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.

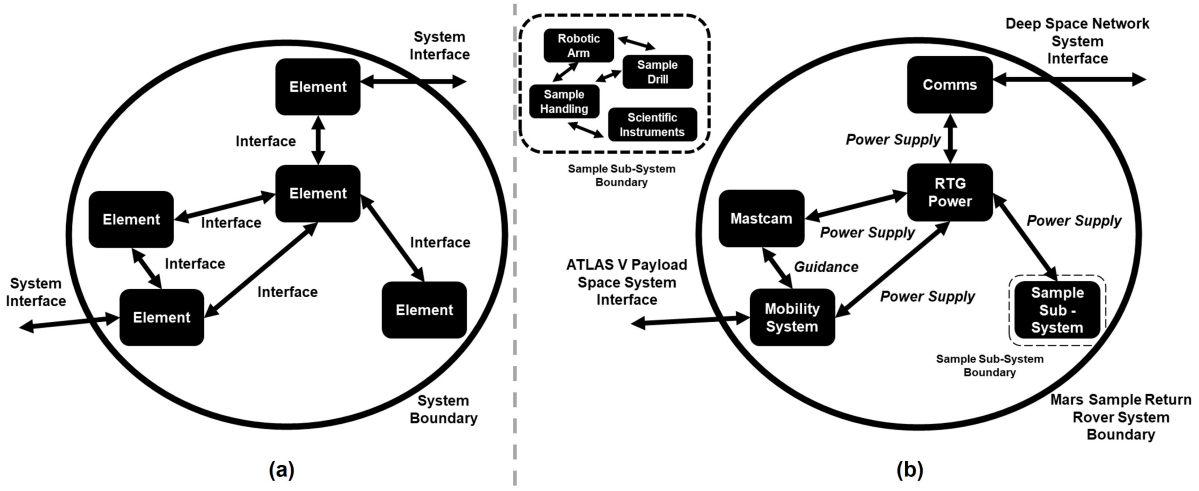


Figure 1: The system model in a) abstract form and b) loosely based on the Perseverance rover

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.

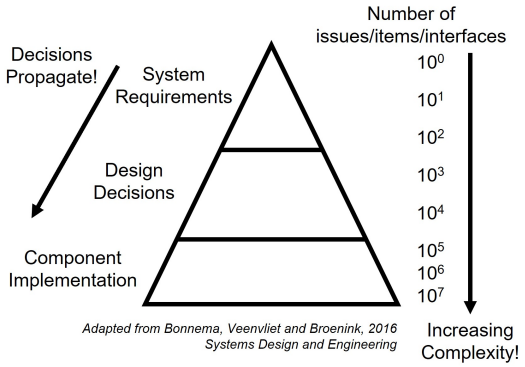


Figure 2: The 'Muller' pyramid showing the increase in complexity as a project progresses, after [3]

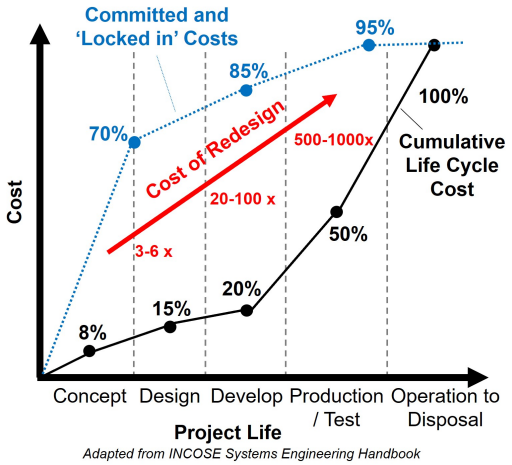


Figure 3: Costs throughout the project lifecycle, after [2]

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

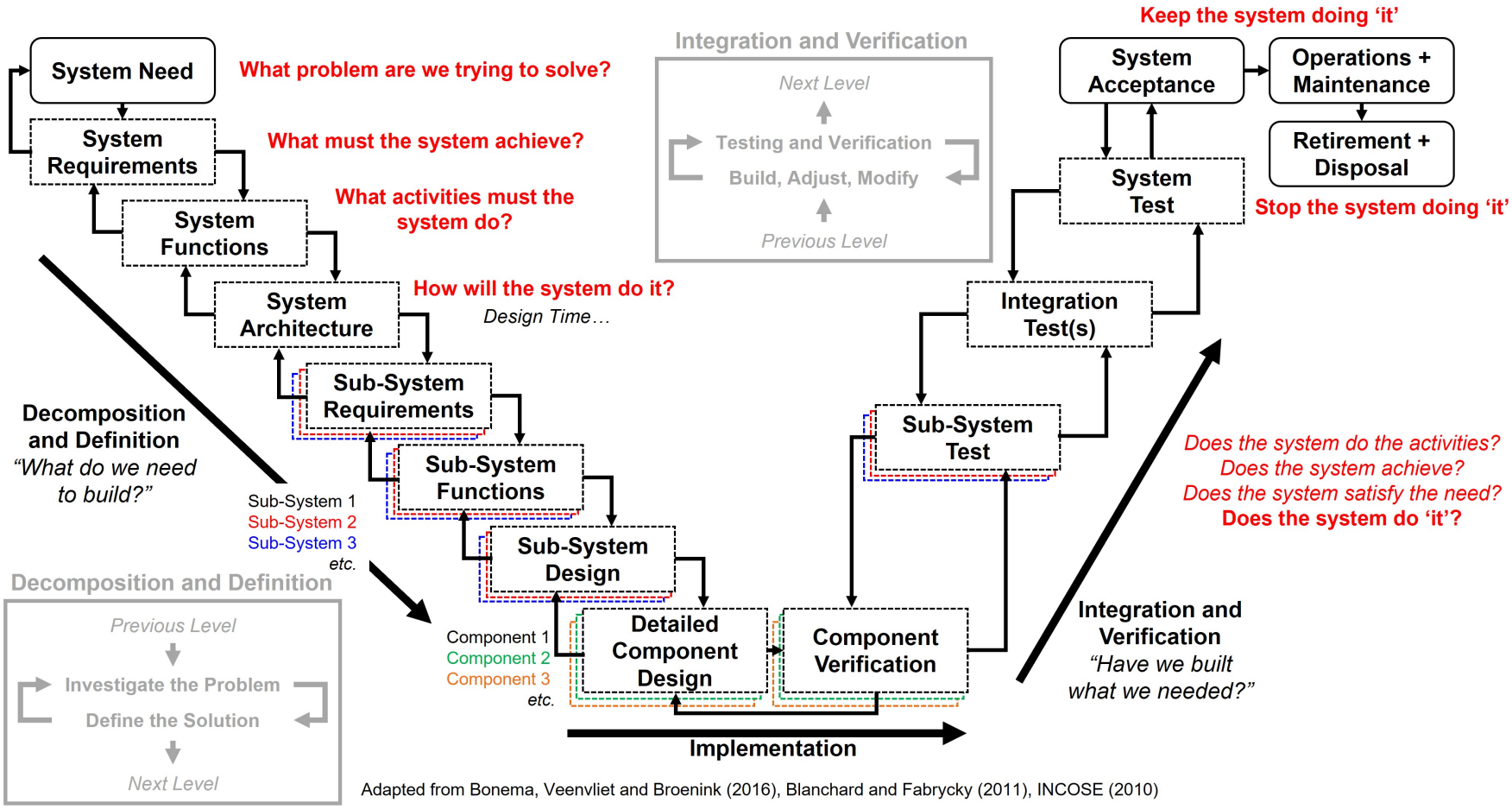


Figure 4: An overview of the V-model for systems engineering, adapted from [2, 3, 5]

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.

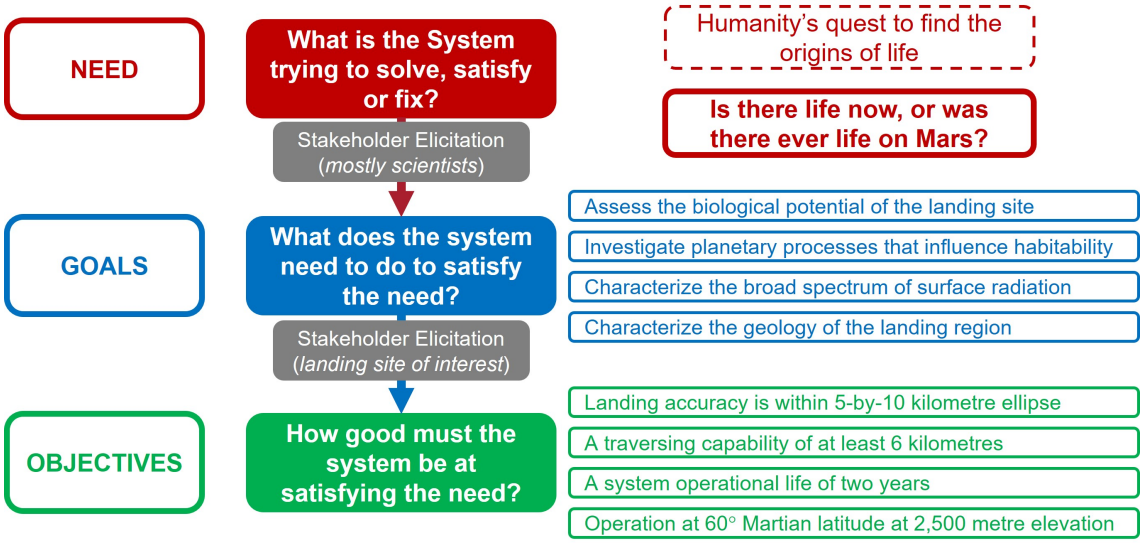


Figure 5: The evolution of a system need to a set of system objectives, using the Curiosity rover as an example [7]

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'.

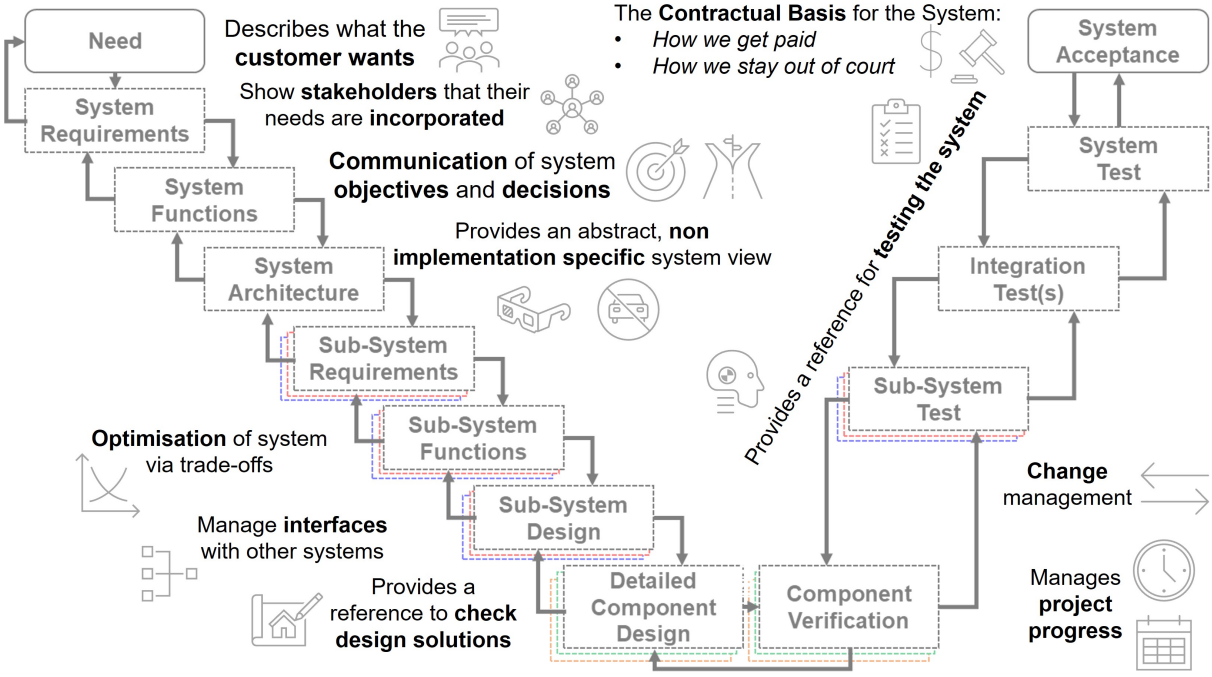


Figure 6: The continual use of requirements throughout the system lifecycle

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:

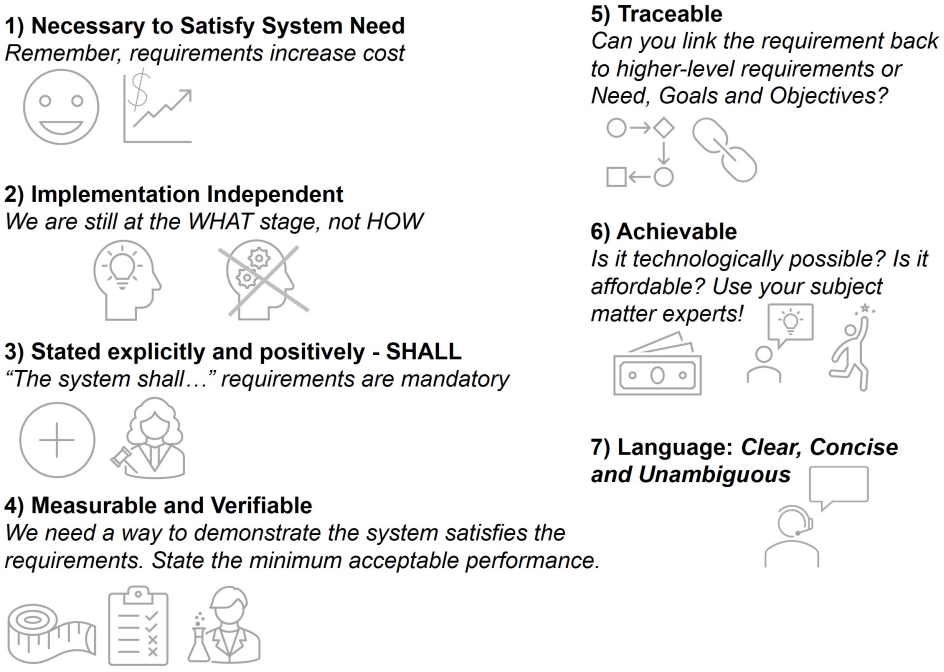


Figure 7: The requirements of writing engineering requirements

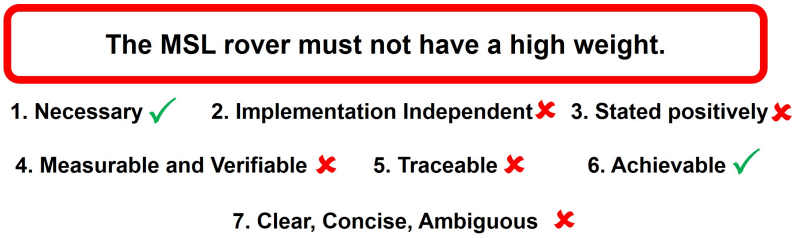
Shall is a five letter word that carries a huge amount of importance in requirements. 'Shall' means that whatever is included in the requirement text is mandatory and the system must satisfy the requirement. 'Shall' is often confused with:

- *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.

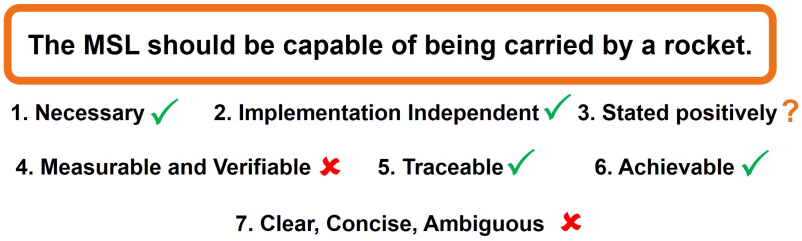
Figure 7 also states that the language within the requirement should be clear, concise and unambiguous. Now not many of us become engineers because we like writing, so here are some elements to **avoid** when writing requirements:

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))

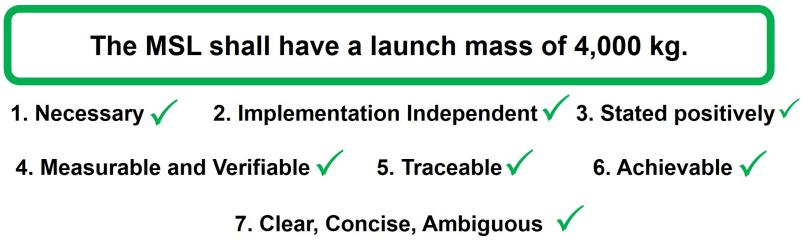
Now I appreciate that I've just given a load of rules. Whilst I am not the rebelious type, the easiest way to understand these rules however is to see what happens when we break them, so it is time to look at the *the good, the bad and the ugly* of written requirements. We'll look at three example requirements 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.

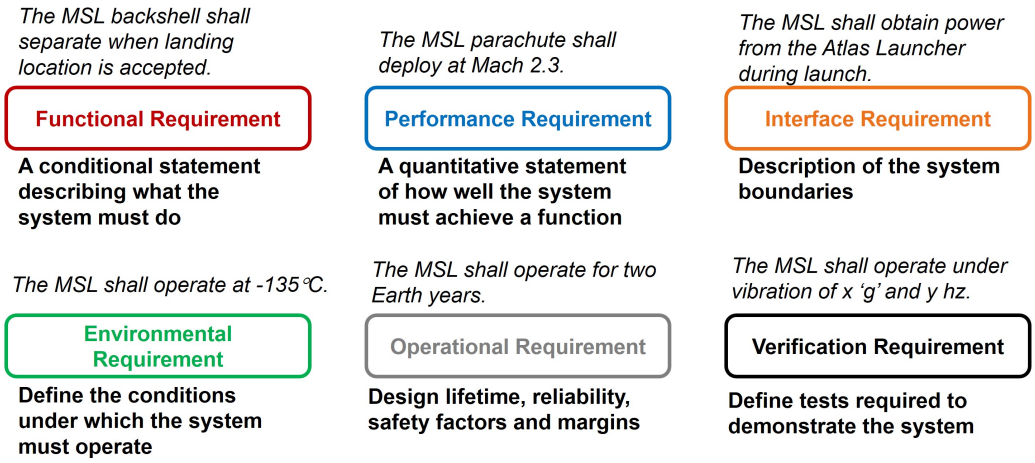


Figure 8: The different types of requirement

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.

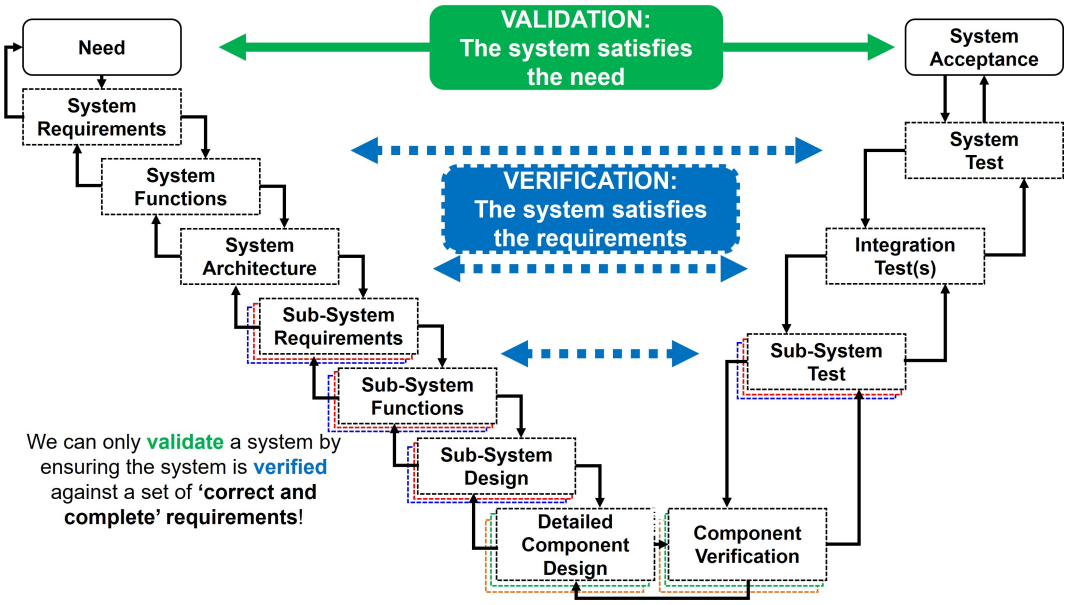


Figure 9: The location of verification and validation activities in the V-model

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!

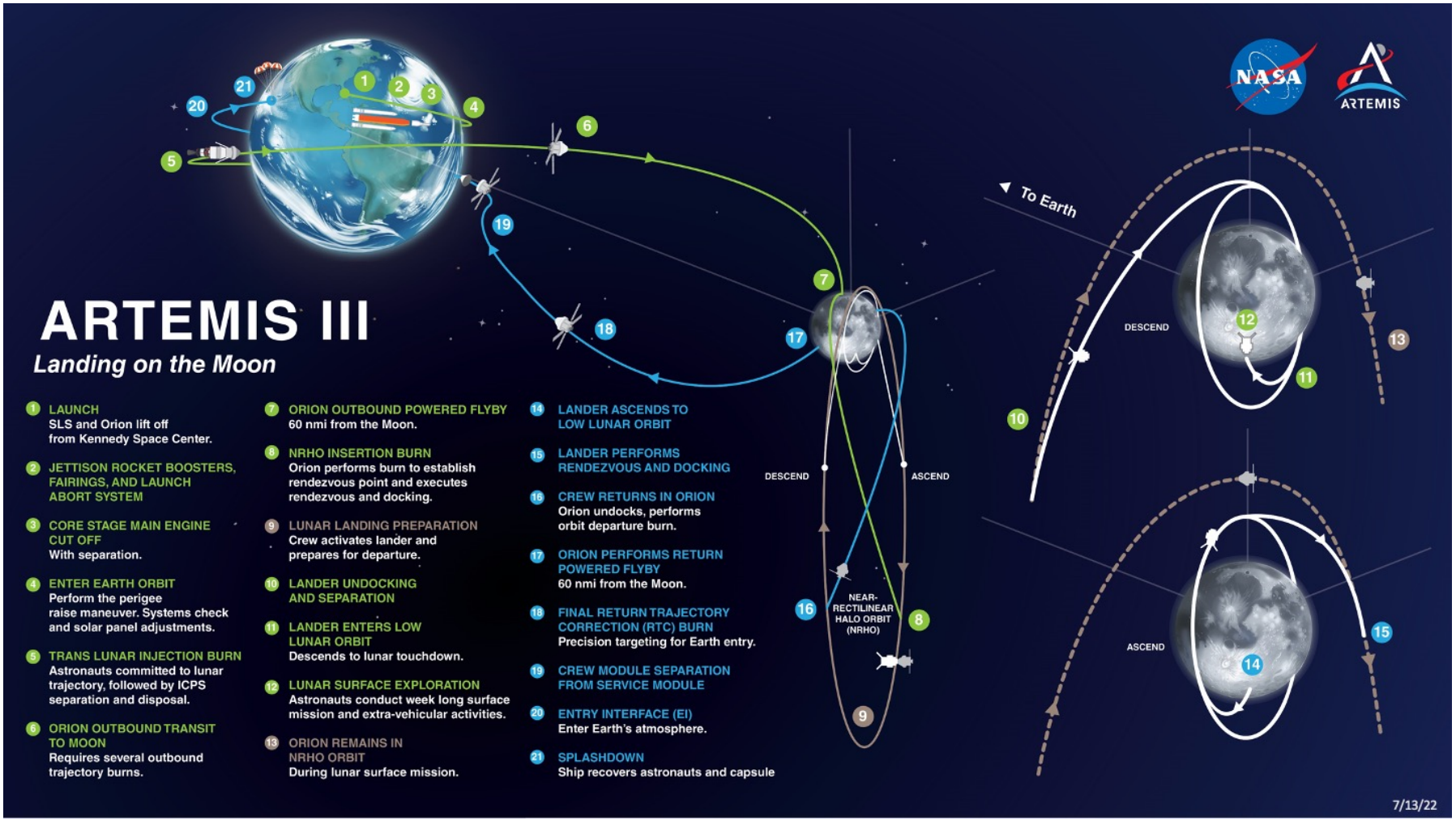


Figure 10: Visual CONOPS of the Artemis III Mission under nominal operation. *Image Courtesy of NASA, Public Domain*

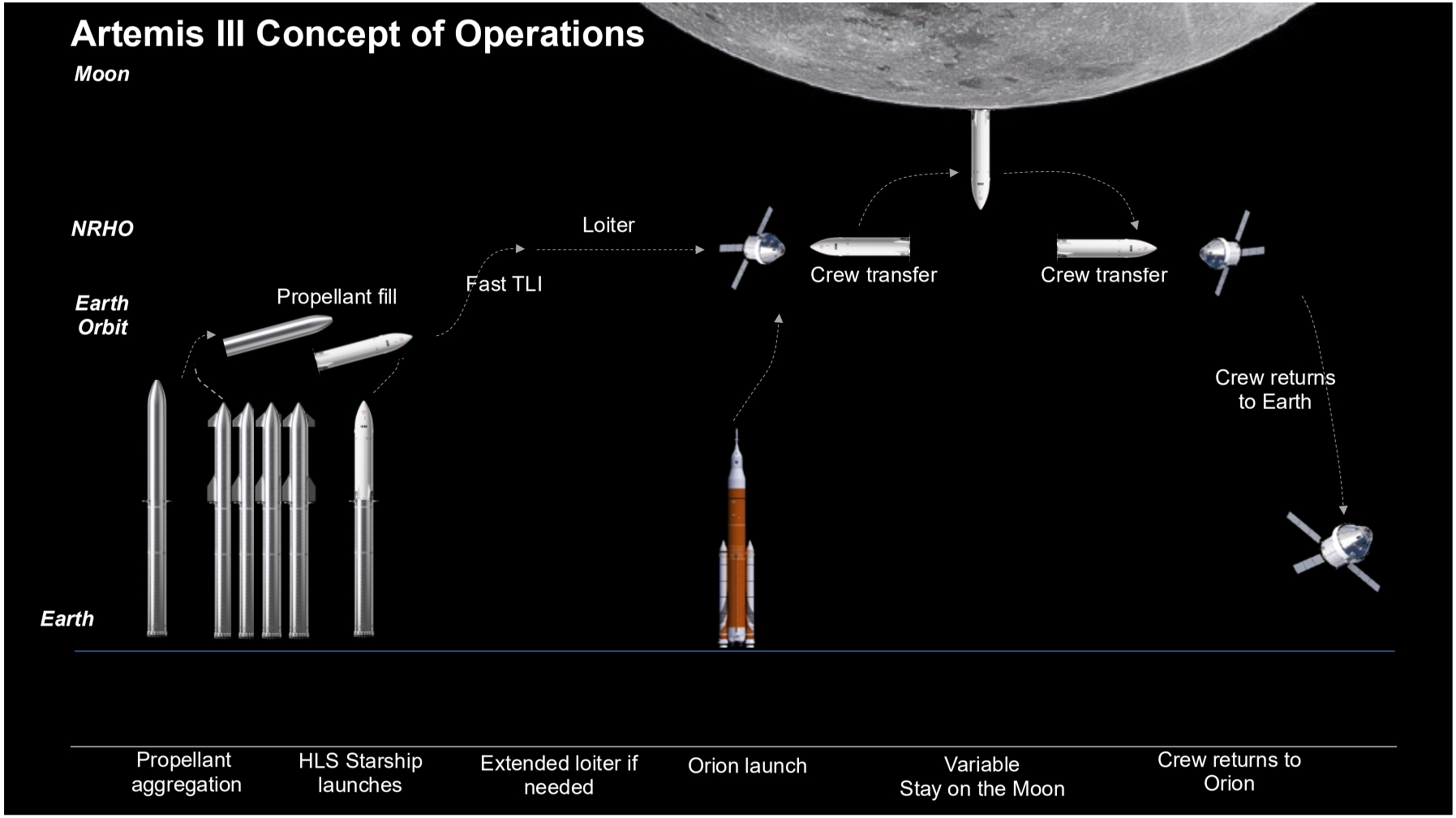


Figure 11: Visual CONOPS of the Artemis III Mission from the Lunar Lander perspective. *Image Courtesy of NASA (Kent Chojnacki), Public Domain*

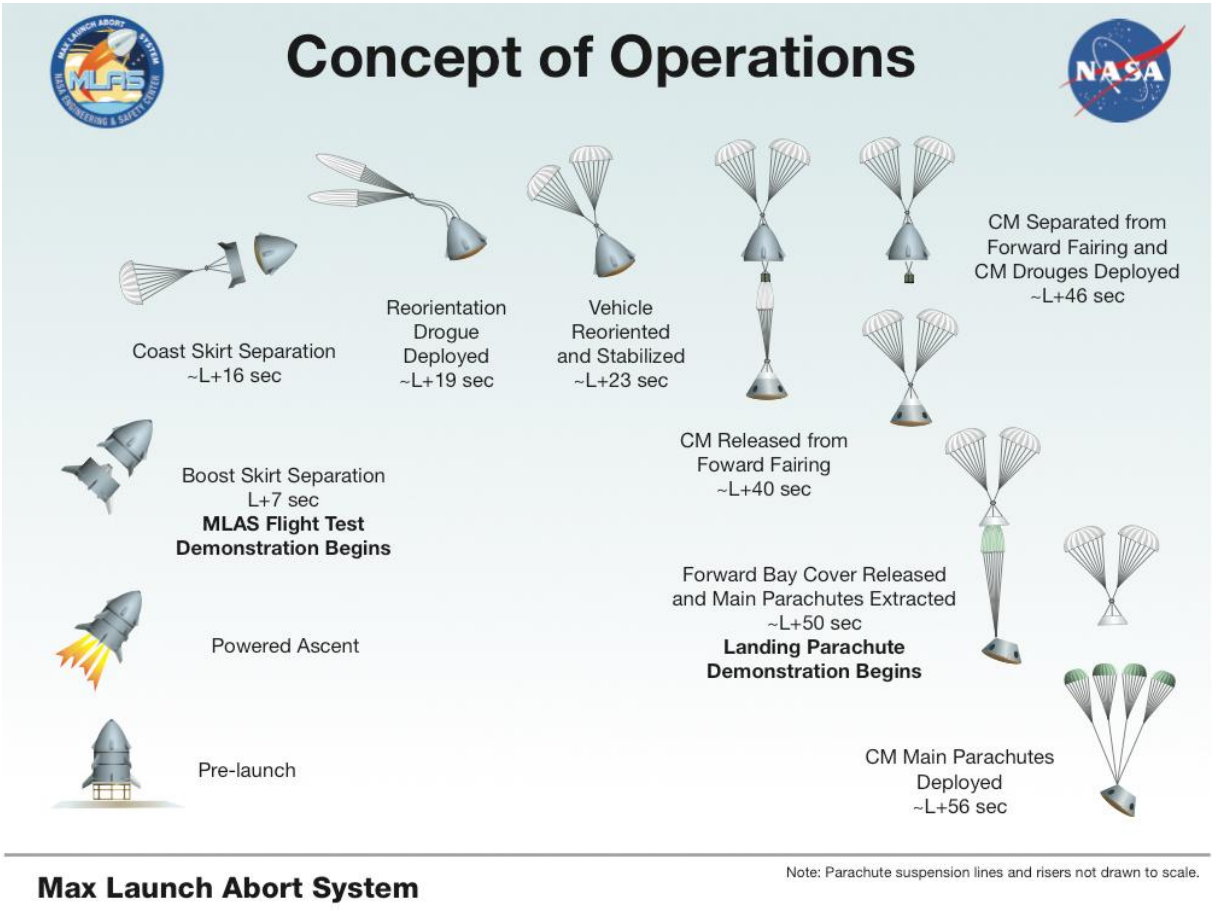


Figure 12: Visual CONOPS concerning the test of the original Maximum Launch Abort System for the Orion spacecraft. *Image Courtesy of NASA, Public Domain*

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.

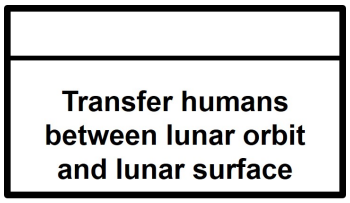


Figure 13: An example of a functional block

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.

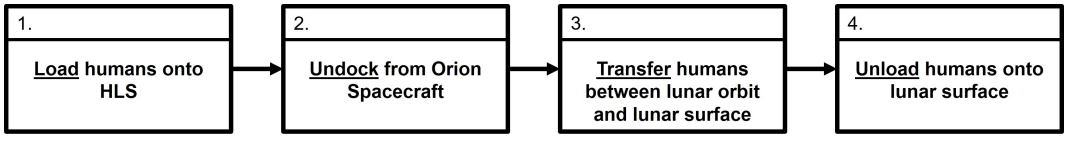


Figure 14: An example of a simple Functional Flow Block Diagram (FFBD)

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.

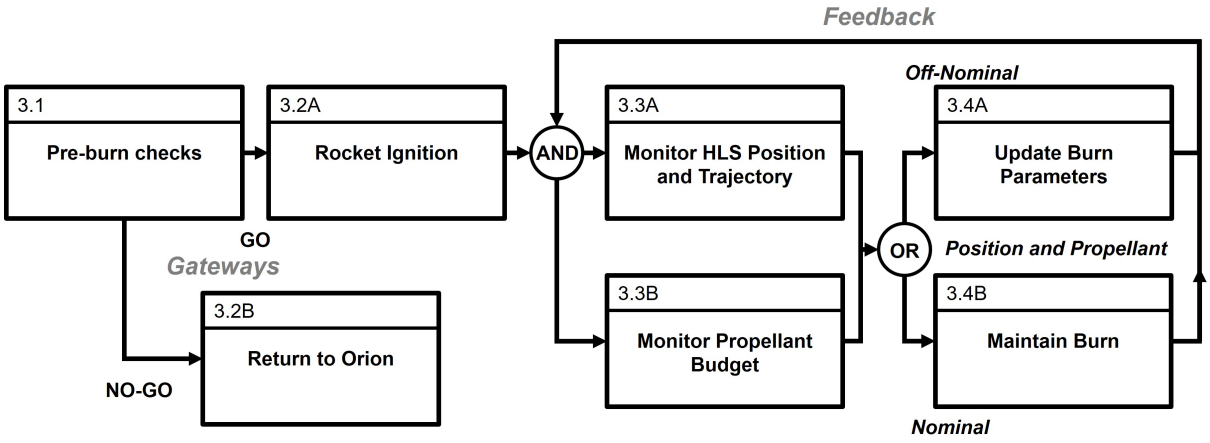


Figure 15: An example of an FFBD that uses gateways, parallel functional paths, conditional functional paths and feedback)

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

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*, 5th 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", 8th 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