# Systems Engineering Handbook

Space Systems Engineering 2024/25
CADE20004

Dr Josh Hoole, Lecturer in Systems Engineering,
School of Civil, Aerospace and Design Engineering (CADE)
University of Bristol

V1.0 July 2024


**Executive Summary**

This document provides a handbook to support the CADE20004 Space Systems Engineering course at the University of Bristol. This handbook introduces topics related to Systems Engineering, including:

- The need for systems engineering
- The V-model of system development
- Guidance for writing engineering requirements
- Verification and validation activities
- Mission and functional analysis
- Engineering budgets and margins
- System architecture definition
- System interfaces
- Handling engineering data
- System safety and reliability

# Contents

# Introduction

Welcome to the Systems Engineering element of the Space SYSTEMS Engineering unit. It's very likely you've experienced Systems Engineering approaches without even knowing it, so it's my job to help you understand the techniques available to us engineers to carry out systems engineering.

But hold on. Just *what* is systems engineering? In fact, what even is a *system*? Let us turn to NASA to guide the way:

**"A '*System*' is the combination of elements that function together to produce the capability required to meet a need"** [1].

Surely, this is the heart and soul of engineering, we fix a problem by defining a solution, which tends to be some form of engineering artefact. But you'll note the definition goes further, it highlights that we are *combining* elements to *function* together. You'll all be familiar with the interconnected nature of the modern world and how this occurs in both physical and digital formats. If we study any engineering artefact, from the most exciting aircraft to the humble electric kettle, we will see that *everything* is made up of individual elements at a smaller scale working together to solve a problem.

But if *everything* could count as a system, why are we learning systems engineering in the context of Space Engineering? Well, many of the techniques we will uncover were born out of the Space Industry during the 1950s to tackle the complexity of engineering systems required to go boldly where no humans had ever set foot. I'm really keen however that you do not think of Systems Engineering as just related to spacecraft. It spreads throughout Aerospace, Engineering and our everyday lives.

Talking of *everything* being a system, the very content we will use to learn about Systems Engineering is in fact, a system. This handbook provides a quick reference guide for the techniques we will cover, but you will find it has full utility when used *together* with the context-setting short 'spark' and cases study asynchronous videos and from your reflections when we 'live' systems engineering together in our lectorial sessions.

As you go through this course, you're probably thinking "*where are the powerpoint slides?*". Whilst there will be some for the live sessions, the pre-reading material feels far more natural as a handbook. In addition, Systems Engineering concepts are usually detailed for practicing engineers in handbooks, such as the NASA Systems Engineering handbook [1] and the INCOSE (International Council on Systems Engineering) Handbook [2]. Please see the document in front of you now as a 'LITE' version of a full Systems Engineering handbook and I heartily encourage you to explore the NASA and INCOSE handbooks in your educational and professional careers (the NASA one is free online!).

Returning to the systems engineering content as a system *itself*, the techniques you will experience will be useful outside of the unit. You are about to embark on the Design Build Test activity in AVDASI2, where you will experience first-hand systems engineering in action and we will be reflecting on this as we make our way through Space Systems Engineering. But please do not stop thinking about systems engineering at the end of Space Systems Engineering and AVDASI2. It is a core part of the engineering mindset that I hope you take forward into your future careers.

I need to address one elephant in the room. Having been part of the team who guided you through technical and laboratory report writing last year in Engineering by Investigation, you will note that I will continually break the rules. This is to try to keep the written content fun, insightful and human, but please do not copy this style in your technical assessments and reports!

In my ramblings above, I've asked you to reflect a lot during the Space Systems Engineering unit and therefore I shall close this introduction with a reflection of my own. Many of us get into engineering because we like fancy technical solutions (*me*), hard maths (*not me*) or are continually amazed at what humans can do when working together (*definitely me*). Systems Engineering forces us to stop, reflect and step away from the hard maths and not be distracted by shiny gadgets to ensure we understand the problem we are facing clearly and then combine the right elements to tackle that problem.

I really look forward to going on this journey with you all to help shape how we think about complex engineering!

***Dr Josh Hoole***

# 1 The what, why and how of Systems Engineering - *How do us humans do complex things?*

From the written introduction and introductory video where we decomposed the Mars Sample Return rover *Perseverance*, you'll have begun to notice that systems engineering focuses alot on decomposing complex systems into individual elements (that we could keep on breaking down to deeper and deeper levels, down to individual washers, capacitors, nuts, bolts, etc.). In this section we will briefly look at how we represent engineering systems in this way, justify why we might want to do this and then show how over a system lifecycle we *break apart* our engineering problem and *build up* our engineering solution.

## 1.1 *What?*: the systems model and systems engineering

As engineers, we naturally gravitate towards a hardware perspective of engineering systems (i.e. the 'thing' we can physically see). However, the secret to complex engineering systems is to view them from different *abstractions* or perspectives. We can convert *any* engineering artefact into a 'black-box' model similar to the one shown on the left in Figure 1.

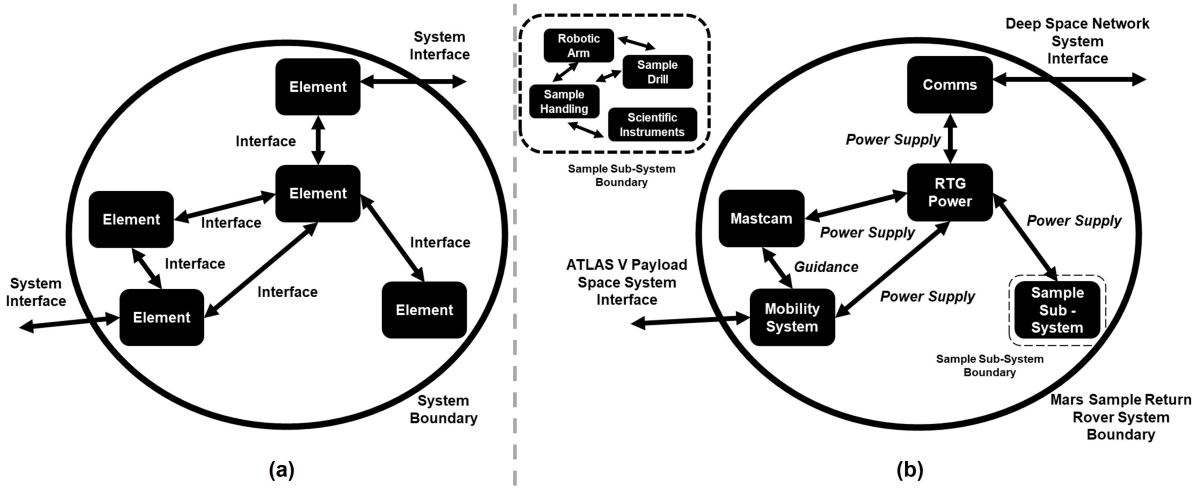



Figure 1: The system model in a) abstract form and b) loosely based on the Perseverance rover

Within the model in Figure 1, you can see that we are joining together various sub-***elements*** with links known as ***interfaces***. Each of the presented elements will have a function or a purpose within our system and may need to work with other elements to achieve the overall system function. Regarding the Perseverance Rover, we saw in the introductory video how we could break the rover into, say, the sample collection system, mastcam, Radioisotope Thermoelectric Generator (RTG) and rover mobility system and this is shown in the right in Figure 1.

You'll also see we have a ***system boundary***. Everything within the boundary is *our* system that we care about and everything outside of the boundary is someone else's problem if we are being lazy. In reality, our system will always need to interact with the world and other engineering systems that co-exist alongside it, so we also have ***system interfaces*** to enable us to do so. For example, the last two martian rovers were launched using Atlas V rockets so had to be designed with this interface (e.g. mass and geometric envelope) in mind.

The system boundary shows what we can influence within it, and the items we are unlikely to be able to change outside of it, so system interfaces are often known as ***constraints***. However, it also gathers together the elements we require to achieve a system function. For example, the Perseverance rover has the function of collecting samples on the Martian surface and hence needs drilling, sample storage, power, mobility, etc. to do this and is therefore within the system boundary. Finally, Figure 1b shows how we view the system model at different levels of sub-system detail, just as we observed in the first 'spark' video for Perseverance's drilling arm.

## 1.2 *Why?*: the need for systems engineering

To put it plainly, modern engineering systems are just too complex for one person to keep all of the detail and understanding of how they work in their head. The detail required to produce safe and efficient aerospace systems is immense and therefore we must share the cognitive load of knowing the intricacies of fluid flow across wings, detailed flight control software programming, understanding the finer points of material science, etc. This is further compounded by the fact that the delivery of such systems requires all of the efforts of thousands of individuals working across capabilities, companies, countries, cultures and continents to be brought together...*and yet we can produce reliable engineering systems of exceptional performance*.

A great visualiser of this complexity is shown in Figure 2, where it is shown that the number 'things' we need to consider and care about 'explodes' the more detailed into the system design we go. Imagine if I continued to disassemble the model rover into the individual 672 plastic building block pieces and then have to remember the orientation and placement of each block - it is not possible. The other challenge Figure 2 infers is that any decision we make at the start will propagate through to millions of individual components, and the decision we make is often based on a large number of factors outside of our control and assumptions when we are at the early stage of a system design.

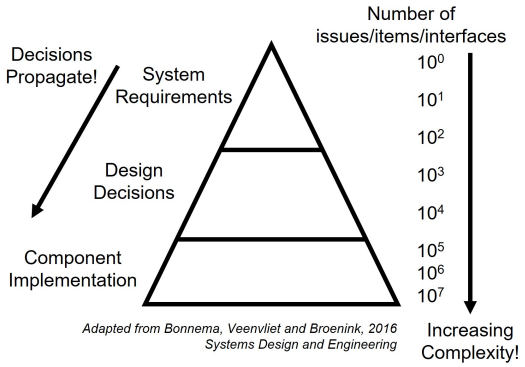



Figure 2: The 'Muller' pyramid showing the increase in complexity as a project progresses, after [3]

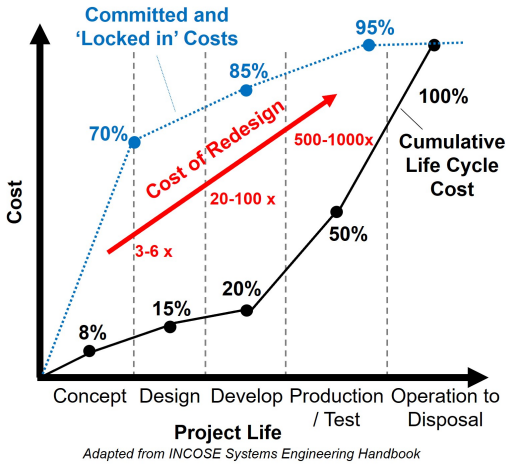



Figure 3: Costs throughout the project lifecycle, after [2]

On the subject of making decisions, Figure 3 shows the cost impacts of making a wrong decision. In large engineering projects the early decisions we make can 'freeze' ≈70% of the total system cost and if we find errors at the test phase of a system their correction can cost up to 1000 times what it would have done if we made a correction earlier. What mistakes can occur to lead to these massive costs? [4]:

- **Faulty requirements:** - *we designed the wrong thing*
- **Interface errors:** - *your thing doesn't fit with another thing*
- **Inconsistent assumptions:** - *you've designed something believing the wrong thing*
- **Production problems:** - *you can't build the thing*

Humanity is also often the problem. Large projects will have lots of competing stakeholders who want their own interest put first. Engineers can also make this worse by being technically and solution driven and we often create problems to fit the fancy solutions we define (innovation for innovation's sake). Alternatively, we can get very *comfortable* in what we do and refuse to innovate.

Systems Engineering helps combat all of the above. It meters the design process, makes us understand the problem first and stops us diving in too quickly. It also provides us the techniques and ways of thinking to manage the thousands of interfaces, parameters and elements we have to care about in engineering systems. This is why I really like the NASA definition of Systems Engineering: "*Systems Engineering* is defined as a methodical, multi-disciplinary approach for the design, realization, technical management, operations and retirement of a system" [1].

### 1.3 *How?*: the V-model for system development

The other element I like about the NASA definition for systems engineering is that it infers that all systems have a lifecycle, from the first observation of the problem to solve, through the retirement of a system. Many systems engineering approaches are based around following the system throughout this lifecycle, and one such approach is know as the V-model, shown in Figure 4 overleaf. Don't panic as it looks a lot to take in, we will walk through a real-life V-model example in the case study video, all this handbook intends to do is show you the key points of the V-model:

1) **Lifecycle:** All projects start with a ***need***, a problem we want to solve. We carefully evaluate the problem to identify ***what*** our system must achieve, before we carry out any design work. We then move onto what ***functions*** the system will need to perform and only then can we start to design the system. Design is the process of identifying ***how*** the system will achieve the functions. Once the system has been designed and implemented, we can begin to test the system to see if it achieves what we wanted and solves our initial problem. If we pass this phase, we then operate the system until its retirement and disposal.

2) **Problem Break-Down and System Build-Up:** By starting from a ***need***, we can then break this need down into ***requirements*** that detail what the system must achieve. These are then decomposed and distributed to specific sub-systems and ultimately components as we move through the design process. This involves asking "*What do we need to build?*" at ever deeper levels of detail. As we implement/construct the system, we then go from the component level and integrate components into sub-systems and the sub-systems in the full system, testing that we are meeting the requirements at every level. This build-up phase asks the question "*Have we built what we needed?*".

3) **Iteration:** All engineering requires iteration and feedback, leading to a non-linear path through system development. You can see that there is iterative feedback between each stage of the V-model. When defining and decomposing the system, we find that investigating the problem helps us define the solution, but a defined solution often leads to more questions about the problem, so we iterate. When implementing the system, we often find that testing highlights elements of the system that need to be modified, after which we re-test the system.

There is also another **very important** set of attributes **missing** from the V-model shown in Figure 4 that we will consider once we have introduced engineering requirements in Section 2.

It seems appropriate to mark the end of this section by thinking about the end of a system's life, which historically engineers are very bad at doing. Remember, the systems we design and implement can outlive us, the companies we work for or even the industrial sectors and their safe and clean disposal can be a real challenge, especially given the sustainable future we all rely on. In your systems engineering future, please always try to think about what will happen to the system when we no longer require its services...

## Section Summary

Systems Engineering is the mindset and techniques we use to develop complex engineering solutions in response to a need:

**WHAT?** - Systems Engineering is the process of breaking-down a problem and viewing our system design from different perspectives as we build-up the solution

**WHY?** - Modern engineering is technically and organisationally complex and the cost of finding mistakes late can be 1,000 times the cost of removing errors early

**HOW?** - Follow the V-model for system design, again its a case of breaking-down the problem and building-up the solution

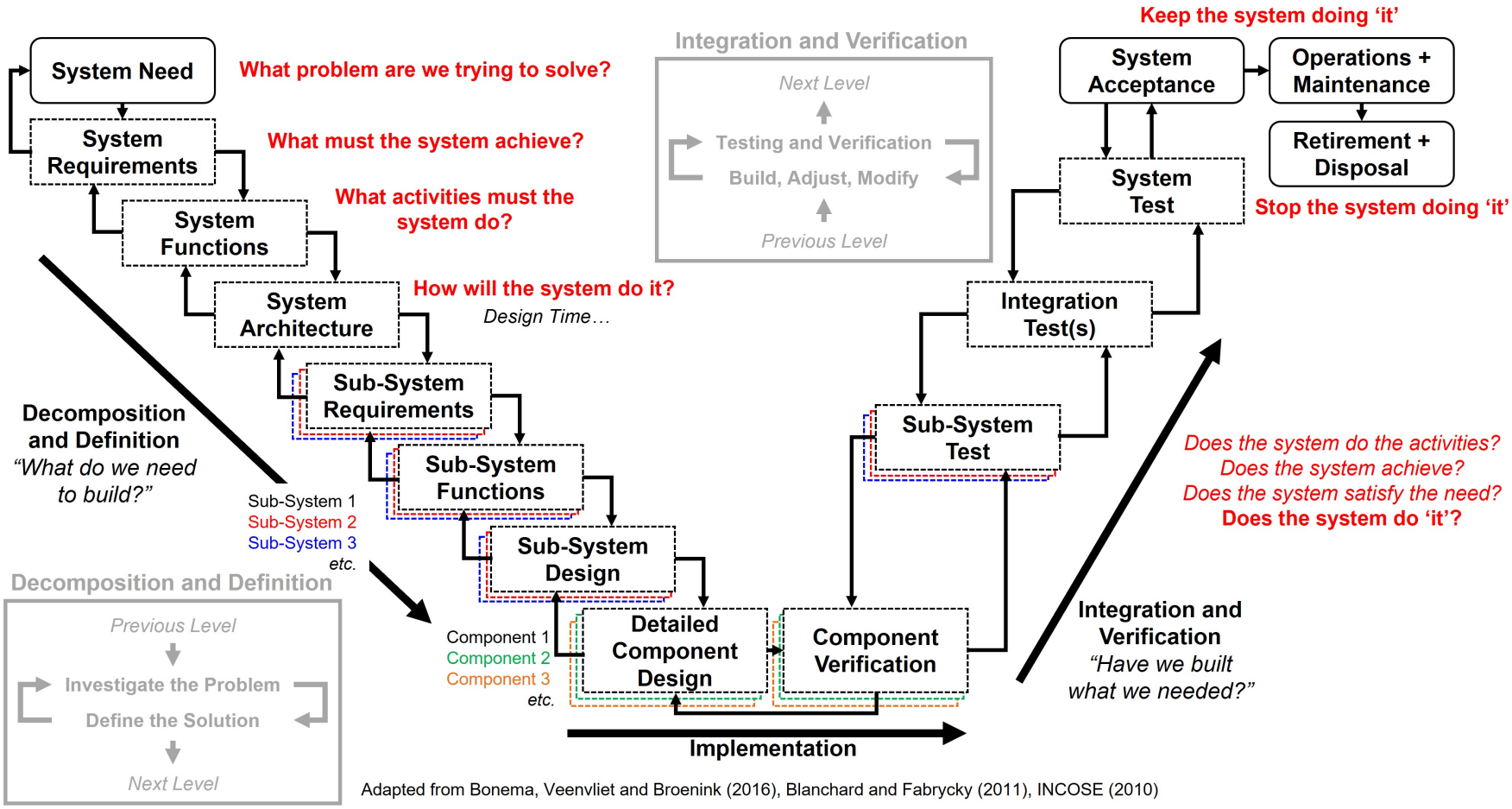



Figure 4: An overview of the V-model for systems engineering, adapted from [2, 3, 5]

# 2 From System Need to System Requirements - *How do we know what our system must ACHIEVE?*

Allow me to start this section with a quote from the Engineering Council (the regulatory body that defines the standards for accreditation of engineering degrees in the U.K.): "*Engineers and Technicians are concerned with the art and practice of changing our world. Responding to the **needs** of society and business, they solve complex challenges and in doing so enhance, welfare, health and safety whilst paying due regard to the environment.*" [6]. A ***need*** is a problem we want solve, and we develop systems to achieve that.

So what specifically is a need? Here are three interpretations, but you'll note they all focus on understanding the problem before we define any form of solution:

- "A single statement that drives everything else. It should relate to the problem that the system is supposed to solve but not be the solution." [1].
- "Needs are defined in the answer to the question, *'what problem are we trying to solve?' [1].*
- "A need...can generally be captured in a single short paragraph that begins with the words '*I want...*'...It will tend to be a qualitative statement..." [4].

So if we generalise the above, a *need* is asking the question: **What is the System trying to solve, satisfy or fix?** A need can come form a multitude of sources including customer requests, internal future project departments, market opportunities, technological advances, replacement of deficient systems, regulatory and political pressure or combinations of all those listed.

However, the statement of a need will often be at an exceptional high and 'fuzzy/woolly' ill-defined level. The remainder of this section will explore how we convert this often general statement into a set of technical requirements that we can design our system in response to.

## 2.1 Requirements and why they are important

In order to demonstrate how we as engineers add detail to the system need definition, we'll again use the Curiosity Rover, Perseverance Rover and Mars Sample Return mission as shown in Figure 5. The need for all of these systems is to answer the question "Is there life now, or was there ever life on Mars?" [7]. In fact, we can even back step this need to humanity's continual quest to find the origins of life.

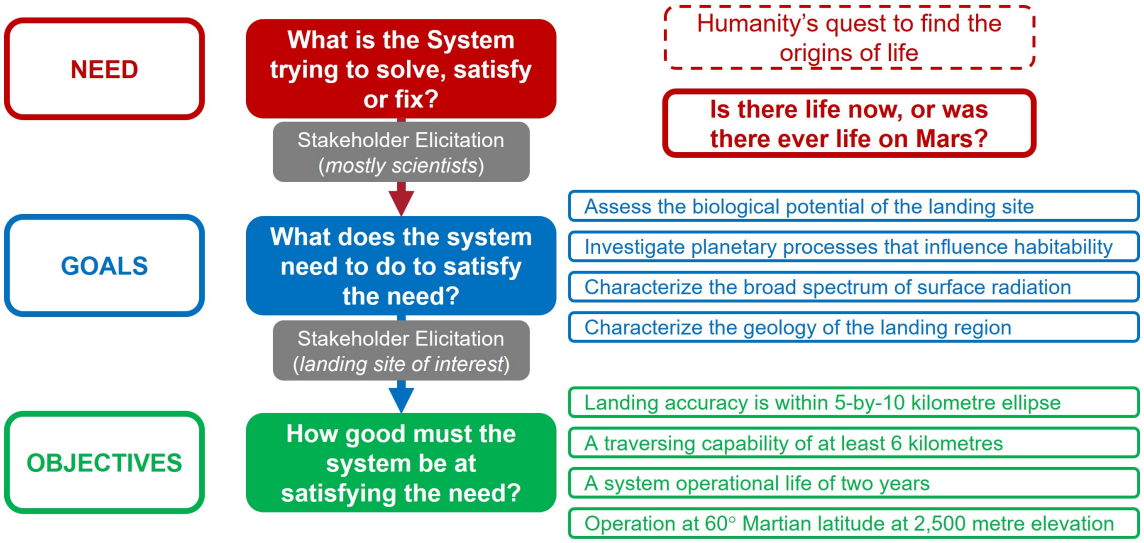



Figure 5: The evolution of a system need to a set of system objectives, using the Curiosity rover as an example [7]

The next stage is to define ***goals*** for the system. Now, the definition of goals aims to answer the question of "*What does the system need to **do** to satisfy the need?* [1]. Note how this is purely functional, its only about the steps the system must carry out, without any detail of how it will perform them. From Figure 5 you can see that there are now qualitative functional statements of what the systems must do.

As most of us engineers are solution-orientated, we can also see the limit in just specifying a goal. We need to add another level of detail. This next level are known as system ***objectives*** and these capture 'how good' the system must be at satisfying the need [1]. We can therefore see in Figure 5 that we can start putting design targets and values to what the system must achieve. The important thing to highlight is that any objectives can be followed all the way back to the system need - they are ***traceable***.

You'll note that I've glossed over how we convert a general need into specific objectives, but that is because it is the hard and time-consuming part of this element of the system lifecycle. In fact, the Mars Science Laboratory and Curiosity programme took two years to go from need to objectives (and that's with teams and expertise from designing previous Mars rovers) [7, 8]. Data from The Planetary Society suggests that somewhere between $100 Million USD and $300 Million USD (inflation adjusted) had been spent during the journey from needs to objectives at the Mission Concept Review in October 2003 [9]. This represents about 3% to 7% of the total system cost [9] and therefore aligns well with the system lifecylce costs shown previously in Figure 3 in Section 1.2.

To jump between need to goals and goals to objectives we have to perform ***stakeholder elicitation***. This is a fancy way to continually engaging with all those with a legitimate interest in the system, which can be considered simply asking 'why?' everytime a stakeholder states something they want from the system. In the case of Martian exploration, engineers would have engaged with scientists to identify what they needed from the mission to answer the need in order to define the functional goals of the system. Further stakeholder elicitation would have defined the landing site of interest on Mars, which would have then led to the objectives shown in Figure 5. The complete definition of objectives will require stakeholder engagement, concept studies, analysis, leveraging of your industry's subject matter experts and a whole amount of work. Perhaps you can see why the V-Model shown previously in Figure 4 is iterative at the need and system requirement loops. We will learn a lot about our system need from the action of developing system goals and objectives.

System objectives are also commonly called system ***requirements***. The INCOSE definition of a requirement is "A '*Requirement*' is a statement that identifies a system characteristic or constraint which is...deemed necessary for stakeholder acceptability" [2]. Our system characteristic is an objective and stakeholder acceptability captures the system need and objectives. I've removed the middle of the definition as we will come back to this aspect of requirements later in Section 2.2.

To summarise, a requirement therefore states something that the system must do and how well it must do it. Ultimately as engineers they provide the contractual basis of the system and therefore are how we either get paid for delivering the system or sued for not delivering the system. However, Figure 6 shows an infographic that details how requirements are actually the lifeblood of the systems engineering process. You can see from Figure 6 how requirements feature at the definition, integration and testing phases of our system.

On real engineering programmes the number of requirements related to a system run into the thousands in number, and we'll have a look at some real engineering requirement documents in the live session. An example of a written requirement taken directly from the NASA systems engineering handbook is as follows [1]:

> **The Thrust Vector Controller shall gimbal the engine a maximum of 9 degrees, $\pm$ 0.1 degrees**

From this, you can see that a written requirement has the following parts:

1. the system or sub-system element that the requirement is placed upon (e.g. thrust vector controller).
2. a function or action that must occur (e.g. gimbal) - this represents a 'goal'.
3. a quantification of how much the function must be achieved (e.g. $9^\circ \pm 0.1^\circ$) - which represents an 'objective'.

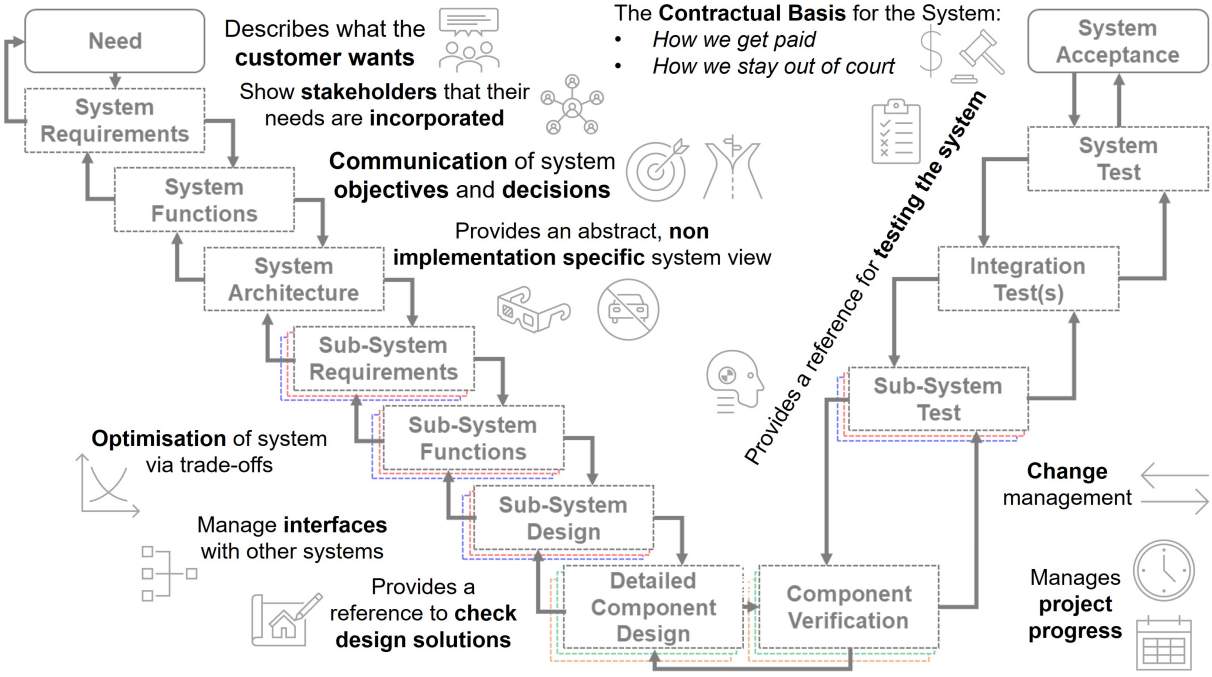



Figure 6: The continual use of requirements throughout the system lifecycle

## 2.2 Writing "Good" Requirements - *the Requirements of Requirements*

Whilst requirements can appear simple at a first glance, writing good requirements that can be used when designing a system is challenging. However, I'm a lover of all things meta, so allow me to present to you the "**The Requirements of Written Requirements**" in the figure shown below:

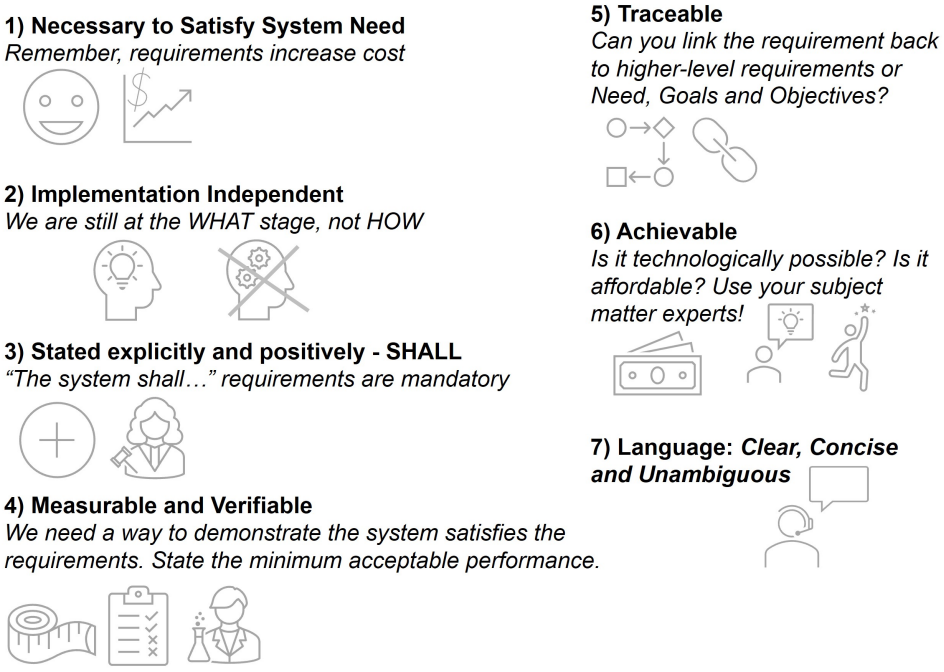



Figure 7: The requirements of writing engineering requirements

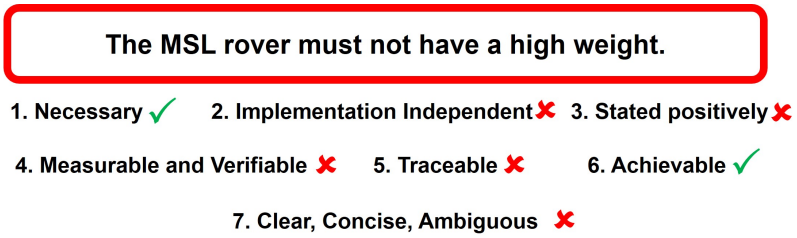
***Shall*** is a five letter word that carries a huge amount of importance in requirements. 'Shall' means that whatever is included in the requirement text is mandatory and the system must satisfy the requirement. 'Shall' is often confused with:

- *Will* - this is a future event, not a mandatory requirement.
- *Must* - this is a strong customer desire, not a mandatory requirement.
- *Should* - this is a customer desire and *guess what?* It is not a mandatory requirement.

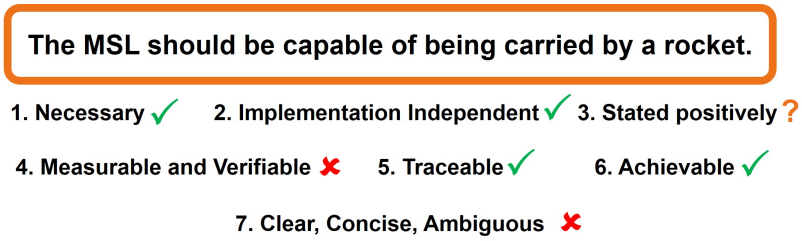
Figure 7 also states that the language within the requirement should be clear, concise and unambiguous. Now not many of us become engineers because we like writing, so here are some elements to **avoid** when writing requirements:

1. using *to be, is to be, are to be, should* and *should be*
2. superlatives (e.g *best* and *most*)
3. comparative phases (e.g. *better than*)
4. loopholes (e.g.*if possible* and *if applicable*)
5. subjective language (e.g. *user friendly*)
6. ambiguous adverbs (e.g. *almost always* and *real-time*)
7. open-ended and non-verifiable terms (e.g. *provide support* and *as a minium*)
8. indefinites (e.g. *To Be Determined* (TBD) and *To Be Confirmed* (TBC))

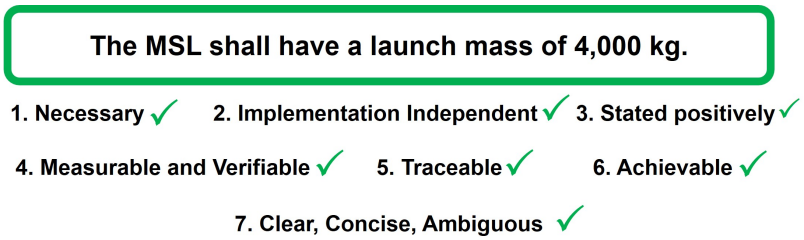
Now I appreciate that I've just given a load of rules. Whilst I am not the rebelious type, the easiest way to understand these rules however is to see what happens when we break them, so it is time to look at the *the good, the bad and the ugly* of written requirements. We'll look at three example requirements and compare them to the 7 requirements of requirements writing from Figure 7. Again we'll use the Mars Sample Laboratory (MSL) as an example. Up first:




Oh dear, oh dear. This requirement has not gone well. I know we can all agree that if we are launching something to orbit that minimising mass is key and that this would probably be achievable with engineering graft. However we have no idea how much we need to minimise the weight by, so we cannot verify our system will do this and after all this requirement infers that it does not really matter because it is only a '*must*'. We've also stated the requirement negatively with 'not'. Let us have another go...




Okay, so this is closer. We can now trace back to that the system should end up in space, and you'll note we've dropped the assumption that we will use a rover (it could be another system architecture after all). We are stating the requirement positively this time but it is still optional through the use of 'should'. The requirement is also ambiguous, after all *which* rocket are we talking about here? This also leads to us not being able to verify that we have satisfied the requirement. Let us try again...




Aha! Third time is the charm. We now have an implementation independent requirement that is measurable and hence verifiable. It most importantly is also now mandatory for us to satisfy the requirement to meet the system need. Job done.

### 2.2.1 Requirement types

When we are dealing with the thousands of requirements on real engineering programmes it can be exceptionally helpful to divide the requirements list into different *types* of requirements. Figure 8 shows 6 different requirement types with examples from the Mars Science Laboratory.

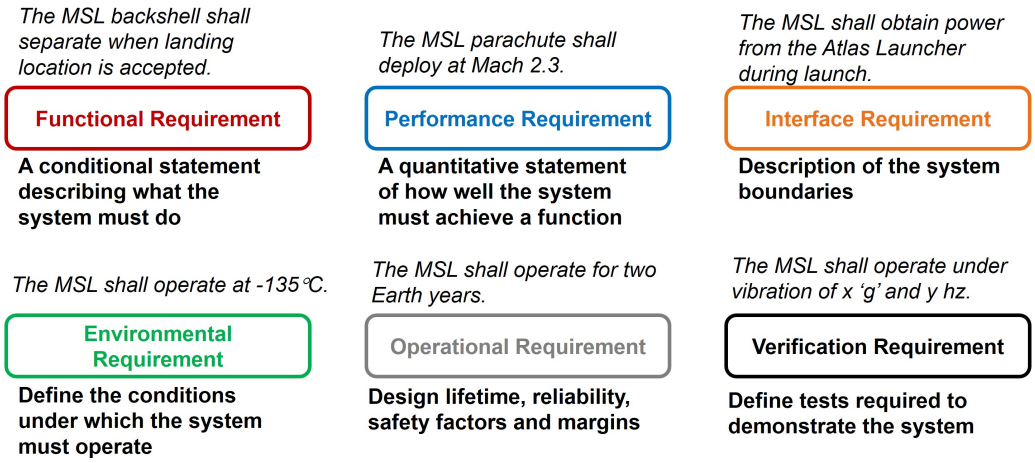



Figure 8: The different types of requirement

You can expect that many requirements for systems will be functional and performance requirements, but also do not forget that many performance requirements could be interface requirements (the good, bad and ugly requirement writing example shown previously is in fact an interface requirement).

The one requirement type that always seems to cause confusion are ***Verification Requirements***. These are requirements that define the tests that we will need to perform to simulate the environment that our system will need to operate in. This is especially true in space systems engineering where the system requirement might be derived from needing to operate in the vacuum and cold of space but naturally we will need to test the system on Earth prior to launch. Hence we'd write a verification requirement related to a climate chamber test of the system.

## 2.3 Verification, Validation and the V-model

Believe it or not, by identifying verification type requirements for defining testing, we've stumbled into one of the most important, yet most easily confused, topics in Systems Engineering: ***Verification and Validation***. *Deep breath, let us give it a go...*

**Verification:** "*Have we built the system right?*" - this is process where we prove that the system we have designed and constructed meets our requirements.

**Validation:** "*Have we built the right system?*" - note the subtle change in wording here. The process of validation is where we demonstrate that the system will satisfy the system need in its intended operating environment.

You may recall back in Section 1.3 that I said we had left something important off the V-model? Well what was missing was infact verification and validation activities. These have been placed in their rightful home in Figure 9. Within Figure 9 you can see that verification and validation activities are how we flow between the left and right hand side of the V-model. You'll also note that verification is performed at component, sub-system and system level, whereas we can only perform validation once the final system is assembled. We breakdown the verification activities to ensure we find errors or problems early and I refer you back to Figure 3 in Section 1.2 to highlight the massive cost of redesign that would occur if found an error only after we had assembled the entire system.

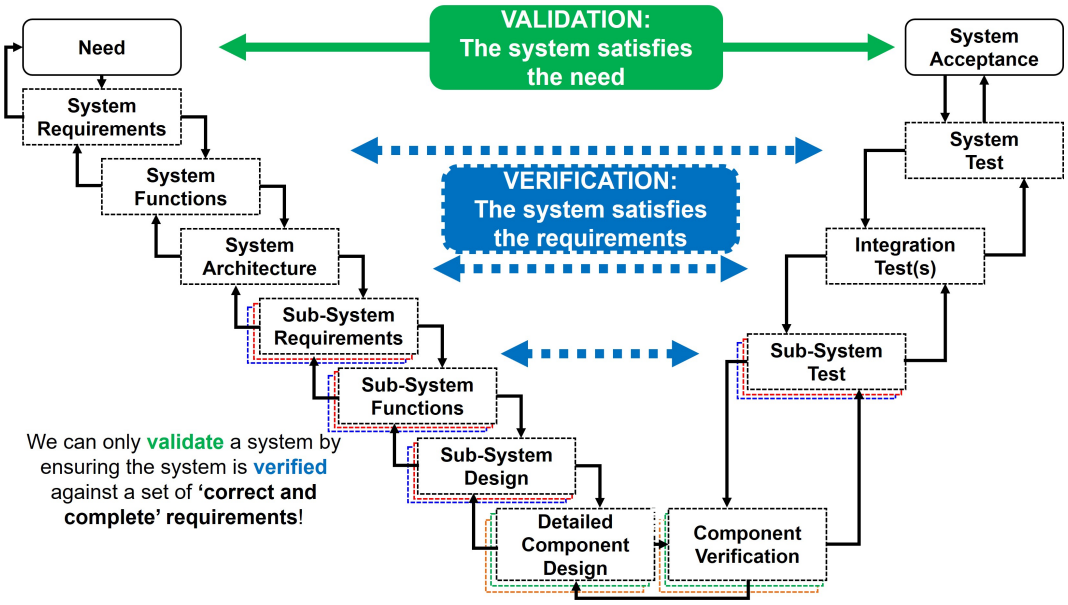



Figure 9: The location of verification and validation activities in the V-model

The concept of *verification* comes quite naturally to us as solution-oriented engineers, as we are used to testing an engineering artefact to see if it does the activity we want it to do, to the required level. Within a real-world programme, you would test your components, sub-systems and system against the requirements you had defined and if it meets those requirements your system has passed verification. There are four core ways in which we can carry out system verification:

1. **By Inspection:** we can directly measure something (e.g. system size)
2. **By Similarity:** our system is something we've previously used for the same mission in the same operating environment
3. **By Analysis:** we have used proven/certified analysis methods to assess the system (probably with some margin/safety factor applied)
4. **By Test:** we have created a representative environment in order to test our system (usually defined by verification-type requirements) - *this is often an expensive and very last minute way to find out your system does not meet the requirements...*

System *validation* is harder to get to grips with. The only way we can guarantee that our system will satisfy the original need, is to demonstrate that the requirements we have used for verification fully encapsulate the system need. We do this using a series of 'completeness and correctness' checks (the NASA Systems Handbook in Appendix C has a super useful checklist! [1]) where we as an engineering team and stakeholder group ask the following questions:

- *Will the system requirements satisfy the need?*
- *Are the requirements written correctly?*
- *Are the requirements verifiable?*
- *Are the requirements necessary, traceable and consistent?*
- *Are the requirements based upon valid assumptions?*

Don't panic, I know this looks like a huge job with loads of grey areas (and on real-world programmes it is!), but we will practice this in the live session where we look at requirements. The key point is, system validation only works if we get our requirements right in the first place.

### 2.3.1 Requirement documentation

We've mentioned multiple times already that real programmes will have thousands of requirements, so as you can imagine there are robust ways of documenting requirements. We will see some real-world requirement documents in the live session (and you'll get very familiar with the AVDASI2 requirements document this year as well), but they often have a table like the one shown below:

Table 1: System Requirements Table Format

| ID | Requirement Text | Rationale Text | Verification Type | Verification Plan |
|---|---|---|---|---|
| *e.g. P1* | *The system shall...* | *Justify your requirement* | *State verification type* | *Detail verification tasks* |

In reality, this information is often spread across different documents or document sections but the key thing that is everything is linked by a unique requirement ID. This is shown as *P1* above, but on real programmes the ID will often state the relevant sub-system and even requirement type along with a unique number.

The rationale is some supporting text that is often there to provide justification for the value stated in the requirement. It will typically reference a specific technical report. It is important to note that having a rationale is not excuse for sloppy or unclear requirements!

It is good practice to also highlight the verification type (see previous section) and the detail of the plan for carrying verification for each requirement, as this demonstrates part of the *completeness and correction* checks for requirements validation.

## Section Summary

**NEED** - What is the system trying to solve, satisfy or fix?

**GOALS** - What does the system need to do to satisfy the need?

**OBJECTIVES + REQUIREMENTS** - How well must the system satisfy the need?

**VERIFICATION** is used to show a system meets a set of requirements...

**VALIDATION** shows our system satisfies the need...

Validation can only be achieved through verification of a system based upon a set of **CORRECT AND COMPLETE** requirements.

# 3 Mission and Functional Analysis - *How do we know what our system must DO?*

Once we have completed the not-so-trivial task of defining the requirements for the system, we can finally engage the solution-orientated part of the engineering mindset and begin to think about *how* we might go about achieving those system requirements.

***But before you go and rush to your Home Lab Kits to build, say a temperature sensing circuit for a CubeSat,*** we are still not yet at the stage of thinking in engineering components, let alone subsystems. We must first think about the general operation of our system. This is one of the most exciting aspects of Systems Engineering, as we get to imagine our way through the system operation - *who doesn't want to pretend to be in Mission Control during a Lunar mission?*. This play-acting is called *Mission Analysis* can take the form of documentation, physical mock-ups or computer-generated images.

Mission analysis requires us to explore how the mission and system operation will unfold in its entirety from start to finish. In doing so, we will identify the useful aspects of our system which can of course be iterative fedback through the V-model to better define the system need and requirements:

1. **The sequence, or schedule, in which things must happen within the mission** - *'when' things happen in a mission can have a significant effect on both the system size and the final architecture and we will explore this in Section 4 and 5*
2. **Other systems outside of our system boundary that are required to complete the mission** - *its all good and well having your rover successfully land on Mars, but it is less useful if we have no means of communicating with it*
3. **System contingency in off-nominal conditions** -*'off-nominal'* is a less-scary way of saying something has gone wrong. Mission analysis helps us identify how we can get out of trouble

Before we look at a specific way of documenting this Mission Analysis, I do need to highlight something. Whilst we have said that we should be implementation independent about the system design so far in this handbook, this is the point where we begin to tip-toe around and cross the line into performing some elements of system design and making some assumptions about what our system might be composed of or look like. *This is okay.* Remember, within matured industries we are often building upon legacy systems, or if we are in the hottest start-up in town, we would generate multiple different Mission Analyses and System Architectures that we would then tradeoff against each other (see Section 5).

## 3.1 CONcept of OPerationS (CONOPS)

The easiest way to get to grips with Mission analysis is to see examples and try it out for yourselves (*spoiler alert for the live session...*). A COncept of OPerationS (CONOPS) is a visual representation of the entire mission cycle for the system, such as the CONOPS for the Artemis III mission in Figure 10. Artemis III will be the first crewed Moon landing mission since the Apollo programme [10, 11].

You should see in Figure 10 that the entire mission is shown, from launch to recovery of the crew following splashdown and the mission steps on the left hand side are complimented by clear schematics. You can also see that systems outside of the Orion spacecraft are captured, such as the Space Launch System (SLS) and the lunar lander. The CONOPS presented overleaf disguises the significant engineering effort and mission analysis that would have already been expended to define the mission, and we'll explore this in the supporting case-study video.

Another important thing is that CONOPS are not static across the development of a system. Whilst we will explore this deeper in the case-study video this week, it can also be seen to some extent in Figure 11, which shows the lander-centric (set to be SpaceX's Human Landing System - HLS) perspective of the CONOPS. This CONOPS details how the lander will be available for rendezvous during Step 8 of the Orion CONOPS in Figure 10. Therefore, we end up with CONOPS that can be effectively 'zoomed-in' to focus on another system boundary to provide more detail on the steps within the specific mission phase.

The only item missing from the two CONOPS is showing off-nominal operations. Rest assured however that there will be multiple CONOPS detailing contingency plans if things go wrong - for example, Figure 12 shows the CONOPS for the test of the original launch orbit system for the Orion spacecraft to be used in Artemis III!

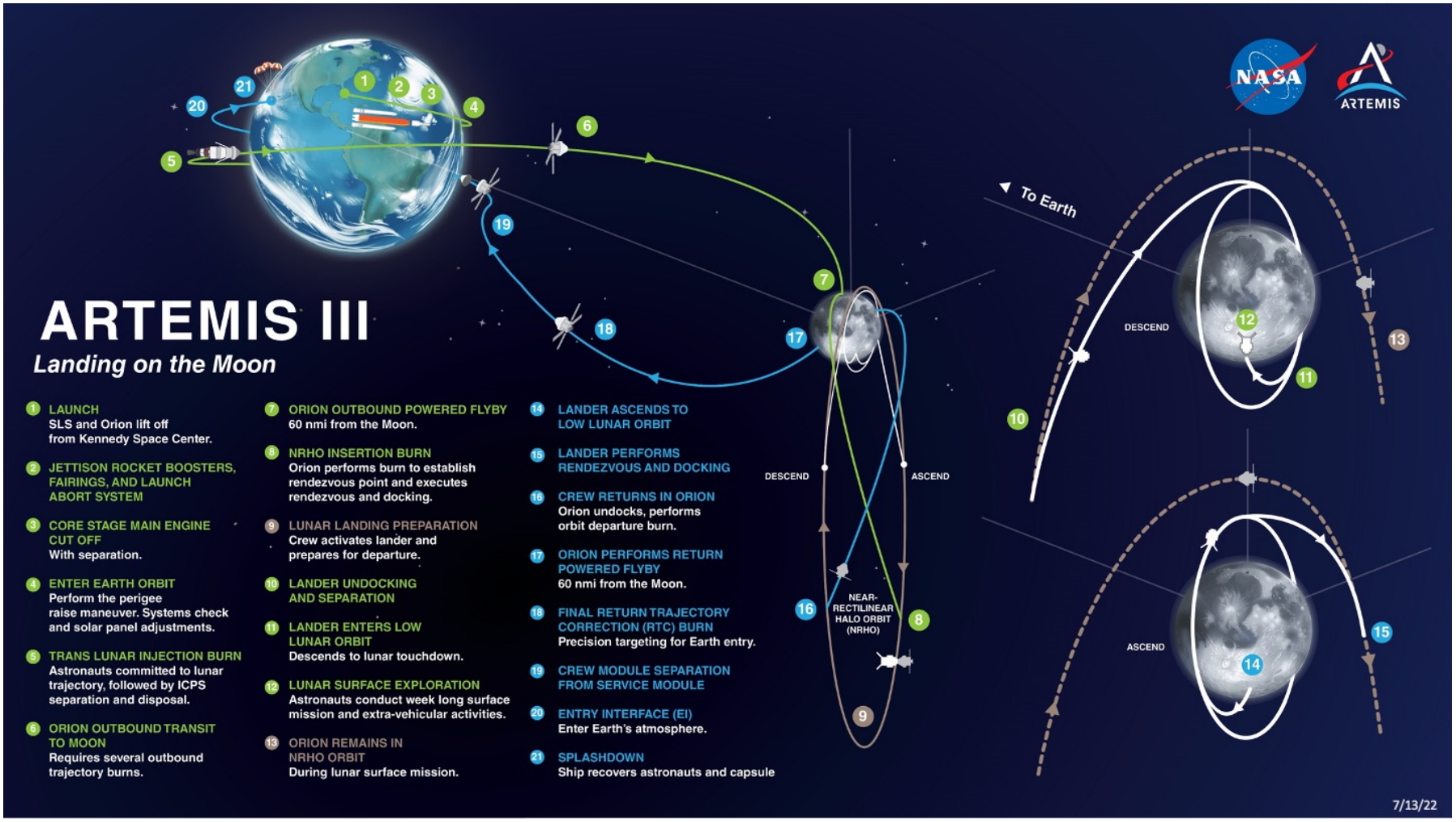



Figure 10: Visual CONOPS of the Artemis III Mission under nominal operation. *Image Courtesy of NASA, Public Domain*

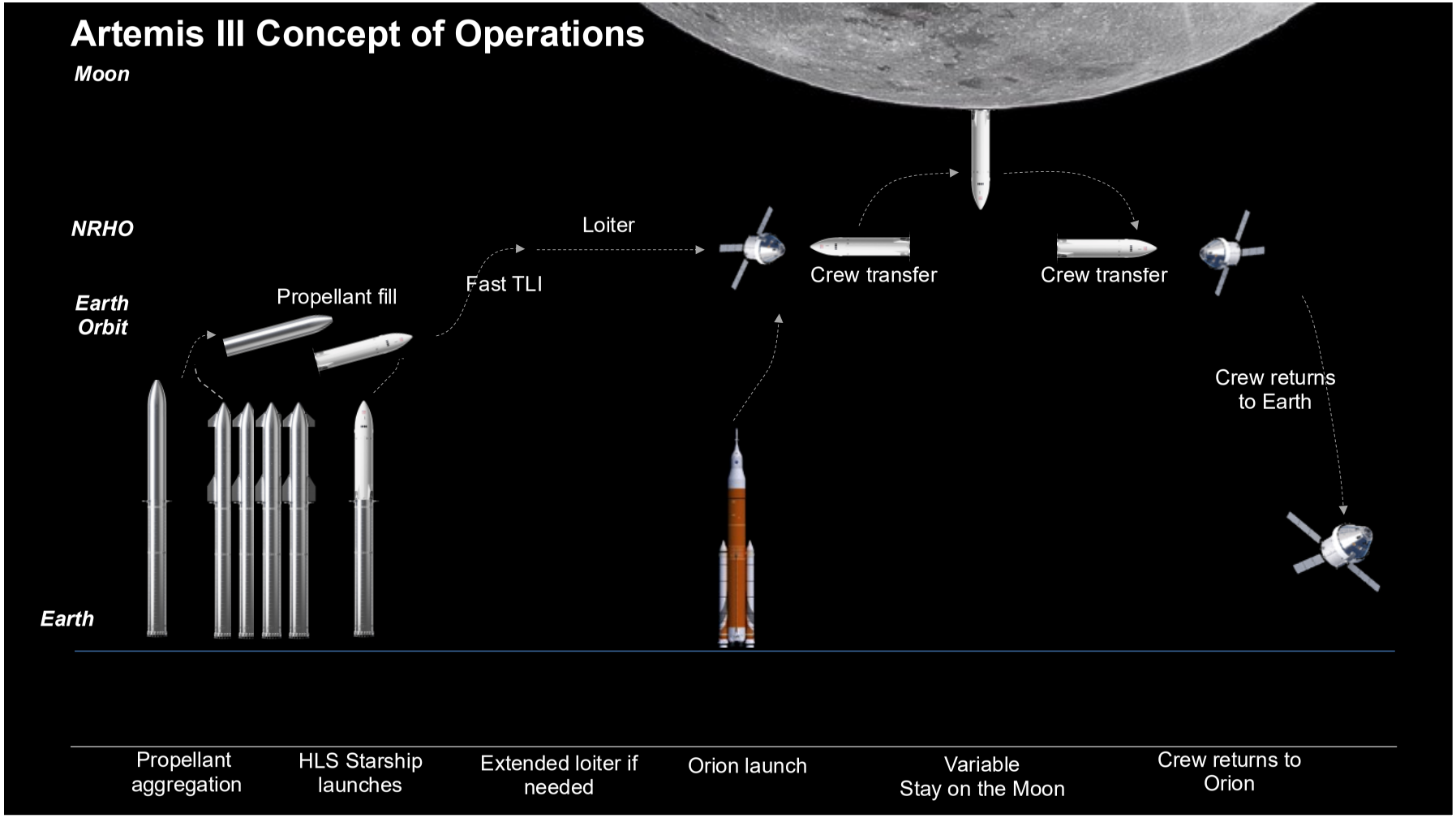



Figure 11: Visual CONOPS of the Artemis III Mission from the Lunar Lander perspective. *Image Courtesy of NASA (Kent Chojnacki), Public Domain*

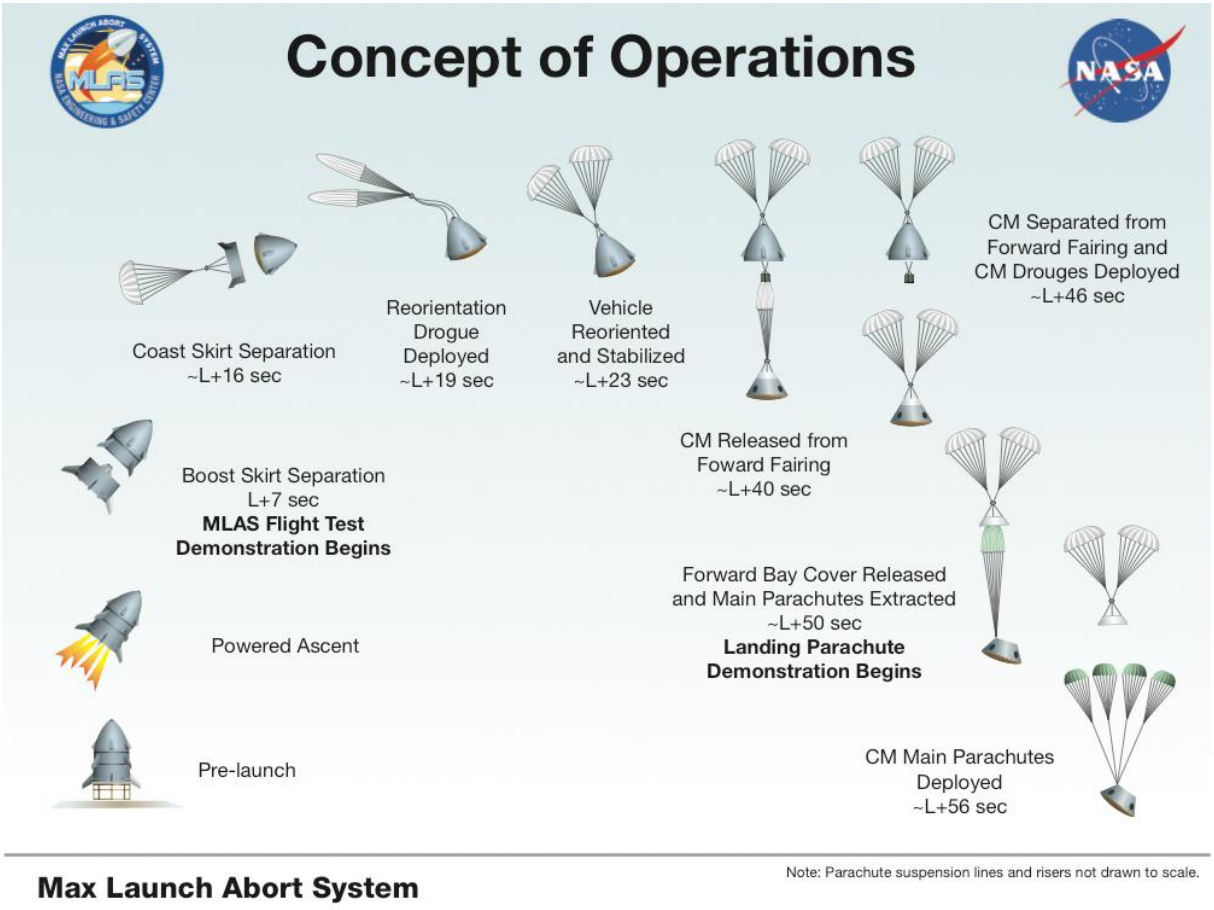



Figure 12: Visual CONOPS concerning the test of the original Maximum Launch Abort System for the Orion spacecraft. *Image Courtesy of NASA, Public Domain*

## 3.2 Functions and Functional Flow Block Diagrams (FFBDs)

So with a CONOPS we have a high-level view of the system mission, but you can imagine that this does not provide enough detail to begin to define a system. We therefore need a mechanism that can help us go from the top-level CONOPS to a detailed set of steps that our system must carry out. Once again, we are having to view our system from a different perspective at a different level of detail.

The transition from CONOPS to a system design requires a significant amount of work and similar to the System Engineer's journey from a system need to a set of requirements. However, to support us on this journey, we do have a specific set of techniques that can assist us. The first of these are known as ***functions***. INCOSE have a formal definition of "*A 'function' is a characteristic task, action or activity that must be performed to achieve a desired outcome"* [2].

To put together a function, we carry out the following steps [4] and we will illustrate these steps using the Artemis III Human Landing System(HLS), based on the SpaceX Starship platform. So just what is the *function* of the HLS aligned with the CONOPS Steps 8 to 12 in Figure 10:

**Step 1: Start with the outputs your are trying to achieve**

*For the HLS, we are trying to transfer humans between lunar orbit and the lunar surface*

**Step 2: Find the other system(s) you need to interact with**

*1)The Orion Spacecraft, 2)The astronauts*

**Step 3: Find the inputs that may be required**

*1)The astronauts are onboard the HLS, 2)The HLS has undocked from the Orion Spacecraft*

**Step 4: Write the function**

*The first function of the HLS is to transfer humans between lunar orbit and the lunar surface after the astronauts are onboard the HLS and after the HLS has undocked from the Orion spacecraft*

From the steps above we can make some observations. Firstly, we can see that functions have a sequence, order and a flow just like the CONOPS. We can also see the definition of interfaces between systems forming and we will explore these later in Section 5. A final very import observation is that the function will contain an 'adverb' or doing/action word. In this instance it is *transfer*.

Naturally the example above is grossly simplified, but you can appreciate that for real systems the description of functions in this way will become very wordy and difficult to trace and design too. Therefore, functions are often captured the *functional requirement* type that we first met in Figure 8 in Section 2. However, all of the conditions and interfaces that can be placed upon a requirement can make it **exceptionally** difficult to write acceptable functional requirements. *We need another way*.

Within complicated programmes we use a technique known as **Functional Flow Block Diagrams** (FFBDs). To create an FFBD, we need to define our functions as functional blocks. Each of these blocks will contain the 'action word' for our function and any related systems. Just like requirements, the golden rule of a functional block is that it only shows a single function. So for the HLS, we might end up with the functional block shown in Figure 13.

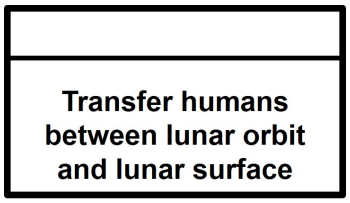



Figure 13: An example of a functional block

Regarding the inputs required for the function from Step 3, we can state these as other functions that lead into our original functional block as shown in Figure 14. We can also complete this chain by looking at the function required after our original functional block. In each block the 'action' word is underlined. Note how we are now graphically demonstrating the sequence in which things must happen as our system operates and we use the ID numbers in the top-left to show the sequence. Consequently, Figure 14 represents a simple FFBD - an FFBD is yet another perspective and abstraction that we can view our system from.

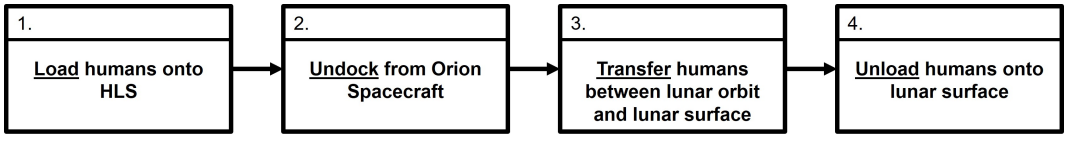



Figure 14: An example of a simple Functional Flow Block Diagram (FFBD)

You're all probably shouting "*it must be more complicated than that!?*" and indeed you are right. We can decompose each of the function blocks into its own FFBD. Figure 15 shows how we have added more detail to the functional steps required to carry out our original function in block 3.

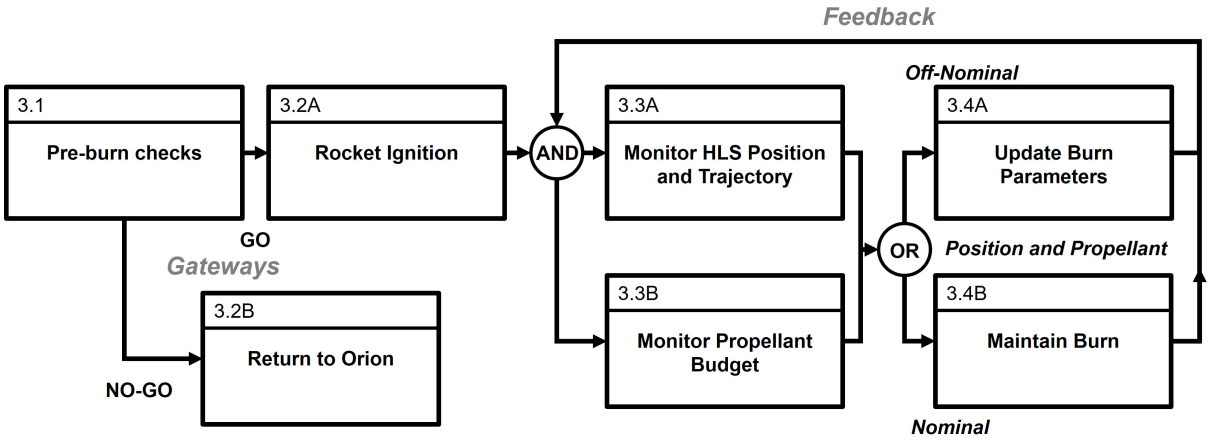



Figure 15: An example of an FFBD that uses gateways, parallel functional paths, conditional functional paths and feedback)

I think we can agree that this FFBD looks more comprehensive and in good news for us it highlights three very useful techniques we can use in an FFBD to capture complex functional behaviour:

1. **Gateways:** We can capture the outcome of tests we need to pass in order for the system to continue nominal operations. We often classify these as 'GO' (i.e. we can continue with the system nominal operation) or 'NO-GO' (i.e. we must stop the system and move to some 'safe' condition/configuration/function). We can see this in Blocks 3.1, 3.2A and 3.2B. Also note how the identification convention groups related functions together. For example if our system passes the pre-burn checks we 'GO' to 3.2**A**, but if we fail them we 'NO-GO' to 3.2**B**.
2. **Parallel Function Paths:** You can appreciate in most complex engineering systems that we will expect our system to carry out two functions in parallel. We can see this in blocks 3.3A and 3.3B where we want to monitor our position and the remaining propellant available at the same time. We can show this using an 'AND' block.
3. **Conditional Function Paths:** Sometimes in normal operation our system will get to a decision point, where it can respond in two or more ways. For example, we may find during our mission that the HLS position or remaining propellant is not as expected and we need to do something about it. An 'OR' block permits us to consider off-nominal and -nominal operations, either to update the burn we are performing, or carry on the thruster burn as planned. You be thinking how 'GO/NO-GO' and 'OR' differ. In FFBDs 'GO/NO-GO' is employed for when we will need to stop the mission under 'NO-GO' condition, whilst an 'OR' captures different functions required under nominal system operation.
4. **Feedback:** You'll note following our decision of either maintaining the burn or updating the burn parameters, that we need to pass this information back through the FFBD to ensure we can continue to monitor the HLS position and propellant budget. This is known as *feedback* and we find the concept of feedback appearing in many areas of Aerospace systems.

As you can imagine, the FFBD shown in Figure 15 is highly simplified from what is required to describe 'real' systems. For example, there would be additional consideration of what functions are required if updating the burn parameters pushes us past the amount of propellant we have to complete the mission. You also need to imagine that each block can be further levels of deeper and deeper functional complexity. Have a think about the different functions that might be required to achieve FFBD block 3.3A for example. There's a link on the unit Blackboard page under this session that will take you to the top-level FFBDs generated for the James Webb Space Telescrope (JWST), to give you a flavour of just how quickly FFBDs can get supremely complicated. Also in the live session we will build a FFBD together of a mystery system, before building upon FFBDs as we start to carry out system design, so please don't worry if the utility of FFBDs is not immediately clear!

## Section Summary

The first stage of system design is to identify **WHAT** the system must **DO** to meet the system need and requirements

LET. YOUR. IMAGINATION. RUN. WILD. when carrying out mission analysis, but **avoid** assuming **how** you will use specific hardware to carry out the mission

**CONOPS** - provides top-level visualisation of the mission

**FFBD** - provides the sequence and detail of functions at the system, sub-system and (eventually) component level

# 4 Budgets, Margins and Apportioning Risk - *How much of a thing does our system HAVE?*

In the previous section we started to identify WHAT our system must do to satisfy a set of requirements. A large part of transferring the functional description of our system into the hardware us engineers get truly excited about is understanding the *constraints* that are put upon our system.

Constraints are essentially requirements that dictate how much of a given physical quantity our system will have. Some examples of constraints include, but are not limited to:

- physical size
- mass
- time
- available power
- fuel/propellant
- data transmission rates
- cost
- *etc.*

***in fact, any quantitative aspect of our system is likely to be constrained!***

The statement of a 'constraint' infers that these physical quantities will have some form of limit (either a maximum or a minimum) that our system must respect. Let us look at a real example from the Curiosity Martian rover. Table 2 shows the propellant mass used by the Mars Science Laborartoy (MSL) cruise stage, which was the spacecraft that transferred Curiosity from launcher separation through to the trajectory correction manoeuvre immediately before the entry, descent and landing phase on the Martian surface.

Table 2: Curiosity cruise stage propellant usage, adapted from Ref [7]

| Start Date | Curiosity Cruise Event | Propellant Mass Used (kg) | Cumulative Propellant Burn (kg) |
|---|---|---|---|
| 26/11/2011 | Launch | 0 | 0 |
| 26/11/2011 | Separation | 1.78 | 1.78 |
| 28/11/2011 | Spindown, turns, calibration | 1.63 | 3.41 |
| 11/01/2012 | Trajectory correction | 18.03 | 21.44 |
| 25/01/2012 | Turns, calibrations | 3.33 | 24.77 |
| 26/03/2012 | Trajectory correction | 2.23 | 27.00 |
| 26/03/2012 | Turns, calibrations | 1.90 | 28.90 |
| 26/06/2012 | Trajectory Correction | 0.14 | 29.04 |
| 26/06/2012 | Turns | 0.19 | 29.23 |
| 29/07/2012 | Trajectory correction | 0.03 | 29.26 |
| 29/07/2012 | Turn | 0.03 | 29.29 |

Now I think some of the numbers in Table 2 are staggering. Firstly, the cruise stage travelled $\approx$350 million miles using only 29.29kg of propellant after separation from the launch vehicle[1]. It is also an incredible feat of command and control that only 30 grams of propellant is being used for the final trajectory adjustments and ultimately Curiosity landed within 2.3km of its intended touchdown site. Wow.

For most of this section, we are going to use the values in Table 2 and from some other case studies to consider how we deal with constraints during system design. Like a number of aspects of Systems Engineering it may feel like engineering 'common sense' but hopefully when you review the case study video you will see that on complex projects constraints can often trip us up....

[1] this works out to be about the mass of three sausage dogs in total or the mass of about 100 human cells of propellant burnt per mile! Now you'll see from Table 2 that these are consolidated into large burns for trajectory correction, but hopefully you'll also see how the first correction has the 'lion's share' of the used propellant mass, and the remaining corrections get smaller and smaller down to only 30 grams worth of propellant.

## 4.1 Engineering Budgets

The arrangement of values shown in Table 2 is known as a **budget**. Each activity, be it separation, trajectory calibration, etc., has an amount of a physical quantity (e.g. propellant mass) associated with it. We also know that the propellant onboard is not infinite, so the physical quantity is constrained.

Now, Table 2 shows what happened in reality for the MSL cruise stage, However a large part of the Systems Engineer's role is to apportion and manage the constrained physical quantity to various parts of the system during design. Perhaps you can even see a link we could form between the functional view of a system we discussed in Section 3 and how we would then apportion the constrained physical quantity to each of the functions we've defined, more on this later...

Allow us to look into the future and consider a Mars exploration system that is yet to exist. In the functional analysis case study video we saw how the Perseverance Rover is currently busying itself collecting and caching drilled samples of Martian rock, which will then be returned to Earth using the Earth Entry Vehicle (EEV). The intermediary in this system is the Mars Sample Return Orbiter (MSRO). In the current system design, it will be the European Space Agency (ESA) who will provide the MRSO, however back in the 2000s, NASA's Jet Propulsion Laboratory (JPL) set about a MSRO concept design [12]. Let us look at an example of a real world system-level mass budget from this document:

Table 3: . Reproduced from Table 3-4 in Ref [12]

| **Parameter** | **Value** | **Units** |
|---|---|---|
| Orbit Parameters (apogee, perigee, inclination, etc.) | 500 km circular orbit, within +/- 30 deg incl. | - |
| Mission lifetime | 5 | years |
| Maximum eclipse period | 42 | minutes |
| Launch site | CCAFS | - |
| Mission lifetime | 5 | years |
| Total orbiter mass **with** contingency | 943 | kg |
| Total EEV **with** contingency | 47 | kg |
| Propellant mass **without** contingency | 1,573 | kg |
| Propellant mass **with** contingency | 2,280 | kg |
| Launch adapter mass **with** contingency (included in orbiter mass) | 30 | kg |
| Total launch mass | 3,720 | kg |
| Launch vehicle | Atlas 551 | type |
| Launch vehicle lift capability | 4,770 | kg |
| Launch vehicle mass margin | 1,550 | kg |
| Launch vehicle mass margin (%) | 31 | (%) |

*Now what can we observe in Table 3?* Firstly, note how we are taking a *bottom-up* approach and summing up the the system element masses of the orbiter (943 kg), the EEV (47 kg) and the propellant (2,280 kg) to get to our total launch mass of 3,720 kg (*943 + 47 + 2,280 = 3,720*). This is the key purpose of a budget, it allows us to add up the contributions to a physical quantity at the system level.

But also note that the budget also operates in a *top-down* manner too. We are often limited on what launcher we can use and this will constrain the mass of our system. For example, the proposed Atlas 551 launch can lift 4,770 kg. The mass budget there also permits us to allocate a constraint across system elements - for example:

1. We might know the maximum mass our launcher can lift
2. We then take away any mass that we might already know about (e.g. the mission requires us to transit to Mars, orbit and return to Earth, so we could make an approximation of the propellant mass required (2,280 kg)).
3. This leaves 990 kg for the system, which some initial analysis and experience might suggest is a 95/5% split between the orbiter and the EEV.

For a moment we will ignore what is meant by the terms *contingency* and *margin* in Table 3 and focus on the last step listed above. At an early system design stage, it is the System Engineers' job to identify how the available constraint must be decomposed or apportioned throughout the system. Here we see the orbiter could be up to 943 kg and the EEV could be up to 47 kg. Now the beauty of a budget is that it can also be decomposed to show how these mass budgets apply in more detail for each system element, as shown for the orbiter and EEV in Tables 4 and 5 respectively. For the time being, try not to worry too much about the Current Best Estimate (CBE) and Maximum Expected Values (MEV) or the 'contingency' (% Cont.) that the table shows. We will come back to this in Section 4.1.1.

Table 4: . Reproduced from Table 3-1 in Ref [12]

| | **Mass** | | |
|---|---|---|---|
| | **CBE (kg)** | **% Cont.** | **MEV (kg)** |
| Structures & mechanisms | 260.9 | 30% | 339.2 |
| Orbiter launch vehicle adapter | 22.6 | 30% | 29.3 |
| Thermal control | 28.1 | 27% | 35.6 |
| Propulsion (dry mass) | 137.5 | 25% | 171.9 |
| Attitude control | 29.0 | 21% | 35.1 |
| Command & data handling | 20.4 | 30% | 26.6 |
| Telecommunications | 29.1 | 13% | 32.9 |
| Power | 98.8 | 30% | 128.4 |
| Cabling | 33.0 | 30% | 42.9 |
| System contingency | - | - | 101.1 |
| **Total Orbiter Dry Bus Mass** | **659.4** | **43%** | **942.9** |

Table 5: . Reproduced from Table 3-3 in Ref [12]

| | **Mass** | | |
|---|---|---|---|
| | **CBE (kg)** | **% Cont.** | **MEV (kg)** |
| Structures and mechanisms | 16.5 | 43% | 23.6 |
| Thermal control | 0.4 | 43% | 0.6 |
| Thermal protection system (TPS) | 15.7 | 43% | 22.5 |
| Range beacons | 0.1 | 43% | 0.2 |
| Sensors and cables | 0.3 | 43% | 0.4 |
| **Total EEV Dry Mass** | **33.0** | **43%** | **47.2** |

Hopefully you can observe from Table 4 and 5 that again the system mass (originally driven by the launcher lift limit) can be budgeted or apportioned across the system elements. And yes, we can go another stage deeper into the detail. Appendix C of Ref [12] (link on Blackboard) shows the next-level mass budget for the Orbiter, and Table 6[2] shows an excerpt that details the thermal control mass breakdown for the orbiter. The large number of specific items (i.e. down to individual thermistors!) that are recorded, along with their itemised count, hopefully demonstrates just how detailed budgets can become as the system design matures. In addition, if you sum up either of the mass columns you should find that they agree with total values stated for the Thermal control sub-system in Table 4.

[2] I won't even pretend to know what every item in the Table 6 is, but it is really useful for you to see the detail that goes into mass budgets and therefore I strongly recommend you check out the full Ref [12] Appendix C section on Blackboard!

Table 6: . Reproduced from Appendix C in Ref [12]

| Component | Flt Units | CBE/Unit (kg/unit) | CBE (kg) | Cont. | CBE + Cont. (kg) |
|---|---|---|---|---|---|
| Multilayer Insulation (MLI) | 32 | 0.38 | 12 | 30% | 15.6 |
| General | 38.325562 | 0 | 0 | 0% | 0 |
| Paints/Films | 10 | 0.09 | 0.86 | 30% | 1.12 |
| General | 1 | 0.84 | 0.84 | 0% | 0.84 |
| Isolation (G-10) | 200 | 0 | 0.86 | 30% | 1.12 |
| Custom | 30 | 0.05 | 1.5 | 30% | 1.95 |
| Propulsion Tank Heaters | 6 | 0.1 | 0.6 | 20% | 0.72 |
| Propulsion Line Heaters | 20 | 0.1 | 2 | 15% | 2.3 |
| Thermistors | 60 | 0.01 | 0.6 | 10% | 0.66 |
| Mechanical | 30 | 0.05 | 1.5 | 30% | 1.95 |
| Other Components | 6 | 0 | 0 | 0% | 0 |
| OS Capture - MLI | 10 | 0.5 | 5 | 30% | 6.5 |
| OS Capture - Cond Iso | 6 | 0.1 | 0.6 | 30% | 0.78 |
| OS Capture - Surf | 5 | 0.05 | 0.25 | 30% | 0.33 |
| OS Capture - Htrs | 10 | 0.02 | 0.6 | 30% | 0.26 |
| OS Capture - Thermostats | 20 | 0.02 | 0.3 | 15% | 0.35 |
| OS Capture - Temp Sensors | 50 | 0.02 | 1 | 10% | 1.1 |

So what are the key takeaway about budgets? Well, firstly it is that they provide a record of how physical quantities are used up across a system design. It is the System Engineers' job to manage the budgets both in a top-down manner buy taking system level constraints and decomposing them across the system design, and in a bottom-up approach by adding up each system element's contribution to the physical quantity. Naturally this task evolves with the maturity of the system design, starting at a very high level such as the budget shown in Table 3 all the way down to near-component level detail in Table 6. Ultimately, the budget is the tool we use to make sure we do not exceed the system level constraint. If we bust the system level constraint its almost certain we've broken a lot of the requirements placed upon our system. A final important concept is that until we've actually built the system, the values shown in budgets are ***estimates***. This statement leads us nicely into the next section...

### 4.1.1 Margins

So far I've avoided talking about certain terms that have come up about contingency, margins, CBE and MEV. All of these terms refer to us having some additional 'safety factor' applied to an estimate of a physical quantity relating to our system. For example, many of the items shown in the previous budgets have a factor of 30% applied to the estimated value to produce the 'Maximum Expected Value' (MEV).

*But why would we want to do this?* - this is a very valid question, especially in the aerospace sector where every extra kilogram we have to lift off the surface of the earth increases cost, fuel burn and emissions. Well there are some very valid reasons:

1. Until we have built, weighed and tested the system, any value we believe about a physical quantity relating to our system is just an estimate and hence full of uncertainty. We need to give ourselves 'room' as we learn about our system in more detail, just incase elements end up heavier, requiring more power or requiring more space etc. than we first thought.
2. System development cycles can take years, so we shouldn't be surprised that stakeholders may want 'more' from the system than we first thought. In addition, individual sub-system engineering teams may have under-estimated the budget they needed at the start of the design.
3. When dealing with safety critical systems we must also account for the 'known unknowns' (*what we know we don't know or what we know we can't control*) and the 'unknown unknowns' (*what we don't know we don't know or can't possibly perceive*). Hence factors of safety are used frequently to provide this additional margin to stop us ever observing failure.

All of the above means that we have to have some mechanism to help us provide this 'room' for quantities in the system to grow right from the very beginning. In a general top-level sense these are known as ***margins*** as they provide a margin between our system and the constraints, or in other cases failure and the negative consequences that come with it. As we'll see in the case study video, mass growth during system design in the aerospace sector occurs on nearly every programme and hence margins play an important role in avoiding the huge costs associated with a late stage re-design shown all the way back in Figure 3.

Now, the concept of a margin sounds sensible and straightforward. However, like many parts of Systems Engineering the simplicity on the exterior hides a huge amount of complexity on the the inside, that unfortunately we can only take a brief look at in this unit. Firstly, the definition of a margin can vary from company to company and industry to industry. We even saw this previously in the acronym-fest of the MSRO with the terms CBE, MEV and Contingency being used. For this unit, we'll simplify the definition of a margin:

$$\text{Margin \%} = \frac{\text{Constraint Value} - \text{Current System Estimate}}{\text{Current System Estimate}} \times 100\% \tag{1}$$

If we return to the MSL cruise stage propellant masses reported in Table 2, we can see that 29.29 kg of propellant was burnt. The MSL cruise stage was designed to carry a maximum of 73.8 kg of propellant onboard. This means that $\approx$ 44.5 kg propellant remained unused. *So what was the margin?* Well, our constraint value is the maximum possible propellant and the current system element is the actual propellant used. Plugging these numbers into Equation 1 **gives a margin of 152%**. Remember that a margin therefore represents how much our current estimate can **grow** by before we are in trouble and exceeding the constraint[3]. In addition, the arrangement of the numerator is important as it sets up the convention that positive margins are acceptable, whilst a negative margin means we've exceeded the constraint.

You might be wondering why such a large margin existed for the MSL cruise stage. A paper concerning the MSL cruise stage by Martin-Muir *et. al* highlights that "*The very precise launch vehicle injection provided ample margin*" [13]. The paper also highlights the challenging probability of success requirement placed on the cruise stage achieving the planned trajectory and this may have led to the large margin observed.

A margin you will come across frequently during an Aerospace degree is the Ultimate Safety Factor of 1.5 that is applied during structural design. Imagine an aluminium alloy lug. If we divide the typical Ultimate Tensile Strength of aluminium alloy (e.g. 340 MPa) by the 1.5 safety factor we would get an allowable stress of 226.7 MPa. If we plug our constraint value of the UTS, and the allowable stress as our current system estimate into Equation 1, you'll see we end up with a margin of 50%.

The above highlights the final challenge of margins, which is deciding what an acceptable margin value is. Too much and we have an inefficient system, whilst too little means our system will not be robust to the in-service life it may lead. Thankfully in aerospace we have many standards that can help guide, based on decades of real world experience, what an acceptable margin is. Once such example are the mass margins defined by the European Space Agency[14][4]:

- 5% margin for a Commercial-Off-The-Shelf (COTS) equipment (e.g. items that we have used before in the same environment)
- 10% margin for COTS equipment requiring minor modifications
- 20% margin for newly designed equipment or equipment requiring major modifications or redesign
- A further 20% margin is applied to the estimated dry (i.e. no propellant) launch mass of a spacecraft

[3] A common mistake in Equation 1 is to divide by the Constraint Value, which can lead to some odd and unrepresentative margin values occurring.

[4] Should you wish to delve into this complex world, come and ask me about some examples and if you wish to make understanding margins your hobby, allow me to leave the lid of Pandora's box ajar by directing you to a paper out of NASA's Goddard Space Flight Center [15] and the European Space Agency's margin philosophy [14].

### 4.1.2 Timelines

A quick flash back to Section 3 will remind us that as a System Engineer it is also our job to map out the sequence of the functions that our system must carry out to achieve its mission. Many functions we carry out will naturally eat into the budget of a physical quantity. For example Function Block 3.4B "Maintain Burn" in Figure 15 will naturally use up some of the propellant mass budget (and in a graceful piece of foreshadowing you can see that Function Block 3.3B provides the function to do this!). Consequently, budgets are not static in time and will vary as our system operates.

The point at which functional analysis and budgets combine is known as Timeline analysis where we map out what function occurs when and for how long, and then consider the budget implications of this. For example, Perseverance's radioisotope thermoelectric generator can produce approximately 100 watts of power, but some of the science equipment onboard will demand 900 watts [16]. This means that very careful planning of the use of Perseverance's batteries is required. In the case study video, we will look into some analysis performed to help identify the time-varying power budget for the Perseverance rover.

## 4.2 Managing Risk - Failure Mode and Effect Analysis

So far we've talked about budgets in terms of physical quantities, but the use of budgeting in Systems Engineering can also be applied to the safety of our system and any risks that we may face on an engineering programme and project. This latter case you will most certainly experience as you continue to embark on AVDASI2. The language around risk can be confusing and misused in everyday life and we also have dedicated techniques for budgeting and prioritising risk within engineering programmes. This final section aims to clarify the terminology around risk and also introduce you to Failure Mode and Effect Analysis (FMEA) techniques. Let us start with some key terms:

> **Hazard:**
>
> A state or set of conditions, known as a **failure mode**, internal or external to the system/project that has the potential cause negative effects (*e.g. harm, environmental damage, equipment loss, increased cost, project delay, project cancellation*)
>
> **Risk:**
>
> The potential for shortfalls, which may be realised in the future, with respect to achieving an intended outcome (*e.g. satisfy requirements, successful project completion*)
>
> **Occurrence:** the likelihood of a hazard occurring
>
> **Severity:** the impact on the system/project of the hazard

Consequently, a hazard is what hurts us and the risk is its potential for harm to occur and is composed of the likelihood of the hazard occurring and how serious the consequences are. As real engineering projects and complex systems can contain thousands of risks that are constantly changing in likelihood and severity we need some methodological process to help us target the most important risks. This is called many different things in the real world, including FMEA, Design Failure Mode and Effect Analysis (DFMEA), Failure Mode, Effect and Criticality Analysis (FMECA), or when dealing specifically with project management, a Risk Register. To keep things simple, we will just call it an FMEA and we will consider it in the context of system safety. *I'd strongly encourage you to think how you could adapt such tools to manage project risk on AVDASI2!*

To explore FMEAs we're going to use an example of risk management in CubeSats, based on a FMEA approach presented by Menchinelli *et al.* [17]. A CubeSat is a small satillite constructed to a standard footprint "1U" of 100 mm x 100 mm x 100 mm and an example of one is shown in Figure 16.

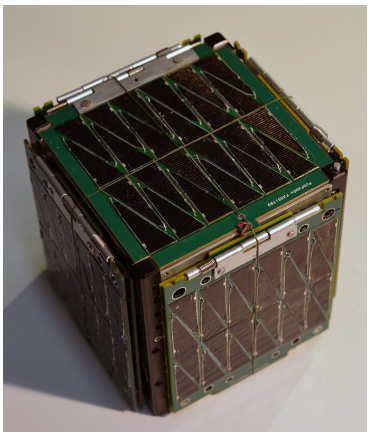


Figure 16: A 1U CubeSat. *Spcutler CC BY-SA 3.0*

The first stage of prioritising risk is to think carefully about our failure mode, to establish the probability it will occur (*occurrence*), the impact it would have on the system (*severity*) but also whether we would have any warning or be able to know where the failure mode had come from (*detectability*). Any FMEA or risk-register based approach will then aim to quantify the occurrence, severity and detectability and the examples for managing risks in CubeSats are shown in Tables 7, 8 and 9 respectively. For each identified Hazard, the scores for occurrence ($O$), severity ($S$) and detectability ($R$) would be defined.

Table 7: Cubesat Risk Occurrence Score ($O$) [17]

| Score (O) | Occurrence of Failure Mode |
|---|---|
| 1 | Likelihood is extremely low or mode has never been experienced in similar missions |
| 2 | Likelihood is low or mode has been experienced once in similar missions |
| 3 | Likelihood is moderate or mode has been experienced twice in similar missions |
| 4 | Likelihood is high or mode has been experienced several times in similar missions |
| 5 | Likelihood is extremely high or mode has been experienced in almost every similar mission |

Table 8: Cubesat Risk Severity Score ($S$) [17]

| Score (S) | Occurrence of Failure Mode |
|---|---|
| 1 | No vital subsystem is affected |
| 2 | 1+ component(s) are affected, mitigated with redundancy |
| 3 | 1 subsystem is affected, but full mission can continue using other subsystems |
| 4 | 1+ subsystem(s) are affected and full mission cannot be completed |
| 5 | Mission is totally compromised |

Table 9: Cubesat Risk Detectability Score ($D$) [17]

| Score (D) | Occurrence of Failure Mode |
|---|---|
| 1 | Only 1 failure source |
| 2 | $<$2 independent possible failure sources |
| 3 | $<$3 independent possible failure sources |
| 4 | 3+ independent failure sources |
| 5 | No ability to identify failure source and impacted sub-systems |

We then combine the $O$, $S$ and $D$ values into a 'Risk Priority Number' $RPN$ using the Equation 2. We then take this value, and *guess what!* We relate it to another table, shown in Table 10.

$$RPN = O \times S \times D \tag{2}$$

Table 10: Cubesat Risk Priority Number ($RPN$) [17]

| RPN | Impact of Failure Mode |
|---|---|
| 1-24 | Failure has potentially no impact on the mission |
| 25-40 | Failure may not have relevant impact on the mission, minor effects observed |
| 50-74 | Failure will produce effects on the mission, which may be subject to limitations |
| 75-99 | Failure will produce major effects on the system, that will likely not perform correctly, or drastically reduce the expected operativeness of the system |
| 100-125 | Failure is catastrophic for the mission |

So, say we were considering that the Mylar Thermal protection of our CubeSat could be torn by vibrations during launch. We know that this will lead to loss of thermal control of the CubeSat which in turn would impact our science payload, battery and communications system. We can then consider the $O$, $S$ and $D$ scores:

- $O$ = 4 because we've seen this failure mode many times before
- $S$ = 5 because loss of the payload, battery and communications leaves us a dead, useless satillite
- $D$ = 4 because on earth all we may know is that the satellite has gone 'dark' and is no longer speaking to us. We might not be able to identify which failure source has caused this.

This would give $RPN$ = 80, which suggests we will drastically reduce the system operation. If we looked at another case, say the battery harness disconnecting under launch vibration loads, we could compute the $RPN$. Imagine that this was a lower probability event, and it would be clear that it was the battery connection that had failed. Yes, loosing a battery is bad, but we still have the solar panels to provide *some* power. We then document these failure modes in a FMEA table, as shown in Table 11.

Table 11: Example FMEA table for a cubesat

| **Sub-system** | **Failure Mode (Source)** | **Effect** | **Affected Sub-systems** | $O$ | $S$ | $D$ | $RPN$ |
|---|---|---|---|---|---|---|---|
| Thermal (Mylar) | Laceration (Vibration) | Loss of Thermal Control | Science, Battery, Comms | 4 | 5 | 4 | **80** |
| Battery (Harness) | Disconnection (Vibration) | Loss of Non-solar Power | Science, Comms | 2 | 4 | 1 | **8** |
| ... | ... | ... | ... | ... | ... | ... | ... |

Consequently, an FMEA table, such as the one in Table 11 provides a tool to ***prioritise*** our attentions on what risks or failure modes need our attention the most. Remember, it is not just our system that has constraints and budgets, our engineering activities do as well, and therefore we must prioritise our efforts. From Table 11 it is clear to see that we would need to focus on the thermal Mylar sub-system failure as a priority when compared to the battery disconnect failure mode. If we really wanted to use our FMEA as a 'live' tool, we would also have columns related to mitigation plans (e.g. *perhaps we could have additional Mylar thickness?*) and also the current status of the risk/failure mode mitigation. It is important to note that every company, industry or application may have its own $O$, $S$, $D$ and $RPN$ definitions and you will see different variations of FMEA tables and processes! Finally, it is fair that the FMEA process feels *pseudo*-quantitative and that we are trying to assign numbers to something that is a qualitative assessment. This is true, but remember a FMEA is there as a relative measure and we will look at mathematical ways for assessing system safety and reliability in Section 6.

## Section Summary

*Gosh, that was a lot of tables to look at wasn't it?*

All physical quantities of a system will be **constrained**. We use **BUDGETS** to keep track of them during system design and operation. Budgets can vary with time.

Usually the values of physical quantities we have are just **estimates**. To account for this, we use **MARGINS**. Margins show show how much a quantity can **grow**.

**Risk** is made up of the severity of harm a hazard causes and the likelihood it will happen.

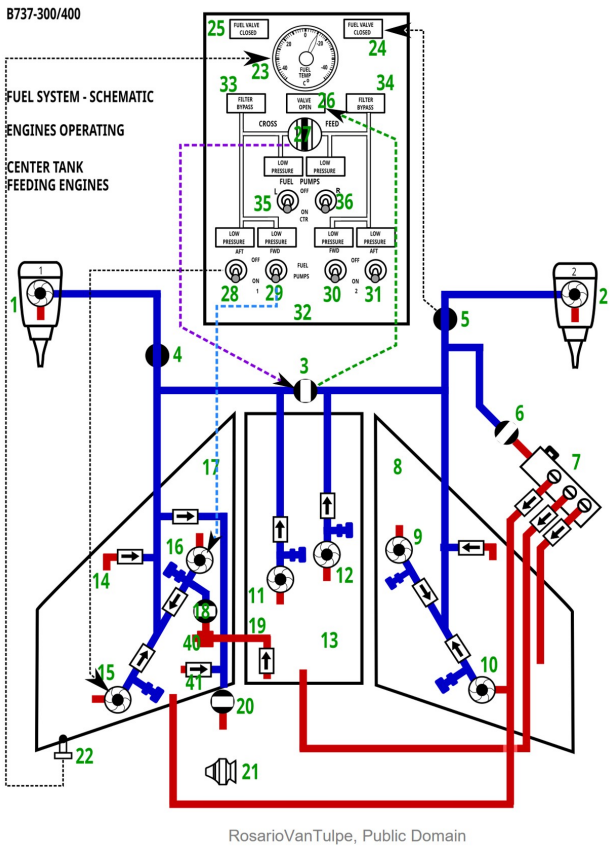
Failure Mode and Effects Analysis (**FMEA**) helps us **prioritise** risks.

# 5 Architecture Definition, Requirements Decomposition and Interfaces - *How do we know what are system must BE?*

So far on our Systems Engineering journey, we've identified what our system must do and how well it must do it, and identified how much of a given resource or physical quantity it might have to carry out its mission. *Surely this means we are approach the threshold of carrying out some maths and design work?*

**Not. Quite. Yet.** Engineering science and mathematics tends to give you the tools required to carry out detailed design on a component level, rather than helping you identify what component you need in the first instance. As you would have seen in the spark video, the characteristics of any engineering sub-system or component are driven by a large number of factors and we will consider each of these in this section.

The missing link between the prior sections of this handbook and the detailed design of components that us engineers relish is known as definition of the **System Architecture**. *But just what is a system architecture?* Ideally, once a system architecture has been defined it should look similar to Figure 17, which is the system architecture *perspective* of a narrowbody aircraft fuel system.




1. Engine Driven Fuel Pump - Left Engine
2. Engine Driven Fuel Pump - Right Engine
3. Crossfeed Valve
4. Left Engine Fuel Shutoff Valve
5. Right Engine Fuel Shutoff Valve
6. Manual Defueling Valve
7. Fuelling Station
8. Tank No. 2 (Right)
9. Forward Fuel Pump (Tank No. 2)
10. Aft Fuel Pump (Tank No. 2)
11. Left Fuel Pump (Center Tank)
12. Right Fuel Pump (Center Tank)
13. Centre Tank
14. Bypass Valve
15. Aft Fuel Pump (Tank No. 1)
16. Forward Fuel Pump (Tank No. 1)
17. Tank No. 1 (Left)
18. Fuel Scavenge Shutoff Valve
19. *Etc…*

Figure 17: An example of a narrowbody aircraft fuel system from a system architecture perspective)

*So, what can we observe about the architecture shown in Figure 17?*:

- **Functionally-Focused:** This is *not* a detailed design of components and systems. Instead, the architecture shows what system elements are required to carry out functions to satisfy the system need. For an aircraft fuel system, the top-level function is to provide fuel to the aircraft engines. Hence we can see the architecture includes fuel tanks (Elements 8, 13 and 17) to store the fuel, and a whole host of pumps to move fuel from the tanks to the engines. But note, there is not a single pump design value (e.g. pressure) in sight.

- **Abstract/Icon Representation:** Because a system architecture is not trying to show the specific design details of each system, they heavily rely on icons or abstract representations of the sub-system and components we wish to depict. For example, the icons used for Elements 1 and 2 in Figure 17 represent the vanes within a typical fuel pump, but do not actually show the full design of fuel pump that we are yet to design in the system lifecycle. Some architectures may even just rely on labelled 'black-boxes' to demonstrate the system architecture.
- **Interfaces:** The final important aspect of the system architecture in Figure 17 is that it shows **interfaces** between sub-system elements. If we cast our minds back to Section 1, we will recall that systems engineering is based upon breaking complex systems up into sub-systems linked by interfaces within the system model. A system architecture perspective shows us our engineering system with all of its interfaces and is therefore the closest to the systems model we can get.

Therefore, we've seen that a system architecture is a depiction of our system design that aims to highlight what system/sub-system elements we need based upon their function, but also the interfaces between them. We'll consider in Section 5.4 how our architecture evolves closer towards a hardware-centric representation of a system design, but first we need to consider how we generate system architectures in the first place. Much like other parts of Systems Engineering, the clarity of the architecture in Figure 17 disguises the long and complex systems engineering journey used to create it. The INCOSE definition for *System Architectures* provides a really nice framework for creating system architectures [2]:

**A *System Architecture* is the selection of the types of system elements, their characteristics and their arrangement, to meet the following criteria:**

1. **Satisfies the requirements**
2. **Implements the functional architecture**
3. **Acceptably close to optimum, within constraints of time, budget and knowledge**
4. **Consistent with the technical maturity and risks of selected elements**

Hopefully the parallels between the case study video and the handbook will begin jumping out at you as the four steps above were highlighted in the context of the landing gear retraction actuator. The remaining sections within the handbook will detail the specific techniques to carry out steps 1-4.

## 5.1 Functional and Requirement Decomposition

In Sections 2 and 3 we observed how we decomposed a system need into a set of system requirements and a Functional Flow Block Diagram (FFBD). Consequently, we shouldn't be too surprised to find that these are the very sources we use to define our architecture. Whilst we started with requirements, a system architecture is more naturally derived from the FFBD. In the live session we will explore converting FFBDs to system architectures in more detail, but let us dust off the FFBD from Figure 15 (which itself was derived from a CONOPS!). The first iteration of constructing a system architecture should try to achieve a 1:1 mapping of functions to a sub-system/component element. This means an architecture is *finally* starting to define **HOW** our system will satisfy the system need. Figure 18 shows the resulting system elements required for each function. From Figure 18 we now have the 'shopping list' of sub-systems that our architecture must contain to achieve Function 3 from the FFBD in Figure 14:

- BITE (Built-In Test Equipment)
- Ignition
- Trajectory Guidance
- Navigation
- Propellant Monitoring
- Propellant Control
- Propellant Delivery

In the live session we will experience what happens when we try to apply this approach across multiple FFBD levels for a system. We will see that the 1:1 mapping approach is not always suitable, but the key takeaway here is that our starting point for defining an architecture (and our list above is a very simple system architecture definition!) and ensuring our system will achieve the functions required to satisfy the system need is identifying sub-systems that map to the system functions.

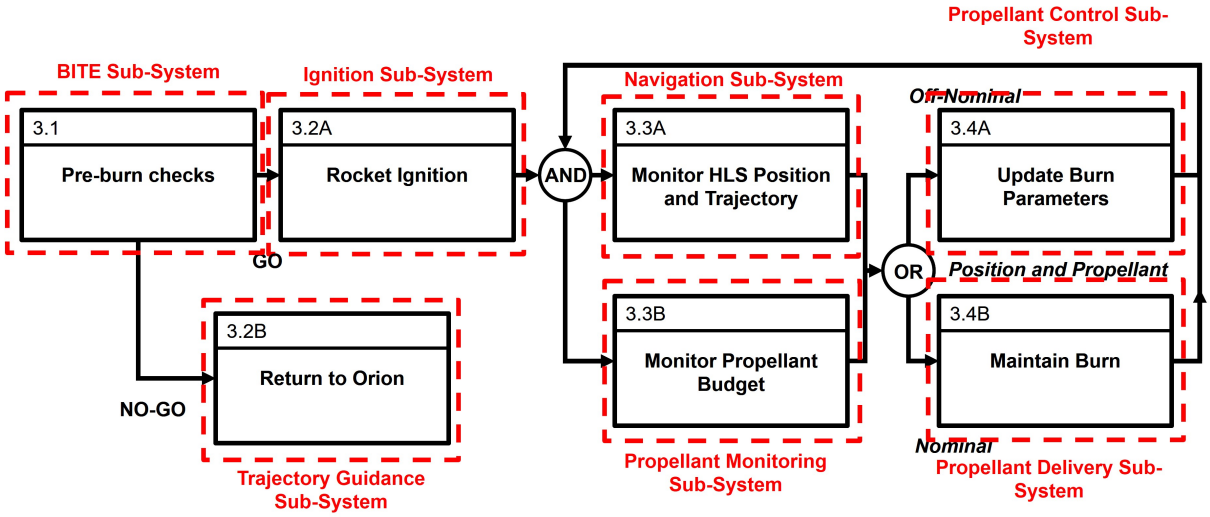



Figure 18: An example of 1:1 mapping from FFBD to required sub-system elements

### 5.1.1 Derived Requirements

As we go through the definition of a system architecture, we will inevitably start to make assumptions about *how* our system is going satisfy the requirements placed upon it. This immediately puts us at risk of breaking one of our requirements of written requirements from Section 2.2, which is that requirements should be *Implementation Independent*. This situation leads us to need to consider two types of requirements, as shown in Figure 19.

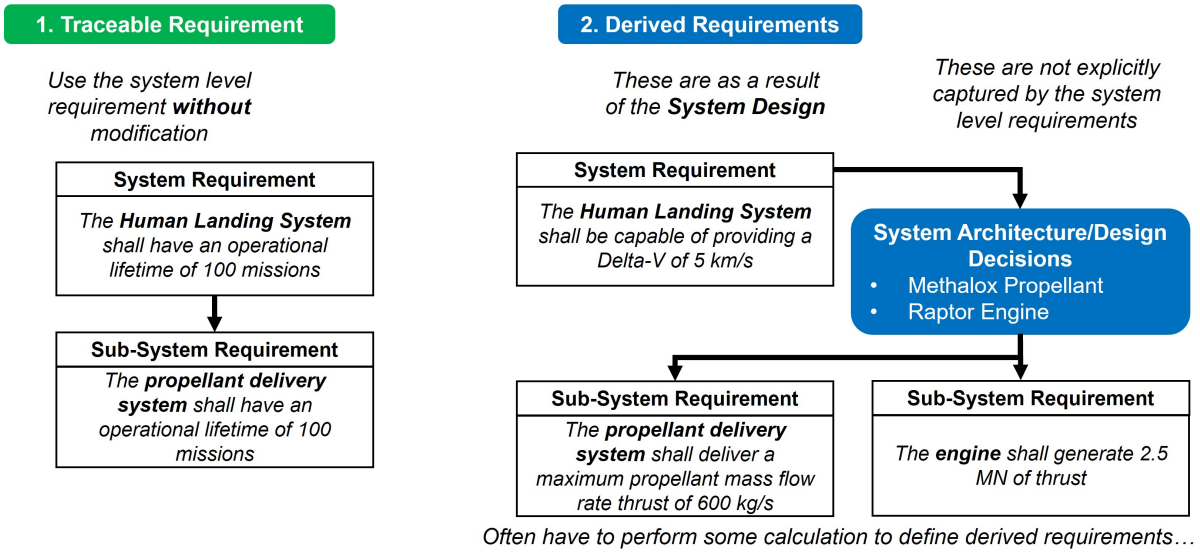



Figure 19: A comparison between Traceable and Derived requirement types

On the left-hand side of Figure 19 we can see ***Traceable*** requirements and these are requirements that can be directly mapped or *decomposed* to our sub-system elements without modification. Traceable requirements can also be linked back to system-level requirements based on any budgets we may have defined using the processes detailed in Section 4.

On the right-hand side of Figure 19, the generation of ***Derived*** requirements is shown. Derived requiremetns are those that can only be written, or only appear, once we've made some decisions about our system architecture. Whilst they will likely have some link back to the system level from

a *functional* perspective, the values used in the requirements will often be reliant on specific decisions and calculations we have made about our system architecture (e.g. the fact we have a propellant delivery system that handles Methalox propellant to feed a Raptor engine) and therefore it can be very difficult to directly trace these requirements back to the system-level.

The processes shown in Figure 19 highlight a continual and iterative process that is performed as we make our way down the left-hand side of the V-model from Figure 4. Remember, the system model permits us to draw a boundary around systems, sub-systems and components and therefore system level requirements are decomposed into a sub-system level, which become the top-level requirements for sub-system of interest and so on and so forth. This is why requirement traceability and rationale becomes so important, because when we eventually get to the sub-system level or component-level we can be many steps away from the original system requirements and it can be hard to remember the justification for why we might have to work so hard to satisfy very specific requirements[5].

## 5.2 System Interfaces: Types and N$^2$ Diagrams

The INCOSE definition of 'System Architecture' refers to the *arrangement* of system elements. At this stage, we would achieve this by identifying the interfaces between the sub-system elements we highlighted in the 'shopping list' in Section 5.1. Figure 20 shows proposed interfaces between 'black-box' representations of the sub-systems from Figure 18.

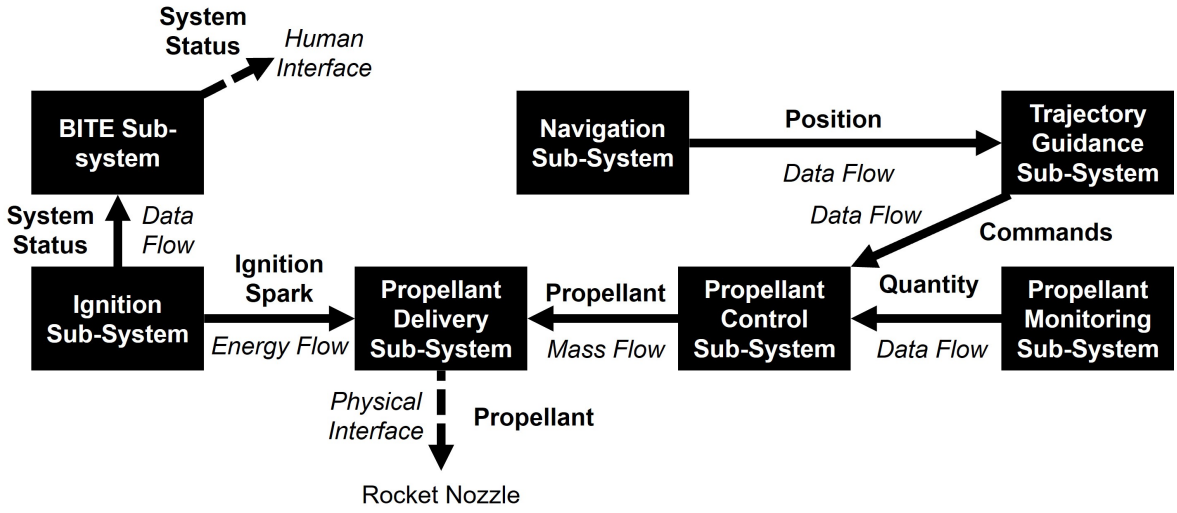



Figure 20: An example of a simple system architecture highlighting interfaces

From Figure 20 we can observe that several different types of interface exist:

- **Physical:** This is where something will physically constrain our architecture. For example, connectors and pipes within fluid systems, or geometrical envelopes that our system must fit within.
- **Energy Flow:** Interfaces that move energy across them (most often as electrical power). Typically, energy flow interfaces will need a corresponding physical interface (e.g. wires, plugs and sockets). Energy Flow interfaces will not always be useful. For example, waste heat is a common energy flow interface and we call such losses 'by product' interfaces.
- **Mass Flow:** Interfaces where a fluid, gas or solid is passed between sub-systems. For example, propellant flow rate. Similar to energy flow interfaces, these often need a corresponding physical interface.

[5]This can go one of two ways and both are not good. 1) We spend a lot of resources satisfying requirements without a clear justification or 2) We feel that a specific requirement may not actually be that important so we ignore it, only to find that it has a large ripple effect all the way up to the system level and then our system does not satisfy the system need, leading to huge costs to rectify this.

- **Data Flow:** Interfaces where 'information' is passed between sub-systems. Information can be in the form of stored binary, electrical signals or any other way of communicating information.
- **Human Interface:** Most systems have to interact with us human beings at somepoint and I'm sure we've all had experiences where the human-machine interface has been very poorly thought out...

Managing interfaces would be a whole load of work if we didn't have a concept called **STANDARDS**. Standards provide an explicit definition of an interface that means any engineer or stakeholder can be confident that they are designing to the same conditions as another stakeholder. As we saw in the case study video, standards are the true secret to managing complex system design because it removes the risk and cost associated with designing bespoke links between sub-systems every single time[6]. If you don't believe me on how important standards are, I found a NASA document from 1967 that lists all relevant NASA standards in use at the time, it is over 300 pages of report titles relating to standards (all written on a typewriter!) [18].

The representation of interfaces can become exceptionally complicated for real engineering systems and it is already starting to look quite messy in Figure 20. Don't panic, we have a specific tool that can help us clearly represent interfaces, known as an $N^2$ (N-squared) diagram. An example for the architecture in Figure 20 is shown in Figure 21.

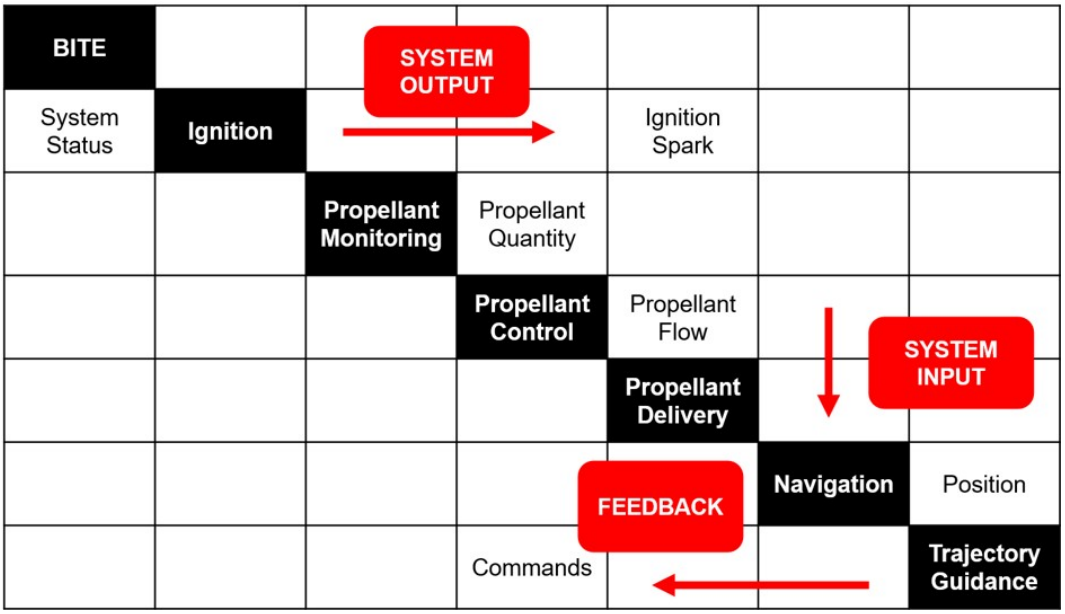



Figure 21: An $N^2$ of the system architecture shown in Figure 20

$N^2$ diagrams are a wonderfully elegant way to show interfaces. We start by placing our sub-system elements on the main diagonal for the table. Anything in a column then represents an input to the system element within that column and anything in a row represents an output from the system element within that row. For example, we can see that 'propellant flow' is an *output* from the Propellant Control system and an *input* to the Propellant Delivery system in Figure 20. As we are considering a simplified system architecture, our $N^2$ diagram does look sparse, but you can appreciate in the real world they can become incredibly complex.

One area of complexity is *feedback* in the $N^2$ diagram. In the event that we see an output on the left-hand side of a system block this infers that it is feedback to another part of the system. Now we have some power over this in the $N^2$ diagram, and we can move system blocks around to eliminate feedback. For example, the 'Trajectory Guidance' system block could be moved to before the Propellant Control block and this would remove the feedback in Figure 21. However, it is important to note that feedback won't always be able to be eliminated and this is where $N^2$ diagrams can be really useful to help identify

[6]Remember we don't get something for nothing, so we will explore the negatives of standards in the live session!

iterative loops or feedback conditions for our system that might be necessary, advantageous, or a real pain to deal with - *all of these outcomes would require more engineering work to deal with!*

A really important thing to highlight is that $N^2$ diagrams are not only useful tools for modelling system interfaces - ***any process with steps can be captured using an $N^2$ diagrams*** and this can be very useful to remember when writing code, or carrying out complex design process and iterations!

## 5.3 Picking the "black-boxes": Tradeoff's and TRLs

Usually when we arrive at the system architecture stage, there will be competing solutions and technologies that must be down-selected to form the final architecture. For example, in the system architecture shown in Figure 20 we have not defined *how* we will measure the quantity of propellant remaining in the propellant monitoring system. A handy study from NASA shows that we could use the following [**?**]:

- Bookkeeping - i.e. monitoring burn times to back calculate propellant usage
- Capacitance Probes
- Wet-dry Resistance Sensor
- Ultrasonic Level Measurement
- *etc.*

*So how would we begin to pick between each of these technical solutions?* In AVDASI2 you will have been learning about down selection strategies such as *Pairwise Comparison* and *Multi-Criteria Decision Analysis*. A general term that is used to cover the use of such tools for evaluating competing system and sub-system architectures is known as a **tradeoff study** or a **trade study**. Some key elements to keep in mind when deploying a tradeoff study (regardless of the tool used) are:

- Be very cautious of tradeoff studies being manipulated to tell you want you want to see - *confirmation bias*
- Tradeoff studies are very good at rejecting many ideas (often quashing creativity) and really poor at ranking or separating system architectures that are close in suitability
- Often the weights and scores used in tradeoff studies are a source of huge subjectivity and uncertainty. Where possible, assess the sensitivity of your ranking to the assumed values in the tradeoff study. Even changing values $\pm$10% can give you an idea of what scores, weights and metrics the study is sensitive too. You can then investigate whether you can source any further definition or justification for these tradeoff study elements
- Remember that optimising at the component or sub-system level can lead to a non-optimal system performance. Always try to build your tradeoff and optimisation processes with the system need and requirements at its core.

As engineers we are always striving to have analysis available to back our decisions, but we must also be aware that really important decisions often have to be made with limited knowledge, time and resources. This is why the systems engineering and the V-model as a process are so important. It helps us 'creep up' progressively to the really important tradeoffs and decisions.

There is another tool that greatly aids system architecture definition when choosing between architectures. Within the Aerospace sector and specifically Space Systems Engineering, we are often reliant on new or 'cutting edge' technology. New technology presents a massive programme and system risk and therefore Technology Readiness Levels (TRLs) are widely employed to enable engineers to refer to a consistent framework (*like a standard*) to ensure new technology is developed and its associated risk level known consistently. The TRLs employed by NASA are shown in Figure 22.

The most important element to take away from Figure 22 is the jumps that occur between certain TRLs, which represent a significant amount of engineering effort to achieve:

- **TRL1 to TRL2:** This is essentially the step where scientists and researchers observe something interesting and engineers then get to work proposing how to exploit the observation.

- **TRL4 to TRL5:** here the testing environment goes from a laboratory to a relevant operational environment. For example, we could test our propellant measurement system on a lab bench in TRL4, but to progress to TRL5 we'd have to carry out the tests in a thermal chamber and a vacuum to provide the operational environment of space.
- **TRL7 to TRL8:** This essentially only occurs when we get to the top of the right hand side of the V-model, where we demonstrate the system has satisfied the system need.

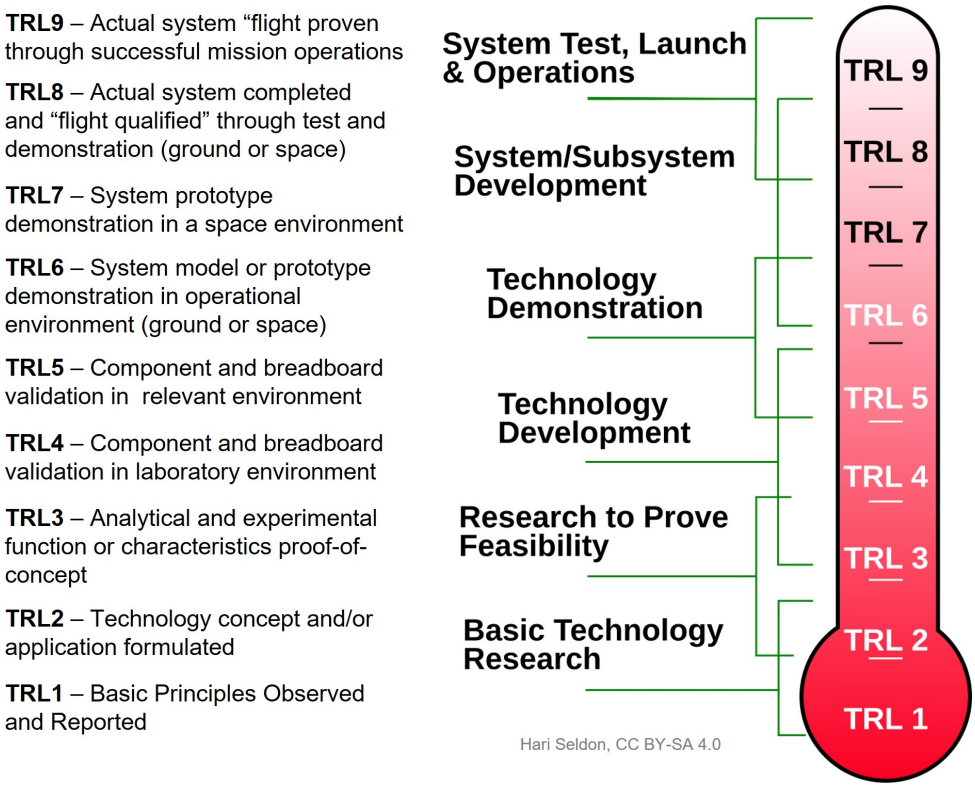



Figure 22: The Technology Readiness Levels (TRLs) employed by NASA

Many of you will experience TRLs in your future working lives. Remember they are a quick, convenient and consistent way for another engineer to tell you the maturity of *their* system such that you can decide how much of a risk it poses to *your* system, without relying on everyone seeing the full development and test campaign of every single new system element.

## 5.4 Onwards to System Configuration

If you have already looked ahead in the assessment, you'll see that we are asking you to prepare a series of possible spacecraft *configurations*, rather than architectures. A configuration is typically a 3D sketch (or in the real world some form of 3D rendering) of what the system might actually end up looking like. If you revisit the case study video 3 for functional analysis, each time an individual system is shown (e.g. Perseverance rover, the Mars Sample Return Orbiter, etc.) you are observing a system configuration.

The step we take to go from a system architecture to a system configuration requires us to visualise what the hardware might actually look like. We will have a go at this altogether in the live session (*yes I'll be handing out big sheets of paper again*). Some useful suggestions when assembling system configurations:

1. Label the configuration where necessary. *Sure, you may know what a propellant tank capacitance probe might look like* - but its likely your stakeholders will not.
2. Use this stage to explore the design space. It will force you to make a list of unanswered questions that you will then iterate on in a real-world systems engineering process.

3. There are many many many possible configurations associated with a given system architecture. At a minimum, consider multiple system configurations and don't be afraid to try some really wild/unusual/novel/'out-there' configuration suggestions!

To close the loop on the system architecture presented in Figure 17, which was the fuel system of a narrowbody airliner. Similar architecture depictions would be produced for pneumatic/bleed air and hydraulic systems across the aircraft. Figure 23 shows the landing gear bay (also known as a wheel well) of a narrowbody airliner and this highlights what a final system configuration can look like when physically realised. Believe me, there would have been many 3D sketches and visualisation of the system configurations to work out the elements and layout of the systems achieved in Figure 23.

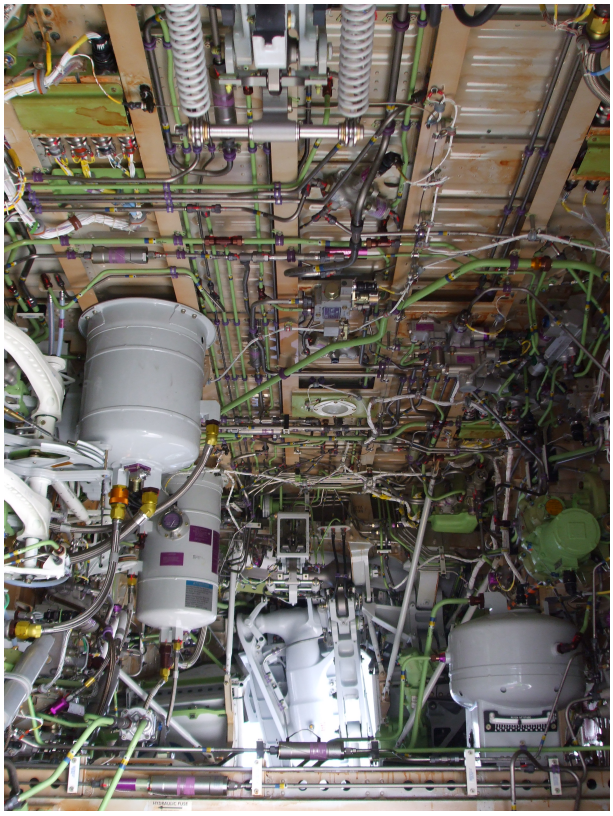


Figure 23: A narrowbody aircraft landing gear bay showing the final configuration of many different aircraft systems. RAF-YYC CC BY-SA 2.0

## Section Summary

System **ARCHITECTURES** are an abstracted view of our system that is built upon, and focuses on, the functionality of the system.

As we construct system architectures we begin to make decisions of **WHAT** system elements are required to satisfy requirements.

**INTERFACES** and specifically **standards** are the key tool that permit complex systems to be designed using systems engineering.

# References


[1] NASA Systems Engineering Handbook, NASA SP-2016-6105 Rev 2, National Aeronautics and Space Administration, 2007

[2] Systems Engineering Handbook: a guide for system life cycle processes and activities, INCOSE-TP-2003-002-03.2, *Version 3*, International Council on Systems Engineering, 2010

[3] Bonnema, G. M., Veenvliet, K. T. and Broenink, J. F., *Systems Design and Engineering: Facilitating Multidisciplinary Development Projects*, CRC Press, 2016

[4] Parkinson, B., *System Design & Management: An Introduction to System Engineering*, Hempsell Astronautics, 2020

[5] Blanchard, B. S. and Fabrycky, W. J., *Systems engineering and analysis*, 5$^{th}$ ed., Pearson, 2011

[6] The Accreditation of Higher Education Programmes (AHEP), *Fourth Edition*, Engineering Council, 2020

[7] Lakdawala, E., *The Design and Engineering of Curiosity: How the Mars Rover Performs Its Job*, Springer, 2018

[8] Welch, R., Limonadi, D. and Manning, R., "Systems Engineering the Curiosity Rover: A Retrospective", 8$^{th}$ International Conference on System of Systems Engineering, pp. 70-75, 2013

[9] The Planetary Exploration Budget Dataset, The Planetary Society (planetary.org) [Accessed 27/08/2024]

[10] Artemis III Science Definition Team Report, NASA/SP-20205009602, National Aeronautics and Space Administration, 2020

[11] Creech, S., Guidi, J. and Elburn, D., "Artemis: An Overview of NASA's Activities to Return Humans to the Moon", 2022 IEEE Aerospace Conference (AERO), 2022

[12] Planetary Science Decadal Survey - MSR Orbiter Mission (including Mars Returned Sample Handling) Mission Concept Study, National Aeronautics and Space Administration, 2010

[13] Martin-Mur, T. J., Kruizinga, G. L., Burkhart, P. D., Abilleira, F., Wong, M. C. and Kangas, J. a., "Mars Science Laboratory Interplanetary Navigation", *Journal of Spacecraft and Rockets*, vol. 51, no. 4, pp. 1014-1028, 2014

[14] Margin philosophy for science assessment studies, SRE-PA/2011.097/, European Space Agency, 2012

[15] Karpati, G., Hyde, T., Peabody, H. and Garrison, M., "Resource Management and Contingencies in Aerospace Concurrent Engineering", AIAA SPACE 2012 Conference & Exposition, 2012]

[16] Mars Perseverance Press Kit, National Aeronautics and Space Administration, 2021

[17] Menchinelli, A., *et al.*, "A Reliability Engineering Approach for Managing Risks in CubeSats", *aerospace*, vol. 5, no. 4, 2018

[18] NASA Specifications and Standards, NASA SP-9000, National Aeronautics and Space Administration, 1967

[19] Dodge, F. T., "Propellant Mass Gauging: Database of Vehicle Applications and Research Development Studies", NASA/CR - 2008-215281, National Aeronautics and Space Administration, 2008