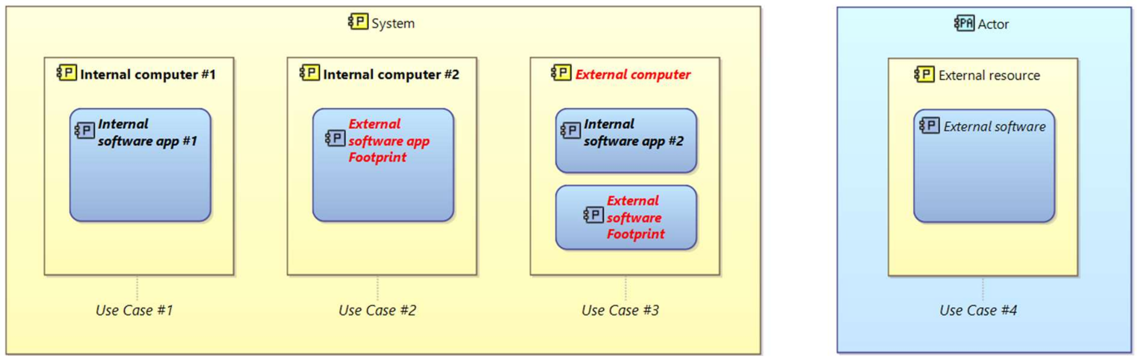
Among other benefits, this allows to move BC or IC from system to actors and vice versa, for different configurations or versions in a product line, and would ease complex transitions from system to sub-systems and Test means; this also allows defining true roles allocated to several actors and possibly to the system (using a Replicable element (REC) BC for role definition and replicas (RPL) BC to allocate a role to an actor or component), which is useful for Human-System Interfaces, or for example to manage variable delegation of service (at some time inside the system, and at other time in an actor). It is also a means to express that the system should use the same kind of IC as external actors (e.g. same kinds of computers), etc.

Leaving theory to meet practice, I propose (as shown in Arcadia reference) that, for sake of simplicity (to avoid creating unused components):

- at PA level, an actor is seen, just as an implementation component which is outside the scope of system engineering; so the actor is linked by physical links to a set of system ICs, and is a container of behavioral components, that will be linked to system BC by behavioral exchanges.
- At SA and LA levels, actors are defined the same way; but the physical links are only defined between the actor and the system itself (not with logical components, because they are not intended to carry physical links/ports).
- shortcuts would be accepted in order to allow actors to not only host BC, but also perform functions, along with associated ports and exchanges.

5.13 How do System and Actors interact and share resources?

In some cases, system and actors can be intricated in a complex manner and system boundaries not so clear, with respect to behavioral components (BC) deployment on implementation components (IC).



Let us consider 4 use cases (numbered 1 to 4):

- UC #1 is traditional: BC (in blue) hosted by IC (in red), both inside the system, and to be integrated/delivered.
- UC #2 illustrates an external BC to be hosted inside the system (e.g. an external software running in one of the system computers).
- UC #3 shows the opposite case where one of the system components is hosted by an external resource (e.g. a customer-supplied processing bord and software to be embedded in the system, or a human-system interface hosted in a multi-purpose customer workstation).
- UC #4 is a standard external BC on an external actor IC. This one is out of our interest.

The idea is to separate the kind of objects (Actor, BC, IC) from their qualification as internal to the system or not (based on the definition above).

A way to address these different cases is adding a property on objects, to define their origin and scope ('INTERNAL' or 'EXTERNAL'). This way, in PA, you could define UC #2 and UC #3 easily, while keeping them inside the system "boundary".

In UC #3, why is 'External computer' IC (red), hosting 'Internal software app #2' BC, located inside the system boundaries? Because it has direct and strong impact on system architecture, hence is to be considered as part of architecture evaluation: if it is a computing board, then depending on its power and available processing capacity, you might have to modify hosted software contents and interfaces; it will consume electrical power and contribute to heating and weight; it shall be part of reliability, security and cyber analyses; etc. This also partly applies if the computer is an external workstation (e.g. it affects system cyber-security analyses).

In UC #2, why is 'External software app' hosted by 'Internal computer #2' internal system IC mentioned here, although not part of system delivery? For the same reasons: if it is a software running on one of the system computers, then you have to provision enough computing resource for it, and in case of its failure, it could shut the computer down.

Furthermore, it might struggle with system own software components for board resources. So at least you should characterize its behavior and resource need under these perspectives. So it should be modelled as well, in a simplified form characterizing its functional interfaces and impact on the system components. This is what is called 'footprint') here.

5.14 How to model communication components?

Let's consider a drone system in which the onboard computer exchanges data with the control station via any type of radio communication channel.

For example, the computer sends sensor data (position, attitude, speed, etc.) produced by a "flight data computing" software component to the "trajectory display" component of the control station (both behavioral components). The communication itself is ensured by a dedicated communication board connected to the computer by an ad hoc link. In the model, this board will implement at least one behavioral component, "data transmission".

A common mistake is to model data passing from "flight data computing" to "data transmission", and then from "data transmission" to "trajectory display" (via exchanges carrying position, altitude, and speed).

Indeed, from the point of view of the "flight data computing" and "trajectory display" components, they are unaware of the communication means employed. In their software code, each component communicates directly with the other, without intermediary.

From the point of view of the "data transmission" communication component, it should not have knowledge of the content of each exchange it carries; otherwise, the specification of this component would have to evolve each time a new exchange is added between application components. Instead, it should receive generic communication service requests, which will be sent to it by the computer's operating system, not directly by the application components.

A much better way to model this case is to connect the application components directly to each other, without going through the communication component. The latter, and the communication board as a whole, remain in the model, as they are part of the architecture, consume resources, and must be included in safety, reliability, etc. analyses, but they are not connected to the application components. Moreover, the implementation details of the application components on the operating system, and the integration between it and the communication component, are irrelevant at this level of engineering modeling and should not be included.

5.15 How to engineer and model communication layers?

The way to model a system which includes communication layers (such as the 7 layers OSI model) is interesting, because it emphasizes the different scopes that different engineering teams have to deal with, and how they should articulate with each other.
Let us imagine a system in charge of supervising scattered installations on the Mars planet, environmental constraints requiring developing a new ruggedized, specific communication stack. Here, we consider the top level system engineering model.

- The top level system engineering team should usually:
    - focus on the application-level processing and communication needs : collect status of installations, send them to a monitoring application, delivering appropriate human-system interfaces etc.. A major goal is specifying this need to application software teams for example,
    - also specify the underlying communication infrastructure to the communication engineering team, by describing what kinds of communications should be supported, with which performance and QoS constraints (see below),
    - and integrate both, mainly focusing, at its high level, on operational, applicative scenarios and application level exchanges, the use of the communication means being implicit and hidden (provided that communication means had been validated separately before being integrated with application software).

- From an application (software) component engineering point of view, the component never has two different kinds of interfaces that cohabit: if you consider the software code of the application component, for example, it only communicates with other application level components, by sending messages, events, requests or so. It has no explicit call to low level communication API for example.
  So what the software application team expects as a specification, only deals with
    - application level exchanges,
    - and possibly the kind of service to be used for each communication exchange (if not to be decided by software team).

  Therefore, putting both application- and protocol-related interfaces on the application components would have no meaning.

- From the communication engineering team point of view, they have no need to get all the application level exchanges as an input, to define and build their communication stack; what they need is:
    - the different kinds of communication services, styles and paradigms that are necessary in order to implement the expected behavior,
    - expected quality of service for each service,

   - some kinds of 'communication performance profiles' that will define how the communication infrastructure will be solicited (how many services of each kind per second, size of transported data, real-time and scheduling constraints, and so on).

  This requires enriching the description of each application level exchange (or exchange item) in top level system engineering model, with complementary properties (such as required communication service or paradigm), and then synthesizing them so as to specify expected communication services.

  The means to define the performance profiles is trickier, and depends a lot from the level of detail in the system engineering model, and the details of the dynamic behavior description. It might have to come from another separated (non Arcadia?) model, more dedicated to performance issues.

Furthermore, system model, application software model and communication model have different lifecycles: adding or removing a new application exchange in the former will (and should) not affect the latter (the communication model).

Now, a question could be raised: is there any reason to include some description of communication subsystem in the top level system engineering model? The answer may be 'yes' if the communication infrastructure is likely to impact the system architecture definition: for example by adding dedicated communication components, or influencing performance analysis (communication layers might consume some resources in the system), and also to specify to the communication engineering team the network nodes that have to be fed with communication capabilities, and more...

In this case, both application level and communication level "sub-models" will coexist in the same top level system engineering model, but the communication sub-model would be less detailed: for example, you could define only communication components (because of their need for resources), communication services functions and functional exchanges (if needed for specification), but no protocol-related component exchange and interface. You should not try to link both sub-models explicitly, with exchanges, for example, because of the reasons above.

One way to express, in the top level system (application) engineering model, 'the relation between "the required communication services in the application model" with "the provided communication services by the communication model"', could be (among others) to

- formalize these services as functions of the communication infrastructure,
- name the 'paradigm' (or 'communication service') properties on application level exchanges accordingly,
- and check, by means of model queries, that each application exchange is typed according to one existing service function name.

Note: this situation also applies to other fields beyond communication stacks engineering, such as cyber engineering for example.

5.16 How to model a communication protocol?

In this meaning, a protocol describes the time-related chronology of interactions between two (or more) entities, so as to establish the connection between them, perform various exchanges of information, material, etc., while ensuring the appropriate quality of communication service, and then ending the connection.

In this case, we use the following concepts of Arcadia to describe the protocol (usually starting at functional level for most details):

- components or actors or system being the entities communicating through the protocol,
- functions allocated to them to express processing of communication and interaction on each side,
- functional exchanges between these functions to describe each elementary interaction,
- data (classes) and exchange items to describe the contents of these exchanges,
- interfaces to group these exchange items, in order to define expectations on components to support this protocol,
- but also component scenarios to illustrate the chronology of the former exchanges, different possibilities, nominal and abnormal behavior...,
- and mode or state machines to describe conditions of evolution and steps in the protocol; usually, one (or more) machine at protocol level, to describe the overall logic of its progress, and one (or more) machine for each entity to express its internal state and protocol progress from its own point of view,
- non-functional property values on these elements to describe expected quality of service (e.g. time constraints, bandwidth, encryption...).

Arcadia also introduces protocol as a language element, and interaction roles (as described in Arcadia reference book), but this is not yet supported by Capella. In the tool, these elements can be grouped into a Replicable Element Collection, and/or in a library for example, in order to be reused.

5.17 How to model the environment and physical sensors?

When modeling a system with an atmospheric pressure sensor, for example, one often encounters an external actor "Atmosphere", to which one tends to allocate a function "provide pressure", which is questionable from a physical point of view and unnatural. The question is even more difficult when it comes to identifying the component exchanges, the ports (the surface of the sensor on one side may be, but what about the atmosphere side?), and the physical link (the air?).

To improve the modeling a bit, I reason (as often) by separating the functional from the structural perspective, then by analyzing the conditions of use of the modeling carried out, including in terms of the level and scope of responsibility of the engineering.

From the functional point of view, we could very well imagine having only a detection function allocated to the sensor, without input; this is precisely the case for simple environmental measurements, in which permanent physical quantities are measured, without any other interaction with the environment. For pressure or temperature, for example, that's what I would do a priori, unless I am in charge of the engineering and model of the sensor itself, where I might accept a function of the atmosphere producing pressure - I would then rather name this function 'is characterized by physical quantities', for example.

But when there is a real interaction (influence of the system on the object following detection, for example), or if the situation of the external actor evolves (e.g. entering the field of a camera...), it is often interesting to start with the external actor equipped with a function providing information to the sensor. In this case, I define:

- an actor function which translates *its own observable behavior* (entering the field, moving; radiating in the electromagnetic spectrum...) and not the "production" of the information that the sensor will interpret (so not 'produce an image', nor 'emit an electromagnetic signature'...); this function is to be considered as a contribution to the specification of what a scenario generator, for example, will have to do to simulate the actor (generator that will move the actor...). One can also try to be generic ('exhibit physical characteristics'...), or even not name it, because it is the exchange that is important.

- a functional exchange, which translates *the nature of the information that the sensor will use* to detect the actor (electromagnetic emission, image, or simply presence information for a proximity sensor, distance for telemetry, landscape or view for a camera, pressure...); this exchange also contributes to the specification of the actor-system interaction simulator, this time by the nature of the information that this simulator will have to provide.

For an inertial navigation system, for example, it is more complicated: one would not have an environment function 'imposes inertia laws'; to test the system, it would be moved, nothing would be done in the environment; so I might not even add an external actor here. Or else, an exchange of 'relative movement', and an actor function of the type 'locates in space'.

This function also allows in particular to build scenarios and functional chains translating the feedback of the system on the behavior of the actor.

Note also that for qualifying the physical quantities involved and their nature, the data model and exchange items may suffice most of the time.

From a behavioral point of view, it could be imagined to do away with behavioral exchanges between the actor and the sensor, insofar as it will not be the responsibility of the system to carry out this exchange, except through technological means of realization that are not within the purview of system engineering and the Arcadia model.

This could be justified, for example, if a sensor receives multiple types of information from the actor, such as temperature, pressure, and humidity, which would be grouped into a single behavioral exchange, but this has no significant impact on engineering at this level, in my opinion.

One of the justifications given for behavioral exchanges, wanting to make them appear in scenarios, does not seem very strong to me, because a scenario will not be less understandable if it starts from the sensor rather than the actor - but it will be simpler...

From an implementation/physical point of view, Here, I would not put any physical link, except for exceptions of course. There is often no reality to this type of link and no responsibility of engineering on it either; engineering will not define a physical interface with the atmosphere, which would appear in the ICD for example, simply to have a line between the system and an "atmosphere" actor.

In this case, it is better not to mention the external actor (the atmosphere here) outside of the functional aspect above. Its definition is useful in the operational and system need levels, because it can evoke constraints, choices, etc., but at the physical level, it is hardly useful anymore.

One potential exception (but I'm not even sure about it) could be, for example, a physical link indicating whether we use infrared or visible image acquisition as a medium for image capture.

In summary,

- For simple sensors and measurements, no external actor, especially in [LA] PA.
- For more complex observations and interactions, only the functional aspect, whose exchange alone carries the characteristic of the acquisition conditions.
- No behavioral exchanges or physical links, except for good reasons and demonstration of interest for the intended uses of the model.

6 Engineering Lifecycle Considerations

6.1 In what order to carry out the modeling activities?

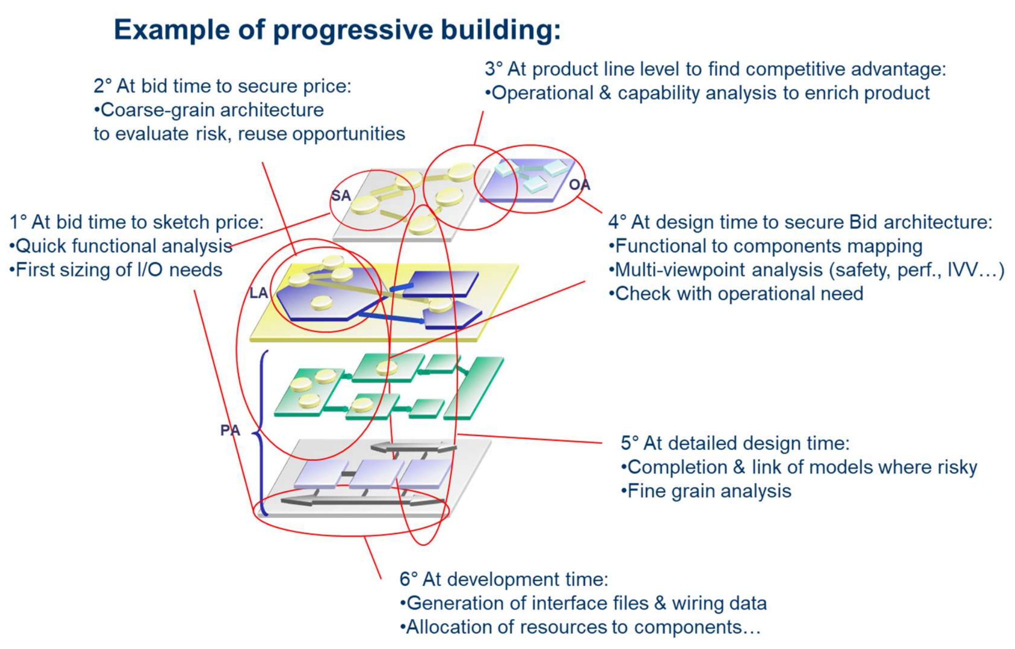
For educational purpose, the elaboration of main Arcadia perspectives is presented as an ordered sequence: Operational Analysis OA, then System Need Analysis SA, then Logical Architecture building LA, Physical Architecture building PA, ending with Building Strategy BS. This reflects the rationale and role of each perspective delivering useful inputs for next ones, whose completion and quality depend strongly on the former.
Similarly, inside each perspective, engineering tasks and activities are also connected by what should be considered as dependency links, but not necessarily time-related ordering of steps & tasks.

In fact, while preserving these dependencies, any process or order can be used to elaborate the definition of the architecture as governed by Arcadia:

- top-down or waterfall approach,
- bottom-up and reuse-driven approaches,
- iterative or incremental processes,
- ...

Furthermore, in real life conditions, iterations and loops are necessary between all engineering activities, yet they are not explicitly represented in Arcadia processes for sake of simplicity.

Consequently, although the workflow may appear to be straightforward, activities may be carried out in a different order; however, for best quality of engineering results, *each activity should not be fully completed without having checked its outcome against its expected entries and dependencies, for consistency*.


Example of progressive building:
2° At bid time to secure price:
•Coarse-grain architecture
to evaluate risk, reuse opportunities
3° At product line level to find competitive advantage:
•Operational & capability analysis to enrich product
SA
OA
1° At bid time to sketch price:
•Quick functional analysis
•First sizing of I/O needs
4° At design time to secure Bid architecture:
•Functional to components mapping
•Multi-viewpoint analysis (safety, perf., IVV…)
•Check with operational need
LA
PA
5° At detailed design time:
•Completion & link of models where risky
•Fine grain analysis
6° At development time:
•Generation of interface files & wiring data
•Allocation of resources to components…

This mainly depends on the process that you want to promote, which itself depends on the specificities of your engineering practices.

For example, when modeling is introduced into an existing engineering process (evolutions of an existing product, significant reuse of already developed elements), it is easier to parallelize the construction of perspectives since the information already exists, and the means of justifying the design are not a priority.
It is just necessary to ensure overall consistency between perspectives and viewpoints, but the existing know-how explains how to do it, and we are rather in a bottom-up approach, particularly to rebuild levels of abstraction and need from a detailed existing product.

Conversely, in the definition of new products or functions or architectures (or parts of them), the construction logic is essential. Here, each perspective must be justified and checked to ensure that it conforms to the expectations of the previous ones, and the dependencies are strong: it is not possible to effectively define the architecture or interfaces without referring to the functional aspects, for example. Furthermore, the verification of needs is essential, and therefore operational analysis is essential, as well as the link with the system need analysis.

6.2 How to iterate, from the OA to the PA, to develop the solution?

The order in which the different Arcadia perspectives are developed is not immutable, and each perspective may have to be reconsidered and evolve throughout the engineering lifecycle. This is particularly true for operational analysis. An example of engineering involving evolving operational concepts or capabilities is as follows:

- Start with an *initial* OA defining the current state of the client's operational processes, organization, etc.
- Build, evaluate, and compare a set of alternative SAs, defining possible contributions of the system under study to this OA
- Define, evaluate, and compare various solution alternatives in LA and PA
- Based on the possibilities and new opportunities offered by the chosen solutions, also consider possible evolutions of the OA to optimize the solution and its use; for example, changing the roles of system operators, simplifying or speeding up operational processes through automation
- Update the SA but also, and above all, the OA, to modify, enrich, or simplify operational processes and entities
- Optionally, draw justification links between the two origin and final OAs.
- The solution provided is therefore the set defined by all the perspectives, including the new OA.

6.3 How to use the OA to describe the solution lifecycle?

The OA (or several specialized or dedicated OAs) can also be used to describe the life cycle of the system itself, including the stages of specification, design, deployment, operations, maintenance and support, evolution, retirement, etc., if necessary, particularly in cases where engineering is involved beyond the delivery of the solution: cyber-security, autonomy, use of learning or big data analytics, for example.

In this case, stakeholders (operational entities, actors: designers, maintainers, etc.) can be added, who have activities (design, IVV testing, analysis of operation data, updating cyber protections, algorithm evolution, etc.), operational processes, information or other needs, successive phases which can be translated into modes, etc. Then, the system can be considered as an object of these activities (which does not mean describing it, which is not recommended in OA!).

Next, in the following perspectives, elements of the system architecture can be related to this description through justification links: for example, a link between a calibration function and the tuning process that requires it at production time, integrated stimulation tools intended for troubleshooting, packaging for transportation, etc.

At the same time, this operational view can also accommodate the description of successive versions and configurations that will need to be delivered to the customer for their "capability roadmap", in the form of user capabilities to be provided over time...

6.4 How to confront an existing system with a new need?

When a product or system exists and a new need arises, it is necessary to verify if the existing product can meet the need, and to determine what changes it must undergo. Arcadia proposes an approach based on the proper use of each perspective:

- Capture the operational need and the expectations of the new client in a dedicated Operational Analysis (OA) and System need Analysis (SA).
- Compare these two perspectives with those that have been defined for the existing product; identify the differences in order to focus the subsequent analysis on them.
- Sketch a principle or notional architecture of the solution adapted to the new need in a dedicated Logical Architecture (LA).
- Compare this LA with that of the existing product; possibly modify it to bring it closer to the existing product; identify the differences in order to focus the subsequent analysis on them.
- Then, compare this LA with the Physical Architecture (PA) of the existing product to analyze the impact of the required changes on it; create a dedicated PA, derived from that of the existing product, on which these changes will be applied.
- Update the OA, SA, and LA accordingly if necessary.

The above confrontation can be done either at the level of a complete model or only in the context of reusing an existing element. In this case, this reusable item (RI) is supposed to be described in a library, as a model element (e.g., for a component, its ports, interfaces, or data, its functional content, its implementation or test scenarios, its non-functional properties such as performance or capabilities, etc.).

At least the following consistencies must be verified:

- Definition of interfaces or exchange items (consistent between RI and the elements related to it).
- Definition of manipulated data.
- Exchanges required by RI and exchanges actually available in the architecture (if an input port of RI is not fed by an exchange, there may be a problem).
- Functional content, implementation scenarios for dynamic behavior definition.

- Consistency of non-functional properties (through multi-point view analysis): performance, consumed resources, domain of application, security, safety, compatibility with system modes and states, etc.

In terms of concrete implementation in the model, several strategies are possible, for instance:

- The RI is directly inserted into the model and connected to its environment; in this case, the verification will probably be done with respect to the functional need, by establishing/verifying the proper traceability links, and then by performing multi-viewpoint analysis.

- Architecture vision is made without inserting RI, and then above consistency is verified with the elements it should replace; here, consistency can be checked directly, maintaining the distinction and marking the modified/reused parts.

6.5 When to stop the design of one level?

Modeling often reveals (but does not cause) ambiguities in the distribution of roles and responsibilities in engineering. These ambiguities are often responsible for teamwork troubles that affect the effectiveness of engineering and the quality of the final product. Clear decisions must be made to avoid such potential problems.

The architects of an engineering level are responsible for engineering choices and decisions, which are translated into information entered into their model, including:

- The allocation of functions to components
- Performance
- Interfaces
- …

This level of engineering is also responsible for the integration/verification of the components it defines and the behaviors it has specified.

It is therefore necessary to ensure a clear boundary between two engineering levels and to specify who is responsible for what, particularly regarding breakdown into components and interface definition.

- If the high-level system engineering (or product engineering PE) cannot finalize the inter-component definition, then it should delegate it rather than impose a breakdown into components; it therefore only transmits functional needs to the hardware or software engineering level (HE/SE), possibly indicating its "preferences" in terms of grouping by first-level functions and associated requirements (for example, specifying needs for modularity, observability, etc.).

Note that delegation does not mean having no right to review and appropriation: the PE can validate the design and use the HE/SE models if necessary, for example, for document generation, etc.

- If the PE chooses to define and therefore impose the inter-component architecture, then they must assume their definition and integration to the end: guarantee the allocation of functions to components (i.e., performance, safety, security, weight, consumption, feasibility – case of an FPGA, for example), freeze the interfaces to their finest level of detail for the HE/SE level, specify the expected dynamic behavior in detail of the interactions between components… so that it is not called into question later; they must also ensure the integration of what they have defined, i.e. integrate the components with each other.

    - In the normal and recommended process, the PE individually outsources the elementary components (leaf components) they define in their physical architecture. Each component description in the PE's physical architecture therefore initializes a system analysis (SA) in the HE/SE model for the hardware or software component, with its environment being represented by external actors.

    - However, there may be different cases: for software or firmware, for example, even if the PE outsources a single leaf component individually, they must sometimes specify the implementation component(s) on which this component will run, for example (ex middleware+operating system+CPU board). This can only be transmitted to the HE/SE at their physical architecture level, since this is where the implementation components appear.

    - Furthermore, and even if it is not recommended, one could imagine a scenario in which the PE wants to define the component breakdown (but then completely, as explained above) but entrusts to the HE/SE all tasks concerning configuration management, production management, unit tests, etc. In this case, the model transmitted by the PE to the HE/SE includes several (behavioral and/or implementation) components; these realization constraints will then be transmitted at the physical architecture level of the HE/SE.
      In this case, since the component breakdown is finalized, it can and should be transmitted to the physical architecture of the HE/SE without any issues (in a multi-level transition from PE model to HE/SE models). This part of the PA model will be read-only; the HE/SE will specify the functions actually performed (still in PA) on these components, refine them, but not challenge their interfaces (except by going back to the PE level of course). The functional need arising from the PE will be transmitted to the SA level (need) of the HE/SE; it will then be necessary, in order to account for the allocation planned by the PE, for example, to create first-level functions grouping those allocated to the same component, in the SA of the HE/SE, and to impose corresponding requirements on them as well as on the components.
      Note that having responsibility does not mean choosing and designing

everything alone: the PE can define the components with the HE and SE if necessary.

6.6 How to collaborate between System and Sub-systems teams?

Arcadia recommends that system engineering team and sub-systems, software and hardware teams work in a collaborative manner, in co-engineering, to elaborate the architecture of the solution.

The place where this co-engineering should take place is the system engineering Physical Architecture. This architecture is defined collaboratively, under responsibility of the system architect, then it is the basis for sub-systems, SW and HW development or acquisition contracts (through automatic transition/generation of sub-systems need analysis SA models, extracted from system PA).

If a modification is to be applied (evolution of customer need, or adjustment of system architecture, or even modification requested by a subsystem team), recommendation is to manage it in the system-level physical architecture, in a collaborative, co-engineering manner (involving system and subsystems teams stakeholders).

Then a new subsystem SA is to be generated accordingly. This subsystem SA is to be considered as read-only by subsystem engineering team; the SS team work is then to confront this new SS need evolution, with the current state of SS architecture design, and make it evolve accordingly.

To summarize, the collaboration workflow may sometimes be bottom-up (e.g. evolution requested by subsystem team, or reuse suggestion), but the model workflow should only be top-down (from system PA model towards subsystem SA model).

6.7 Can we merge two engineering levels in the same model?

There is no fundamental opposition to a model reflecting several levels of engineering, but there are several conditions for this to work, and experience shows that it can be very risky and generally not recommended. Essential conditions to reduce the risks of such operation would be:

- Truly integrated team responsible for only one level of engineering in the eyes of the client, quality, organization (and agreement of the hierarchy on this point)

    - to avoid the difficulty of managing a technical contract between the two levels, the difficulty of formalizing/verifying it

- System and software (or subsystem) scopes close to each other: substantially equivalent functional contents between the two levels, not too many components other than software, simple execution platform viewed from the system...
    - to limit the number of stakeholders in the model with too different concerns and modeling and stopping criteria;
    - to avoid too heterogeneous levels of modeling details that will make scenario and functional chains definition complex, evolution costly,
    - to reduce the complexity of intertwined lifecycle/configuration management,
    - to reduce multi-user constraints, model and diagram size...
- Only one level of documentation, for example (even if there can be an additional level of detail: e.g., SSS, SSDD, software SDD)
    - to avoid complicating the document generation and easily allocate model elements to documents and simplify the document configuration management
- Only one level of IVV, even if it can be progressive,
    - to avoid having to systematically go down to the finest level of detail required by the software to define test procedures, scenarios, and associated FCs;
    - to avoid forcing system IVV teams to go down to the software level to conduct the IVV strategy and locate problems

Note: the few elements I give here are not to be taken literally; they depend on the context and are not exhaustive either.

At a minimum, if we wanted to have truly distinct two levels of engineering in the same model, while separating them (let's say to fix ideas a system or product engineering level and a software engineering level),

- OA/SA/LA would then need to be the responsibility of Systems Engineer – but with only the level of detail required for IS product choices;
- The LA would be both the description of the System engineer solution but also the need of software engineer, *and only that need;*
- While the PA would be the responsibility of software engineering, translating the software solution (at the first level).

But if the previous conditions are not met, we can clearly see the problem: systems engineers need to go all the way down to the physical description for the platform and other subsystems without being polluted by the software detail... And the software has no reason to be disturbed by an interface evolving between two subsystems outside its scope - and yet they are in the same model...

Furthermore, I fully understand the need for co-engineering and also the need for systems engineers to access the details of software engineering design if necessary, to better understand, verify, etc. *But this does not mean one single model*: simply, the models at each level must be developed together, lead to a consensus, and be accessible and navigable by each level of engineering.

6.8 How to manage the transition from System to Software?

The system-software transition takes place at the architecture level, not at the software design level.

The decomposition into components as defined in the system PA (preferably in co-engineering jointly between systems and software engineering) must be respected by the SW, even if it means going through a negotiation process again. Therefore, it must be transmitted in one way or another to the software architecture, just like the data model, as it describes the external interfaces that must be respected. This transition from the system to the software can generally be automated, for example, by code generation (definition of interfaces, containers/components/microservices, assembly and deployment of services or components, etc.).

For the functional part, on the other hand, it cannot be as direct: this functional part expresses the expected need (in the same way as textual requirements), to which the detailed design and software code responds in its own way. Such a function can be found distributed in N methods of P classes, cut into pieces due to threading, use of libraries or reused components, etc.

Therefore, for the functional part, one must just try to trace it if possible in the code or detailed design, in order to refer to it in the definition of versioning, testing, allocation of technical facts, etc. But, in the general case, this traceability is necessarily manual.

Of course, if software engineering chooses to use Arcadia for software architecture design, then the continuity will be total, including for the functional content.

6.9 Can we use Arcadia to work on Software?

Regarding the use of Capella (and therefore Arcadia), it has always been said that it is not intended to be a software design tool: it will not replace either the code or a possible modeling tool such as UML, Matlab, or other, aimed at the almost complete generation of the code itself.

However, there is a need for software architecture engineering on many software developments (especially constrained ones) before embarking on design and code. The absence of such reasoned, justified, and ideally formalized architecture is a proven weakness in several domains.

In many cases, the use of UML can be adapted to this formalization. However, if the major constraints of the architecture require an analysis in viewpoints, and/or if the criteria for defining and choosing this architecture are based on functionality (which is also more frequent than one might think), then Arcadia and Capella are natural candidates.

There are many cases of software architectures defined with Capella. Again, this does not mean software development or even software code design itself (which are outside the scope of Arcadia).

6.10 What kind of document can be produced from the models?

Arcadia favors model-driven engineering and focuses on formalized assets rather than on documents. Outputs of each Arcadia perspective are pieces of models (Operational Analysis, System Need Analysis, Logical and Physical Architectures, sub-systems contracts, building strategy, etc.). The models are the reference, and internally, engineering teams only use them.

However, for various reasons, such as customer requests, certification processes, sub-contractor or customer interactions, traditional documents can be extracted and generated from these models. Typically:

- Operational Concepts documents (OCD) are built from Operational Analysis (for the CONOPS part), and from System Need Analysis (for CONOPS, CONUSE and CONEMP) contents; they may be complemented by physical architecture elements if needed
- System/Segment Specifications (SSS) and Interface Requirement Specification (IRS) documents are built from System Need Analysis
- System/Segment Design Documents (SSDD) and Interface Control Documents (ICD) are built from Logical Architecture (preliminary documents) and Physical Architecture (final documents)
- Sub-systems SSS/IRS, Software Requirement Specifications (SRS) and Hardware Requirement Specifications (HRS) and IRS/ICD are built from System Physical Architecture contents, filtered and restricted to the SS/SW/HW component context
- Test plans, IVV strategy and more are also generated from the models, but are out of scope here.

6.11 How to use models to communicate with the client?

Delivering an engineering model as-is to a client has always proved counterproductive, both due to an excess of information that hinders the client's analysis and due to a lack of adaptation to the client's usage.

Here are some recommendations to facilitate the client's appropriation of the model, if this mode of communication has been chosen with them.

- The model must be reviewed beforehand by the engineering team from the perspective of supplying it to a client.
- The non-confidentiality of the information provided must be validated by management; if necessary, the model must be filtered from what should not be disclosed.
- The status of the model must be recorded in written form with the client: it must not be contractual in any case, and you should be able to change it at any time, etc.
- A reading guide must be developed, with conventions, reading order, etc. (at least in the form of the documentation field of the overview in Capella, which also serves as the entry point for the html version of the model).
- The diagrams presented to the client must be simple, expressive, commented, contextualized; above all, do not use construction diagrams, working diagrams... that are not intelligible to them. The rule should be: no more in a diagram than what you would put on a ppt slide (so zoom in once or twice, then forbid yourself from scrolling the diagram...).
- Training for the client's readers, in both the method and the tool, is almost imperative and must absolutely be offered to them (they are free to refuse).
- The readers must be accompanied in their discovery of the model by someone from the engineering team who knows the model well.

6.12 Model-based Testing (MBT)?

The principle of MBT adopted here is to build a model of the system's *use* (or the solution) as expected by users and clients (we could also speak of a model of usage, of the system's interaction with its environment), while making as few assumptions as possible about the design of the solution itself, in order to remain in the world of needs (with a focus on verification and validation, or even qualification).

The aim is therefore to build a formalized model of the expected behaviors of the system based solely on the specifications (in theory), and then to generate (preferably automatically) test patterns to which the real system will be subjected.

In the case of Arcadia, the method involves building mission-driving scenarios for the system, *independently of the system itself*, from an operational analysis (OA) perspective.

These scenarios are then translated into system interaction *needs* from a system analysis (SA) perspective. These scenarios are natural candidates for initializing test conditions for verification/validation - thus for initializing the test model that will feed the MBT.

However, an aspect often overlooked in MBT approaches is that in order to concretely generate finalized test scenarios (and even more so for automatic generation), it is necessary not only to formalize the expected conditions of use of the system as described in the specifications (OA SA), but also to link them - or even "translate" them - into interactions of the system with its environment *as designed by the engineering* (and not just as specified), otherwise automatic generation is not possible: real external interfaces, sequencing of the dialogue with external systems and operators (which are a product of the design and development of the solution system), etc.

This "translation" or realization of test scenarios is carried out in the test construction approach recommended by Arcadia as follows:

- By translating the specifications (and/or operational analysis) into capabilities, illustrated by functional chains and scenarios described in SA,
- Then by building the solution in LA PA, and transforming the scenarios (and functional chains) from SA to express them in LA PA (with appropriate traceability and justification links).

If we wanted to carry out a complete MBT approach, then for the rest of the approach, the most economical approach would probably be to:

- start with these capabilities, scenarios, and functional chains to initialize an executable model (possibly in a tool dedicated to this use of MBT),
- use this tool to add exploration of the test space (values, abnormal conditions, sequential conditions, etc.), and generate executable scripts on test means. Though, depending on the formalism used by MBT tools, the transition is more or less easy to achieve and often requires some manual user input.

In the current state of the tools, this second part is done manually by building test campaigns and cases based on the previous capabilities in the model, while maintaining the associated traceability.

7 Arcadia comparing to other approaches

7.1 Arcadia Vs Architecture Frameworks?

Architecture Frameworks (AF) were initially mainly issued for customer-oriented concerns: DoDAF or NAF (and TOGAF to some extent) were expected to:

- Help defining the way to acquire and deploy new operational capabilities (including time-related issues, programmatic constraints, staffing…).
- Demonstrate to the customer that their operational needs were clearly understood by the bidding supplier(s), and that their solution would functionally meet these expectations.

See US DoDAF : To enable "the development of architectures to facilitate the ability of Department of Defense (DoD) managers at all levels to make key decisions more effectively through organized information sharing across the Department, Joint Capability Areas (JCAs), Mission, Component, and Program boundaries."

The answer to these expectations is (or should be!) globally described eg in terms of Doctrine, Organization, Training, Materiel, Leadership and education, Personnel, Facilities and Interoperability (US DoD DOTMLPFI). As you can see, the solution (system) is only a limited part of these concerns, mainly in 'Material'.

This is extremely useful to shape customer organization and means, and for the supplier to master customer need and the way its solution should fit in, mainly in a top-down approach. But of course, it is not so much adapted to describing the solution architecture itself in details, and even less to support solution detailed architecture design and justification, along with system engineering structuring.

Can Arcadia be used instead of Architecture frameworks to meet their objectives? No. In contrast with them, Arcadia (as a method) targets structuring, guiding and supporting System, software, hardware [architecture] engineering.

It deals notably with architecture definition, justification, integration and validation, thus from a supplier point of view.

It is much more adapted to this, thanks to unique features such as:

- adaptation to multiple industrial lifecycles: top-down, bottom-up, middle-out, incremental, iterative, legacy reuse or brand new…,
- clear separation and justification of solution vs need,
- strong coherency by design thanks to modelling language and rules,

- support of co-engineering between specialties (safety, security, performance, product line…) and architect,
- support to co-engineering and articulation between multiple levels of engineering (system, sub-systems, hardware and software…)and with integration verification validation teams,
- check and justification of architecture definition by means of multi-viewpoints analysis,
- detailed description of final 'as designed' architecture: interface definition & justification, performance allocation, IVV strategies & releases…

However, Arcadia has included concepts from Architecture Frameworks in its own language: eg operational entities, activities, operational processes, system components, functions… and some views compatible with them: eg OV2, OV4, OV5, OV6, OV7; SOV; SV1, SV2, SV4, SV5, SV10…

This has been done for several reasons:

- to drive architecture justification against operational need
- to initialize system need description starting from Architecture Frameworks, and ensure easy coupling with models supporting AF as an input
- to encourage people not used to operational need analysis, to consider this activity, even if they are not using Architecture Frameworks, thus having a simple but limited means to justify their architecture from operational need description
- to deliver to customer familiar with AF, description of Arcadia models and analyses outputs under a formalism close to AF views.

So if you have a significant work to be done in 'architecting' phase, with your customer (or substituting to him), then you should consider using Architecture Frameworks, then reusing these models in Arcadia and Capella as [requirement] inputs for system engineering (to design, develop, integrate and verify the solution). Traceability and justification links should be maintained between AF models and Arcadia models accordingly.

If you have limited freedom of action in architecting phase or in customer analysis, then you should at least fill Arcadia Operational Analysis (OA) and System Need Analysis (SA) phases, even if you cannot perform an extensive architecting analysis using AF.

7.2 Arcadia and ISO 15288-2015?

I made a (maybe too?) quick analysis of 15288, to see the points of intersection with Arcadia.

Obviously, Arcadia only covers a small part of each process, being focused on engineering. But when I note the elements presented in the standard that I also find in Arcadia as concerns and recommendations, I still find quite a lot of things (which is good news ☺) .

Being well aware that we are talking about a very partial coverage, I find the following elements:

- Business & mission analysis :
  - The Arcadia OA contributes to it to some extent, I think - still from the engineering perspective only, I won't repeat it ;-) - on the stakeholder/activity analysis, required operational capabilities, and also "Operational concepts include high-level operational modes or states, operational scenarios, potential use cases," for example.

    A beginning of SA or even LA would be possible in B&M Analysis if an organic projection, for example, was needed (different sites, etc.), but most often it will be limited to a multi-criteria or causal analysis (e.g. CID causal influence diagrams), for an initial exploration of the solution space.

    By the way, in Arcadia, the OA does not position itself on system capabilities and interaction with the system but before: the system should not appear in the OA. The approach seems to be fully in line with the 15288 presentation.
- Stakeholder needs and requirements definition :
  - This is the first part of the Arcadia SA perspective, analysis of the system need in customer requirements analysis and associated negotiation. It is centered on the system, "express the intended interaction the system will have with its operational environment," seen from the customer. A reference to the OA (traceability with that of the B&M analysis, or even a new dedicated OA, e.g., to translate part of the CONOPS CONEMP CONUSE) could be justified.
- System requirements definition :
  - This is the second part of the Arcadia SA perspective, which finalizes the system need from the customer need and derives system requirements.
- Architecture definition :
  - Here, a small clarification is needed: for me, this activity is done in continuity (which does not mean that there is no break in the nature of the description, detail, what constitutes it ...) - it starts from very broad, unformalized, unstructured, imprecise, and undetailed considerations but "generative" (of ideas, alternatives ...), and will bring out possibilities of solutions, initially only evocative of its possible orientations, then increasingly precise in details details as the field of possibilities narrows down.

I am well aware that Arcadia only covers the end of this cycle (if there is an end), and moreover, not over the entire scope, far from it. It still helps, from experience, to raise engineering and architectural questions, to consider alternatives, and to make evaluation and choice criteria concrete; but again, this is only on a part of this reflection. So for me, it clearly contributes, but within these limits, to this architecture definition activity.

  - The LA, and to some extent also the PA, that carry this contribution.

- Design definition :
  - This is the actual domain of the Arcadia PA, more precisely the use of this PA to build component or subsystem contracts and the IVV strategy.
- System analysis :
  - I see it as transverse to all the others, it is an activity of "verification" or evaluation ("assessment") on what has been developed up to a certain point. For engineering, this concerns both the verification of understanding, coherence, completeness of the need, the validity of the solution with respect to it, the validity of that same solution with respect to various points of view, the validity of the integration strategy or tests, etc., and also the comparison of the merits of architectures.

    So Arcadia's contribution is probably in the multi-viewpoint analysis and impact analysis in general.

Note :

In the reality of a complex project, all of this needs to be put into perspective in terms of roles and responsibilities: business & mission analysis will often be done under the auspices of the client, using architecture frameworks, for example (or via OA SA Arcadia at worst in a preliminary version oriented towards clients, if it is the supplier who does it and wants to preserve the continuity of the modeling). The stakeholders' needs could be partly constituted by the associated solution views (MSV, NSOV more or less), or a preliminary Arcadia SA. The system need analysis would either be the same final SA or another SA traced with respect to the first.

7.3 Why did Thales embark on the development of a new method and tool?

Before conceiving Arcadia and developing the associated modeling tool, Capella, Thales deployed operational approaches based on NAF/DoDAF and SysML-based commercial tools, without success.

In these projects, the feedback revealed numerous shortcomings or incapabilities of both Architeture Frameworks- and SysML-based approaches and tools available in the market in defining the solution architecture.

Why not using Architecture frameworks?

Could "Architecture Framework" languages (such as the NATO AF (NAF)) and supporting modeling tools, be used for system and architecture engineering? What does Arcadia bring as compared to them?

As demonstrated by experience of real projects, architecture frameworks are not so well adapted to describing the architecture of the solution itself in detail, let alone designing the solution architecture and providing detailed justification, driving IVV, product line and more. For example:

- There is no separation between need and solution (equivalent to the SSS / SSDD separation).
- The consideration of engineering specialties and non-functional aspects is not natively integrated.
- The articulation of different levels of engineering and support for IVV are missing.
- The control of the size and complexity of the model is limited.
- Moreover, there is currently no precise method supporting this approach targeting the engineering domain, since it was not originally intended for it.

In contrast to this final customer need, Arcadia (as a tool-based method) aims to define and verify system, software, and hardware architectures, as well as associated testing means.

This method is based on the Capella modeling tool, which takes into account the complexity constraints of current projects.

Arcadia deals with architecture definition, its justification in relation to needs and constraints, particularly non-functional and industrial ones, integration, and validation, from the supplier's point of view ("design, develop, integrate, and verify the solution").
It is much more suited to the overall design of the system, thanks to unique features such as:

- Adaptation to several industrial lifecycles:
    - Top-down for new systems
    - Bottom-up for existing reuse
    - Composite (middle-out) for model evolutions
    - Incremental and iterative

    - Product line...

- Support for cooperation between engineering specialties (security, safety, performance, product policy...) and the architect
- Support for co-engineering and articulation between several levels of engineering (systems, subsystems, hardware, and software...)
- Control and justification of the architecture through analysis from multiple viewpoints
- Detailed description of the finalized architecture: definition and justification of interfaces up to generation, resource allocation, and performance, IVV strategies and optimization...

However, Arcadia still integrates some concepts from architecture frameworks, such as operational entities, activities, operational processes, system components, functions, and some views (OV, SV) compatible with them.

Why not using SysML, then?

Regarding SysML, weaknesses raised by projects were notably:

- In terms of modeling approach:
    - No operational analysis (thus no reference to missions, contexts, ops constraints, etc.)
    - No traceability and compatibility with architecture frameworks (NAF & co)
    - Functional analysis is often not core part of the approach
    - No clear and justified definition/management of interfaces, of implementation
    - Limited behavioral and non-functional description (no functional chains, no supervision/states-and-modes engineering)
    - No support for product line construction, system/subsystem transition, IVVQ
    - …
- In terms of language, inheriting the complexity and limitations of UML: for example,
    - Complexity of concepts and their articulation (no unified and minimal metamodel for the connectivity between concepts)
    - Inadequacy for non-top-down approaches (not based on hierarchical decomposition), thus difficult to manage Reuse, collaborative definition, model evolution
    - Difficulties in scaling large models (no multi-level synthesis capabilities)

- ○ Class/type-based approach (unnatural for system engineers) rather than replicable instances (closer to practice)
- ○ ...

Of course, some of these limitations are also present in the tools (particularly the ability to manage complex models collaboratively and the complexity of architecture diagrams). Most of these limitations are still present in current tools.

Moreover, most of these approaches and tools provided little assistance in managing the complexity of engineering as a whole, in particular:

- Collaborative model-building approach
- Management of large models (several thousand to tens of thousands of main elements), their maintenance and evolution
- Different intertwined lifecycles in the same project
- And above all, support for engineering in all its end-to-end global processes, not just the capture of the system definition.

This is why Thales decided in 2007 to invest (significantly and for an extended period!) in the development of a tooled method, with the objectives of:

- Guiding each major engineering activity by framing the work with a detailed method
- Covering all phases of engineering (from operational analysis to IVV)
- Supporting and facilitating collaboration between disciplines (architects, requirement analysts, specialty engineering, IVV, detailed interfaces...)
- Ensuring continuity and consistency throughout the depth of engineering (from the complete system to software and hardware architectures)
- Adapting to most real-life lifecycles (new, reuse, legacy, product, bottom-up, evolution, incremental...)

7.4 Summary: What is Arcadia’s value-added compared to existing modeling languages and tools?

Here is a quick summary of Arcadia features and benefits developed in the former Questions and Answers.

- More than just a language or environment for describing a system, Arcadia is a method supporting collaboration between stakeholders in "end-to-end" engineering.
    - Covering the entire engineering cycle, from "Business/Mission Analysis" technical Process up to Verification & Validation Technical Process (ISO 15288)
    - Implementing multiple and intertwined engineering levels (system/sub-system, specialties & disciplines)
    - Supporting various engineering approaches (Reuse, Bottom/Up approach, collaborative, Product Baseline…)
    - On board support & practice recommendations helping system engineers in the unfolding of their activities

- The Arcadia language aims at supporting engineering and the collaboration between its actors, focusing on their needs
    - Language adapted to system engineering & architecture definition
    - Considered as simple and natural by system engineering stakeholders
    - Close to SYSMLV2 core concepts(under definition)
    - Covering from Operational Analysis up to Product Breakdown Structure (PBS)
    - Ensuring consistency by construction between all sets of data (interfaces, Modes & States, Configurations, Functional, Resources,…)
    - Adapted to model complexity, collaboration, and various model lifecycles
    - NAF standard formalism compatible (to some extent)

- Arcadia's operational analysis (like NAF-type architecture frameworks) is not limited to the system environment: it is a view with a different level of abstraction that does not presuppose the limits of the system, but only deals with missions, goals, objectives, issues, activities... of future users of the system and their environment. This view is essential to better control customer needs, to emerge new concepts, and to analyze human factors, for example (otherwise, the operator's task analysis and the operator/system allocation issue are omitted).
  The view of the system environment also exists in Arcadia, but in SA.

- Arcadia explicitly separates the expression of the need (constituted by OA and SA) and the description of the solution (LA PA); this is quite natural (as is done by separating SSS and SSDD, for example) and allows in particular existing solutions to be compared with a new need, which is otherwise not possible, and to manage different lifecycles of needs and solutions, and different visibilities.

- Arcadia manages complexity using various abstraction views that separate levels of complexity and detail, while allowing impact analysis. For example, the logical architecture LA allows both a higher-level view of the solution, independence from technological choices, and a place for reconciliation between existing (reuse) and desired architecture. Similarly, functional and organic (or structural) views coexist without mixing at each level of abstraction, including for interface justification (structural) by the functional view. The articulation and separation of engineering levels by models deduced from each other preserves the lifecycles specific to each level, confines complexity while ensuring support for collaboration and overall coherence.

- Arcadia's physical architecture allows for precise and complete definition/justification of interfaces, crucial in engineering: in particular, for example, it distinguishes between interactions (flows, energy, data, couples...) and vehicles of these interactions (pipes, conductors, data links, axes of rotation...). It is also the functional view that defines and justifies the interfaces - hence the importance of being able to describe it at different levels (need, notional solution, final solution).

- To fully take advantage of the previous possibilities, dedicated tooling for implementing the method is essential; this is the purpose of Capella:
  - Assisting in engineering MBSE set up and in applying Arcadia.
  - Easy to start up, concepts & diagramming easily understandable by system engineering community.
  - Large Tooling scope to master model complexity & diagrams & various lifecycles.
  - Industrial & Academic References (Rolls Royce, Framatome, Virgin Hyperloop, Continental, Ariane group, Safran, Bombardier, COMAC…) in several countries (Europe, China, India, Japan, Brazil, USA, Canada…).
  - Open Source Product, free for local license, moderate costs for team license (SME company).
  - THALES and others able to provide operational support, including training

In general, Arcadia favors an approach that assists and secures end-to-end engineering, unifying engineering practices, products, and collaboration conditions, rather than a smaller common denominator type of language/tool that does not ensure success in collaboration and leaves it up to each individual to build their approach.