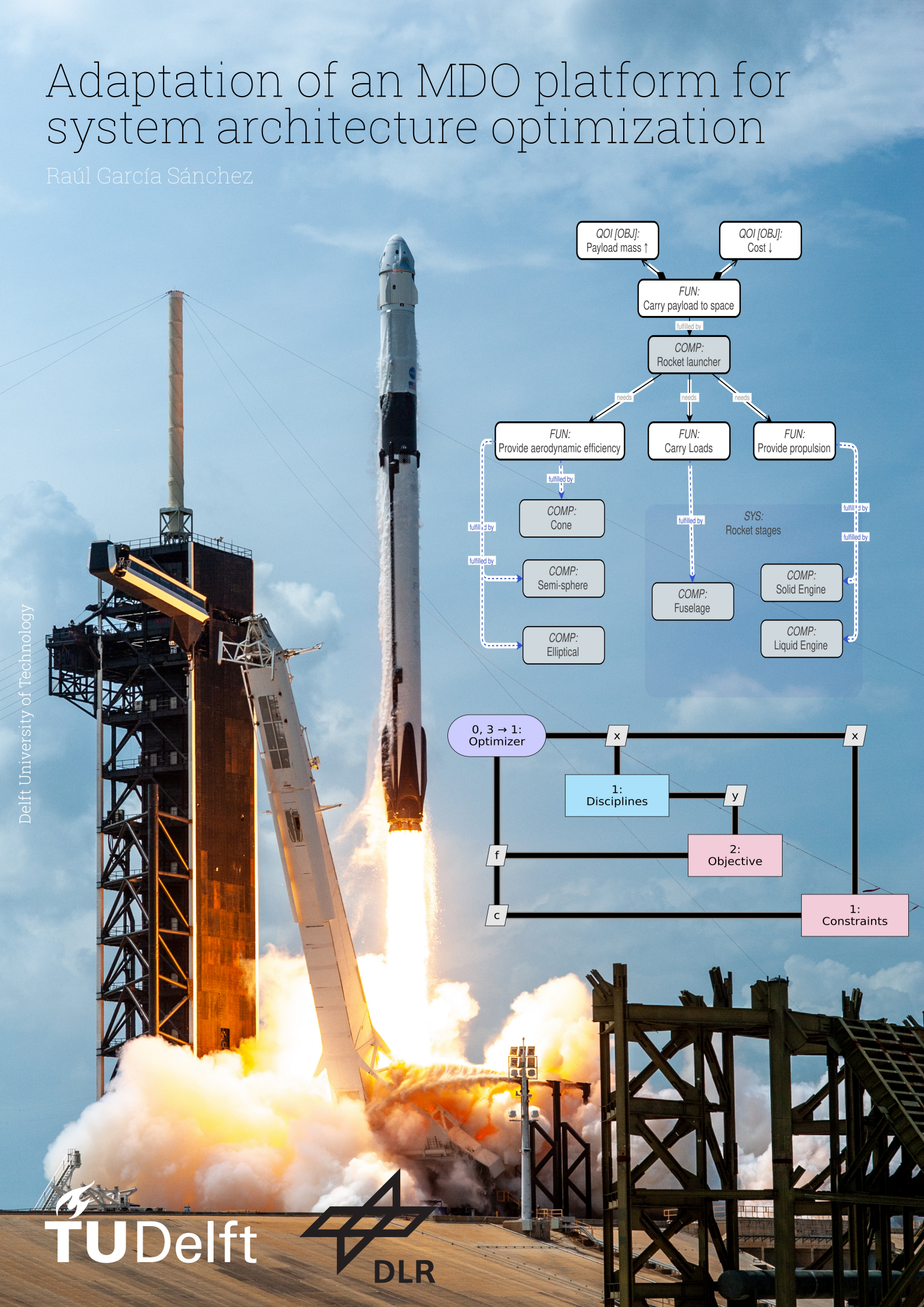
Adaptation of an MDO platform for system architecture optimization

Raúl García Sánchez



Delft University of Technology

TU Delft

DLR

Adaptation of an MDO platform for system architecture optimization

by

Raúl García Sánchez

to obtain the degree of Master of Science in Aerospace Engineering
at the Delft University of Technology,
to be defended publicly on Friday January 26, 2024 at 9:30 AM.

Student number:	5628458	
Project duration:	March 1, 2023 – December 15, 2023	
Supervisors:	Dr. ir. G. La Rocca,	TU Delft
	J. H. Bussemaker,	DLR
	S. Garg	DLR
Thesis committee:	Dr. M. Pini	chair
	Dr. ir. G. La Rocca,	Main TU Delft supervisor
	J. H. Bussemaker,	DLR supervisor
	Dr. J. Guo,	Examiner

Preface

This is the last chapter to be written, not only of this thesis, but also of this amazing story in TU Delft. After these 2 years (and a half) experience I have learnt lots of thing I did not even knew they existed, and what is more important, I did it while enjoying (most) of the process. This chapter is to thank everyone who made it possible.

First, I have to start with my whole family. My parents have been supporting me from the start, and sometimes I forget how fortunate I am of having them with me. I would not be here (literally) if it was not because of them. Also my brother/sister helped me a lot, making me laugh even being thousands of kilometers away. I would also like to mention here my godparents, who have been always there when I needed them, and were so important in starting my curiosity for science. The support of my cousins has also been essential to be successfully in this challenging project.

Another key has been my friends. Although I could not spend as much time as I wanted with my all-life friends in Spain, the good times we have had together were really helpful to always go ahead and continue. However, what I could not imagine is that I could make so many new friends here in Hamburg. I would like to thank them for all the interesting conversations and good moments we had (and I hope we will have) together. Specially, I would like to mention Nik, Menno, Naz, Matheus and of course Vincenzo and Caro, for being always there.

This experience has also allowed me to meet astonishing people from the research world. First, my TU Delft supervisor, Gianfranco La Rocca. I was surprised since the first day of the master by his passion for knowledge and science. He is able to really connect with students and share his passion for what he does. I would like to thank him for all his pieces of advice during the research, and for all the nice conversations we had. It was also a pleasure to have the opportunity to discuss with Daniël de Vries, being really helpful to setup the space problem and to know more about the industry.

Also here I would like to include all my colleagues of DLR. I really loved working there. They were really supporting from the first day, and I am just willing to start researching with them. I would like to specially thank my group for all the nice moments (Pina, Luca, Adrian, Carlos, Jasamin, Francesco, ...). I also want to thank Erwin for giving me the opportunity of continuing my experience here in DLR for the next years.

Finally, I would like to end this preface with the main actors in the success of this project, my DLR supervisors. They were always there to answer any question I had. I cannot thank enough all the time they have dedicated to me. First we have Sparsh, who has become a real friend for me. His role as supervisor has been absolutely amazing, and the experiences we shared outside of office helped me considerably to be successful at the end of the project. Finally, the last person I would like to say thanks is my other supervisor, Jasper. He is responsible for returning back to me the passion for aerospace that I thought I had lost. I will always thank him all my life for the things he has taught me and for his support.

Raúl García Sánchez
Hamburg, December 2023

Summary

Early design decisions have a significant influence in the final success of the project. One of the most important decisions is to determine the system architecture, as it highly impacts the performance of the system. System architecture optimization can be used to determine the best possible architectures through the formulation of an optimization problem, allowing to explore the design space without traditional bias and conservatism.

MDO can be used to evaluate the performance of each architecture, allowing to consider the interactions between the multiple and coupled disciplines involved in the design process. To do this, MDO platforms have to satisfy multiple requirements, including the automatic readjustment of the MDO problem for each system architecture. They also have to be adapted to collaborative MDO, so that they can be used in real industrial cases.

Before this research, there was not MDO platform that satisfied all these requirements, impeding the integration of system architecture optimization in the industry. The MDO platform consisting on MDAx and RCE was adapted to collaborative MDO and satisfied all requirements to be used as architecture evaluator, except the automatic readjustment of the MDO problem. To fill the previous gap, the main objective of this research has been to extend MDAx backend code to allow the formulation of these dynamic MDO problems, allowing to use it in the system architecture optimization process.

To achieve this, first the possible modifications that the system architectures can cause in the MDO problem, called architectural influences, are determined. Then, some possible implementation strategies MDO platforms can use to deal with these influences are presented. After that, the actual implementation process used to extend MDAx backend code is widely discussed.

Afterwards, a benchmark problem based on Fourier series is used to verify the implementation. A real engineering problem, based on the design of a space multistage rocket, is also used as validation to show the potential tool, and more generally, of the methodology. Finally, some conclusions and possible future steps are drawn.

In conclusion, this research allows to reduce the existing gap between system architecture optimization and MDO by obtaining an MDO platform that can be used as an architecture evaluator. Also, the different requirements identified for the inclusion of architectural influences, as well as the benchmark problems discussed, are aimed to help developers to extend their MDO platforms to be adapted to system architecture optimization, reducing the barriers for its implementation in the industry.

Contents

List of Figures

List of Tables

Nomenclature

Abbreviations

Abbreviation	Definition
ADS	Architectural Design Space
ADSG	Architectural Design Space Graph
AIAA	American Institute of Aeronautics and Astronautics
CDS	Central Data Schema
CMDOWS	Common MDO Workflow Schema
CPACS	Common Parametric Aircraft Configuration Schema
DAU	Defense Acquisition Univ
DLR	Deutsches Zentrum für Luft- und Raumfahrt e.V
DSM	Design Structure Matrix
GUI	Graphical User Interface
I	Architectural Influence
INCOSE	International Council on Systems Engineering
IRT	Technological Research Institute
ISAE	Institut Supérieur de l'Aéronautique et de l'Espace
JSON	JavaScript Object Notation
KADMOS	Knowledge and graph-based Agile Design for Multi-disciplinary Optimization System
LEO	Low Earth Orbit
MBSE	Model-Based Systems Engineering
MDAx	MDAO Workflow Design Accelerator
MDO	Multidisciplinary Design Optimization
NASA	National Aeronautics and Space Administration
NSGA-II	Nondominated Sorting Genetic Algorithm II
ONERA	Office national d'études et de recherches aérospatiales
OpenLEGO	Open-source Link between AGILE and OpenMDAO
QOI	Quantity of Interest
RVF	Requirement Verification Framework
RCE	Remote Component Environment
SAO	System Architecture Optimization
SBO	Surrogate-Based Optimization
SSBJ	Supersonic Business Jet
XDSM	eXtended Design Structure Matrix
XML	Extensible Markup Language

Symbols

Symbol	Definition
AR	Aspect ratio
Batt.	Batteries
C_d	Drag coefficient
C_f	Skin Friction coefficient
D	Drag (N)
dpdx	Pressure ratio
D_T	non-Dimensional Throttle setting

Symbol	Definition
ESF	Engine scale factor
h	Altitude (ft)
L	Lift (N)
L/D	Lift to Drag ratio
l_w	Winglet length (m)
M	Mach number
MTOW	Maximum Takeoff Weight (N)
N_z	Maximum load factor
Prop.	Propeller
R	Range (m)
SFC	Specific Fuel Consumption
S_{ref}	Wing reference surface (m^2)
T	Thrust (N)
Temp	Temperature (K)
t/c	Thickness to chord ratio
W_{BE}	Baseline Engine weight (N)
W_E	Engine weight (N)
W_F	Fuel weight (N)
W_{FO}	Miscellaneous fuel weight (N)
W_O	Miscellaneous weight (N)
W_T	Total weight (N)
x (SSBJ)	wingbox section height (m)
Y_{ribs}	Location of wing ribs in the y-axis (m)
δ	Winglet cant angle (°)
Λ	Wing sweep (°)
λ	Taper ratio
σ	Stress (Pa)
Θ	Wing twist (°)

1

Introduction

The decisions taken in the early phases of the design process have a great influence on the final success of the project (Wichmann et al., 2015). One of the most important tasks of the early design phase is to determine the architecture of the system, which can be defined as a description of the different components of the system and the relationships between them (Crawley et al., 2015), as it has a large effect on the whole performance of the system (Bussemaker and Boggero, 2022).

The problem is that even with a low number of architectural decisions, a combinational explosion of alternatives appears (Iacobucci, 2012). This means that it is impossible to create all the architectures for a given problem and evaluate all of them. What has been done traditionally is to select a small number of architectures based on the knowledge of experts to enter the conceptual and preliminary design phase. However, this approach may lead to bias and conservatism (Roelofs and Vos, 2018) and is becoming more challenging to apply due to the increase of complexity of aerospace projects over the last decades (DeTurris and Palmer, 2018). Furthermore, it does not allow to completely explore the design space, leading to solutions far away from the optimum.

System architecture optimization can be applied to overcome these hurdles (figure 1.1). It allows to systematically explore the design space without having to evaluate all the architectures (Bussemaker and Ciampa, 2022), taking into account the interactions between the different components of the system. To carry out system architecture optimization, an architecture evaluator is needed, allowing to provide quantitative feedback of the performance of each system architecture to the optimizer. To accurately determine the performance of each architecture, Multidisciplinary Design Optimization (MDO) can be used (Sobieszczanski-Sobieski et al., 2015), so that the interactions between the different design disciplines are taken into account.

MDO platforms, defined as pieces of software used to formulate and execute MDO problems, could be used to evaluate the different architectures using MDO. There are three main challenges that MDO platform have to deal with when used as evaluators for system architecture optimization. First, they must be able to deal with mixed-discrete variables. Second, they must implement/be able to integrate complex optimization algorithms. The third and main one is related with the fact that each architecture has different components and connections between them. This means that the design variables of these architectures and the constraints that they have to satisfy will vary from an architecture to another. Furthermore, even the design disciplines involved might change.

As a consequence, to include MDO in system architecture optimization, it is important that MDO platforms support to readjust the MDO problem automatically (changing the disciplines, connections and design variables of the MDO problem) depending on the architecture that is being evaluated. There are two approaches MDO platforms can follow to deal with this. The first solution is to generate automatically a different MDO problem for each system architecture. The second approach is to generate an unique MDO problem that can be readjusted automatically during the execution process, depending

on the system architecture being analysed.

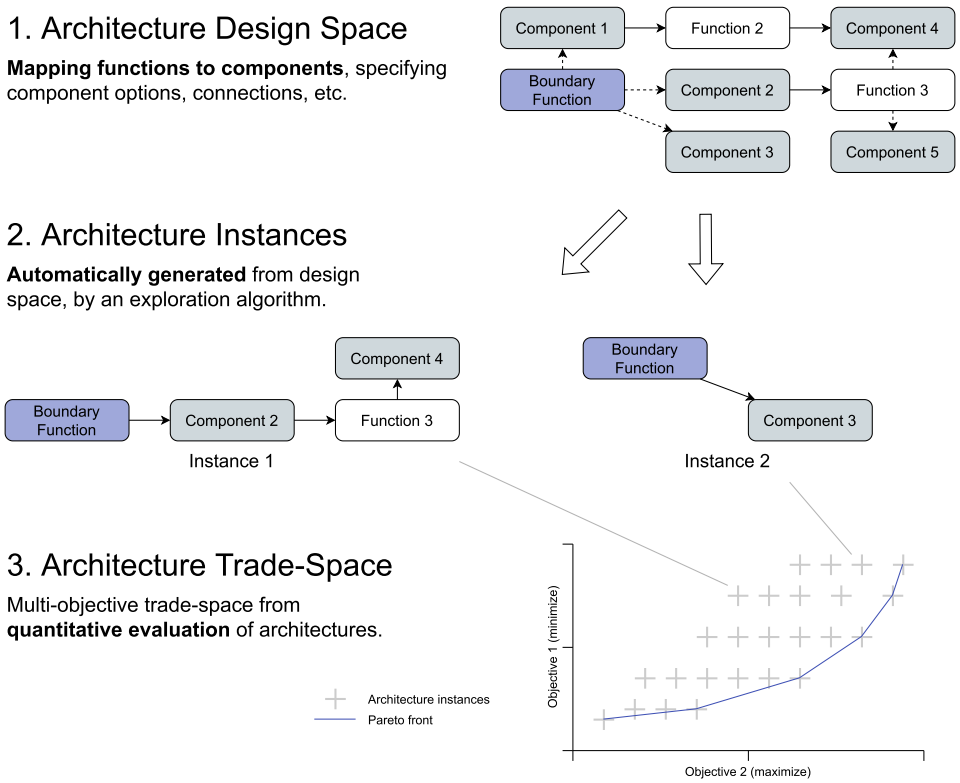


Figure 1.1: Example of the system architecture optimization process. First all the possible architectures are defined. After that, the optimizer will generate different architectures and evaluate them, deciding the architectures being generated depending on the quantitative feedback it obtains during the optimization process. Finally, the solution (usually in the form of a Pareto front) is obtained. Figure taken from Bussemaker and Ciampa, 2022.

When MDO has to be used in real industrial projects, it is common that the tools needed for the MDO problem execution belong to different departments, or even to different companies. As a consequence, multiple experts from different fields and backgrounds are needed to be integrated together. To coordinate all these experts, so that the MDO problem can be formulated and executed efficiently, MDO platforms have to incorporate the methodologies encompassed by collaborative MDO (Ciampa and Nagel, 2016).

To achieve this, they need to help and guide designers for the formulation of the MDO problem, without the necessity of knowing beforehand all the variables, disciplines and connections that exist in the problem. This can be done including some capabilities such as collisions warning and resolution methods, or through the inclusion of a simple graphical user interface (GUI), which allow designers to propose and try different problem formulations. From the execution point of view, collaborative MDO also demands some capabilities from MDO platforms, such as the protection of confidential data or the real time supervision of the problem execution by the different experts (section 2.3.1).

If system architecture optimization is desired to be applied in the industry, it is necessary to obtain an MDO platform adapted to collaborative MDO, but also capable of dealing with the requirements needed to be used as an architecture evaluator. There are some platforms that already include the methodologies encompassed by collaborative MDO, such as the union of KADMOS (van Gent and La Rocca, 2019), OpenLEGO (de Vries et al., 2017) and OpenMDAO (OpenMDAO, 2019), or the union of MDAx (Page Risueño et al., 2020) and RCE (Boden et al., 2019). However, at the beginning of the project there was no MDO platform that could deal with the automatic readjustment of the MDO problem on its own, existing a gap in the implementation of MDO in system architecture optimization, specially when applied to industrial cases.

1.1. Thesis objective and methodology

MDAx is an MDO problem formulation platform specially tailored for collaborative MDO (section 3.3.3). It guides the problem formulation process though several tools, such as its GUI. It also allows to try multiple configurations of the MDO problem in a simple manner, thanks to the graphic problem definition based on Pate et al., 2014. However, MDAx is not able to execute the MDO problem itself, it needs an additional platform for this task. One of the options is RCE, which allows to execute the problem according to collaborative MDO principles, such as remote execution or real time supervision (section 3.3.3).

The MDO platform consisting on MDAx and RCE satisfies the first requirement to be used for architecture evaluation, dealing with mixed-discrete variables, as they are based on XML. This platform also satisfies the second requirement, as RCE already includes complex optimization algorithms able to deal with system architecture optimization problems. The platform also allows to perform collaborative MDO.

However, there is still a missing requirement it has to satisfy to be used for system architecture optimization, and it is readjusting the MDO problem automatically for each system architecture. Generating a different MDO problem for each system architecture is not possible, as once MDAx export to RCE is done, some manual steps are needed before execution. This means that it is necessary that the MDO problem generated by MDAx already include all the information regarding possible modifications in the disciplines, connections and design variables of the MDO problem, so that the MDO problem can be readjusted during the execution process.

To obtain an MDO platform that can be used for system architecture optimization, and therefore reduce the existing technological gap, the objective of this thesis will be **to adapt MDAx to be used as an architecture evaluator, by extending its backend code to model and export MDO problems that can be readjusted automatically during the optimization process, depending on the architecture being analysed.** Achieving this objective would be an important step forward in the inclusion of system architecture optimization in the industry, accomplished by reducing the gap in the implementation of MDO in system architecture optimization.

To do so, the following methodology will be used. In the first part of the thesis, the possible modifications in the MDO problem due to the different system architectures will be determined, as well as some possible strategies to deal with them. Then multiple requirements will be derived to adapt MDAx to deal with these modifications. After that, the implementation of all these new features will be tested using a benchmark problem that includes all the possible modifications. Finally, a realistic case system architecture optimization problem will be solved to validate the system.

1.2. Thesis structure

This section introduces the different chapters of the thesis, which are:

- **Systems engineering and MDO:** In this chapter, an introduction to the fields of systems engineering and MDO is provided. An introduction to MBSE and collaborative MDO is included too. This introductory chapter will help the reader to understand the necessity of using system architecture optimization in the industry.

- **Systems Architecture Optimization:** This chapter is aimed to provide a thorough explanation about system architecture optimization. It starts with a brief introduction of the field, including some basic definitions of the different elements needed to perform system architecture optimization. Then, the architecture generation process is introduced, including the different types of architectural decisions.

  After that, the architecture evaluation process is discussed, including the main challenges that MDO platform have to deal with when used to formulate and execute MDO problems for system

architecture optimization. Afterwards, a brief literature review about some of the most popular MDO platforms is carried out, including a discussion about their adaptability to be used as architecture evaluators within the system architecture optimization problem. Finally, the technological/literature gap is presented.

- **Architectural influences:** First, the possible modifications that can take place in the MDO problem formulation/execution due to the different system architectures will be studied, called architectural influences. These architectural influences will be obtained studying how multiple MDO problems are modified when the system architecture is not fixed.

  The second part of the chapter covers some possible approaches/implementations that MDO platforms can use to deal with the previous modifications automatically, called strategies.

- **MDAx adaptation to system architecture optimization:** This will be the main chapter of the thesis. Multiple requirements are identified to implement each of the different architectural influences. This chapter provides a wide explanation of how the existing MDAx backend code was extended to satisfy all these requirements, including the different problems that were found in the process and how they were tackled.

- **Verification & Validation:** In this section, to verify that MDAx implementation has been carried out successfully, a benchmark problem containing all the possible architectural influences will be formulated and executed using MDAx/RCE. Then a more complex system architecture optimization problem based on a real case scenario will be also solved, showing how this platform can be used to help designers with real architecting engineering design problems.

- **Conclusions & Recommendations:** This last chapter of the thesis aims to summarize the importance of this research to reduce the existing gap in the inclusion of MDO in the system architecture optimization process, specially when used for real industrial cases. After that, some recommendations are given to achieve the inclusion of system architecture optimization in the industry.

2

Systems Engineering and MDO

To understand the goals and the necessity of using system architecture optimization, this chapter introduces first the broader fields of systems engineering and MDO.

2.1. Systems engineering introduction

According to the International Council on Systems Engineering (INCOSE), ” systems engineering is an interdisciplinary approach and means to enable the realization of successful systems. It focuses on defining customer needs and required functionality early in the development cycle, documenting requirements, and then proceeding with design synthesis and system validation while considering the complete problem: operations, cost and schedule, performance, training and support, test, manufacturing, and disposal” (Walden et al., 2015).

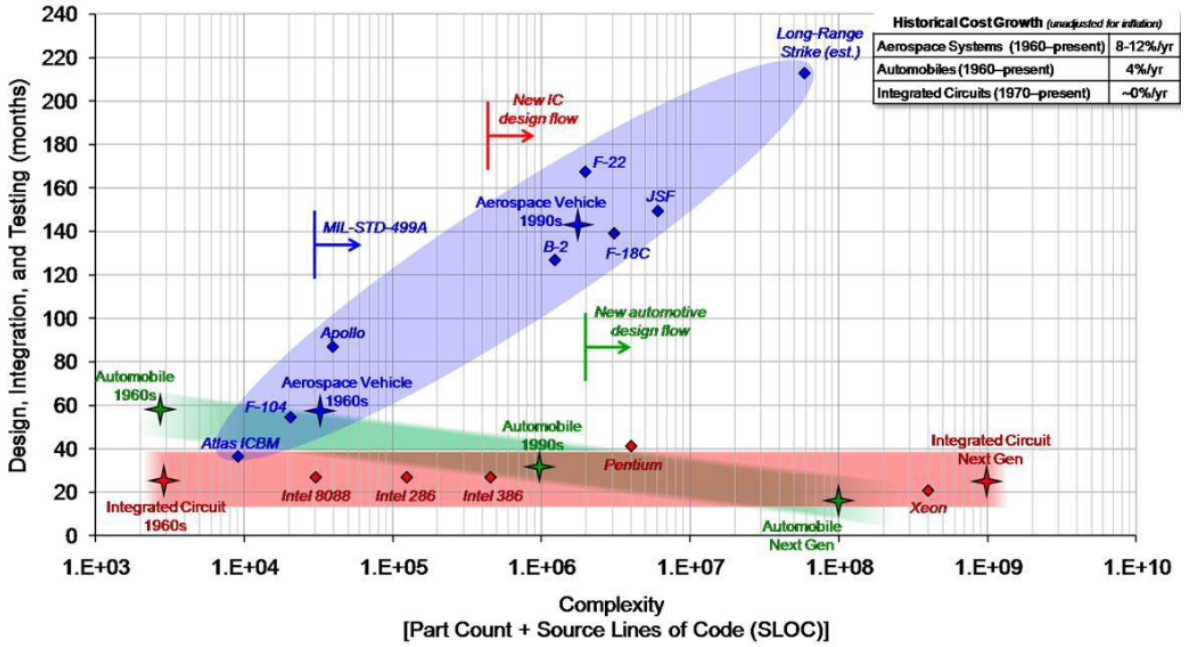


Figure 2.1: Increase of complexity (defined as number of parts and source lines of code in the system) in the last years in multiple engineering fields, including the aerospace sector. Although this description of complexity is not complete, a relationship can be established between the increase of complexity and the development time. Figure extracted from DeTurris and Palmer, 2018.

During the last decades, the number of different disciplines involved at the design is increasing, leading to an increase of complexity of the systems (DeTurris and Palmer, 2018). Complex systems are systems where there are numerous elements and multiple connections between them, especially when these connections change with time (Haberfellner et al., 2019). This increase of complexity is also the case in the aerospace sector, as observed in figure 2.1.

Systems engineering emerged in order to help engineers dealing with this increase of complexity, allowing to take into account the interactions between the different components and reducing the risk of the decisions taken at the early stages of the development phase (Haberfellner et al., 2019).

According to a statistical study carried out by the Defense Acquisition University (DAU) about the life cycle cost of projects in the US department of defence, only 8 percent of the total cost was spent on conceptual design, although this phase would determine 70 percent of the total cost of the product (DAU, 1993), as observed in figure 2.2.

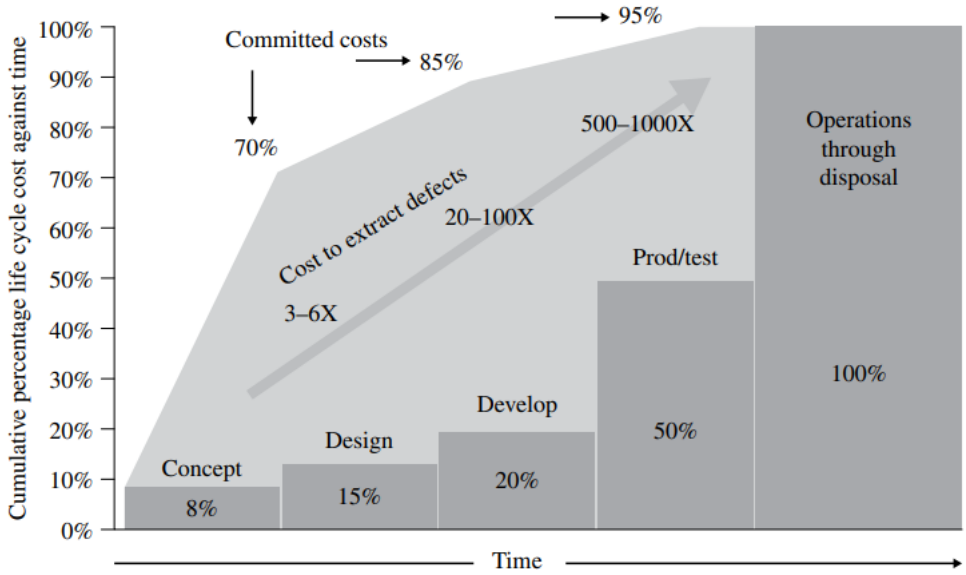


Figure 2.2: Committed life cycle cost against time. Although only 8 % of the total cost is spent to conceptual design, this phase determines 70 percent of the total product development cost. Figure extracted from DAU, 1993.

This confirms that the decisions taken at the early phases of the design process hugely influence its final success (Wichmann et al., 2015). Systems engineering can be really useful to take these decisions, allowing to consider the multiple relationships existing between the different components of the system. Traditionally, it has been based on documents. However, this document-based approach has some disadvantages, such as ambiguity, lack of clarity or poor traceability (Boggero et al., 2021).

The increase of complexity of modern products and the desire of achieving shorter development times has led to the need of introducing new methodologies for the design, verification and validation of complex systems (Broy, 2013). This, combined with the disadvantages of the document approach, is leading to the inclusion of a new methodology inside systems engineering (Chaudemar and de Saqui-Sannes, 2021), called Model-Based Systems Engineering or MBSE.

2.2. Model-Based Systems Engineering (MBSE)

MBSE has become more important inside the systems engineering field over the last years. According to INCOSE, MBSE can be defined as the ”formalized application of modeling to support system requirements, design, analysis, verification and validation, beginning in the conceptual design phase and continuing throughout development and later life cycle phases” (Friedenthal et al., 2007).

The current trend from a document-based to an MBSE approach in complex engineering fields, such as the aerospace sector (Caliò et al., 2016), is a consequence of the multiple benefits MBSE provides, such as its ability to help to understand better the complexity of the system while simplifying the communication inside the development team (Eigner et al., 2015). The introduction of MBSE also leads to the reduction of both error rates and costs (Obstbaum et al., 2017).

MBSE involves a higher investment at the first stages of the development product, as observed in figure 2.3. This can lead to an overall reduction of costs, as an improvement in the first stages of the product development is translated into a lower number of corrections at the later stages.

However, the previous reduction of cost is only true when the complexity of the product is high and when it is expected to have a long life span (Madni and Purohit, 2019), justifying the previous increase of investment in the first stages of the design.

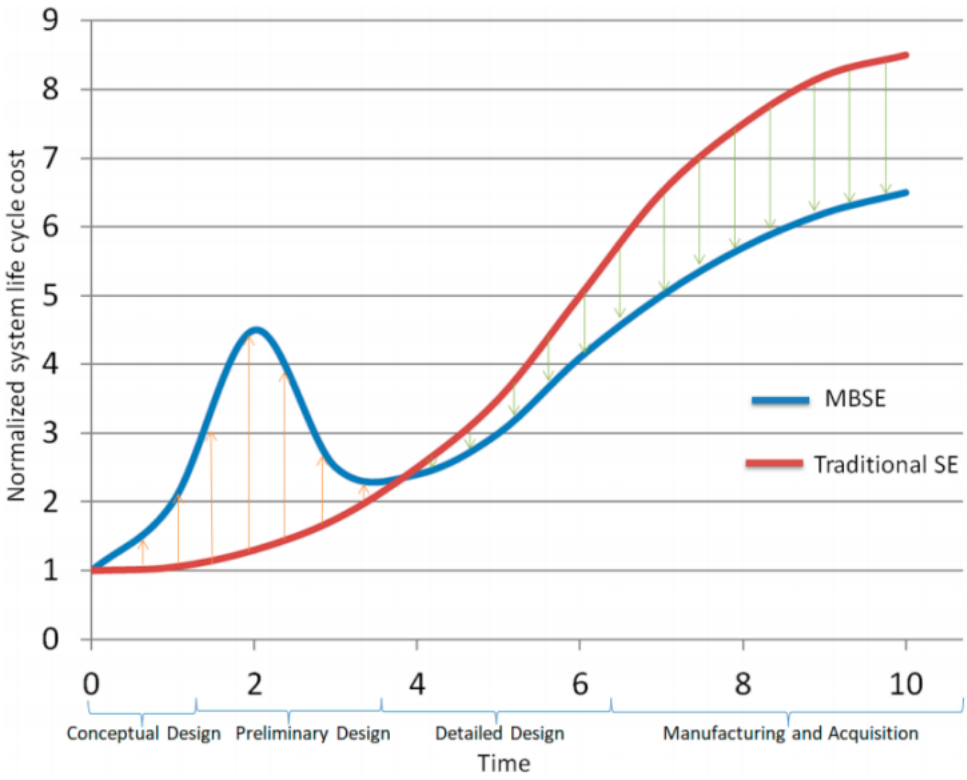


Figure 2.3: Normalized system life cycle cost versus time. These data are based on a cost statistical analyses of system engineering projects carried out by DAU. The cost in the earlier stages using MBSE is higher, although the final cost of the project reduces with respect to the traditional system engineering approach. Figure taken from Madni and Purohit, 2019.

The aeronautical industry is an ideal field for the implementation of MBSE, due to the increase of complexity of aerospace products (DeTurris and Palmer, 2018), combined with the poor integration of the horizontal and vertical levels of the current aeronautical supply chain (Ciampa et al., 2020) and the high life span of the product.

Inside the design phase step covered by this methodology (MBSE), the aerospace sector has another particularity, and it is the usual existence of multiple and highly interrelated design disciplines. To take into account the interactions they have between each other, so that an optimum solution can be obtained, Multidisciplinary Design Optimization (MDO) can be used.

2.3. Multidisciplinary Design Optimization (MDO)

Until recently, if optimization was introduced in the aerospace industry, it was carried out in a sequential manner, leading to sub-optimal designs (Martins and Ning, 2021). However, most aerospace design processes are characterized by the existence of multiple and highly coupled disciplines. If a balanced solution taking into account the contributions of all disciplines at the same time is desired, MDO can be used.

MDO can be defined as a group of methods, procedures and algorithms that are used to find the best design in complex systems whose behaviour is governed by multiple disciplines coupled between each other (Sobieszczanski-Sobieski et al., 2015). The main motivation for using MDO in the aerospace industry is that the performance of a multidisciplinary system is driven by not only the performance of the individual disciplines, but also by their interactions (Martins and Lambe, 2013). This means that the optimized solution of the whole system can only be obtained considering the interactions existing between the different disciplines.

As an example, consider the case of an aircraft. Multiple disciplines are involved in the design, such as propulsion, aerodynamics or structures. Any change in a design decision related with any of these disciplines will affect the others. All of them have to be considered simultaneously to obtain an aircraft with a good performance, and MDO can be used for that.

The MDO problem formulation is usually represented by its XDSM or eXtended Design Structure Matrix as it can be observed at figure 2.4. This is an extension of the Design Structure Matrix (DSM), where the data dependency and process flow are shown at the same time on a single diagram (Lambe and Martins, 2012). In other words, in this diagram all the variables, disciplines and how they are connected between each other are represented (data graph). This diagram also includes the execution workflow (process graph), which is the order in which the different disciplines are executed.

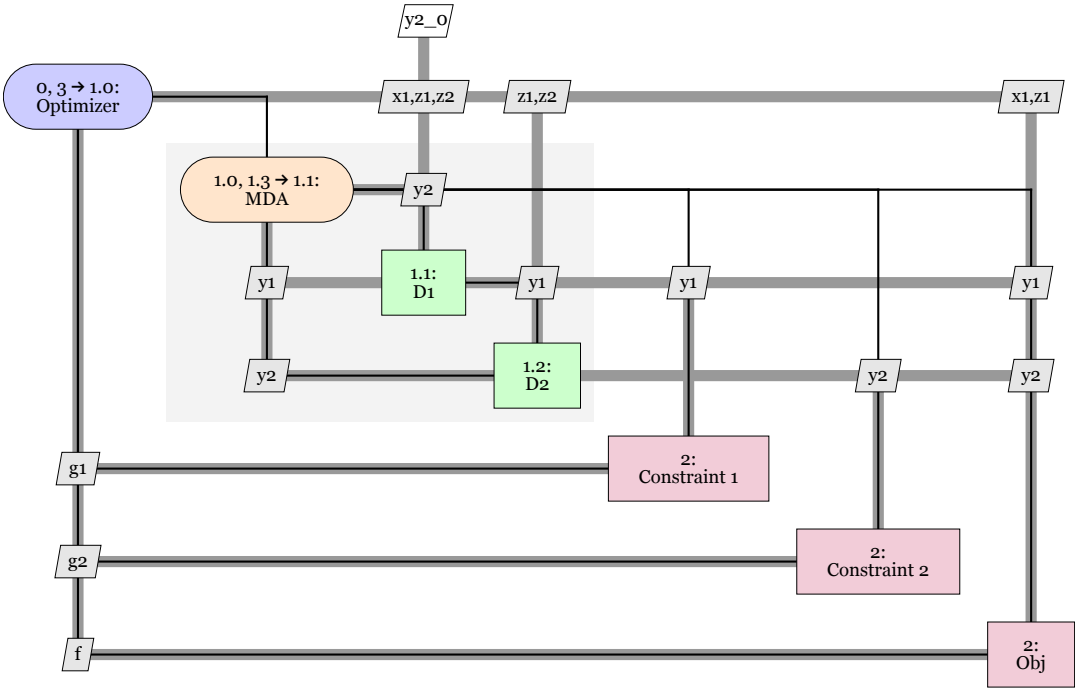


Figure 2.4: Example of an MDO problem XDSM, in this case of the Sellar problem (Sellar et al., 1996).

In the MDO process, the optimization algorithm proposes a value for each of the different design variables (this is a design vector). Then after the execution of the different disciplines, the optimizer receives the value of the objective(s) function(s) which measure quantitatively how ”good” the design vector was. Usually, some constraints are evaluated too to determine the feasibility of this design vector. With the previous results, the optimizer can propose new design vectors until an optimized solution

is obtained.

MDO has already been used in multiple engineering fields (Martins and Ning, 2021). The aerospace sector is where most examples of MDO can be found, where it has been used for different topics, from the aerostructural design of fan stages (Pokhrel et al., 2023) to the whole design of drones (Aïello et al., 2022). However, it has already been introduced in other engineering fields such as the automotive industry (Crea, 2022), the mining vehicles design (Vidner et al., 2021) or the design of wind turbines (Hegseth et al., 2020).

MDO has already started to be implemented partially in the aerospace industry. However, some difficulties arise when MDO is desired to be implemented in big industrial aerospace projects. These demand the participation of multiple experts from different disciplines (Moerland et al., 2020), companies and with different backgrounds, leading to different national, organizational and human barriers (Baalbergen et al., 2022). To tackle these difficulties, collaborative MDO can be used.

2.3.1. Collaborative MDO

Collaborative MDO is a group of technologies and methodologies that aim to facilitate the implementation of MDO in the aerospace industry (Ciampa and Nagel, 2016). Some of the technologies necessary to carry out collaborative MDO are:

- **Centralized Data Schema (CDS)**: This is a naming convention of variables where all the tools share a common language. This allows to reduce the MDO problem formulation time considerably thanks to the connections made automatically between the different disciplines, just by declaring their inputs and outputs (van Gent et al., 2017). Also the number of connections is highly reduced. An example of a CDS in the aerospace sector is CPACS (Alder et al., 2020).

- **Remote and protected workflow execution**: In big aerospace design projects, multiple companies are involved. These companies usually don´t want to share their software. Therefore, to carry out MDO in these environments, it is usually necessary to execute tools remotely, while ensuring that the confidential information of the companies is not broken. To achieve this, tools like BRICS can be used (Baalbergen et al., 2017).

- **Real time supervision**: During the execution of the workflow, it is necessary that the MDO platform used provides information in real time about the execution process that can be supervised by the different experts. This allows to discover errors faster and in an easier manner, allowing to reduce the time necessary to obtain the solution.

- **Guidance in the MDO problem formulation:** In these big projects, usually the variables, disciplines and constraints to consider in the MDO problem formulation are not know beforehand. Several iterations are usually needed until a definitive MDO problem formulation is achieved. Therefore, MDO platforms used for collaborative MDO should allow to modify the MDO problem easily, which can be achieved using a Graphic User Interface (GUI).

Including these methodologies, the desired coordination between the different ”actors” in the design process can be achieved. This allows to reduce the set-up time of MDO problems by a 40 % (Ciampa and Nagel, 2020), and therefore facilitates the final implementation of MDO in the aerospace industry.

2.4. System architecture: a link between MBSE and MDO

The product development can be divided in two parts. First, the upstream phase, where the different needs of the stakeholders are identified and translated into requirements. This is where MBSE can be used to reduce costs in the product development. The second phase is the product design phase where different methodologies, such as MDO, can be used to design a product that satisfies the previous requirements.

Different methods have been proposed to achieve this link, such as the presented in A.-L. Bruggeman et al., 2022, where a framework called RVF is used to link requirements to different elements in the MDO problem. However, to fully achieve this link between MBSE and MDO, it is necessary to include the system architecture in the process (figure 2.5).

A system architecture can be defined as the group of components of a system and the relationships between them (Crawley et al., 2015). Some steps have already been taken to fully achieve this link between MBSE and MDO including system architecture, such as Bussemaker, Boggero, and Ciampa, 2022, where a tool called MultiLinQ is used. This tool allows to join each component or quantity of interest (QOI) of the system architecture to the corresponding variable in the MDO problem.

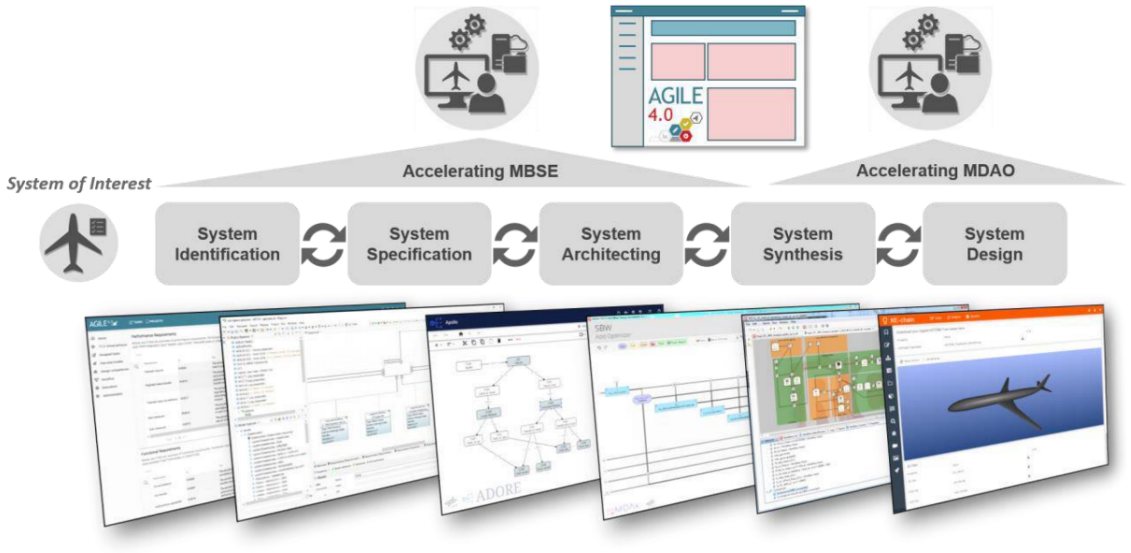


Figure 2.5: Both MBSE and MDO implementation in the aircraft industry have been extensively researched during the last years, for example in the AGILE (Ciampa and Nagel, 2020) and AGILE 4.0 (Ciampa et al., 2020) projects. In AGILE 4.0 an almost complete link between MBSE and MDO was achieved. Figure taken from (Baalbergen et al., 2022).

What has been done traditionally when trying to implement system architecture in the design process is to manually select the architectures that are going to be analysed based on experts knowledge (Roelofs and Vos, 2018). However, this exposes the process to bias and conservatism (Mesmer et al., 2022).

To overcome these hurdles, specially when dealing with novel technologies, a systematic exploration of the different possible architectures constituting the design space should be carried out. This can be done using system architecture optimization. Chapter 3 will provide an introduction to system architecture optimization, as well as to the different components needed to carry it out and the main challenges it involves.

3

System Architecture Optimization

This chapter aims to provide a more detailed explanation of the system architecture optimization process. The first section is an introduction. After that, the different components needed to carry it out will be presented. Finally, the main challenges of including MDO in system architecture optimization will be discussed, ending with the identification of a gap in the implementation of MDO in the system architecture optimization process, specially when applied to real industry.

3.1. Introduction to system architecture optimization

During the design process, one of the most difficult decisions to be taken is to determine the architecture of the system. This decision, which is going to hugely influence the final success of the project, has to be taken at an early stage of the process. However, the number of possible architectures that could be chosen is significantly high, as it grows exponentially with the number of possible decisions (Iacobucci, 2012). At this stage it is also exceedingly challenging to predict which is going to be the influence in the performance of the system of the different architectural decisions.

To overcome these hurdles, system architecture optimization can be used, aiming an unbiased automated search for the best architectures by defining the design problem as a numerical optimization problem (Bussemaker et al., 2023). To do so, two main components are needed, which are the architecture generator and the architecture evaluator.

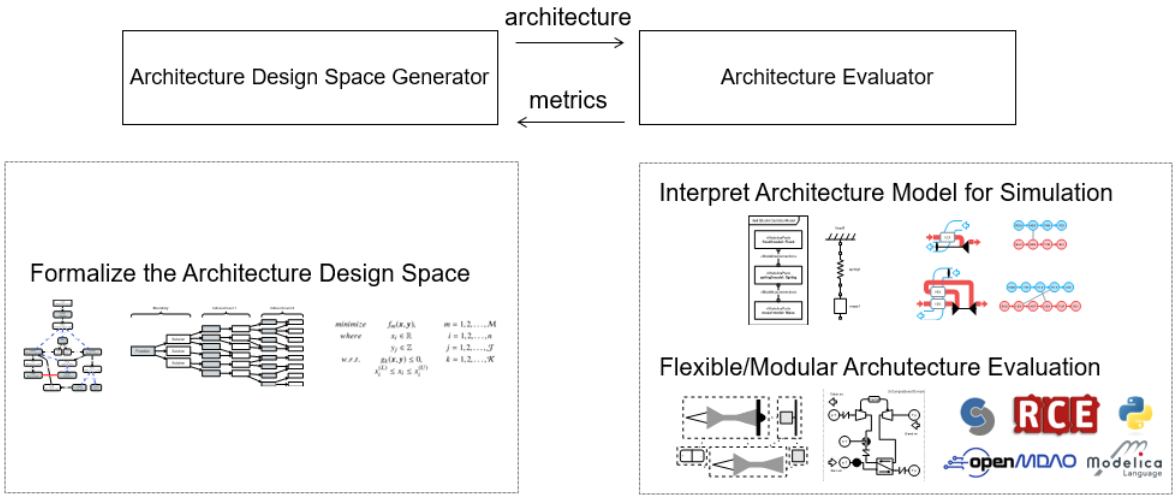


Figure 3.1: Relationship between architecture generator and architecture evaluator, reproduced from Bussemaker and Boggero, 2022. The architecture generator is in charge of formalizing the architectural design space (all the possible architectures of the system). Then the architecture evaluator will provide quantitative feedback for the different architectures, so that the optimum architectures can be obtained.

The architecture generator will be in charge of formalizing the whole architecture design space (ADS), which is the group of all the possible architectures of the system. Then the architecture evaluator will provide quantitative feedback of each of the different architectures being generated, so that the optimization process can be carried out (figure 3.1).

The process is usually as follows. First the optimizer proposes a design vector, which is translated into an architecture by the architecture generator. After that, the architecture evaluator provides quantitative feedback about that architecture to the optimizer. Then depending on the results, the optimizer will propose a new design vector until the solution is obtained.

Usually, system architectures have to be designed considering multiple and conflicting objectives. In this case, the solution is not an unique architecture, it is a Pareto front (figure 3.2). A Pareto front is a set of design points that share the following condition: it is impossible to find another point in the design space without making at least one of the objectives worse (Sobieszczanski-Sobieski et al., 2015). All the system architectures found in the Pareto front are optimized solutions, as it is impossible to find another architecture that is better at all the objectives at the same time.

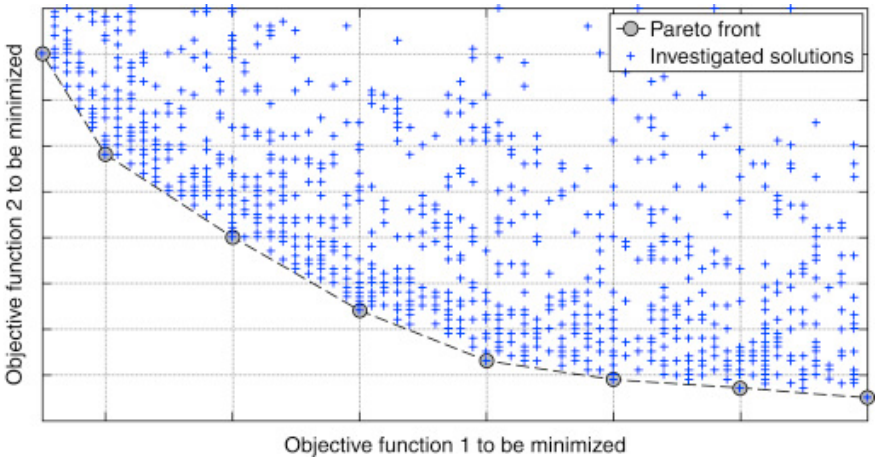


Figure 3.2: Schema of an explored design space and the resulting Pareto front. Figure reproduced from Dincer, 2018.

In the next sections, more details will be provided about the different phases necessary to perform system architecture optimization, with the next section focusing on the formalization of the different possible architectures of the system.

3.2. Architecture generation

Each system architecture is the result of a combination of architectural decisions. To understand how the different architectures can be generated, it is necessary to know what are architectural decisions, and what different types of architectural decisions exist.

Architectural decisions are high-level design decisions that lead to different system architectures, leading to significant differences in the system performance (Crawley et al., 2015). Examples of architectural decisions in an aircraft could be the type of engine (turbofan or turbojet) or the source of energy for propulsion (electric or conventional-fuel based).

The impact of architectural decisions on the design and the importance of keeping traceability of each of them is something well known in the engineering sector (Tyree and Akerman, 2005), being these decisions the variables optimized in the system architecture optimization problem.

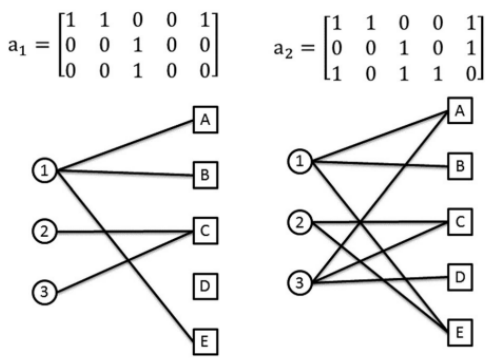
3.2.1. Types of architectural decisions

To determine what are the possible architectural decisions of the system, and ultimately to be able to perform system architecture optimization , it is necessary to know what are the different possible types of architectural decisions. Selva et al., 2016 proposes a set of six different architectural patterns representing the different types of architectural decisions. These six patterns allow to generate the different parts (referred as fragments) of an architecture. These are:

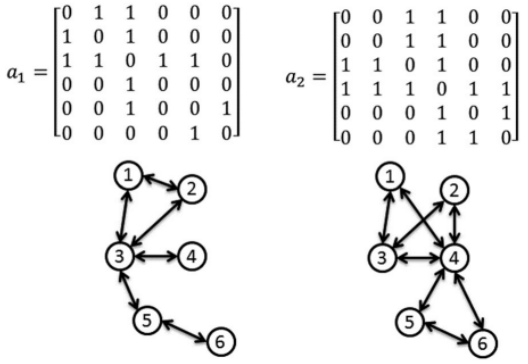
- **Combining pattern:** Given a set of n decisions, and each decision with its own set of discrete options, an architecture fragment would be the result of the combination of exactly one of the options for each of the possible decisions. As an example, if there are two decisions which are the type of engine (turbofan or turbojet) and the source of energy (electric or conventional) an architecture would be an electric turbofan, or a conventional turbojet. Morphological matrices are also an example of a combining pattern.

- **Assigning pattern:** Given two sets , each component of a set has to be linked to certain components of the other set. As an example, consider the case of a group of sensors and a group of computers. Using this pattern, each architecture would be one of the possible ways of connecting the computers with the sensors (each computer to one sensor, all sensors with all computers,...)

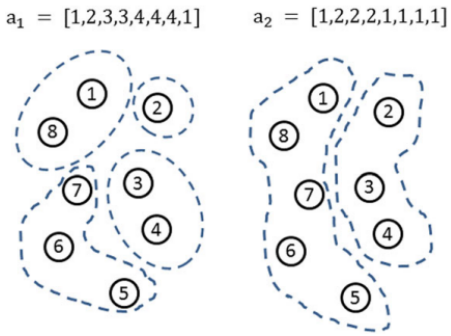
- **Partitioning pattern:** Given a set with multiple entities, these entities have to be grouped in an undetermined number of subsets. It is similar to the assigning pattern but no overlap can occur. Using the previous example, it is necessary to assign the sensors to the computers, but each sensor can only be attached to a maximum of one computer (multiple sensors could share the same computer).



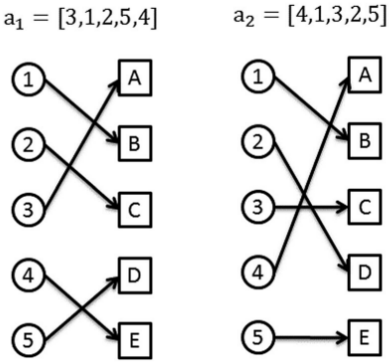
(a) Assigning pattern. Each number can be assigned to one or multiple letters.



(b) Connecting pattern. For each couple of numbers it has to be determined if they are linked.



(c) Partitioning pattern. A group of numbers has to be divided into an undetermined number of groups with no overlapping.



(d) Permutation pattern. Each number has to be joined to only one letter (and there is no overlap at any of the two sets).

Figure 3.3: This figure shows four different architectural patterns. Figures taken from Selva et al., 2016.

- **Downselecting pattern:** Given a set of entities, it is necessary to determine which entities are going to be selected and which are not. As an example, consider a case where there are 8 different types of instrument to be included in a satellite, but only 5 can be equipped. Each architecture would be any combination of 5 different instruments.

- **Connecting pattern:** Given a set of entities, for each couple of entities it has to be determined if they are connected or not. As an example of this pattern, consider the problem where it is necessary to determine the connections between the main train stations of a country. Each architecture will be the result of deciding if there is a connection for each couple of train stations.

- **Permuting pattern:** Given a set of entities, each architecture fragment/architecture is given by a certain order of these entities. Considering again the case of sensors and computers, this pattern would exist if the number of sensors and computer is the same, and it is necessary to link exactly one sensor to only one computer.

According to Selva et al., 2016, all the possible architectures of a system can be obtained as a result of different architectural decisions that are expressed as a combination of the six previous patterns. As an example, consider the guidance problem proposed in Apaza and Selva, 2021. The problem consists on designing a guidance system where it has to be determined how many components are wanted of each element (computer, sensor, actuator). Then, for each of them, the type has to be chosen from a catalog, and finally the connections between them have to be determined. The architectural decisions regarding the number and type of components are example of combining patterns. The connections can then be modelled using the assigning pattern.

These patterns own a characteristic property, and it is that all of them are exchangeable (although always one of them is more efficient to model each architectural decision, from a computational point of view). They allow to obtain a standard and complete formalization of the different types of architectural decisions. However, these decisions are considerably abstract, making it challenging to systematically build the whole architectural design space based on them.

3.2.2. Architectural Design Space Graph (ADSG)

The Architectural Design Space Graph or ADSG is a directional graph where all the possible different components of a system architecture and the relationship between them are represented based on a functional decomposition approach, with the aim of representing the architectural design space in a more practical manner (Bussemaker et al., 2020).

In this functional decomposition approach, each architecture is represented by a set of functions that the system has to fulfill and by a set of components that perform these functions (Bussemaker and Ciampa, 2022).The origin of this decomposition is the boundary function (or boundaries functions), which is/are the main function(s) that the system has to provide. The boundary function could be provided by one or different components, and each component will induce additional functions that are needed to be fulfilled, as shown in figure 3.4.

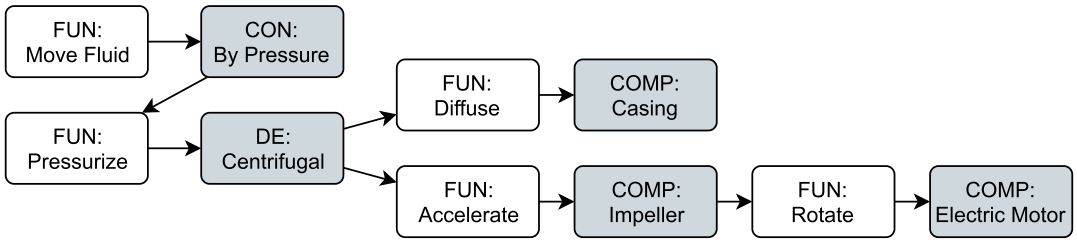


Figure 3.4: Example of the ADGS of a water pump. The functions to be fulfilled are represented in white (FUN), and the components fulfilling them are represented in grey (COMP). Additional blocks are added to provide more information about how a function is fulfilled (Concept, CON) and to decompose a function into smaller functions (decomposition, DE). Figure taken from Bussemaker and Ciampa, 2022.

The ADSG is based on two types of architectural decisions, which are the selection choices and the connection choices. Selection choices are decisions where there are mutually exclusive options, being directly linked to the combining and the downselecting patterns introduced by Selva et al., 2016. They can be further divided into two new types of architectural decisions:

- **Function fulfillment:** Functions specify what the system should do. Each function is fulfilled by one or multiple components, and each component can demand a new function to be fulfilled. There are numerous times where the same function can be performed by different components. Determining which of the possible components performs a certain function is a function fulfillment architectural decision.

  There is an example of this architectural decision in figure 3.5. The system must provide a function which is to provide longitudinal stability to the aircraft, and there are two different components that can perform this function, the canard or the horizontal stabilizer. This type of architectural decision can be linked to the combining and the downselecting patterns.

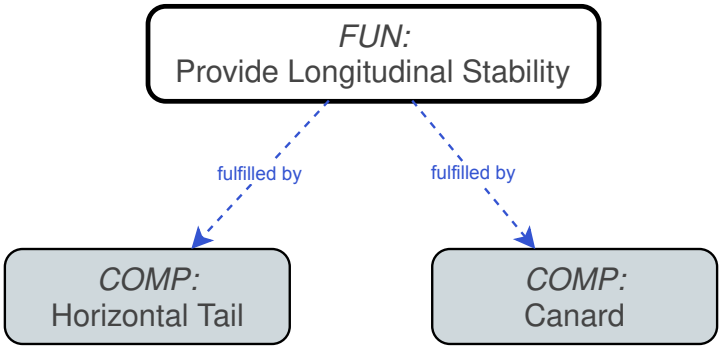


Figure 3.5: Example of a function fulfillment architectural decision. There are two possible components to provide longitudinal stability, which are the horizontal tail or the canard.

- **Component characterization:** This architectural decision is found at a component level, providing additional information about the components being included in a certain system architecture. This architectural decision is further divided into two architectural decisions.

  The first one of these two architectural decisions is the **number of instances**. When a component is included at a certain architecture, it might be necessary that the number of times it is included can vary. In that case, the number of instantiations of that component is considered to be an architectural decision.

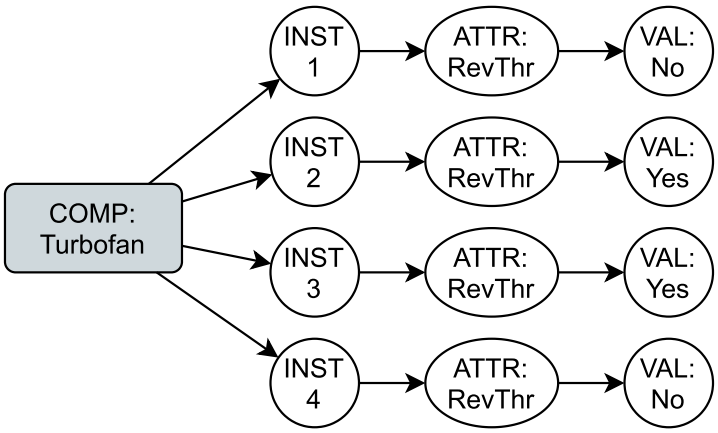


Figure 3.6: Example of both types of architectural decisions at a component level. First it is necessary to determine how many turbofan instances are generated (**number of instances decision**). Then for each turbofan, it has to be determined if it has or not thrust reverse system (**component properties decision**). Figure taken from Bussemaker and Ciampa, 2022.

The second architectural decision at a component level are the **component properties**. Each component is defined by a group of different properties, which provide further information and details about a certain component. For example, in the case of a wing, a property could be if it has winglets or not. The value of each of these properties for each component is an architectural decision too. If a component is repeated multiple times, each instantiation of the component could have different values of the properties. This can be observed in figure 3.6.

To generate the ADSG, connection choices are needed too. These choices cover the decisions regarding the possible connections between some sources and targets, being directly linked with the 6 patterns from Selva et al., 2016 (section 3.2.1) . These connections are usually implemented using ports, where each connection can represent anything, from energy flows to physical connections between components, as shown in figure 3.7.

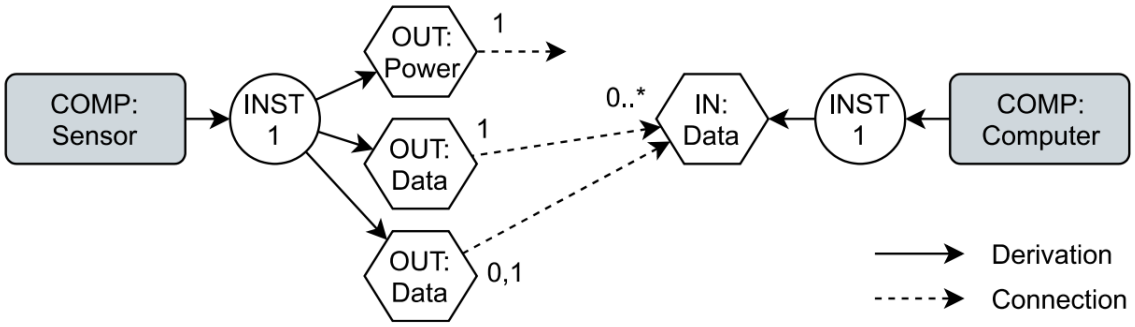


Figure 3.7: This figure shows the logic inside the port. In this case, the possible connections between a sensor and a computer are modelled inside the port (same example as in section 3.2.1). Figure taken from Bussemaker and Ciampa, 2022.

Four types of architectural decisions (function fulfillment, component number of instances, component properties and component connection choices) have been introduced. These decisions allow to generate the ADSG using software such as ADORE (Bussemaker, Boggero, and Ciampa, 2022), formalizing the whole architectural design space. However, to carry out system architecture optimization, it is necessary to provide quantitative feedback about each of the different generated architectures. The next sections will cover the methodology that can be used to achieve this, as well as the main challenges that this process arises.

3.3. Architecture evaluation

To compare the different architectures being generated, so that an optimization can be executed, it is necessary to evaluate quantitatively each of the different architectures being generated without any user interaction. This quantitative information has to be given to the optimizer, so that it can determine which architectures are better than others and obtain an optimized solution in the end.

To accurately evaluate each of the different architectures generated, MDO can be used, so that the couplings between the multiple disciplines involved in the design process are taken into account. When carrying out MDO, information regarding the objectives and the constraints is given to the optimizer for each architecture evaluated, determining the performance and the feasibility of the system architecture.

In this section, first the requirements MDO platforms desired to be used as architecture evaluators have to satisfy will be discussed. After that, some of the main MDO platforms are going to be introduced, paying special attention to their advantages and disadvantages when used as evaluation method for system architecture optimization, specially when applied to industrial cases. Finally, some conclusions will be drawn, the gap in the literature and in the existing technology will be identified and the thesis objective will be formulated.

3.3.1. Requirements for the implementation of MDO in system architecture optimization

MDO platforms used as architecture evaluators for system architecture optimization have to satisfy multiple requirements. A total of three different requirements have been identified.

Mixed-discrete variables

A differentiating characteristic of MDO problems used for system architecture optimization is the types of variables they involve, which are both continuous and discrete (mixed-discrete variables). Continuous variables are numerical variables defined by a lower and an upper bound, taking any possible value between bounds. An example of this type of variables could be the aspect ratio or span of a wing.

Discrete variables are defined by a set of possible values, meaning the number of values they can take is finite. Usually, a further distinction is made between integer variables and categorical variables. Integer variables are numerical variables where the order of the values inside the set is important, being possible for optimizers to use this information for the optimization. Some examples of these variables could be the number of engines or the number of ribs of a wing. Categorical variables can be of any type and the order inside the set is irrelevant. Some examples would be the material used for the fuselage or the type of energy used by the propulsion unit.

When performing system architecture optimization, all these types of variables appear in the MDO problem. One of the main reasons is because architectural decisions are usually treated as discrete variables in the optimization process (Chepko et al., 2008). As a consequence of all this, any MDO platform intended to be used as architecture evaluator **must be able to deal with mixed-discrete variables**, both as design and state variables.

Optimization algorithms

Architecture evaluators have to provide the optimizer with quantitative feedback of each proposed architecture. There are two possibilities for this. The first one is to use an optimizer external to the evaluator. The second one is that the platform used as evaluator also runs the system architecture optimization problem itself (the optimizer is integrated in the MDO platform too).

In both cases, the optimizer has to be able to deal with multi-objective problems, as usually system architectures are built for multiple and conflicting objectives (Hirshorn et al., 2017). Due to the complexity of system architecture optimization and the consequent usual participation of multiple companies/institutions in the process, usually tools used in the process are considered to be black-boxes, where only the outputs for a certain combination of the inputs can be known.

Another problem is that novel architectures usually demand physics-based tools for evaluation. When using these tools in the MDO problem execution it is common that hidden constraints appear. Hidden constraints are constraints that are not known by the optimizer until the evaluation has ended (Müller and Day, 2019). These constraints also mean that the optimizer won´t be able to evaluate that architecture. Hidden constraints appear in most cases because the simulation used for the evaluation has crashed (Le Digabel and Wild, 2023). It is key that the optimization algorithm being used is able to deal with this type of constraints.

Finally, as each architecture is the result of a set of architectural decisions, the variables to be optimized might change depending on the architecture being evaluated. For example, in the design of a hybrid-electric aircraft propulsion system, depending on the selection of turboshafts or electric motors to provide mechanical power, the design variables will be different (Bussemaker et al., 2023). An example would be the batteries weight, which would only exist in the MDO problem if motors are chosen. It is necessary that the optimization algorithm used can deal with hierarchical design variables like this.

It can be concluded that MDO platforms used to as architecture evaluators **must include or be able to connect to algorithms able to deal with mixed-discrete, multi-objective, hierarchical, black-box optimization problems**, so that they can solve the system architecture optimization problem.

Some examples of these algorithms are evolutionary algorithms, such as the Nondominated Sorting Genetic Algorithm II (NSGA-II) (Deb et al., 2002), or Surrogate Based Optimization (SBO) algorithms (Bussemaker, Bartoli, et al., 2021).

Automatic readjustment of the MDO problem

Each system architecture is composed of different components. As each component is defined by different variables, this means that variables included in the MDO problem formulation will change (Bussemaker, Garg, and Boggero, 2022). Furthermore, the design disciplines, constraints and the possible connections between them might also change depending on the architecture that is analysed. As an example to illustrate this, consider the case of an aircraft propulsion system. In the case of an hybrid architecture, additional disciplines with respect to the conventional architecture (with their corresponding inputs and outputs) will be included to size and optimize the electric part of the propulsion system .

Traditionally, as the possible architectures of the system were manually selected and the number was not too high, a different MDO problem was formulated for each architecture. However, for the case of system architecture optimization, the number of possible architectures is excessive to follow this procedure. As a consequence, any MDO platform used for system architecture optimization **must be able to readjust the MDO problem automatically depending on the architecture being analysed**.

There are two approaches to deal with this automatic readjustment of the MDO problem. The first approach is that the MDO platform formulates and executes a different MDO problem for each architecture (figure 3.8). The second approach is that the MDO platform formulates just an unique MDO problem and that during the execution process the necessary modifications are made for each specific architecture (figure 3.9).

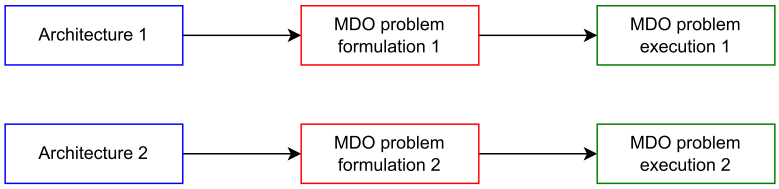


Figure 3.8: This figure shows the first possible approach to adapt the MDO problem automatically, where for each system architecture (blue) proposed by the optimizer, a different MDO problem is formulated (red) and executed (green).

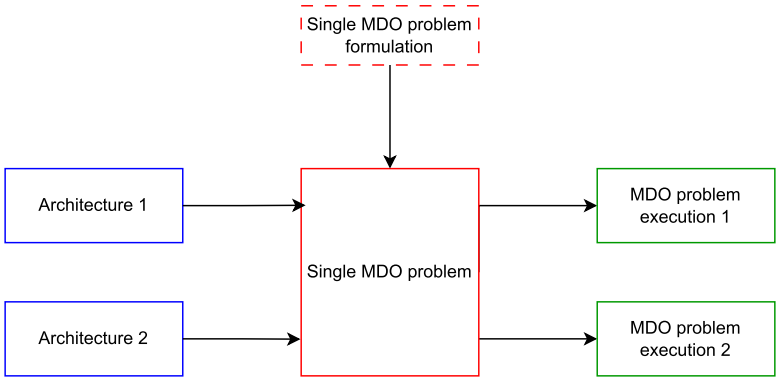


Figure 3.9: This figure shows the second possible approach to adapt the MDO problem automatically, where a single MDO problem is formulated and the execution process is adapted automatically for each system architecture.

The first approach should be more efficient from the computational point of view, as only the elements necessary to exist in the MDO problem for each system architecture are included. However, there are some cases where only the second approach is possible as the execution platform might not allow to change the MDO problem formulation once the execution starts.

It is important to mention that on some occasions the input to the architecture evaluator will be the optimizer design vector. Nevertheless, there are other cases where it is the actual system architecture (translated or generated from the optimizer design vector). However, in both cases the MDO problem is readjusted according to a certain system architecture (directly or indirectly).

3.3.2. Possible MDO platforms for system architecture optimization

The different requirements that an MDO platform has to satisfy to be used as an architecture evaluator have been introduced. Now the main MDO platforms in the aerospace industry will be introduced, including a discussion about their suitability to be used as architecture evaluators. Also their suitability for real industrial projects will be covered through the study of their adaptability to collaborative MDO.

OpenMDAO

OpenMDAO is an open-source optimization framework, but also a platform to build new analyses tools based on Python (OpenMDAO, 2019), which has been developed by a collaboration of the MDO lab of the University of Michigan and NASA.

OpenMDAO has multiple advantages that make it a competent MDO platform, although the two distinctive ones are the development of own algorithms for the solution of coupled systems and the development of methods for the effective calculation of derivatives (Gray et al., 2019). This allows OpenMDAO to deal (from the mathematical point of view) with design problems characterized by complex and extensive design spaces, while maintaining a great efficiency, specially when gradient-based optimization can be used.

OpenMDAO has already been used as part of more complex codes to formulate and execute MDO problems for system architecture optimization (Bussemaker et al., 2023). It is able to deal with discrete variables and also owns some of the necessary algorithms to solve these MDO problems. However, on its own, it cannot readjust the MDO problem automatically for each system architecture. The main reason for this is that to run the optimization using OpenMDAO as an optimizer, the MDO problem formulation and execution cannot be changed (figure 3.10).

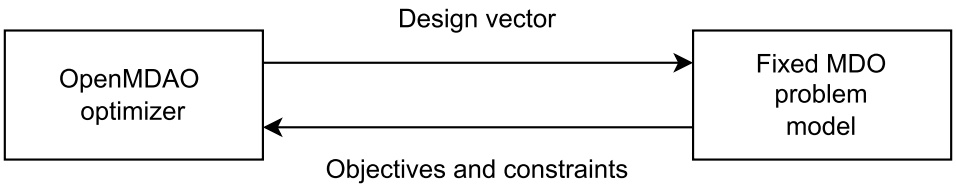


Figure 3.10: This figure shows that when OpenMDAO is in charge of running the optimization, the model containing the MDO problem formulation and execution has to be fixed. As a consequence, it cannot solve system architecture optimization problems by itself.

In the literature, OpenMDAO is used as an architecture evaluator, existing an external optimizer to run the whole system architecture optimization problem in a higher level. Then, based on the architecture proposed by this optimizer, a different problem was modelled and executed using OpenMDAO, and the results were given back to the external optimizer (figure 3.11). However, it is important to remark that this automatic readjustment of the MDO problem was possible only when there was an external optimizer, and what is more important, though the addition of additional code by the user (as it is Python based). Using default OpenMDAO properties it is not possible to use it as an architecture evaluator.

Regarding collaborative MDO, it is not adapted much to it. It does not own a GUI. As a result, great quantities of coding are needed in order to formulate the MDO problem, which could take considerable time depending on the size of the problem and the experience of the designer. This coding also results in a huge opening barrier, which is a serious problem when performing system architecture optimization, due to the numerous amount of experts that are involved in the process and that have different knowledge levels about MDO and coding.

To formulate the MDO problem, it is necessary to know beforehand all the variables, disciplines and constraints of the problem, including all the possible connections between them, which is usually not the case in system architecture optimization. Finally, it is also not very flexible when introducing new disciplines or connections in the MDO problem.

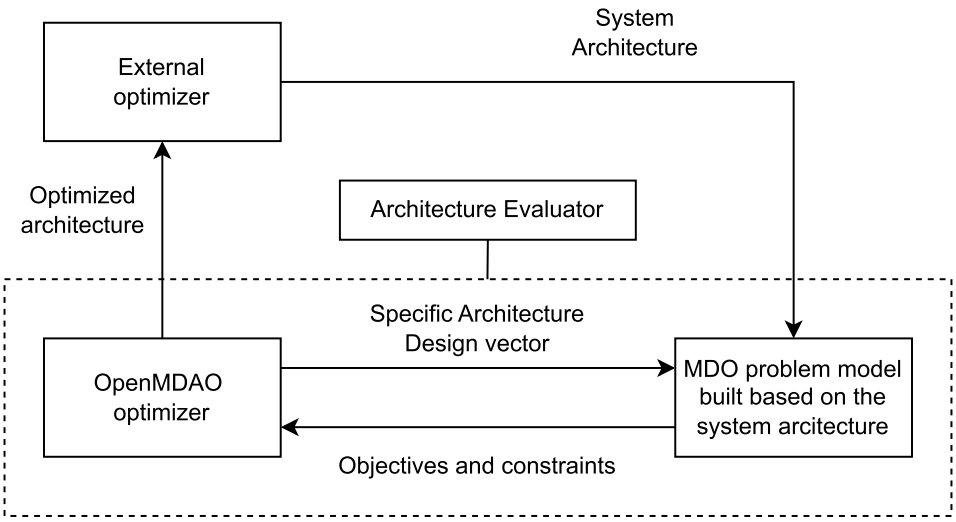


Figure 3.11: OpenMDAO can be used as an architecture evaluator, but only through the addition of additional code and using an external optimizer to run the optimization. The external optimizer proposes the different architectures and a different MDO problem is formulated and executed in OpenMDAO to optimize that specific architectures, giving back the results to the external optimizer.

GEMSEO

GEMSEO is an open-source MDAO platform developed by the IRT Saint Exupéry consortium (Gallard et al., 2018), including partners such as Airbus and ISAE-Supaero, which has already been used by one of the main players in the aerospace industry (Airbus). It works in a similar way to OpenMDAO as it is also based on Python. It shares all the advantages of OpenMDAO for system architecture optimization (mixed-discrete variables and algorithms). However, it shares the same disadvantages as OpenMDAO too.

It also needs additional code to readjust the MDO problem, and consequently, to be used as an architecture evaluator. Regarding collaborative MDO, it also lacks a GUI and a great amount of coding is necessary to formulate the MDO problem, making it difficult to formulate complex system architecture optimization problems using this platform. It also requires again to know the MDO problem formulation beforehand and is not flexible to modifications.

RCE

RCE (Remote Component Environment) is an open source software environment that allows to both generate and execute workflows by integrating different tools into a common network (Boden et al., 2019). RCE is highly adapted for the execution of complex MDO problems. Each expert can introduce their tool on the workflow execution and check on real time the value of all the variables. The workflow can also be executed in a distributed manner, with the tools located on different computers. Intellectual property can be also protected adding additional software such as BRICS.

However, formulating the MDO problem in this platform is tedious and time consuming. High loads of work are necessary to integrate the different tools between each other, including making all the connections manually. The tool is not really flexible when integrating new disciplines neither, and the debugging process is time consuming. These are the main reasons why RCE is usually used just as an execution platform, defining the MDO problem in platforms specially designed for that task, such as KADMOS.

KADMOS, CMDOWS and OpenLEGO

Knowledge and graph-based Agile Design for Multidisciplinary Optimization System (KADMOS) is an open-source graph-based MDO problem formulation platform developed by TU Delft (van Gent and La Rocca, 2019), based on graph-analyses.

KADMOS cannot execute the MDO problem itself. Common MDO Workflow Schema (CMDOWS) is used to store the MDO problem formulation in an XML file. This file can be later used to execute the MDO problem in various MDO execution platforms (van Gent et al., 2018). These platforms include RCE or OpenMDAO. Plugins are need for both cases, such as Open-source Link between AGILE and OpenMDAO (OpenLEGO, de Vries et al., 2017).

KADMOS owns some characteristics necessary to be used as an architecture evaluator, such as the ability to deal with continuous and discrete variables. KADMOS and its associated tools allow to formulate and execute MDO problems that satisfy many of the requirements necessary for collaborative MDO, such as the inclusion of a GUI called VISTOMS (Aigner et al., 2018). A framework has already been proposed to allow KADMOS to readjust the MDO problem automatically for each system architecture (A.-L. M. Bruggeman and La Rocca, 2023), and research is being carried out by TU Delft in this field right now to implement this functionality.

WhatsOpt

Finally, to reduce the opening barrier of both OpenMDAO and GEMSEO, WhatsOpt could be used. WhatsOpt is a collaborative environment to support MDAO (Lafage et al., 2019) created by ONERA. Basically, it is a GUI where the MDO problem can be formulated.

WhatsOpt allows to reduce the opening barrier, as now the MDO problem is modelled and not coded, but it achieves that by reducing the freedom of the user. Also some capabilities of OpenMDAO and GEMSEO are lost, specially those related with changing dynamically the MDO problem, making it right now not suitable for system architecture optimization.

3.3.3. MDAO Workflow Design Accelerator (MDAx)

None of the previous platforms satisfy all the requirements necessary to implement MDO in real industry system architecture optimization process. Another option is going to be studied in this section, which is MDAx. MDAx is a Python-based MDO workflow modelling environment developed by DLR that allows to model, inspect and explore workflow components and their relationships, as well as exporting the workflow configuration for the execution on integration platforms (Page Risueño et al., 2020).

To formulate the MDO problem, MDAx uses a GUI based on the XDSM which allows to drag and connect the different components of the workflow. This aims to solve existing drawbacks of other MDO platforms related with the MDO problem formulation, such as the difficulty to set up and modify the workflow.

To study MDAx adaptability to be used as an architecture evaluator as part of the system architecture optimization problem, the methodology used to formulate the MDO problem will be discussed first. After that, the workflow execution process through RCE will be presented. Finally, it will be determined how does MDAx+RCE currently deal with the three different requirements previously discussed, as well as with collaborative MDO.

MDO problem formulation

MDAx uses the graph definition of MDO problems introduced by Pate et al., 2014 and further developed by van Gent, 2019 as a base to formulate the MDO problem. In this methodology, each discipline is represented by a block, and its inputs and outputs are attached to it as variable nodes. To formulate the MDO problem, it is necessary to obtain two graphs, which are the data graph and the process graph (figure 3.12).

The **data graph** includes all the data connections existing between the different disciplines (which represent design tools). Then, the **process graph** includes the information regarding the execution order of the different disciplines.

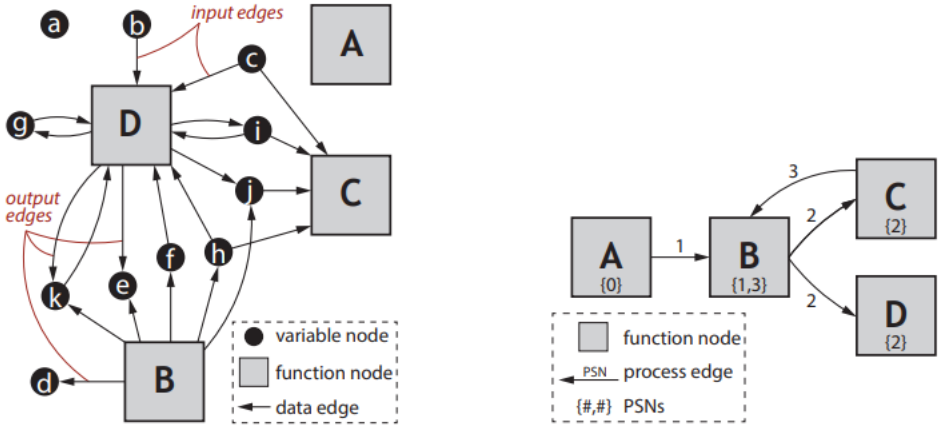


Figure 3.12: On the left , example of a problem data graph. On the right, a possible execution graph. Figures taken from van Gent, 2019.

This graph-based methodology has the great advantage that it is not necessary to know the whole MDO problem beforehand to formulate the MDO problem (Pate et al., 2014). It allows designers to try different combinations of tools and to study the connection between them. Then, after some iterations, a valid MDO problem can be formulated. This process simplifies the MDO problem formulation, allowing to tackle the complex MDO problems needed for system architecture optimization.

Three steps are necessary to formulate the MDO problem in MDAx. These are:

- **Load blocks inputs and outputs**: Each discipline is defined by its inputs and outputs. To include a discipline in MDAx, it is necessary to declare its inputs and outputs. This can be done directly in the GUI, or in a more efficient manner by uploading the inputs and outputs of the discipline in two different XML files. XML files are used because they use an universal language and data structure, helping to integrate the different tools used in the execution phase.

  These files have a tree structure, consisting of nodes, where each node can store multiple types of data (such as strings, numbers, arrays,...), or even multiple children nodes . Each node, and its information, can be accessed through its corresponding xpath, which is just a string specifying the location of the node inside the XML tree. This xpath expression is built including all the nodes from the root (the most outer node of the XML file) to the selected node (figure 3.13) .

  MDAx includes two particularities with respect to the approach introduced by van Gent, 2019 to formulate the data graph (Page Risueño et al., 2020). First, tools that don´t have inputs and/or outputs are allowed, as in many cases tools to visualize the process or the results are needed in the workflow. Secondly, tools with self-loops (tools that have a variable node with the same name as input and output) are allowed, assuming that the output is just an updated value of the input . An example would be a tool that needs an estimation of the MTOW to calculate the real

MTOW of the aircraft.

```
<aerodynamics>
  <aircraft>
    <wing>
      <span>17</span>
      <airfoil>NACA0012</airfoil>
      <root_chord>3</root_chord>
      <tip_chord>1</tip_chord>
      <sweep_angle>15</sweep_angle>
    </wing>
  </aircraft>
  <flow>
    <speed>200</speed>
  </flow>
</aerodynamics>
```

Figure 3.13: Possible XML used as input of an aerodynamic tool. The wing geometry and the flow conditions are given as input. The input values necessary for the tool can be obtained specifying the nodes xpaths where the information is stored. For example, the value of the wing span could be obtained through the xpath "/aerodynamics/aircraft/wing/span".

Knowing this, and once all the tools have been uploaded, MDAx will make the connection between them joining the output variable nodes of disciplines with input variable nodes that share the same name. This can be done because MDAx assumes that a Central Data Schema is used, allowing to make connections automatically.

- **Resolve collisions**: A collision occurs when two different disciplines have an output variable node with the same name. This is a common scenario, for example in cases where a tool has to be executed in different parts of the workflow or when two tools provide the same output but with different levels of fidelity.

  MDO problems with collisions are considered still valid, but they won´t be able to be executed. MDAx allows to identify which variable nodes are causing a collision automatically, and provide several methods to solve them automatically.

- **Resolve feedback**: The final step to end the problem formulation is to obtain the process graph, which indicates the execution order of the workflow. In MDAx, all the different disciplines are found in the diagonal of the XDSM, where blocks found on the left are executed first.

  The user can modify the order easily by dragging the different blocks. Then MDAx will check if the workflow is executable. This is done by iterating the diagram backwards and determining if there are uncoverged feedback between disciplines. Unconverged feedback happens when a discipline demands an input that has not been computed yet. These have to be fixed for the workflow to be executed.

  There are two ways to do this. The first one is to rearrange the disciplines, which will change the connections and might solve the problem (MDAx included automatic tools to order the disciplines minimizing feedback connections). Sometimes unconverged feedback cannot be solved just by reordering the disciplines, so it will be necessary to add convergers.

When these three steps are executed successfully, a valid and executable MDO problem will have been formulated. However, MDAx is not a complete MDO platform by itself, as it cannot execute the MDO problem. There are different options to execute the workflow (OpenMDAO, Optimus,...), although the most common is RCE.

MDAx export to RCE

When exporting the workflow to RCE, there are three high-level groups of elements that can exist in the workflow:

- **Input provider and Output writer**: The input provider is an RCE element that contains all the necessary inputs necessary to start the workflow. These inputs are given in the form of an XML file. The data file contains usually a value for all input variable nodes that are unconnected, as well as an initial value for the design variables (if there is an optimizer). The structure of this data XML file can be created automatically by MDAx, but it is the user who has to add the initial values manually. The file must be also selected by the user in RCE to start the optimization.

  In the case of the output writer, it is just another RCE element where the final values of the different variables are stored in an XML file.

- **Converger and Optimizer**: These blocks will only exist if they appear in the XDSM. RCE contains already default blocks for both of them. MDAx will set them up automatically depending on the MDO problem formulation, including the design variables, the optimization algorithm or the variables used for convergence, among other specifications.

- **Tool**: Each discipline in MDAx will lead to the creation of at least 5 blocks in the RCE workflow (figure 3.17). This is because inputs and outputs of tools have to be specially prepared before and after execution. To show how these components work, consider the case of a discipline that needs as input two variables (x and y). Then this discipline provides as output z, which is the addition of both previous variables.

  The first block of this discipline is the base splitter. This block will create a copy of the workflow XML. The first copy goes to the input filter block, which selects from this file those nodes (and the data they include) that represent input variables of the tool. Then it creates a new XML file only with them (figure 3.14).

```
1 <workflow>
2   <x>1</x>
3   <y>1</y>
4   <a>1</a>
5 </workflow>
```

```
1 <workflow>
2   <x>1</x>
3   <y>1</y>
4 </workflow>
```

Figure 3.14: Figure showing the process carried out in the input filter. The figure on the left shows the workflow file before the discipline. The second figure shows the XML file to be used as input for the tool, where only those variables to be used by the tool are included.

  Then this new file will enter into the tool block. Finally, from the XML file generated by the tool, only those nodes that represent variables existing in the MDO problem will be included in a new file created by the output filter block (figure 3.15).

  The last step is to merge the second XML copy created by the base splitter and the XML coming from the output filter into an unique XML file, which is done by a new block denominated merger block. This block is a script block generated by MDAx (figure 3.16).

```
1 <workflow>
2   <z>2</z>
3   <b>1</b>
4 </workflow>
```

```
1 <workflow>
2   <z>2</z>
3 </workflow>
```

Figure 3.15: Figure showing the process carried out in the output filter. The figure on the left shows the XML file generated by the tool. In this file, variables that are not included in the MDO problem could be found, as b in this case. The output filter will delete these unnecessary variables from the file, leading to a new file as shown on the picture on the right.

There is an additional important detail to be mentioned, and it is the case of tools executed in parallel. In this case, additional merger blocks will be needed to be included to ensure that all the results are included in the workflow file to be used in the next execution step.

```
1 <workflow>
2   <x>1</x>
3   <y>1</y>
4   <z>2</z>
5 </workflow>
```

Figure 3.16: The original workflow file and the output filter file are merged in the merger block. An example of the merged file is shown in this figure.

Finally, an example of an MDAx discipline export in RCE can be found in figure 3.17

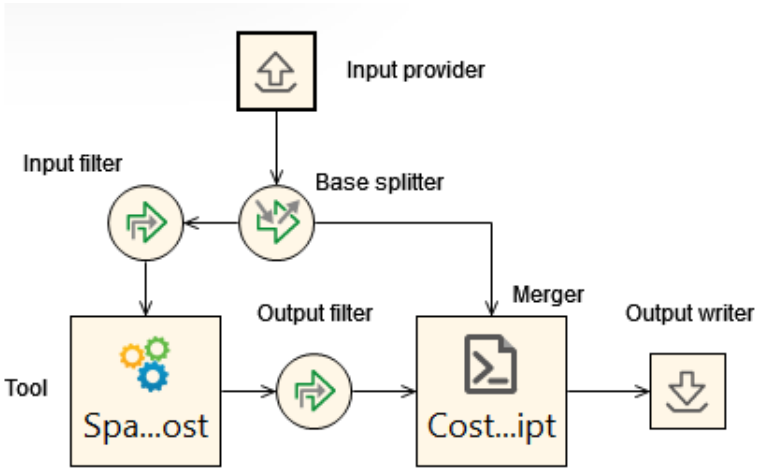


Figure 3.17: Example of a discipline generated by MDAx in RCE. First the input filter provides the workflow XML file. The splitter will generate two copies. The first one goes to the left to the input filter. After preparing the input file for the tool, executing it and extracting the necessary outputs in the output filter, the generated file will be merged with the second workflow file generated by the base splitter in the merger block. Finally, the results are stored in the output writer.

MDAx state of the art adaptability to system architecture optimization

Three different requirements were identified to be fulfilled by an MDO platform to be used as an architecture evaluator in the system architecture optimization process. The first one was dealing with mixed-discrete variables. This is not a problem for MDAx and RCE, as they work with XML files, and nodes inside XML files can store many different types of data. Even more complex data structures, such as arrays, can be exchanged between disciplines.

For the case of the optimization algorithms, RCE includes several default algorithms that are able to deal with mixed-discrete, multi-objective, hierarchical, black-box optimization problems. It also allows to include custom tools with more advanced optimization algorithms, such as those presented in SBArchOpt (Bussemaker, 2023).

Both MDAx and RCE are fully adapted to collaborative MDO, being the strongest characteristic of the platform. In the case of MDAx, it includes a GUI that guides user to formulate the MDO problem (figure 3.18). As mentioned before, this simplifies enormously the time required to set up the problem, as the user does not need to know all the disciplines and their connection beforehand. The GUI also allows to modify the problem in a much easier and faster manner. MDAx also includes auxiliary tools necessary to deal with complex MDO problems, such as the option of redo/undo and the automatic detection/resolution of collisions, simplifying enormously the formulation of the MDO problem.

RCE is also adapted to collaborative MDO, including the possibility to supervise the results in real time. Tools can be executed remotely and additional tools such as BRICS can be used to protect confidential data too. MDAx already exports to RCE a workflow that is almost ready for execution. Before that, two additional steps are necessary to be done in RCE by the user, which are including the execution tools (if not detected automatically) and implementing the input file.

The only requirement left to be implemented in MDAx to be used as an architecture evaluator adapted to collaborative MDO is to model MDO problems that can be readjusted automatically depending on the architecture analysed. This is going to be the central topic of the following section.

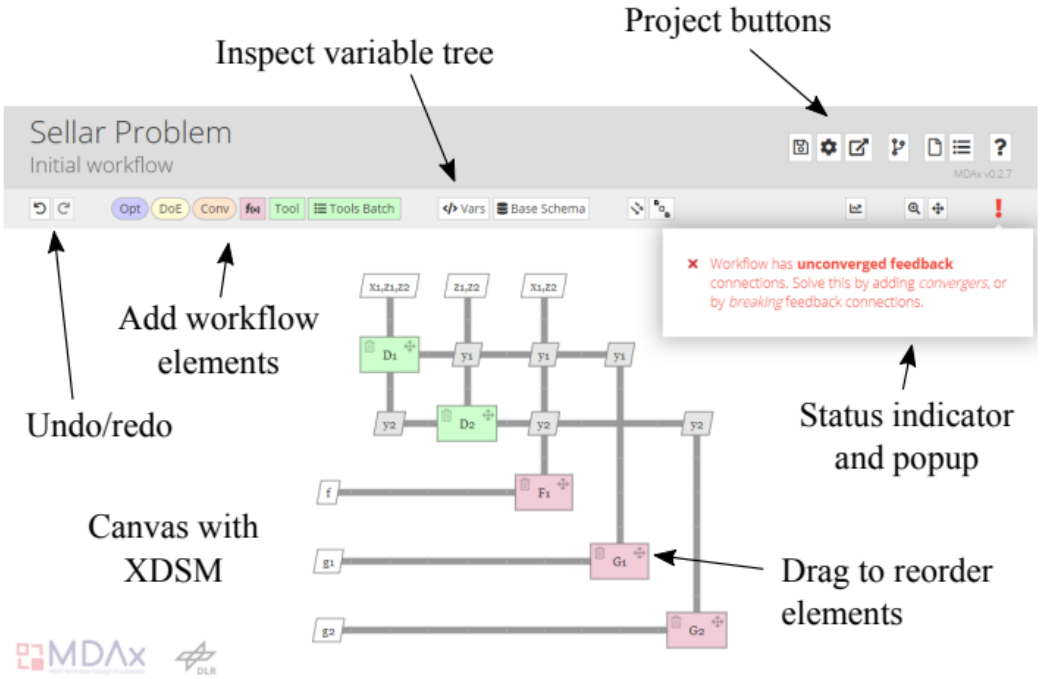


Figure 3.18: MDAx GUI. The XDSM is shown in the main part of the GUI. The toolbar includes several features to modify the workflow, such as the inclusion of tools, the undo/redo or the inspection of the variables existing in the workflow (variable tree). Also features characteristics of collaborative MDO, such as automatic detection and resolution of collisions, are included.

3.4. Technological gap

System architecture optimization allows to obtain the optimum architecture of a system by formulating a numerical optimization problem. To evaluate the different candidates proposed during the optimization, MDO can be used to take into account the couplings interactions existing between the different design disciplines.

To do this, MDO platforms can be used as architecture evaluators, evaluating each architecture proposed by the optimizer through MDO and giving back their performance and feasibility in the form of objectives and constraints. Three different requirements have been identified that are needed to be fulfilled by an MDO platform to be used as an evaluator for system architecture optimization. These are:

- It must be able to deal with mixed-discrete variables.
- It must include or be able to connect to algorithms able to deal with mixed-discrete, multi-objective, hierarchical, black-box optimization problems.
- It must be able to readjust the MDO problem automatically depending on the system architecture being analized.

If system architecture optimization is desired to be used in the industry, the MDO platform used as evaluator also has to be adapted to collaborative MDO. A research has been carried out on some of the main MDO platforms in the industry, determining that there are some platforms adapted to collaborative MDO, but there is none that can readjust the MDO problem automatically, existing a gap in the literature.

The platform consisting on MDAx and RCE is able to deal with the two first requirements to be used as architecture evaluator. It is also adapted to collaborative MDO. To fill this literature/technological gap, the objective is going to be to **adapt MDAx to be used as an architecture evaluator, by extending its backend code to model and export MDO problems that can be readjusted automatically during the optimization process, depending on the architecture being analysed.**

Every time that MDAx formulates and exports the MDO problem to RCE, some manual modifications are needed before executing it, as mentioned previously. Also, if tools are executed remotely and credential are needed, it is before the execution starts that these have to be checked. As a consequence, to integrate the automatic readjustment of the MDO problem in this platform, the approach consisting on the automatic readjustment of the execution process has to be used.

Only the back end part will be considered for time reasons, although it is necessary to implement in the GUI the corresponding modifications in the backend to obtain an MDO platform to be used as an architectural evaluator and fully adapted to collaborative MDO.

Nevertheless, implementing all the necessary features for the automatic reformulation of the MDO problem in the backend will be already an important step forward in the integration of MDO in the system architecture optimization process and in the inclusion of the latest in the industry. To do so, the first step will be to determine what are possible modifications that the system architectures can cause in the MDO problem formulation/execution, as well as determining possible strategies to deal with them from the implementation point of view. This will be the goal of the next chapter.

4

Architectural influences

When performing system architecture optimization, MDO can be used to evaluate the different architectures. MDO platforms used for this function need some special capabilities, including the necessity of readjusting the MDO problem for each different architecture automatically. Each of the different modifications in the MDO problem formulation/execution caused by the different system architectures will be denominated **architectural influences**.

The possible modifications that an MDO platform needs to deal with to reformulate the MDO problem for system architecture optimization are the same necessary to model an unique MDO problem whose execution process is adapted automatically to the different architectures. This is the reason why the previous definition of architectural influences includes both the formulation and execution.

These architectural influences are the topic of this chapter. First, the methodology used to determine the different architectural influences will be introduced. Then, an example case of how the methodology was applied will be discussed. After that, the four architectural influences discovered will be presented, as well as a couple of verification cases. Finally, the different manners in which MDO platforms can deal with these architectural influences, from an implementation point of view, will be studied.

4.1. Methodology

Determining the modifications that all different architectures of a system could cause in the MDO problem formulation/execution might seem as an impossible task, as the number of architectures is usually excessively high. However, all the possible architectures can be obtained as a combination of only four different types of architectural decisions (section 3.2.2). Therefore, these modifications or influences in the MDO problem can ultimately be attached or linked to one of those four architectural decisions.

To determine what are the architectural influences caused by these decisions, the following methodology is proposed. First, a sample of MDO problems is collected. These problems will cover different fields, including multiple problems related with the aerospace sector. Each of these problems is formulated for a certain system architecture, which has been previously fixed.

Then, for each of the problems, different ”thought experiments” will be performed where the original system architecture is modified using the architectural decisions previously mentioned. The aim is to observe the modifications that would be caused by the inclusion of the different architectural decisions, obtaining all the possible architectural influences. To show the procedure, an example application of this methodology is presented in the next section.

4.2. MDO Example case: Supersonic Business Jet Problem (SSBJ)

To show how the presented methodology has been used to determine the different types of architectural influences, an example case is widely discussed in this section. The problem chosen is a slightly modified version of the Supersonic Business Jet Problem. The SSBJ problem is a well-known MDO aircraft design problem presented by Sobieszczanski-Sobieski et al., 1998, as an adaptation of the problem proposed by the 1995-1996 AAIA Student Competition.

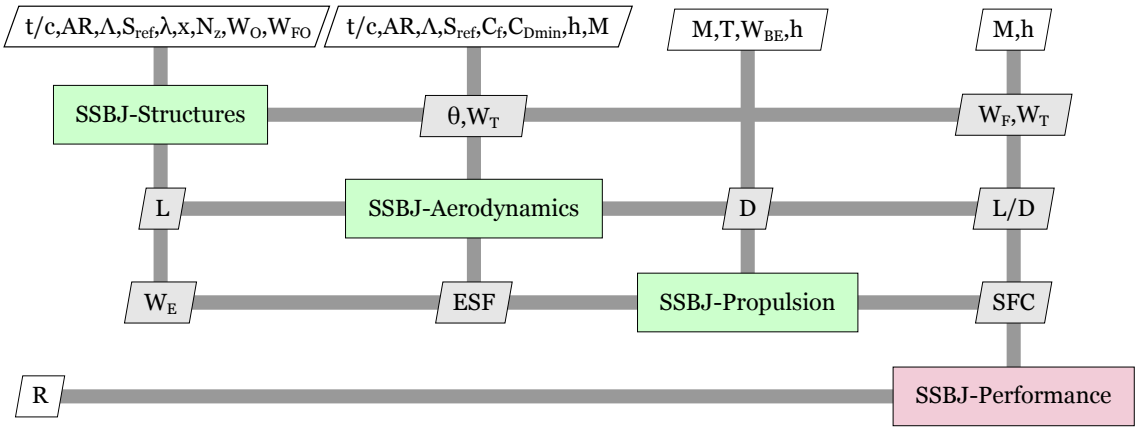


Figure 4.1: Data dependency between the different design disciplines of the SSBJ.

The aircraft design process is implemented using four different design disciplines coupled between each other, which are structures, aerodynamics, propulsion and performance. Each discipline consists on a set of numerical equations which are valid for the early conceptual design phase. The inputs and outputs of these disciplines, as well as the coupling variables, are shown in figure 4.1.

The optimization problem can be formulated as:

$$\begin{aligned} &\text{maximize} : R \\ &\text{with respect to} : t/c, M, h, AR, \Lambda, S_{ref}, \lambda, x, C_f, T \end{aligned}$$

Finally, the XDSM of the SSBJ would look similar to the one shown in figure 4.2.

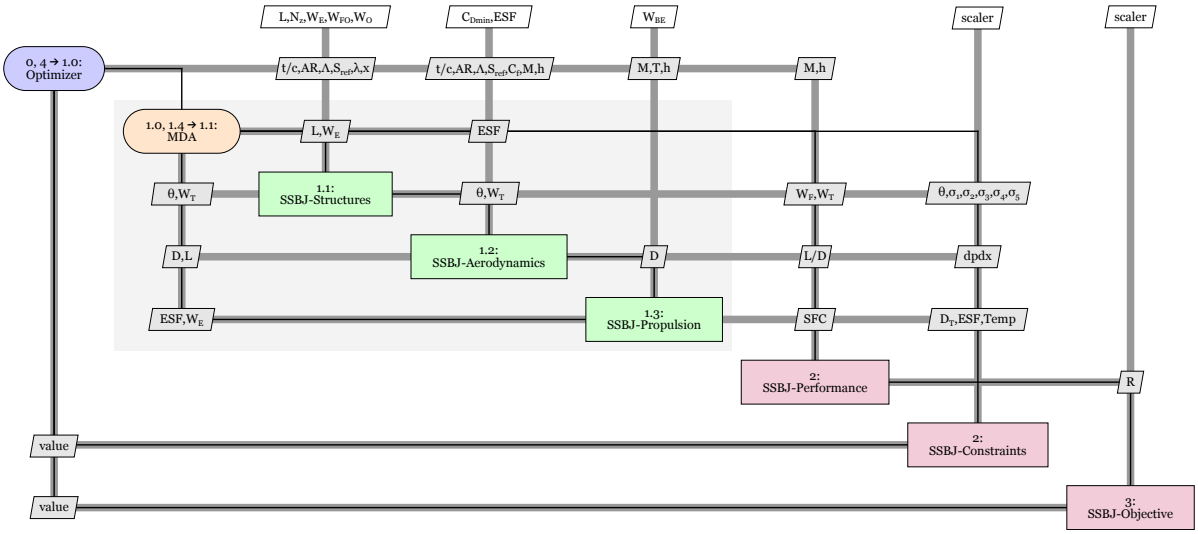


Figure 4.2: XDSM of the SSBJ problem.

Some constraints have been added to check that the design is feasible, as well as some reference values (scaler) to make the constraints and the objective non dimensional. A convergence loop has been included to ensure that the solution is compatible with all the different design disciplines. Finally, an optimizer block is added, being responsible for providing the different design vectors.

Architectural influences in the SSBJ problem

In the SSBJ problem, the geometry of the wing can be determined using five design variables (t/c,Λ,λ, AR and S_{ref}). These variables are inputs of design disciplines where the wing geometry is needed. However, it could be an architectural decision to include winglets to reduce the induced drag generated by the wing (function fulfillment architectural decision).

In this case, two new variables would be added to define the geometry of the wing, which are the length of the winglet (l_w) and the cant angle (δ), as shown in figure 4.3. As a result, depending on an architectural decision, these variables would be included or not in the MDO problem formulation/execution. These variables are going to be called **Conditional Variables**.

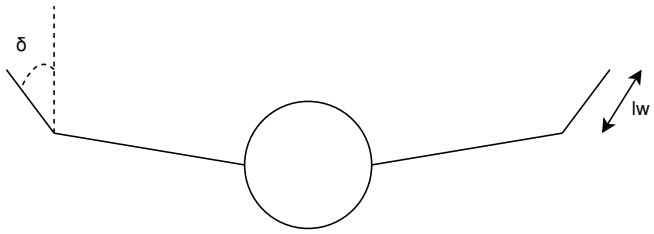


Figure 4.3: If a winglet is included in the architecture, the length of the winglet (l_w) and the winglet cant angle (δ) have to be included when defining the wing geometry.

Sometimes more complex data structures are exchanged between disciplines, being recommendable that the MDO platform can also deal with modifications in these data content. Consider as an example a structure discipline which needs as an input an array representing the location in the y axis of the wing ribs (Y_ribs).

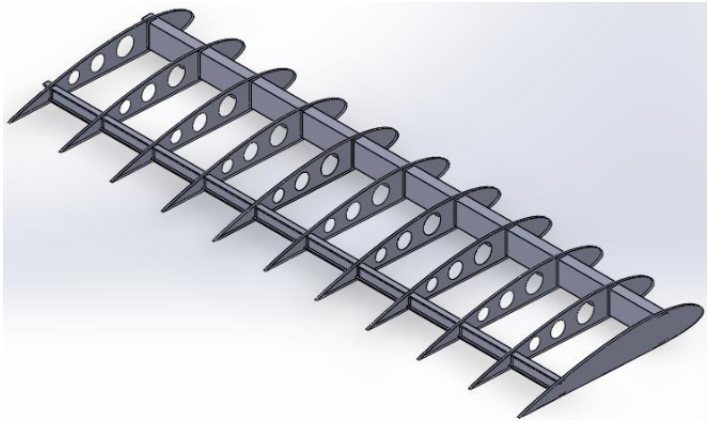
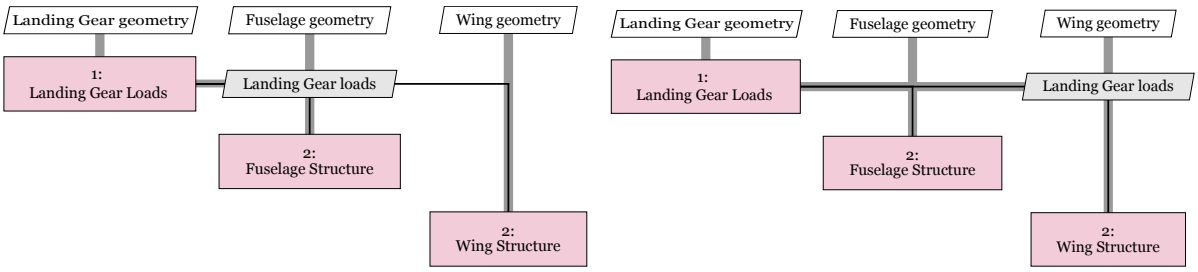


Figure 4.4: The number of ribs is now another design variable of the optimization problem. As a result, the number of y locations to be optimized will change, modifying the data length to be inputted to the structures discipline. Figure taken from Ali et al., 2021.

When performing system architecture, the number of ribs (component instance architectural decision) could be another design variable of the problem (figure 4.4). As a result, the length of this array would change depending on the architecture, being another more complex case of Conditional Variables MDO platforms should be able to deal with.

Connection choices decisions also lead to influences in the MDO problem. This is what happens in the case of the landing gear location. Suppose there are two structure disciplines, one for the fuselage and one for the wings. Depending on where the landing gear is located, the landing gear loads will be an input for the wing or the fuselage structure disciplines. This results in a change in the **connections** existing in the MDO problem, as shown in figure 4.5.



(a) Landing gear attached to the fuselage

(b) Landing gear attached to the wing

Figure 4.5: When the landing gear is attached to the fuselage, the landing gear loads discipline will be connected to the fuselage structure discipline. When the landing gear is attached to the wing, this connection is deleted and a new one is generated between the loads discipline and the wing structure one.

Another possible architectural decision that could be allowed to change would be the number of engines. Consider the case where the propulsion discipline also needs as an input the geometry of the engine. Then, if each engine has different geometrical properties, the propulsion discipline would be necessary to be repeated a number of times equal to the number of engines **(discipline repetition)**.

The final architectural decision that is going to be included in this example is the fuselage material (component property architectural decision), which could be chosen to be metallic or composite. Depending on which decision is taken, the discipline **activated**/used for the structure analyses could be different, leading to two possible structure disciplines, as shown in figure 4.6.

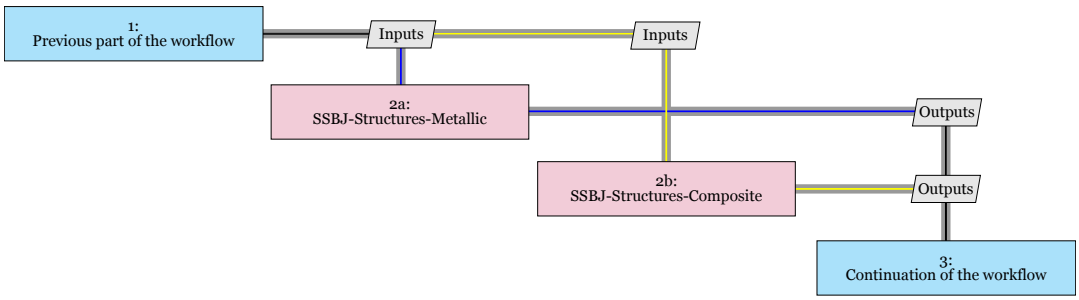


Figure 4.6: This figure shows an example of discipline activation. When a metallic material is chosen, the metallic structures discipline is chosen and the necessary connections are made (dark blue path). In the case of composites, another discipline is used to perform structural calculations, so new connections are needed (yellow path). When multiple disciplines can be added or excluded, connections always change, and even variables could be different too.

As it can be observed, using the different types of architectural decisions, four different types of possible modifications in the MDO problem (influences) were found. All these influences can usually also be found in the rest of the problems in the sample, as shown in appendix A. In the next section, these influences are formalized.

4.3. Architectural Influences

Applying the previous methodology to all the sample problems (appendix A) and using as a base the influences discussed in Bussemaker, Garg, and Boggero, 2022, a total of four different architectural influences in the MDO problem have been discovered. In this section, each of them will be widely introduced, including an example where they occur.

4.3.1. Conditional variables (I1)

Each system architecture is made of different components. Each component is defined by different variables, and as a result the variables existing in the MDO problem might change depending on the system architecture that is evaluated. The variables that might or might not exist in the MDO problem because of an architectural decision are denominated conditional variables.

Apart from the effects it has for the optimizer (discussed in section 3.3.1), the existence of conditional variables leads to the modification of inputs and outputs of the different disciplines (where disciplines refer to both design disciplines and constraints). This means that the inputs and outputs of the disciplines have to be readjusted or modified depending on the architecture that is analysed.

As an example, consider an engine design disciplines that needs as inputs the variables defining the geometry of the engine. The diameter of the fan or the number of fan blades will be an input only if a turbofan architecture is chosen (instead of a turbojet or a turboprop, for example).

When dealing with complex system architecture optimization problems, MDO disciplines usually don´t exchange variables individually, but in more complex data structures (for example arrays). It could also happen that because of an architectural decision, part of these data (or all) is included or not in the MDO problem formulation.

All these complex data could ultimately be modelled as multiple individual variables, and that is why modifications of this data are not considered a separated architectural influence (and ultimately a requirement for system architecture optimization). However, two special cases have been found that should be covered by MDO platforms used for complex problems that are worthy to be mentioned.

First, when arrays are used, it could be that an architectural decision causes the length of an array to change. MDO platforms should be able to deal with dynamic length arrays. An example would be the case where the location in the y-axis of a wing spars are given as inputs of a discipline, being the number of wing spars an architectural decision.

The second case is where multiple variables related with the same component are grouped together under one single instance. For example , two disciplines could exchange all the geometric and aerodynamic properties of a wing on a single data structure. Big parts of this data structure could be modified because of an architectural decision. An example would be the configuration of high lift system to be used. The wing data structure will include multiple different variables depending on the high lift devices chosen, and the MDO platform should deal with these phenomena too.

4.3.2. Data connection (I2)

In some occasions, the data flow connections between disciplines could be modified as a result of an architectural decision. This happens when the disciplines exchanging an existing state variable change because of an architectural decision. MDO platforms used for system architecture optimization must allow to dynamically reroute the connections between disciplines automatically.

An example of an architectural decision causing this influence is the landing gear placement in an aircraft (section 4.2). In the previous example, depending on where it is chosen to place the landing gear (under the wing or under the fuselage), the landing gear loads will be an input to the wing or to the structure discipline, leading to changes in connections between disciplines.

4.3.3. Discipline repetition (I3)

There are some cases where a discipline has to be repeated multiple times, being in some cases the number of repetitions dependant on an architectural decision. This involves that for each repetition of the discipline, some inputs and outputs will have to be modified. New connections will be needed to be generated or activated too, depending on the implementation (more information in section 4.5.2).

A simple example could be the case of a design discipline that calculates the performance of an helicopter rotor depending on the rotor parameters. If the number of rotors is an architectural decision, and they have different parameters, the number of times the discipline will be repeated will vary depending on the architecture being evaluated.

4.3.4. Discipline activation (I4)

Similar to the case of conditional variables, there are some cases where as a result of an architectural decision, a discipline can be included or not in the MDO problem formulation/execution. This is usually found in cases where two different technologies are available to perform a certain function, and each one demands different disciplines (or tools).

An example would be the design of an aircraft propulsion system where the source of energy is an architectural decision. In the case that an electric architecture is chosen instead of a conventional one, the discipline(s) and the tools needed to evaluate its performance will be different. The inclusion or exclusion of the different possible disciplines will lead to the inclusion/exclusion of variables and connections in the MDO problem too.

4.4. System Architecture Optimization Example cases: Aircraft propulsion system

Two system architecture optimization problems dealing with aircraft propulsion system design are going to be presented. The aim is to show that the architectural influences previously discovered are real modifications that take place when MDO is wanted to be included in system architecture optimization.

4.4.1. Jet engine problem

The first problem is the one presented in Bussemaker, De Smedt, et al., 2021. This problem aims to design a conventional aircraft propulsion system, existing different possible architectural decisions such as the inclusion of a fan or the number of compressor stages (figure 4.7).

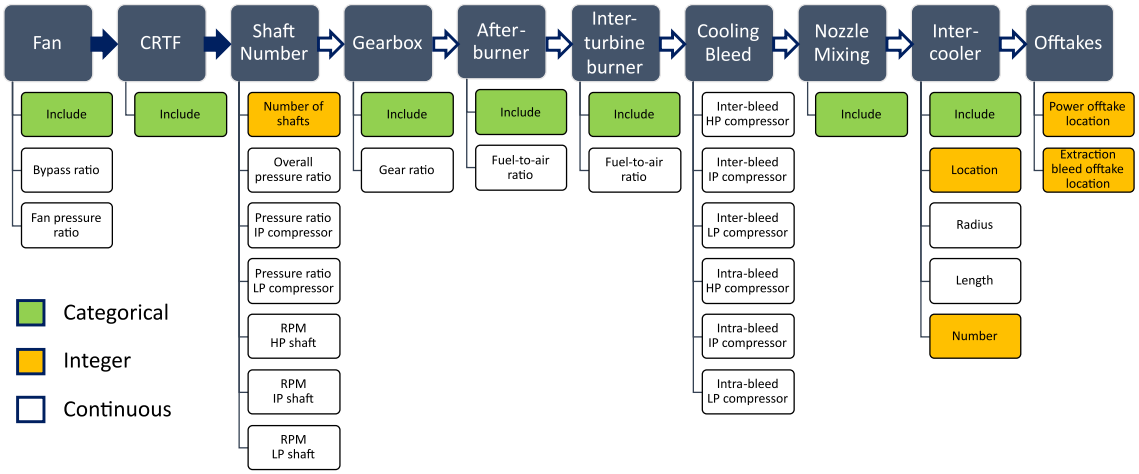


Figure 4.7: Possible architectural design space of an aircraft jet engine . Figure taken from Bussemaker, De Smedt, et al., 2021 .

This problem is characterized by the existence of multiple **Conditional Variables (l1)**. Only if a component exists in the architecture, its corresponding variables (such as weight, length or emissions) will also exist in the MDO problem. An example could be the gearbox. If a gearbox is included in the architecture, then its associated variables (the weight in this case) will be included in the MDO problem. If not, those variables will be excluded from the MDO problem formulation/execution (figure 4.8).

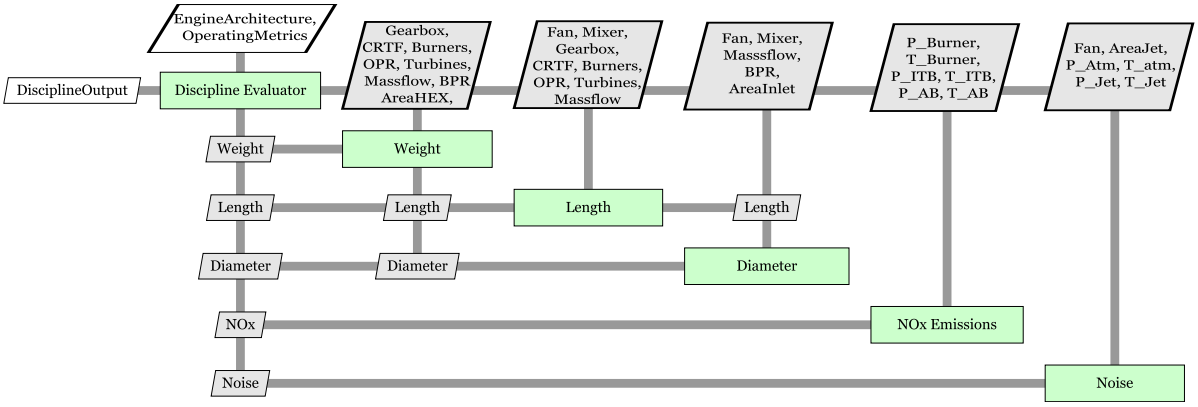


Figure 4.8: Data graph of the design disciplines used to evaluate the performance of the different engine architectures. Figure taken from Bussemaker, De Smedt, et al., 2021 .

4.4.2. Hybrid electric aircraft propulsion system problem

Architectural influences can also be found in the problem presented in Bussemaker et al., 2023. The aim of this problem was to design the propulsion system of an aircraft considering different possible sources of energy (hybrid-electric). To do so, it connects an overall aircraft design program called OpenAD (Woehler et al., 2020) and a mission performance analyses tool called OpenConcept (Brelje and Martins, 2018).

During the optimization process, there is a certain section where the propulsion system thrust is calculated using as an input (among others) the mechanical power generated either by the turboshaft or the motor. This process is shown, in a simplified version, in figure 4.9.

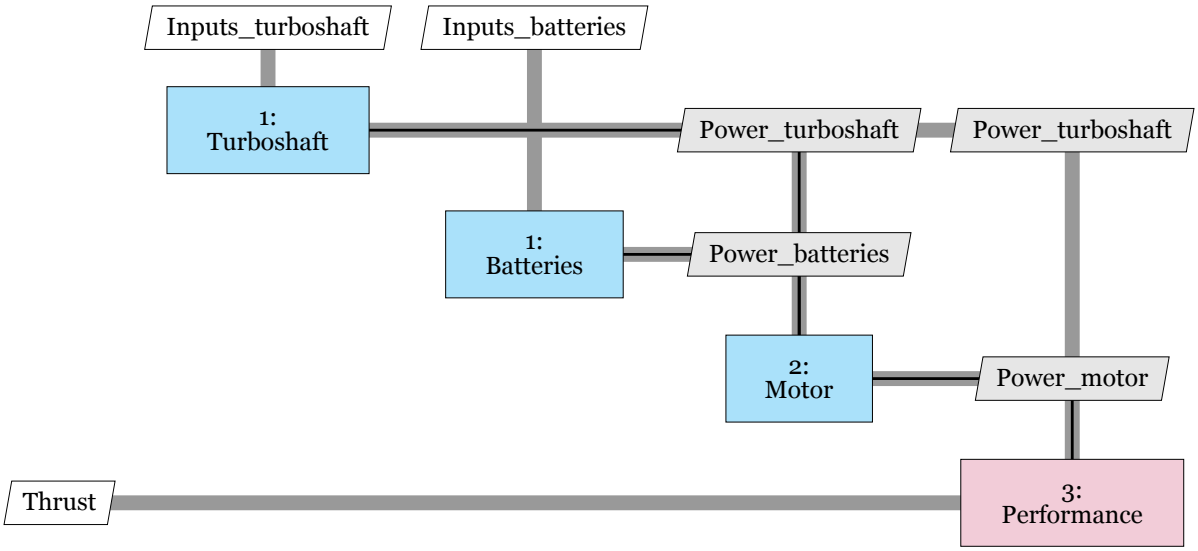


Figure 4.9: Simplified process for the calculation of the aircraft propulsion thrust considering different sources of mechanical power.

Each of the different components that can intervene (directly or indirectly) in the generation of mechanical power are modelled as disciplines (Turboshaft, Batteries and Motor). These disciplines will only exist in the optimization process if their associated component are included in the system architecture being analysed, and are consequently an example of **Discipline activation (I4)**. Also, the number of engines is an architectural decision and each engine can have different components, so **Discipline repetition (I3)** is also found.

Finally, there are two possibilities where the turboshaft and the motor disciplines are included at the same time. The first one is when both are used to generate mechanical power (parallel architecture). The second one is when the power generated by the turboshaft is used to feed the motor (series architecture). Depending if the mechanical power of the turboshaft is used to generate thrust or is used to feed the motor (architectural decision), the ”Power_turboshaft” variable will be an input to the Performance or Motor discipline respectively. This would be an example of **Data Connection (I2)**.

4.5. Strategies to deal with architectural influences

In the previous sections, the different types of architectural influences were discussed. Also some real system architecture optimization problems with these influences were shown. Now, some of the possible methods that MDO platforms can implement to deal with these influences (called strategies) will be presented . To do so, a research has been carried out in the MDO platforms introduced in section 3.3, paying close attention on how they try to deal with each of the different influences.

Of all these different platforms, only OpenMDAO and GEMSEO will be considered for this discussion, as there are examples where their code was extended by the user to deal with architectural influences. Apart from this research, multiple conversations with experts on the field have been carried out. This allowed to obtain more strategies that still have not been implemented, but that should also be considered. They were also used to confirm the validity of the different strategies.

4.5.1. Conditional Variables (I1)

Two different approaches can be used to implement conditional variables in the MDO platform. The first approach is to define all possible inputs and outputs of each discipline, and then ignore the components that are not necessary, usually by inputting a default value to them such as a zero.

As an example, consider the winglet inclusion case discussed in section 4.2. All the geometric variables of the winglet (l_w and δ) would be included as inputs of the discipline. Then, when a winglet does not exist, these variables would be given a value of 0. This strategy is not efficient from a computational point of view, as there are unnecessary values being dragged during the calculations. Also in the case of using external tools, some mapping might be needed to be used to delete these unnecessary elements.

A different and better approach is to change automatically the inputs and outputs of disciplines depending on architectural decisions. Therefore, variables that don´t exist in the MDO problem will be deleted automatically from the inputs and outputs of the different disciplines.

These strategies can also be applied to more complex data structures, such as arrays, assuming that each component of the array is an individual variable. It could also be applied to group of variables, if they are allowed as inputs and outputs of disciplines.

4.5.2. Data connection (I2)

To understand the possible strategies to deal with this influence, it is necessary first to understand how connections between disciplines can be generated. Two different approaches are discussed in van Gent, 2019, depending on the naming convention used for the problem variables.

The first one is a decentralised approach. Here, each discipline can have its own naming convention. As a consequence, every time that a variable has to be exchanged between different elements of the optimization problem, the user has to manually define the connection. As a result, the user needs to know and keep track of all possible connections to formalize the optimization problem.

However, this is not needed in a centralised approach (as discussed in section 2.3.1). As all variables adhere to a common protocol (Common Data Schema), connections between disciplines can be made automatically, once all inputs and outputs are declared (van Gent et al., 2017). Depending on the naming convention that is used, the strategy to deal with data connection architectural influence will be different.

When a decentralized approach is used, data connection is implemented by creating and deleting connections depending on the architectural decision. In the case of the centralized approach, connections are made automatically depending on the disciplines inputs and outputs. As a result, data connection is implemented by deactivating disciplines inputs and outputs depending on the architectural decision, which at the end will deactivate the different connections.

As a summary, in a centralized approach what are included or excluded are the variables determining the connection between the disciplines, and in the decentralized approach it is the connection itself.

4.5.3. Discipline repetition (I3)

There are two different procedures to implement discipline repetition. The first one is to generate a different discipline for each time that the discipline has to be repeated (parallel configuration), as shown in figure 4.10.

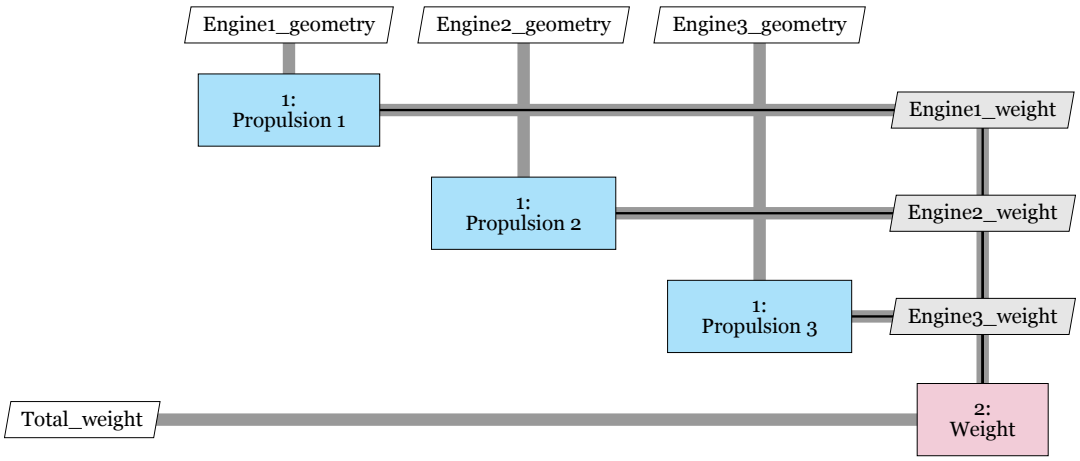


Figure 4.10: In the parallel implementation, each time that the discipline has to be repeated, a new discipline block is created. This method allows to execute the different instances in parallel.

However, there can be another approach where only one discipline instance is used for all the repetitions. This is the series configuration (figure 4.11), where the workflow ”enters” several times into the same discipline. This approach requires that inputs and outputs are redeclared each time the workflow enters into the same discipline. It also needs to save the different variables values until the last iteration is finished and they can be passed to the next correspondent elements in the workflow. The great advantage the series approach has is that it is not needed to know the maximum number of possible repetitions of the discipline for its implementation.

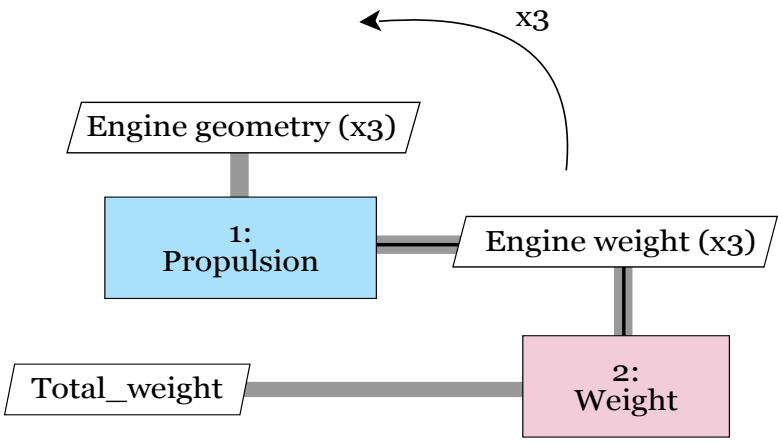


Figure 4.11: In the series implementation, the workflow enters into the same discipline multiple times. In the example, the engine geometry parameters and the engine weight are going to be different at each iteration. This approach does not allow parallel execution, although allows to implement repetition without knowing the maximum number of repetitions.

4.5.4. Discipline activation (I4)

To implement discipline activation, the only procedure is to define those disciplines that might or not be included in the MDO problem together with the condition for their inclusion in the formulation/execution process. This condition is then used to check certain architectural decisions, and, depending on the result (True or False), the discipline is included or not.

Including a new discipline involves several additional modifications to the MDO problem. First, new variables might be needed to be included in the MDO problem. Also, as each discipline has several inputs and outputs, new connections will be needed to be generated.

It is at this latest step where the only difference between platforms could be found, similar to the case of data connections. If a centralised naming convention is used for variables, these connections will be created automatically. In the case of a decentralized convention, these connections have to be stated manually under the same condition used for the discipline activation.

5

MDAx adaptation to system architecture optimization

To model an MDO problem that can be readjusted automatically during the execution process to each system architecture it is necessary to deal with four possible modifications, called architectural influences. This chapter will focus on how to extend MDAx backend code to deal with each of these four architectural influences, considering only the case where the workflow is exported to RCE.

The structure of this chapter in the following one. Each architectural influence implementation will be first divided into smaller requirements that are needed to be satisfied. After that, each of these sub-requirements will be further explained. Finally, the methodology used to satisfy them will be discussed.

5.1. Discipline activation (I4)

As mentioned in section 3.3.3, some modifications are needed to be done manually before the workflow exported by MDAx can be executed in RCE. As a result, there is only a possible strategy to implement discipline activation in RCE. MDAx must export an unique workflow with all the possible disciplines that can exist in the MDO problem formulation. Then RCE has to determine during the execution if a discipline has to be executed or not, depending on the system architecture being analysed.

The methodology proposed is the following. Those tools whose execution depend on an architectural decision will have attached to them the condition determining their inclusion in the MDO problem execution (called from this point activation logic assertion). When a tool has one of these assertions attached in MDAx, in the RCE export it will include an additional script which is executed before the tool. This script will check if the assertion is satisfied. If it is, the tool will be executed as normal, and if it is not, the workflow will ”skip” the tool and continue to the next discipline (figure 5.1).

The five steps or requirements needed to implement discipline activation in MDAx are:

1. **Create a configuration file**
2. **Attach the activation logic information to MDAx disciplines**
3. **Determine and verify activation logic assertions**
4. **Adapt the collision detection process to activation logic**
5. **Include activation logic in the RCE export file**

The next sections will cover with more detail each of these requirements, including the implementation process.

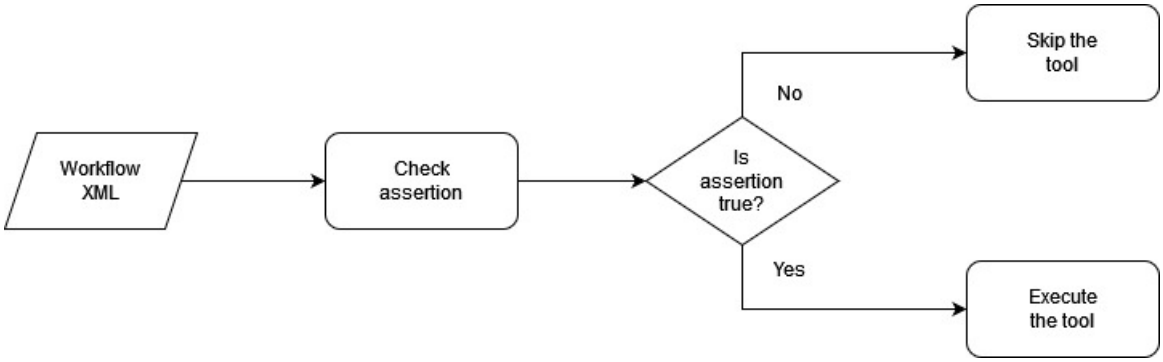


Figure 5.1: Schema of the process used in RCE to determine if a discipline is executed or not. At the start of those tools that can or not be included in the MDO problem execution there will be a script. This script will check if the condition (assertion) for the inclusion of the tool is satisfied. If it is, the tool will be executed. If not, it will be skipped and the workflow will directly ”jump” to the next tool.

5.1.1. Creation of a configuration file

To include all the different architectural influences in MDAx, it is necessary to create a new file where all the information of a tool regarding these architectural influences is stored. This file, called configuration file, is necessary to import any tool related with architectural influences, together with the input and output files.

This configuration file has been decided to be created in JSON to improve the human readability with respect to XML. JSON files can store many types of information, from strings or numbers to more complex structures such as arrays. Each piece of information has associated a certain key that identifies it.

For the implementation of activation logic, a key called ”activationLogic” has been added to store the condition (assertion) that determines if a tool is included or not in the MDO problem execution. Additional metadata that might be useful in the future has been added to the configuration file too, such as the name of the tool, its version or some additional notes. An example of a tool configuration file can be observed in figure 5.2.

```
{"activationLogic": "XPathExists('/sellar/variables/variable/x1')",
 "executionName": "D1",
 "name": "D1",
 "notes": "",
 "version": "1.0"
}
```

Figure 5.2: Example of a tool configuration logic file. A key called ”activationLogic” is used to store the condition/assertion determining the inclusion of the tool in the MDO problem execution.

5.1.2. Attachment of activation logic information to MDAx disciplines

Each discipline in MDAx is represented by a Python class called ”function block”, where a class is just a data structure that represents a certain object (in this case a discipline in the MDO problem). Each class can contain different characteristics that differentiate each object from another and that are called attributes. Attributes can be simple variables or arrays, or even more complex data structures such as another Python class[1].

[1] More information about classes and attributes in Python can be found in https://docs.python.org/3/reference/datamodel.html

The first step to attach to a tool in MDAx the assertion determining its inclusion in the MDO problem execution has been to create a new attribute for the function block class called activation logic. Then, the import function used by MDAx to create a discipline has been extended to also accept a configuration file. Finally, all the different keys existing in the configuration file are parsed, and when the ”activationLogic” key is found, its information is stored in the corresponding attribute of the tool´s function block.

5.1.3. Determination and verification of activation logic assertions

The next step to implement activation logic in MDAx is to determine the different types of conditions (assertions) that could be used to assert if a discipline is included in the MDO workflow execution. These conditions depend on architectural decisions, which can always be expressed as a function of the information existing in the workflow XML file.

More specifically, these assertions can ultimately be expressed though the existence of certain nodes in the XML file, or through the information stored inside them. Three big different types of assertions have been identified, which are:

1. Existence of a node
2. Number of nodes
3. Content of a node

The three different assertion types are going to be introduced, including the different sub cases. Also an example based on a rocket design will be introduced for each of them, based on the workflow XML shown in figure 5.3.

```
1  <rocket>
2    <Propulsion>
3      <Liquid_engine>
4        <VULCAIN></VULCAIN>
5      </Liquid_engine>
6    </Propulsion>
7    <Geometry>
8      <material>aluminium</material>
9      <number_of_fins>4</number_of_fins>
10   </Geometry>
11 </rocket>
```

Figure 5.3: Workflow XML used as example to show the different types of assertions. It shows some possible components/attributes of a rocket, such as the number of engines or the material used.

- **Existence of a node**: On some occasions, the condition determining if a tool is included in the MDO problem execution is the existence of a component in the system architecture. The existence of a component in the system architecture usually can be translated in the existence of a node in the workflow XML file.

  An example would be the case of a tool which only exists if a liquid propulsion engine is found in the system architecture. In this case, the tool would only be included if the node ”’rocket/Propulsion /Liquid_engine’” exists in the workflow XML file.

- **Number of nodes**: Sometimes, a component number of instances architectural decision could cause a certain tool to be included or not in the MDO problem execution. This architectural decision is usually translated in the workflow XML file through the repetition of the node representing the component multiple times.

Three different subcases can be differentiated, depending if it is wanted to be checked if the number of components is lower, equal or higher than a certain value. Using the rocket design example, a possible activation condition could be that the number of engines in the system architecture is less, equal or more than 2. This would be determined checking if the corresponding node (”’rocket/Propulsion/Liquid_engine/engine’”) is repeated less, equal or more than 2 times respectively.

- **Content of a node**: This case can usually be attached to component properties architectural decisions, although all architectural decisions could lead to modifications in the content of a certain node in the workflow XML tree. Six different subcases can be differentiated.

  1. Contain any value
  2. Contain an exact string
  3. Contain a fragment of a string
  4. Contain a value lower than a reference
  5. Contain a value equal to a reference
  6. Contain a value higher than a reference

  The first possible condition would be the existence of any kind of information inside a certain node. For example, it could be a condition to include a tool that a material has been selected for the rocket. Then, it would be needed to be checked if the node ”’rocket/Geometry/material’” contains any data.

  The second and third conditions are used when the inclusion of a tool depends on the existence of an exact string, or a fragment of it, in a node, respectively. As an example of them, it could be that the condition determining the inclusion of a tool is that the material chosen is aluminium 6061, or that it is at least any alloy of aluminium. This would mean checking if the node ”’rocket/Geometry/material’” contains exactly ”aluminium6061” or that it at least contains the string ”aluminium” respectively.

  The three final sub cases are conditions where the number inside a node has to be lower, equal or higher than a certain reference number. If the condition is that the number of fins is lower, equal or higher than 3, then it has to be checked if the node ”’rocket/Geometry/number_of_fins’” contains a value lower, equal or higher than 3.

Different Python classes have been implemented in MDAx to check each of these different assertions. These can be found in table 5.1, which shows the Python class name and its arguments for checking each type of assertion. The last column shows an example of how each different type of assertions has to be indicated in the configuration file, taking the previous rocket design as reference.

Table 5.1: Python classes implemented in MDAx to check activation assertions.

Assertion	Class	Arguments	Example
Existence of a node	XPathExists()	node xpath	XpathExists(’rocket/Propulsion /Liquid_engine’)
Number of nodes (lower)	XPathNLt()	node xpath, reference	XPathNLt(’rocket/Propulsion /Liquid_engine/engine’,2)
Number of nodes (equal)	XPathNEq()	node xpath, reference	XPathNEq(’rocket/Propulsion /Liquid_engine/engine’ ,2)
Number of nodes (higher)	XPathNGt()	node xpath, reference	XPathNGt(’rocket/Propulsion /Liquid_engine/engine’,2)
Contain value	ElHasValue()	node xpath	ElHasValue(’rocket/Geometry /material’)

Contain exact string	ElStrEq()	node xpath, string	ElStrEq('rocket/Geometry /material','aluminium6061')
Contain fragment of a string	ElStrContains()	node xpath, string	ElStrContains('rocket/Geometry /material','aluminium')
Contain value lower than reference	ElNumLt()	node xpath, reference	ElNumLt('rocket/Geometry /number_of_fins', 3)
Contain value equal than reference	ElNumEq()	node xpath, reference	ElNumEq('rocket/Geometry /number_of_fins', 3)
Contain value higher than reference	ElNumGt()	node xpath, reference	ElNumGt('rocket/Geometry /number_of_fins', 3)

Finally, it should be remarked that more complex conditions consisting on multiple assertions can also be used. This is possible thanks to the inclusion of logical operators such as OR (|), negation ($\sim$) and AND (&). For example, if the condition to include a tool is that it has 2 engines and is made of aluminum, the assertion included in the configuration file would be "XPathNEq('rocket/Propulsion/Liquid_engine/engine' ,2) & ElStrEq('rocket/Geometry/material','aluminium')".

5.1.4. Adaptation of the collision detection process to activation logic

At this point, MDAx is already able to accept a possible configuration file to define a tool that includes (among other information) the assertion determining the inclusion of the tool when executing the MDO problem. It is also capable to attach this assertion to the Function block class defining each discipline. Python classes to determine if these assertions are true or false have also been created and implemented in MDAx.

However, there is a still a task to be solved regarding the MDO problem formulation, and this is the collisions detection process. When two disciplines provide the same output, then a collision is detected, impeding to export the workflow. However, when implementing activation logic, it is common that two disciplines have the same output and that they don´t exist at the same time in the workflow execution. In this case, MDAx would detect a collision, although it would not be a real collision.

To exclude these cases, the collision detection process has been modified. For each variable causing a collision, it is checked if the disciplines providing that variable as output include any activation logic assertion. In the case all of them do, the collision will be ignored. It is assumed that the activation logic assertions included by the user are exclusive between each other, meaning that those disciplines will never provide the same output at the same time (as only one of them will be executed in the workflow for each system architecture).

5.1.5. Inclusion of activation logic in the RCE export file

The last step to include discipline activation in MDAx is to adapt the RCE export to this architectural influence. In the function used to generate the MDO workflow to be executed in RCE, each discipline is represented by a Python class (called RCEToolNode). This Python class contains all the different RCE elements for each discipline (also represented by Python classes), including their inputs, outputs and the connections between them (section 3.3.3).

Three new elements have to be added to the RCEToolNode class. These are:

- **Activation logic script**: This element contains all the different assertion classes described previously, as well as the activation logic assertion of the tool. Taking the XML workflow file as an input, it will check if the assertion is satisfied. It will provide as output the same XML workflow file, as well as a boolean variable stating if the assertion was satisfied or not.

- **Switch**: This element is based on an existing element with the same name in RCE. To script it, reverse engineering has been carried out, researching RCE workflows that include switches and looking how they are scripted. It takes as input the workflow XML file and the boolean given by the activation logic script. If the boolean is true, then the workflow XML will be passed to the tool (specifically to the base splitter). If not, the workflow XML will go to the third new element, the joiner.

- **Joiner**: This element, also based on an existing one in RCE, takes as input two files. As soon as it receives one of the files, it will give it as output. Joiner are usually used when only one of the inputs can exist, which is this specific case. The first input would be the workflow XML file coming from the switch when the tool is not executed. The second input would be the XML coming from the merger when the tool is executed. Therefore, independently of executing or not the tool, the workflow will continue to the next discipline.

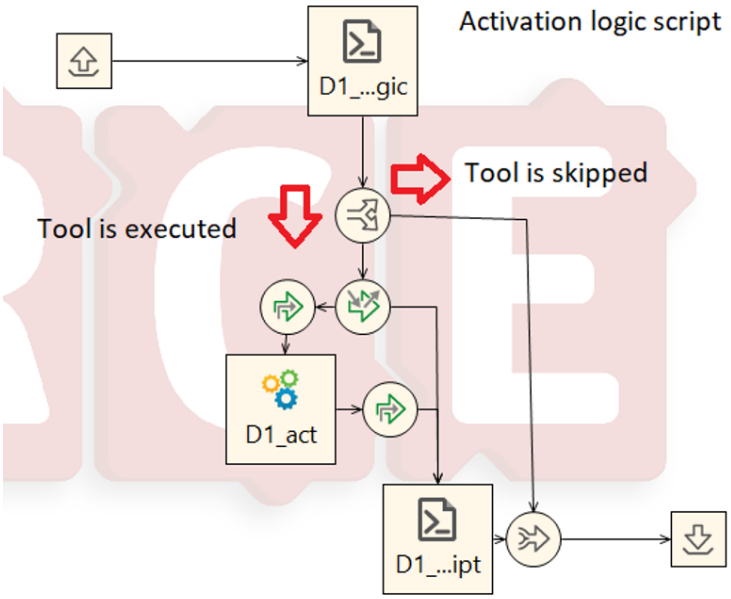


Figure 5.4: Example of a tool with activation logic in RCE. First, the workflow XML enters into the activation logic script, which will check if the activation logic assertion attached to the tool is true or false. If is true, the switch (circle with two arrows) will pass the workflow XML to the tool and the tool will be executed. If not, the workflow XML will pass directly to the joiner (the block next to the merger). In both cases, after the joiner the XML workflow file will be passed to the next discipline.

These three new elements will only be included in the RCEToolNode class when the MDAx discipline has attached an activation logic assertion. Finally, the connections between the different elements constituting each MDAx discipline in RCE have been also modified to include the activation logic. The RCE implementation of activation logic according to figure 5.1 can be observed in figure 5.4.

5.2. Discipline repetition (I3)

The methodology that is going to be followed to implement discipline repetition in MDAx is going to be the serial configuration presented in section 4.5.3. There will be an unique discipline instance where the workflow will enter multiple times, changing automatically the inputs and outputs on every iteration.

To achieve this, it is important to know that the existence of multiple instances of a component manifests in the workflow XML file by the repetition of the nodes representing the component (and their properties). Therefore, it will be necessary to select what parts of the workflow XML file will be used to take the inputs and write the outputs at each iteration, as these will change. These processes will be carried out by the ”Global to Local” block. The ”Local to Global” block will prepare the tool outputs to be merged correctly. Finally, a component called ”Iterator” will be used to determine when the repetition has ended. A schema of this methodology is shown in figure 5.5.

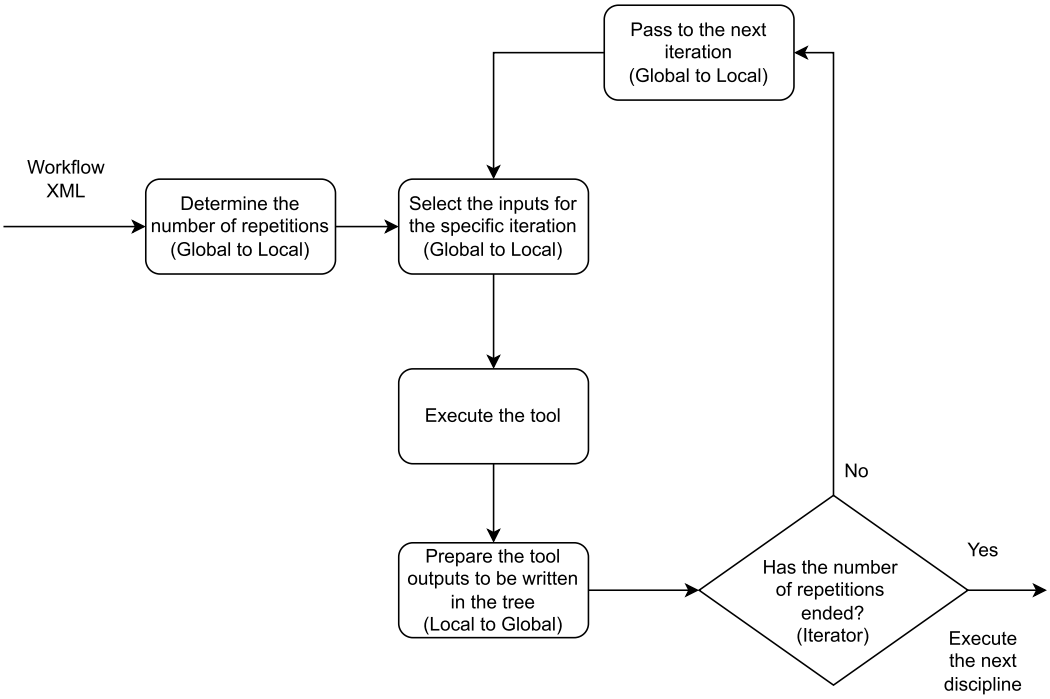


Figure 5.5: Flowchart of the procedure used to implement discipline repetition in MDAx. The Global to Local component will be in charge of multiple processes, such as determining the number of iterations, keeping track on the iteration the execution is at each moment or selecting the inputs/outputs for each iteration. The Local to Global will prepare the tool outputs to be merged in the correct place in the workflow XML. Finally, the iterator block will determine if the repetition of the discipline has ended or not.

The requirements identified to implement discipline repetition according to the methodology described before are:

1. **Extend the configuration file and the function block attributes to deal with discipline repetition**
2. **Develop a methodology to determine the number of repetitions**
3. **Develop a methodology to differentiate the variables to be selected for each iteration**
4. **Select the correct inputs/outputs for each iteration (Global to Local)**
5. **Prepare the tool outputs to be merged (Local to Global)**
6. **Determine the end of the repetition process (Iterator)**
7. **Adapt the RCE export to discipline repetition**

5.2.1. Extension of the configuration file and the function block attributes to deal with discipline repetition

The configuration file created for discipline activation can be reused to indicate discipline repetition. A new key has been added for this, called ”repetition” (figure 5.6). This key will contain a number if the discipline has to be repeated a fixed number of times, or a more complex expression if the number of repetitions depends on an architectural decision (more information in the next section).

Once the configuration file has been adapted, a new attribute has been created in the function block where the information regarding discipline repetition included in the configuration file is stored. Finally, the import function of MDAx has been extended to also read the information of this new key and attach it to this new attribute of the function block.

```
{"executionName": "D1",
 "name": "D1",
 "notes": "",
 "repetition": "NumberofInstances('aircraft', 'engine')",
 "version": "1.0"
}
```

Figure 5.6: Example of a tool configuration file including discipline repetition. A key called ”repetition” is used to express how many times a discipline has to be repeated. This number can be fixed or depend on an architectural decision.

5.2.2. Determination of the number of repetitions

Two different cases will be allowed in MDAx for repeating a certain discipline. The first one is to select a fixed number of repetitions for the discipline. The second one is that it depends on the number of instances of a certain component in the system architecture. To calculate the number of repetitions for the second case, a new python class (similar to the number of equal nodes assertions of discipline activation) has been developed. This python class takes as input a node´s xpath (representing a system architecture component) and checks how many times it is repeated in the workflow XML.

The expression to be used in the configuration file to indicate this case of discipline repetition will be ”NumberofInstances()”. As an example, if a discipline has to be repeated a number of times equal to the number of engines in a propulsion system, and the node´s xpath is given by ”aircraft/engine”, then this condition would be expressed in the configuration file as ”NumberofInstances(’aircraft’,’engine’)” [2](figure 5.6).

5.2.3. Differentiation of the variables for each iteration

If a component of a system architecture has multiple instances, the nodes representing these components and their variables in the workflow XML file will usually be repeated multiple times. Consider the case of a system architecture with two different turboprops. As it can be observed in figure 5.7, each engine has a different node for each of their properties (number of blades and thrust). The only exception is the case of the turboshaft model. This is because usually variables with a common value for all the component instances are not repeated in the workflow XML file.

```
1  <Propulsion_system>
2    <engine>
3      <number_of_blades>2</number_of_blades>
4      <thrust>17000</thrust>
5    </engine>
6    <engine>
7      <number_of_blades>4</number_of_blades>
8      <thrust>34000</thrust>
9    </engine>
10   <turboshaft>
11     <model>CT7-9</model>
12   </turboshaft>
13 </Propulsion_system>
```

Figure 5.7: This figure shows how multiple instances of a component translates into multiple nodes in the workflow XML file. If their properties are different, their corresponding nodes will also be repeated multiple times.

[2]The node xpath has to be indicated in two different parts, where the first part contains all the parents nodes and the second one contains the node itself

Consider now a discipline that takes as inputs the engine properties (number of blades and turboshaft model) and gives as output the total thrust it produces. To implement discipline repetition, it is necessary that only the first engine properties are taken as input for the first iteration, and the second engine properties are taken for the second one. For the case of outputs, the tool should write the value of the calculated thrust in different parts of the tree for each iteration.

The problem is that all the nodes (both inputs and outputs) for the different iterations have the same xpath. Therefore, right now it would not be possible to determine which specific nodes are wanted to be selected for each iteration, as all of them are equal. The first step to solve this is to differentiate explicitly these nodes in the workflow XML file.

This can be done adding an attribute to each of these nodes xpath. An attribute in XML is a tag with two parts, the key and the element. For example, for the first engine number of blades, the xpath could be given by "aircraft/engine[UID = 'engine_1']/number_of_blades", where the key is "UID" and the element is "engine_1" (figure 5.8).

```
1  <Propulsion_system>
2    <engine UID="engine_1">
3      <number_of_blades>2</number_of_blades>
4      <thrust>17000</thrust>
5    </engine>
6    <engine UID="engine_2">
7      <number_of_blades>4</number_of_blades>
8      <thrust>34000</thrust>
9    </engine>
10   <turboshaft>
11     <model>CT7-9</model>
12   </turboshaft>
13 </Propulsion_system>
```

Figure 5.8: Possible implementation of attributes to differentiate nodes representing the same variable, but corresponding to different component instances.

Forcing the designer to use a different attribute for each component instance node, the variables for each iteration can be differentiated. The only condition will be that all components instances share the same words for the key and the element, and that the latest include the component instance number in its definition.

Now nodes corresponding to different iterations can be differentiated, as they have different attributes. However, there is still no methodology to determine which specific part of the XML should be taken for each iteration. This is going to be solved in the Global to Local component.

5.2.4. Selection of inputs/outputs for each iteration: Global to Local

The Global to Local component will be a script inside the workflow (similar to activation logic script) with two different main functions. The first one will be to **determine how many times it is necessary to repeat the discipline**. This can be done including inside this component the information stored in the function block attribute regarding repetition. This information will contain the number of repetitions directly (with an integer) or indirectly (with an assertion of the type "NumberofInstances"). In both cases the number of repetitions will be determined. Then this information will be passed to the iterator, so that it knows which is the maximum number of iterations.

The second main function of this component is to select **which part of the workflow XML file is going to be used for the inputs and the outputs at each iteration**. It converts global data (workflow XML file) to local data (tool). First the case of the inputs is going to be considered. To understand the procedure that is going to be used, it is necessary to explain with more detail the methodology used by the input filter to determine the inputs of the tool.

The input filter stores the tool´s input file. It reads each of the different xpaths of this file and try to search them in the workflow XML file. Every time it finds the xpath, it takes this xpath from the workflow XML file (and the value of the node it represents) and include it in the file to be given as input for the tool execution.

The problem is that when looking for these xpaths in the workflow file, xpath attributes are not taken into account if they don´t exist in the tool input file xpaths. This means that the input filter will still take the nodes for all the iterations at once, instead of taking just the part of the workflow XML corresponding to that iteration.

To solve this, the following idea was proposed. When repetition is used and a variable is going to have different values for each iteration, then its node in the tool input file will have an attribute similar to the ones added in the workflow file, but substituting the component instance by the keyword ”INDEX” (figure 5.9).

```
1 <Propulsion_system>
2   <engine UID="engine_{INDEX}">
3     <number_of_blades></number_of_blades>
4   </engine>
5   <turboshaft>
6     <model></model>
7   </turboshaft>
8 </Propulsion_system>
```

Figure 5.9: When a variable is going to take different values for each iteration, it should be indicated in the tool input file with an attribute in its corresponding node. This attribute must include in the element the keyword ”INDEX” and has to be the same to the ones used for the workflow file.

Then, a counter could be added that increases an unit per iteration. This counter could be passed to the input filter, where every time that the INDEX keyword was found in an xpath, it would be substituted by the iteration counter. As an example of this procedure, for each iteration the ”INDEX” in figure 5.9 would be substituted by a 1, by a 2,... allowing to select the correct part of the workflow XML for each iteration.

However, there were two problems that lead to small modifications to this approach. The first one is that the number of iterations has not to be equal to the number of the component instance to be used as input for the discipline. This is because both activation and repetition can happen at the same time in a discipline. Consider the case of a workflow XML file with three engines, and that only the first and the third have to enter to a certain discipline as inputs. For the first iteration, the first engine is needed (component instance 1). However, the second iteration demands the third engine (component instance 3). As it can be observed, the iteration counter and the instance number have not to be the same.

This problem has been solved introducing two different counters, one for the iteration and another one for the component instance to be considered for that specific iteration. With this, it is possible to know which component instance should be considered for each iteration. Nevertheless, there is still a problem in the previous procedure. This is that the xpaths in the filters have to be fixed during the workflow

execution.[3].

As a consequence and using the previous example, the input filter will always look for engines nodes that have the keyword INDEX in the attribute, instead of the corresponding component instance number (1,2,..). To solve this, the Global to Local component will have two different XMLs as output. One for the merger (equal to the workflow XML file received as input) and another one modified for the input filter. In this latest file, the component number instance of the specific iteration will be substituted by the INDEX keyword (figure 5.10). As a result, the input filter will select the correct nodes to be used as inputs.

```
<Propulsion_system>
  <engine UID="engine_1">
    <number_of_blades>2</
        number_of_blades>
    <thrust>17000</thrust>
  </engine>
  <engine UID="engine_2">
    <number_of_blades>4</
        number_of_blades>
    <thrust>34000</thrust>
  </engine>
  <turboshaft>
    <model>CT7-9</model>
  </turboshaft>
</Propulsion_system>
```

```
<Propulsion_system>
  <engine UID="engine_{INDEX}">
    <number_of_blades>2</
        number_of_blades>
    <thrust>17000</thrust>
  </engine>
  <engine UID="engine_2">
    <number_of_blades>4</
        number_of_blades>
    <thrust>34000</thrust>
  </engine>
  <turboshaft>
    <model>CT7-9</model>
  </turboshaft>
</Propulsion_system>
```

Figure 5.10: At each iteration, the attributes of the nodes with the component instance number for that specific iteration will be modified, so that they include the "INDEX" keyword. In the figure, it is the first iteration, that is why "engine_1" is substituted by "engine_INDEX".

This solves the problem for the inputs. For the outputs, the procedure is explained in the next section.

5.2.5. Preparation of outputs for merging process: Local to Global

Usually, a tool used in MDAx/RCE gives as output an XML file with no attributes. To merge this file correctly with the original workflow file in the merger, it is necessary that the attributes corresponding to that specific iteration are added in the output file. This is done by the Local to Global component .

This file, which is a script similar to the case of the Global to Local, receives as inputs the tool output file and the nodes attributes necessary to be added for this specific iteration, so that the information is written in the correct part by the merger (figure 5.11). These attributes and their location (their corresponding nodes) are given by the Global to Local too.

```
<Propulsion_system>
  <engine>
    <thrust>17000</thrust>
  </engine>
</Propulsion_system>
```

```
<Propulsion_system>
  <engine UID="engine_1">
    <thrust>17000</thrust>
  </engine>
</Propulsion_system>
```

Figure 5.11: This figure shows an example of the input and output files in the Local to Global component. The attributes of the nodes for the specific iteration are added, allowing to later merge correctly the tool output file.

[3]Indeed they could be modified, but this would require to include multiple new files in the workflow. This would increase the computational time and make the integration more complex, so it has been discarded.

5.2.6. Determination of the end of the repetition process: Iterator

This is the last main component necessary to include discipline repetition. This component will be located after the merger. It will receive as input the workflow XML file and the number of times the tool has to be repeated. It will provide as output the same workflow XML file and a boolean stating if the repetitions have ended or not. This boolean is later passed to a switch, which will send the workflow file again to the Global to Local if a new repetition is needed, or will send it to the next tool if not.

5.2.7. RCE export adaptation to discipline repetition

In this section, the different modifications needed to be implemented in the RCE export regarding discipline repetition will be discussed. This will be carried out using figure 5.12 as reference, which shows an example of a tool with discipline repetition in RCE.

The first step has been to include the Global to Local block (G/L) and the joiner above it. This joiner is needed because the workflow XML file used as input for the Global to Local comes from different blocks depending on the iteration. If it is the first iteration, the file will come from the previous tool (or from the input provider). If an iteration has already been performed, then the file will come from a switch at the end of the tool (so that the outputs from previous iterations are saved).

The first function of the Global to Local is to determine the number of repetitions needed, which is passed to the new Iterator block located after the tool merger. Then, the Global to Local creates a new workflow XML file for the input filer, where the input nodes attributes are modified so that only the part of the workflow for each specific iteration is considered.

The Global to Local also determines which are the specific nodes where the tool has to write its output for each iteration. These nodes xpaths are saved and given to the Local to Global block. Finally, the Global to Local will also provide the original workflow XML file to the merger. As a consequence of providing different files for the input and the merger, the base splitter has been deleted.

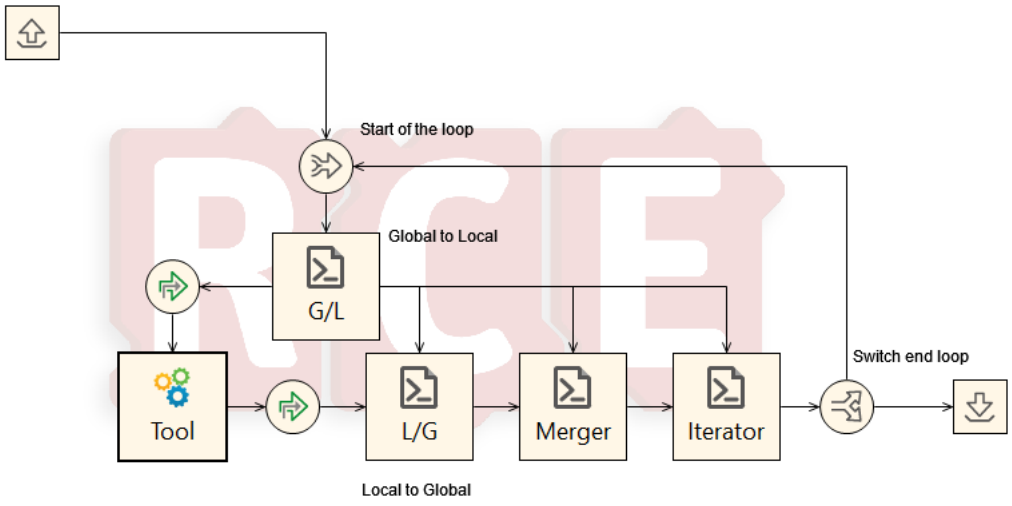


Figure 5.12: This figure shows the different elements and their connections in RCE of a discipline with repetition.

The next modification was the inclusion of the Local to Global block, to implement the necessary modifications in the tool output file, so that the data is merged correctly. Then the iterator blocked has been added as well as a new switch element. This switch will read the boolean given by the iterator. If it is false, the new workflow XML file will be given back to the Global to Local to start a new iteration. If is true, the workflow XML file will be provided to the next tool.

There are some cases where discipline activation and repetition take place at the same time. In those cases, elements from both influences are found in the export, but the connections are different, as shown in figure 5.13.

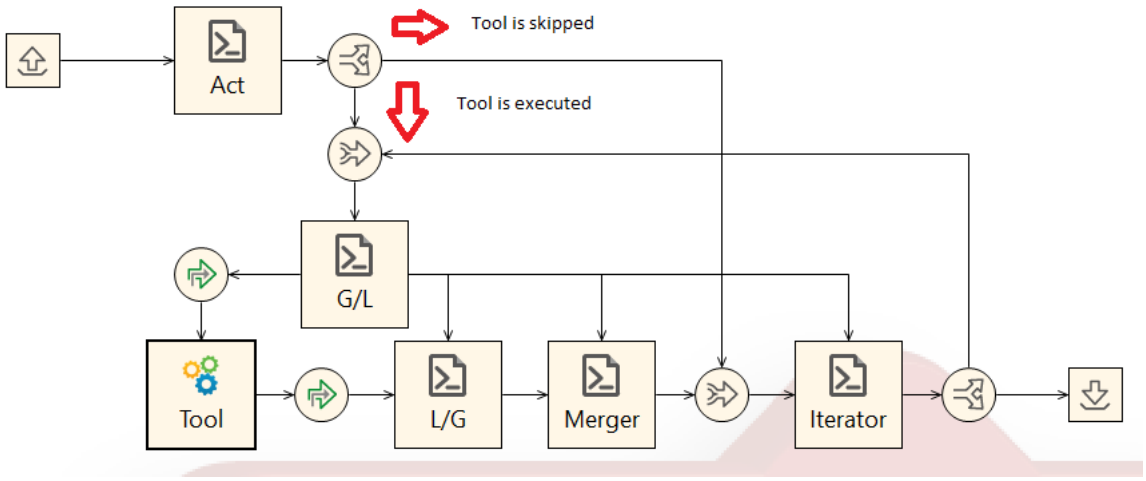


Figure 5.13: This figure shows the case of a discipline with activation and repetition in RCE. New connections have to be added to ensure both influences work together correctly.

The logic is similar, although the locations of some elements are changed, and new connections have to be made. First it is checked if the tool has to be executed. If not, the workflow XML file is passed from the first switch to the iterator. Then the iterator indicates that the process has ended and the second switch passes the XML to the next tool. If the tool has to be executed, the procedure described before will be followed.

5.3. Data connection (I2)

There are some occasions where the connections between disciplines change because of an architectural decision. This leads to variables being exchanged between different disciplines depending on the system architecture. As MDAx forces to use a central data schema, this influence can be implemented by deactivating inputs of disciplines depending on architectural decisions (section 4.5.2).

All possible inputs and outputs are always defined and exported to RCE, meaning that all possible connections between disciplines will exist in the workflow. When a connection is wanted to be deactivated between two disciplines, it can be done by just deactivating the variable involved in that connection from the inputs of the second discipline.

To do so, before the second discipline´s tool is executed, the architectural decision that determines if the variable has to be included or not in the inputs will be checked. In the case it has to be included, the tool will be executed as normal. If not, this variable will be deleted from the XML workflow file going to the input filter. This can be achieved by extending the Global to Local block implemented for discipline repetition.

The following requirements have been identified to implement data connection in MDAx/RCE:

1. **Extend the configuration file and the function block attributes to deal with data connection**
2. **Develop a nomenclature to indicate the condition determining the exclusion of variables from the tool inputs**
3. **Extend the Global to Local block to include inputs deactivation**
4. **Adapt the RCE export to data connection**

5.3.1. Extension of the configuration file and the function block attributes to deal with data connection

As in the previous architectural influences, the first step has been to extend the configuration file. A new key called ”input_deactivation” has been added to store variables that might be excluded from the tool inputs, and the architectural decision causing this effect.

```
{"executionName": "D1",
 "name": "D1",
 "notes": "",
 "input_deactivation": [{"Xpath" : "/sellar/variables/option1/x2",
 "Architectural_decision" : "ElNumEq(',/sellar/architecture/arch', 2)"}],
 "repetition": "NumberofInstances('/sellar/variables', 'variable')",
 "version": "1.0"
}
```

Figure 5.14: Example of a tool configuration file including data connection repetition.

A new attribute with the same name has been added to the function block in MDAx. Finally, the import function was extended again to read information from this key in the configuration file and store it in the corresponding attribute of the function block.

5.3.2. Nomenclature for data connection

The next objective is to develop a nomenclature to indicate what variable can be deactivated from the tool inputs and the corresponding condition originating this effect. To do so, it has been decided to use a list of dictionaries in the configuration file of the tool, where each dictionary is attached to a certain variable. This dictionary has two pieces of information.

The first one, which will be included in a key denominated ”Xpath”, contains the node’s xpath corresponding to the variable that might be deactivated. The second key, called ”Architectural decision”, contains the condition leading to the deactivation of the variable. As in the case of discipline activation, these conditions are expressed using the activation logic assertions introduced in section 5.1.3.

5.3.3. Extension of the Global to Local block to include inputs deactivation

In the case of repetition, the Global to Local block already had to generate a new workflow XML file for the input filter. As this is also the case of data connection, it has been decided to reuse this block. However, all the information regarding discipline repetition (such as the determination of the number of iterations, or the selection of output xpaths for the Local to Global) won´t be included, except in the case that data connection and repetition exist simultaneously.

The modifications done to the input file will be different. In this case, the Global to Local will contain the dictionary extracted from the configuration file stating the nodes to be checked and the conditions for their exclusion from the input file. Then, it will check the condition for each of these nodes, and if it is true, the node will be deleted from the input file going to the input filter, as shown in figure 5.15.

Then, when the input filter tries to search for the variable deleted, it will just ignore it as the variable won´t exist in the file (this will be discussed further in the next section). To evaluate the different conditions, the Python classes used to evaluate activation assertions will be included in this block when there is data connection.

```
<Aircraft>
  <Wing>
    <Span>10</Span>
    <t_c>0.15</t_c>
    <Chord>2</Chord>
    <Airfoil>NACA2315</Airfoil>
  </Wing>
  <Landing_gear>
    <Placement>Fuselage</Placement>
    <Loads>10000</Loads>
  </Landing_gear>
</Aircraft>
```

```
<Aircraft>
  <Wing>
    <Span>10</Span>
    <t_c>0.15</t_c>
    <Chord>2</Chord>
    <Airfoil>NACA2315</Airfoil>
  </Wing>
</Aircraft>
```

Figure 5.15: On the left, this figure shows part of the possible workflow XML file for the modified SSBJ problem. For the case of the wing structure discipline, the inputs are the wing geometry and the landing gear loads if the landing gear is attached to the wing. On the right it can be observed the XML file to be used as input for the tool. The landing gear loads have been deleted from the input file going to the input filter as the landing gear is attached to the fuselage.

Data connection and repetition can exist at the same time. In these cases, all the necessary information for repetition and data connection will be included. However, there is an important detail to be mentioned before discussing the RCE export, and it is that when these two influences happen simultaneously, the necessary tasks for repetition will be performed before the data connection ones in the Global to Local. This ensures that for each iteration of the tool, it can be decided if the input variable is included or not, as connections for each discipline instance might be different.

5.3.4. RCE export for data connection

For the case of data connections, no new elements have been added to the RCE export. Indeed, apart from modifying some connections, the main modification has been to also include the Global to Local component when the data connection influence is included in a tool. In this case, the Global to Local component only has two outputs instead of four, as there is not a Local to Global or an iterator. An example in RCE is shown in figure 5.16.

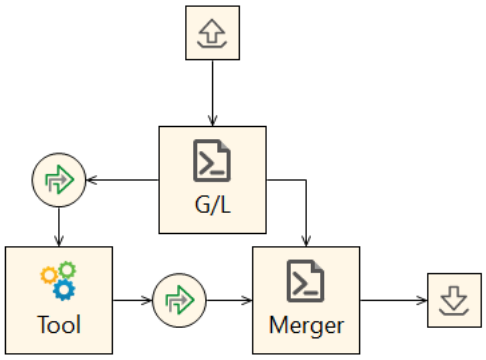


Figure 5.16: This figure shows the different elements and their connections in RCE for a discipline with data connection architectural influence.

In the case that there is data connection and repetition, the implementation is the same as in figure 5.12, being the only difference the content inside the Global to Local. For the case of activation logic, the modifications in the export are the same as in the case of repetition (figure 5.17).

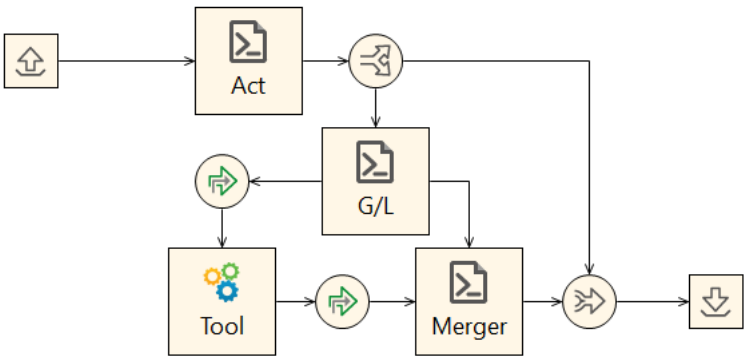


Figure 5.17: RCE discipline with activation and data connection. Data connection adds the Global to Local block. In the case of activation, first it is checked if the tool has to be executed in the activation logic script (Act). If true, then the tool will be executed. In the opposite case, the tool will be skipped, as in the previous cases.

5.4. Conditional variables (I1)

Finally, the case of conditional variables is discussed in this section. Two cases are going to be differentiated, depending if the variable that might or might not be included in the optimization problem is an input or an output for the different disciplines. In the case of inputs, when the input filter of a tool looks for the xpath of an input node and it does not exist in the workflow XML file, it will just ignore that node. This behaviour was implemented already in MDAx and means that is able to deal with conditional variables when they are inputs of disciplines.

Indeed, MDAx can also deal with more complex cases related with conditional variables, such as arrays of different variable length. This is because arrays can be stored in just an unique node, and MDAx/RCE are indifferent of the data stored inside nodes. It can also deal with big missing parts in the workflow XML file. An example could be a discipline that needs the wing geometry as input, and because of an architectural decision there is no winglet. As a result, a considerable part of the workflow XML tree would be missing (the winglet geometry), but the workflow would still continue to be executed (figure 5.18)

```
1  <Wing>
2    <span>4</span>
3    <t_c>0.3</t_c>
4    <chord>2</chord>
5    <airfoil>NACA2315</airfoil>
6    <winglet>
7      <span>0.5</span>
8      <cant_angle>30</cant_angle>
9    </winglet>
10 </Wing>
```

Figure 5.18: This is an example of a discipline input file. It expects as input the wing geometry (all the information stored inside the wing nodes). In case there are big missing parts in the tree, such as information regarding the winglet, the workflow execution will still continue.

The case of the outputs is different. There was no mechanism in MDAx to deal with conditional output variables. The methodology developed for data connection could be used to ”skip” this type of variables during the workflow execution. All the disciplines that received this specific conditional variable as input

could just deactivate the variable from their input files, ignoring the variable.

However, this would be just a provisional solution as the variable would still exist in the workflow XML file. To deal with this influence correctly, it would be necessary to add the Local to Global block in the RCE tool. This block would just delete from the tool output this variable when a certain architectural decision was taken. This is left as a future step.

The four different architectural influences are already implemented in MDAx and the corresponding RCE exportable workflow. In the next section, two new benchmark problems with all the architectural influences will be presented and executed in order to verify that MDAx+RCE can deal with all the aspects concerning the automatic readjustment of the MDO problem necessary for system architecture optimization.

6

Verification & Validation

To verify that all the architectural influences have been implemented correctly in MDAx, a mathematical benchmark problem based on Fourier series is solved. This problem contains all the possible architectural influences and has been designed to be a fast method to check architectural influences implementation in any MDO platform. Then, for validation, a second problem based on the design of a multistage rocket is proposed, aiming to show the potential of system architecture optimization when applied to real engineering design problems.

6.1. Verification: Mathematical benchmark problem

To verify the implementation of architectural influences in MDAx, a system architecture optimization problem based on Fourier series is proposed. Fourier series are expansion series used to approximate periodic functions using trigonometric functions. They can be represented by the following formula:

$$fourier(x) = A_0 + \sum_{n=1}^{\infty}(A_n cos(w_n * x) + B_n sin(w_n * x)) \tag{6.1}$$

where A_0, A_n and B_n are the Fourier coefficients, and w_n are the frequencies multiplied by 2π. The optimum values for these coefficients (and for the frequencies) is already known and can be calculated using algebraic expressions.

However, in this benchmark problem, new couplings between the different coefficients are introduced to include all the architectural influences in the problem formulation. This leads to a new approximation function where the number of sines and cosines terms is independent. The frequencies are also allowed to be different for each A and B coefficient (equation 6.1). The objective of the problem is to determine the best approximation function F(x) to approximate a periodic objective function f(x)

$$F(x) = A_0 + \sum_{n=1}^{N_A}\sum_{j=1}^{Nwa_n} A_n cos(w_{nj} * w_0 * x) + \sum_{m=1}^{N_B}\sum_{k=1}^{Nwb_m} B_m sin(w_{mk} * w_0 * x)$$

Figure 6.1: This formula is valid if $N_A > 0$ and $N_B > 0$. In the case one of them is zero, their corresponding terms (cosines or sines respectively) won´t exist in the approximation function.

To obtain the best possible approximation, the optimizer has control over architectural decisions, such as the number of terms, and design variables, such as the values of the coefficients or the harmonics to be included. More information about the objective function, constraints, architectural decisions and the design variables can be found in the next section.

6.1.1. Problem formulation

In this section, the different design disciplines are going to be introduced. After that, the different architectural decisions will be shown. Then, the optimization problem will be presented, including the objectives, constraints and design variables. Finally, this section will end with the introduction of the problem´s XDSM.

The approximation function F(x) consists of trigonometric functions (sines and cosines) and a constant term. In the workflow execution, the first step will be to determine these trigonometric functions. To do so, two different design disciplines are needed, which are:

- Y: This discipline provides as output the cosine terms (g) of F(x) for a given coefficient A_n, a reference w_0 (reference frequency multiplied by two π) and the desired harmonics to be used w_{nj}. As an example, for A_n = 3, w_0 = 2π , w_{n1} = 1 and w_{n2} = 2 , the output will be $g = 3(cos(2\pi x)+cos(4\pi x))$. If more than one A_n coefficient is chosen as an architectural decision, this discipline will be needed to be repeated multiple times.

  Finally, this discipline also provides an additional output, which is the value to be used as input for the constant term calculation (A_0). This value (called c) is calculated evaluating the expression g for a certain x_0 (fixed at the problem formulation). Therefore $c = g(x_0)$.

- Z: This is a similar discipline as Y, but for sines terms. The inputs will be in this case a certain B_m coefficient, the same reference w_0 and the harmonics w_{mk}. The outputs will be the trigonometric expression h, and another possible value for the constant term, also named c, calculated with the same method as before.

It is possible that there are no cosines or no sines in the approximation function (in the case of N_A or N_B equal to zero), meaning that sometimes the Y or the Z disciplines might not be executed.

Once the trigonometric functions have been obtained, the only term necessary to define the approximation function is the constant term A_0. This one will be calculated by the C discipline. To do so, it will take as inputs the function evaluations (c) of Y, Z or both (depending on an architectural decision called x_{con}). This is shown in figure 6.3.

Adding all the selected ”c” terms together, it will obtain a new value used as the height and the radius to define an ”imaginary cone”. The C discipline calculates both the surface and the volume of this cone (in case that the height or the radius are negative, the volume and the surface will be both set to 0), and the output to be taken as the constant term A_0 (the surface or the volume) depends on another architectural decision, called x_c.

Once all the terms have been obtained, the objective function block will calculate the error between a periodic function f(x) and F(x). The periodic function f(x) that has been chosen from the literature is the well known sawtooth wave (figure 6.2). The amplitude of the wave has been set to 1, as well as the period T (the frequency can be obtained knowing it is the inverse of T). The wave has been also centralized on the x axis and translated by half unit in the y axis.

The error between both functions is calculated using expression 6.2

$$error = \int_{-1}^{1} [f(x) - F(x)]^2 dx \tag{6.2}$$

Multiple architectural decisions can be found in this optimization problem. First, the number of A and B coefficients (N_A and N_B respectively). Then for each A_n and B_m coefficient, it has to be determined how many harmonics are wanted (Nwa_n and Nwb_m). The inputs to be taken by the C discipline, in cases where both Y and Z exist, is also an architectural decision (x_{con}). The final architectural decision, called x_c, is to select the output from the C discipline to be used as the constant term A_0 . The relationships between all these decisions, and how the architectural design space is built, can be found in appendix B.

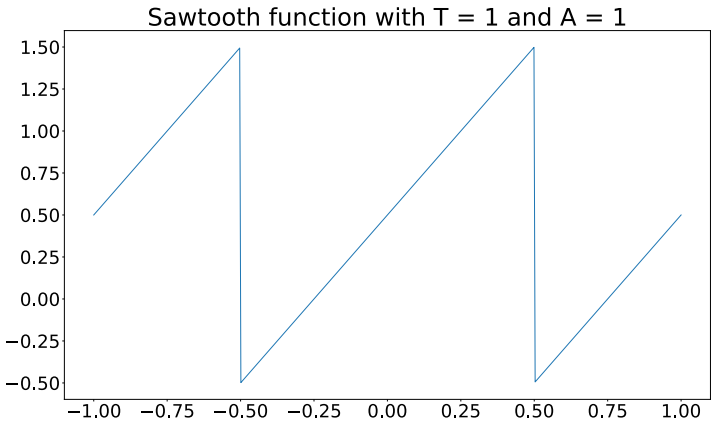


Figure 6.2: Sawtooth wave function to be approximated.

Finally, two constraints have been introduced. First the maximum number of trigonometric terms of F(x) has been set to 2. Second, it has been set that harmonics cannot be repeated for each A/B subsystem too. These constraints, as well as the objective function and the variables bounds, are expressed in the following problem formulation:

$$
\begin{aligned}
\text{minimize :} \quad & error \\
\text{with respect to :} \quad & N_A = [0,1,2] \\
& N_B = [0,1,2] \\
& Nwa_n = [1,2] && n = [0, N_A] \\
& Nwb_m = [1,2] && m = [0, N_B] \\
& w_{nj} = [1,2] && n = [0, N_A], j = [1, Nwa_n] \\
& w_{mk} = [1,2] && m = [0, N_B], k = [1, Nwb_m] \\
& -1 \leq A_n \leq 1 && n = [0, N_A] \\
& -1 \leq B_m \leq 1 && m = [0, N_B] \\
& x_{con} = [Y, Z, Both] \\
& x_c = [Surface, Volume] \\
\text{subject to :} \quad & wn1 \neq wn2 \\
& wm1 \neq wm2 \\
& 1 \leq \sum_{n=0}^{N_A} Nwa_n + \sum_{m=0}^{N_B} Nwb_m \leq 2 \\
\text{given :} \quad & x_0 = 0.3 \\
& w_0 = 2\pi
\end{aligned}
$$

Finally, the XDSM of the problem is shown in figure 6.3. The architectural influences found in the problem, as well as the new notation added to the XDSM to express them, are discussed in the next section.

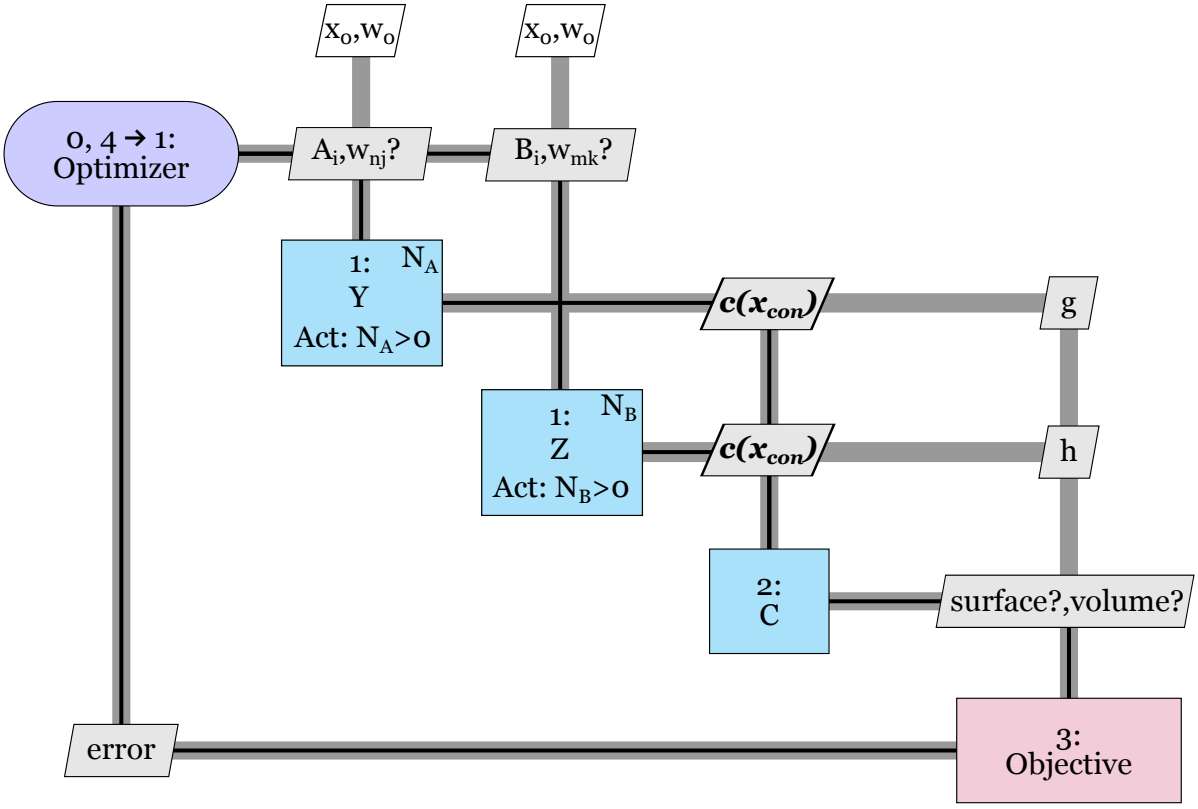


Figure 6.3: XDSM of the mathematical benchmark problem.

6.1.2. Architectural influences

All the four types of architectural influences can be found in the mathematical benchmark problem. In this section, it is discussed where they are found, and also the necessary new notation used to show them in the XDSM.

- **Conditional variables (I1)**: This influence can be found twice in the problem. The first case are the possible outputs of the C discipline. Depending on an architectural decision (x_c), only the volume or the surface variable will exist in the MDO problem. The second case are w_{nj} and w_{mk}, as the number of harmonics associated to each A_n and B_m coefficient depends on an architectural decision (Nwa_n and Nwb_m respectively). These conditional variables are indicated in the XDSM with an interrogation mark symbol next to them.

- **Data connection (I2)**: The c variables will always be an output of both Y and Z. However, there are some cases where only the variable from Y is taken as input for the C discipline. In other occasions, it is the one from Z that is taken. Finally, there are other times where the quantity from both are used as input. This is determined by the architectural decision labelled as x_{con}. As a result from the different values of this variable, there is rerouting of the variable c, and as a consequence new connections have to be made for each case. This is represented in the XDSM by writing the c variable in bold and italics, as well as indicating the architectural decision causing this influence.

- **Discipline repetition (I3)**: The number of times that the Y and the Z discipline have to be repeated depends on an architectural decision (N_A and N_B). This is expressed in the top right corner of the discipline, stating the variable whose value determines the number of repetitions.

- **Discipline activation (I4)**: As mentioned before, there are some cases where there are no cosines or sines in the approximation function. This means that the Y discipline or the Z discipline won´t be included in the MDO problem execution, respectively. The condition that determines their inclusion in the MDO problem is stated in a third line inside the discipline box.

6.1.3. Architectural influences verification

Finally, once the problem has been introduced, MDAx/RCE capability to deal with this automatic readjustment of the MDO problem is going to be proven. To do so, different architectures are going to be executed, demonstrating that MDAx/RCE are capable of dealing with the different architectural influences [1].

Discipline Activation (I4)

To show that discipline activation is implemented correctly, the Y discipline will be executed for two different cases. In the first case, the approximation function only contains a sine term (figure 6.4). As a consequence, when executing the workflow, the Y discipline is not executed, it is skipped (as confirmed by the Y tool execution counter, which is equal to 0).

```
1  <disciplines>
2    <z UID="z_1">
3      <B>0.5</B>
4      <wb>1</wb>
5    </z>
6    <x>0.3</x>
7    <Architecture>
8      <Method>Volume</Method>
9      <Input>z</Input>
10   </Architecture>
11 </disciplines>
```

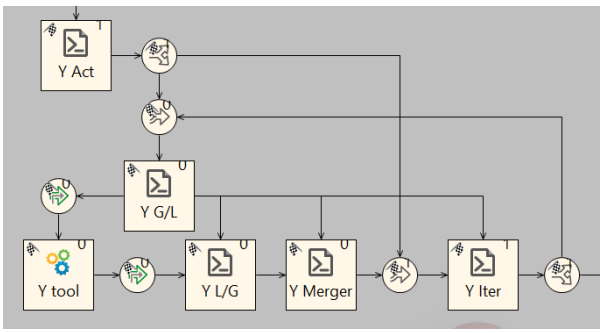


Figure 6.4: On the left, a possible workflow XML file for an architecture with only a sine term. As a consequence, it can be observed on the right that the Y discipline is skipped.

However, when an architecture with a cosine term is analysed (figure 6.5), the tool is executed.

```
1  <disciplines>
2    <y UID="y_1">
3      <A>0.5</A>
4      <wa>1</wa>
5    </y>
6    <x>0.3</x>
7    <Architecture>
8      <Method>Surface</Method>
9      <Input>y</Input>
10   </Architecture>
11 </disciplines>
```

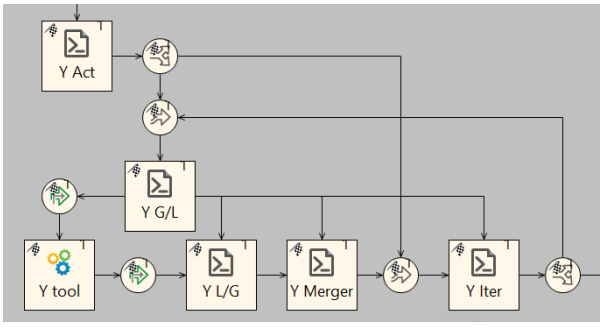


Figure 6.5: In this case, as there is a cosine term in the approximation function, the Y tool is executed.

[1] Conditional variables are not included, as this architectural influence was already implemented at MDAx.

To achieve this, it was necessary to indicate in the Y discipline configuration file that the tool would only be executed when the number of cosines term was equal or higher than one (figure 6.6). This would be the case when their corresponding node ("disciplines/y") exists in the XML workflow file, as shown by the "XpathExists" expression.[2].

```
{"activationLogic": "XPathExists('/disciplines/y')",
 "executionName": "Y",
 "name": "Y",
 "notes": "",
 "repetition": "NumberofInstances('/disciplines', 'y')",
 "version": "1.0"
}
```

Figure 6.6: Configuration file of the Y discipline. It is indicated that the tool has only to be executed when the "discipline/y" node exists, or in other words, when there are cosine terms in the approximation function.

This example shows that MDAx and RCE are now able to deal with discipline activation. MDAx allows to attach to each tool the condition determining their inclusion in the workflow execution. In these cases, the RCE export is modified, including new elements for this purpose, such as the activation logic script (figure 6.7). Finally, depending on the assertion being true or false, the tool will be executed or not.

```
424 #tool_activation_logic is inputted here
425 tool_activation_logic = XPathExists('/disciplines/y')
426
427 #Assertion is checked
428 check_activation_logic(tool_activation_logic)
```

Figure 6.7: This figure shows part of the code stored in the activation logic script. The information stored in the MDAx Y discipline regarding discipline activation is automatically passed to the activation logic script. This allows RCE to check if the tool has to be executed or not.

Discipline repetition (I3)

To show how MDAx and RCE deal with discipline repetition, an architecture with two sines is going to be used (figure 6.8).

```
1  <disciplines>
2    <z UID="z_1">
3      <B>1.0</B>
4      <wb>1</wb>
5    </z>
6    <z UID="z_2">
7      <B>0.5</B>
8      <wb>2</wb>
9    </z>
10   <x>0.3</x>
11   <Architecture>
12     <Method>Volume</Method>
13     <Input>z</Input>
14   </Architecture>
15 </disciplines>
```

Figure 6.8: Mathematical benchmark problem workflow XML file for an approximation function with two sine terms.

[2]The rest of activation logic assertions were checked in python using unit tests

To indicate MDAx that the discipline has to be repeated once for each sine term in the approximation function, the "NumberofInstances" assertion is used, as shown in figure 6.9.

```
{"activationLogic": "XPathExists('/disciplines/z')",
 "executionName": "Z",
 "name": "Z",
 "notes": "",
 "repetition": "NumberofInstances('/disciplines', 'z')",
 "version": "1.0"
}
```

Figure 6.9: Configuration file of the Z discipline. It is indicated that the tool has to be repeated once for each sine term in the approximation function. Each sine term will have its corresponding node in the workflow XML file ("discipline/z)".

The first step is to determine the number of iterations, which is checked by the Global to Local component. Then this block will also select the correct inputs of the workflow for each iteration. This can be observed in figure 6.10, where the inputs xpath attributes corresponding to each iteration are modified, indicating to the input filter the inputs to be used.

```
<disciplines>
  <z UID="z_{INDEX}">
    <B>1.0</B>
    <wb>1</wb>
  </z>
  <z UID="z_2">
    <B>0.5</B>
    <wb>2</wb>
  </z>
  <x>0.3</x>
  <Architecture>
    <Method>Volume</Method>
    <Input>z</Input>
  </Architecture>
</disciplines>
```

```
<disciplines>
  <z UID="z_1">
    <B>1.0</B>
    <wb>1</wb>
    <output_z>1.0*sin(6.2832*x)</output_z>
    <c>0.951</c>
  </z>
  <z UID="z_{INDEX}">
    <B>0.5</B>
    <wb>2</wb>
  </z>
  <x>0.3</x>
  <Architecture>
    <Method>Volume</Method>
    <Input>z</Input>
  </Architecture>
</disciplines>
```

Figure 6.10: The INDEX attribute used to indicate the inputs is set in different parts of the workflow XML file on each iteration. On the left, the first sine components are used. On the right, the second sine components will be the input.

The Global to Local component will also store the original attributes of these xpaths and will pass them to the Local to Global component. Then this component will add them to the tool output files, allowing to merge these outputs in the correct part of the workflow XML file (figure 6.11).

```
<disciplines>
  <z UID="z_1">
    <output_z>1.0*sin(6.2832*x)</
        output_z>
    <c>0.951</c>
  </z>
</disciplines>
```

```
<disciplines>
  <z UID="z_2">
    <output_z>0.5*sin(12.5663*x)</
        output_z>
    <c>-0.294</c>
  </z>
</disciplines>
```

Figure 6.11: The Local to Global adds the necessary attributes to the tool output, so that it is merged correctly. This can be observed on the different attributes (UIDs) added on each iteration.

Finally, the iterator will keep track of the number of iterations, ending the loop when the number of repetitions stated in the configuration file is achieved (figure 6.12).

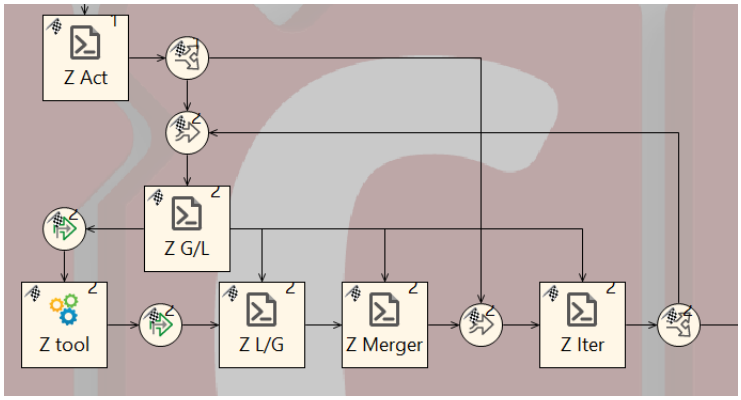


Figure 6.12: The Z tool is executed twice, as there are only 2 sines terms in the approximation function.

```
<disciplines>
  <z UID="z_1">
    <B>1.0</B>
    <wb>1</wb>
    <output_z>1.0*sin(6.2832*x)</output_z>
    <c>0.951</c>
  </z>
  <z UID="z_2">
    <B>0.5</B>
    <wb>2</wb>
    <output_z>0.5*sin(12.5663*x)</output_z>
    <c>-0.294</c>
  </z>
  <x>0.3</x>
  <Architecture>
    <Method>Volume</Method>
    <Input>z</Input>
  </Architecture>
</disciplines>
```

Figure 6.13: Merged XML given by the Z discipline. Different parts of the workflow were selected automatically for the inputs. The outputs location was also readjusted automatically.

Data connection (I2)

To demonstrate the successful implementation of data connection, the automatic rerouting of the variable c (used as input of the C discipline) will be shown. Two architectures will be used. Both of them have a cosine and a sine term. However, the discipline providing the input for the C discipline will be different on each case. In the first scenario, it will come from the Y discipline. In the second case, from the Z discipline (figure 6.14).

```
<disciplines>
  <y UID="y_1">
    <A>1.0</A>
    <wb>1</wb>
  </y>
  <z UID="z_1">
    <B>1.0</B>
    <wb>1</wb>
  </z>
  <x>0.3</x>
  <Architecture>
    <Method>Surface</Method>
    <Input>y</Input>
  </Architecture>
</disciplines>
```

```
<disciplines>
  <y UID="y_1">
    <A>1.0</A>
    <wb>1</wb>
  </y>
  <z UID="z_1">
    <B>1.0</B>
    <wb>1</wb>
  </z>
  <x>0.3</x>
  <Architecture>
    <Method>Surface</Method>
    <Input>z</Input>
  </Architecture>
</disciplines>
```

Figure 6.14: Workflow XML files to be used for the data connection demonstration. On the left, the c variable coming from the Y discipline is taken as input. On the right, it is the one coming from the Z discipline.

On each case, the Global to Local component will delete from the input filter XML file certain nodes, leading to the deactivation of specific connections. The nodes to be deleted , and the condition determining their suppression, is stated in the configuration file (figure 6.15).

```
{"executionName": "C",
 "input_deactivation":
    {"Xpath" : ["/disciplines/y/c","/disciplines/z/c"],
     "Architectural_decision" : ["ElStrEq('/disciplines/Architecture/Input', 'z')",
     "ElStrEq('/disciplines/Architecture/Input', 'y')"]},
 "name": "C",
 "notes": "",
 "version": "1.0"
}
```

Figure 6.15: Configuration file of the C discipline. The inputs to be deactivates, as well as the condition for the deactivation, are stated in the "input deactivation" key.

In the first case, as it is only wanted to take the c variable coming from the Y discipline, the node representing the c variable of the Z discipline has to be deleted. This allows to suppress the existing connection between the Z and the C discipline. In the second scenario, it will be the connection between the Y and the C discipline the one deleted (figure 6.16).

```
<disciplines>
  <y UID="y_1">
    <c>-0.309</c>
  </y>
</disciplines>
```

```
<disciplines>
  <z UID="z_1">
    <c>0.951</c>
  </z>
</disciplines>
```

Figure 6.16: Input files of the C discipline for both cases. The input comes from the Y or the Z discipline, depending on the architectural decision.

6.1.4. Mathematical problem results

Finally, the results of the optimization problem are going to be presented. A surrogate based optimization algorithm has been used, with a total of 1400 evaluations. The value of the design vector obtained is:

$$N_A = 0,\ N_B = 2,\ Nwb_1 = 1,\ Nwb_2 = 1,\ w_{11} = 1,\ w_{21} = 2,\ B_1 = 0.6275,\ B_2 - 0.3246,\ x_c = Volume$$

As it can be observed in figure 6.17, the result is similar to the best possible approximation of the original function using only two trigonometric terms (given by Fourier coefficients theoretical solution). Although it is not exactly the same (as how the problem is formulated, the Fourier coefficients would give an A_0 that slightly increases the error), the value of the objective function only differs 0.25 percent with respect to the ideal Fourier solution, concluding that the optimizer was able to find a solution at least close to the optimum.

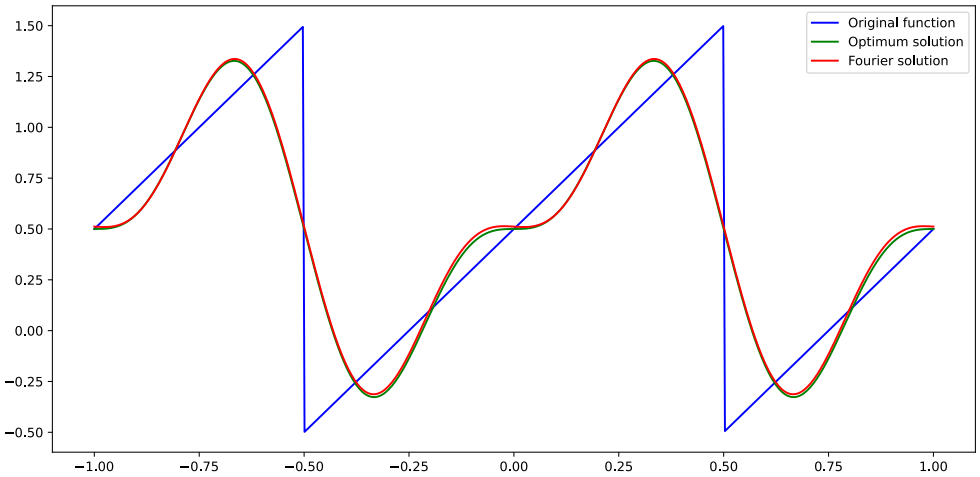


Figure 6.17: This figure shows the optimum solution found by the optimizer. The approximation is close to the optimum value of Fourier series.

The Fourier problem satisfies many of the requirements needed to be used as a system architecture optimization benchmark problem. It is simple , easy to implement and includes all the architectural influences. However, its main drawback is that it is not a real engineering problem, no real system architectures are used. This is the topic of next section.

6.2. Validation: Space benchmark problem

To show the potential of system architecture optimization in a real case design problem, a new and more complex benchmark problem is going to be proposed. The objective of this problem will be to design a multistage rocket that maximizes the mass of payload that can be lifted to a circular orbit of 400 km while minimizing the cost.

To achieve this, the optimizer will have control over different architectural decisions (head shape, number of stages, type of engines,...) and conventional design variables (length of each stage, length to diameter ratio, etc..). Also some constraints will be added to ensure that the design is feasible, such as the maximum dynamic pressure that the vehicle can stand. In the next section, all these disciplines and constraints, as well as the interaction between them, are widely introduced.

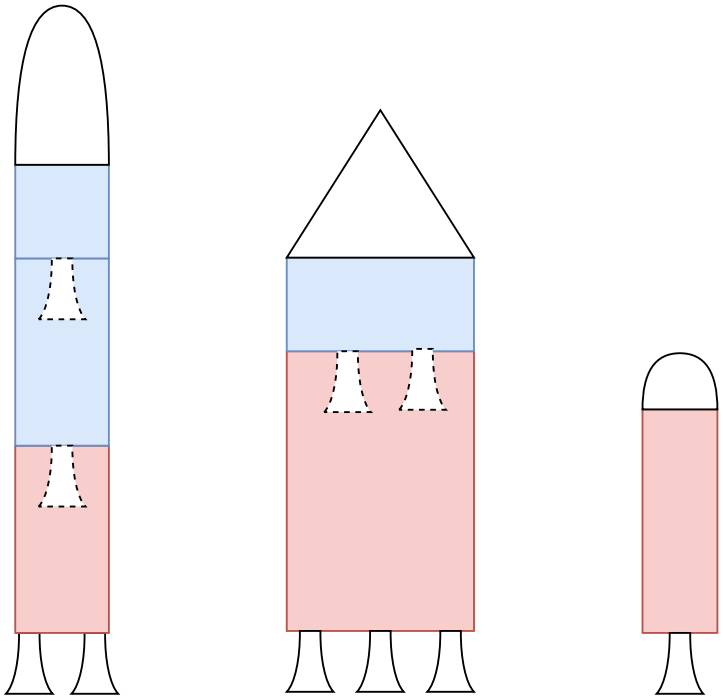


Figure 6.18: Examples of different possible rocket architectures. Some architectural choices can be observed, including the type of propellant (blue=liquid and red = solid), the number of stages, the type of head or the number of engines per stage.

6.2.1. Problem formulation

Multiple disciplines/constraints are necessary to be taken into account when designing a space rocket. In this benchmark problem, seven different design disciplines and two constraints have been considered. In this section these disciplines/constraints, and their corresponding variables, will be introduced. Then the architectural decisions of the problem will be presented. Finally, the mathematical problem formulation will be added, including the problem XDSM.

Propulsion

The first two design disciplines are related with propulsion, using a different discipline depending on the type of propellant. For the propulsion calculations, it is assumed that each stage can only consist on liquid or solid propellant (not hybrid). It is also assumed that the engines used for each stage are always the same. With these assumptions, the following disciplines are proposed:

- **Solid propulsion**: This discipline will be included when solid propulsion is used for at least one of the stages of the rocket. Three different types of engines will be allowed to be chosen, based on the characteristics of real solid propulsion engines. The engines used for reference are the SRB (used in the space shuttle), the P80 (used by ESA in the Vega rocket) and the GEM60 (used in the DELTA IV, among other rockets).

This discipline takes as input a list with the engines used for a certain stage. The number of engine nozzles is a design variable, and to indicate how many nozzles there are, the name of the chosen engine has to be repeated a number of times equal to the number of nozzles. For example, if a SRB with two nozzles is wanted, the input would be: Engines = [SRB,SRB].

Then, depending on the engines chosen and using the characteristic shown in tables 6.1 and 6.2, this discipline will provide as an output the maximum thrust of that stage and the corresponding mass flow rate ($\dot{m}$) for that specific thrust.

Table 6.1: Thrust of reference solid engines.

Engine	Thrust (MN)
SRB	12.45 (Kanner et al., 2011)
P80	2.1 (ESA, 2002)
GEM60	1.245 (ATK, 2020)

Table 6.2: Mass flow rate of reference solid engines.

Engine	Mass flow (kg/s)
SRB	5290 (Borghi and Spinozzi, 2017)
P80	764 (ESA, 2002)
GEM60	814.19 (Grumman, 2023)

- **Liquid propulsion**: This is the analogous propulsion discipline for the case of liquid propellant. The inputs and the outputs are going to be exactly the same, except for an additional output, which is the nozzle expansion ratio (A_e/A_g). This is defined as the ratio between the nozzle exhaust area and the nozzle throat area.

  As in the case of solid propulsion, three real engines that have been already used in rocket design are going to be used as reference. These are the Vulcain (used by ESA for the Arianne family), the RS68 (used for the space shuttle too) and the S-IVB engine (used for the last stage of Saturn V). The data used for the thrust, the mass flow and the expansion ratio is presented in tables 6.3, 6.4 and 6.5.

Table 6.3: Thrust of reference liquid engines.

Engine	Thrust (MN)
Vulcain	0.8 (Chemeurope, 2005)
RS68	2.891 (Sumrall and McArthur, 2007)
S-IVB	0.486 (Astronautix, 1997)

Table 6.4: Mass flow rate of reference liquid engines.

Engine	Mass flow (kg/s)
Vulcain	188.33 (Chemeurope, 2005)
RS68	807.39 (University, 1998)
S-IVB	247 (Astronautix, 1997)

Table 6.5: Expansion ratio of reference liquid engines.

Engine	Expansion ratio
Vulcain	45 (Chemeurope, 2005)
RS68	21.5 (Creech et al., 2008)
S-IVB	28 (Astronautix, 1997)

These tools only accept as input the engines used for a certain stage. Therefore, if multiple stages of the same type of propellant are chosen, these disciplines will have to be repeated multiple times.

Geometry

The next discipline in the workflow is a **geometry calculator**. The first geometrical properties given as output of this discipline are the total rocket length (L_t) and the total diameter (D_t). To calculate them, the length of each stage (l_i) and the length to diameter ratio ($L_t\_D_t$) are given as inputs. Once these properties have been calculated, the volume for each stage (V_s) is calculated, assuming that it has a cylindrical shape

Another output of this discipline is the calculation of the surface and the volume of the rocket head, which are later used by different design disciplines. There are three different types of heads, which are elliptical, semi-spherical and conical (figure 6.19). The discipline knows the head shape depending on the inputs that it receives, which depend on an architectural decision (H_s). If a conical head is wanted, the cone angle (ϵ) will be given as an input. If a semi-elliptical head is chosen, the ratio between its length (L_e) and the total length of the rocket will be given ($R_e = L_e/L_t$). If none of these design variables are inputs to the discipline, it will assume that a semi-spherical head has been chosen for the architecture.

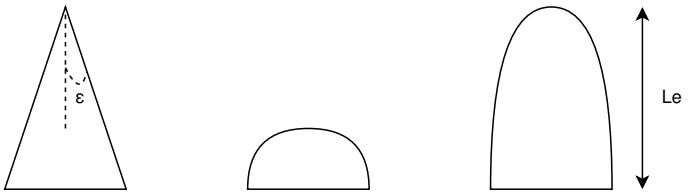


Figure 6.19: Possible head shapes of the rocket. These are conical, semi-spherical and semi-elliptical.

The formulas for the surface and the volume calculations used for each head shape are provided in the next table:

Table 6.6: Geometric formulas for the head surface/volume calculations.

Shape	Surface	Volume
Cone	$\pi R g$	$\frac{1}{3}\pi R^2 h(\epsilon)$
Semi-sphere	$2\pi R^2$	$\frac{2}{3}\pi R^3$
Semi-elliptical	$\pi L_e^2 + \frac{\frac{\pi R^2}{\nu} ln[(1+\nu)/(1-\nu)]}{2}$	$\frac{2}{3}\pi R^2 L_e$

where:

$$h(\epsilon) = R/tan(\epsilon) \tag{6.3}$$

$$g = \sqrt{R^2 + h(\epsilon)^2} \tag{6.4}$$

$$\nu = \frac{\sqrt{L_e^2 - R^2}}{L_e} \tag{6.5}$$

Finally, the engines used for each stage are given as input. If liquid fuel engines are used, the geometry calculator will provide four additional outputs for that stage. These are the surface/volume for the fuel tank (St_F and Vt_F) and for the oxidizer tank (St_O and Vt_O).

All the reference liquid engines use H_2 as fuel and LOX as oxidizer. It will be assumed that the oxidizer to fuel mass ratio is equal to 7.937 (Gordon and McBride, 1959). Assuming that the addition of the tank volumes of a certain stage is equal to the stage total internal volume , the following formulas can be used for the volumes:

$$Vt_F = \frac{V_S}{\frac{7.937\rho_{H2}}{\rho_{LOX}} + 1} \tag{6.6}$$

$$Vt_O = V_s - Vt_F \tag{6.7}$$

where the density values are $\rho_{H2} = 71 kg/m^3$ and $\rho_{LOX} = 1140 kg/m^3$. Assuming that the tanks have cylindrical shapes (as the stage), the surfaces can be calculated too.

Rocket mass

With the geometric values already calculated, the next step is to determine the masses of the different rocket stages. To do so, two disciplines are used and repeated for each stage. First the **propellant mass** discipline, which takes as an input the engines of a certain stage (as each engine has an associated propellant) and gives as an output the propellant mass (m_p) in the case of a solid propulsion stage, or the fuel and oxidizer masses (m_F and m_O) for a liquid propulsion stage.

For solid propulsion, it is assumed that the whole stage inner volume is filled with propellant. This is not true for the case of liquid propulsion, where it is necessary to leave some free volume in the tanks (called ullage) for pressurization, boil-off... This volume will be assumed to be equal to 6 % (Hedayat et al., 1998). Knowing these assumptions, the inner volume of each stage, and the densities shown in table 6.7 , the different propellant masses can be calculated.

Table 6.7: Propellant densities used for each reference engine.

Engine	Propellant	Density (kg/m^3)
SRB	PBAN	1715 (Braeunig, 1996)
P80	HTPB1912	1810 (Wingborg et al., 2017)
GEM60	HTPB-APCP	1650 (Thomas, 2018)
Liquid	H2/LOX	71/1140 (Verfondern et al., 2021)/(Sakaki et al., 2017)

For the calculation of each stage mass, a discipline called **structural mass** is used. The components contributing to the total mass of a rocket stage highly differ depending on the source of propulsion, leading to a different number of outputs of this discipline. The masses of these components will be calculated using simple numerical equations taken from Akin, 2016.

The first component that can contribute to a stage mass (if it the last stage) is the head structure mass (M_s). To calculate it, it is necessary as input the head surface. Then, assuming a constant thickness of 0.005 m (SpaceX, 2021) and that the material used for all this structure is aluminium 2024 (commonly used in aerospace industry) with a density of 2780 kg/m^3, the total mass of the rocket head structure is given by:

$$M_s = 2780 * 0.005 * Head\_surface \tag{6.8}$$

For the case of solid propulsion, only one component contribute to the structural mass of the stage. This component is the engine casing used to contain the combustion chemical reaction. The mass of the casing (M_{casing}) can be calculated when the propellant mass is given as an input using the following equation:

$$M_{casing} = 0.135 * m_p \tag{6.9}$$

For the case of liquid propulsion, three outputs are provided. These are the tanks mass (M_{tanks}), the tanks insulation mass ($M_{insulation}$) and the pumps mass (M_{pumps}) necessary to pump both the fuel and the oxidizer. For the case of the tanks mass, it is obtained using the graph shown in figure 6.20, where tank volumes are the only necessary inputs.

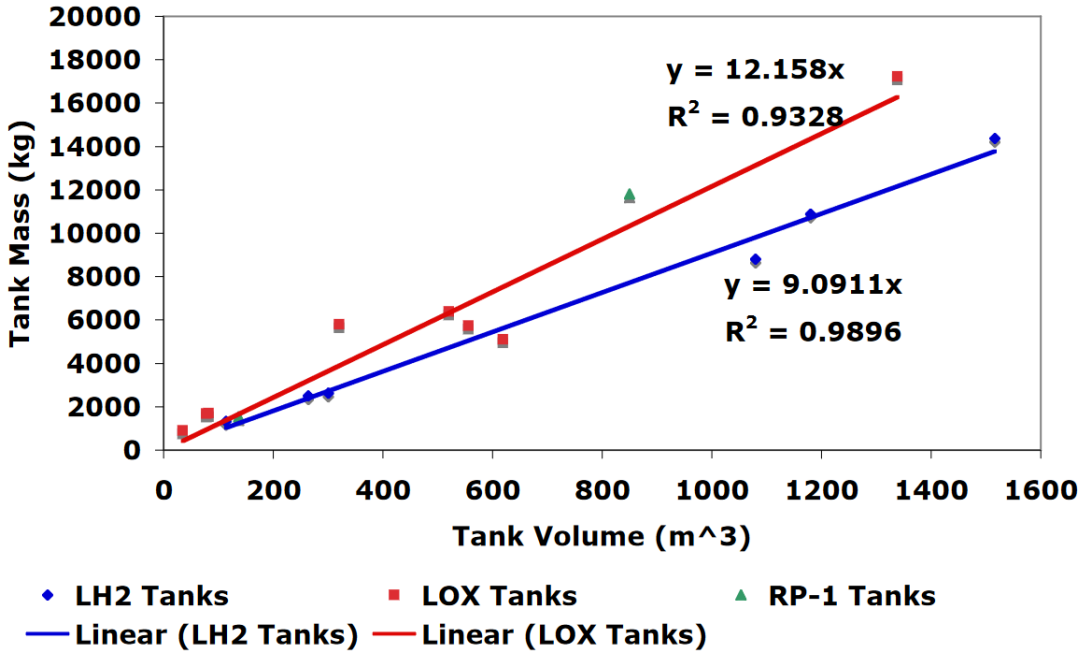


Figure 6.20: Regression data used to calculate tanks mass as a function of the tanks volume. Figure taken from Akin, 2016.

The insulation mass of both the fuel and the oxidizer is calculated with numerical equations using the tanks surfaces as inputs (given by the geometry calculator).

$$M_{insulation,H2} = 2.88St_F \tag{6.10}$$

$$M_{insulation,LOX} = 1.123St_O \tag{6.11}$$

For the case of the pumps, it is necessary as input the thrust and the expansion ratio calculated by the propulsion discipline. Assuming a combustion chamber for each nozzle, the total weight of the pumps for each combustion chamber is given by:

$$M_{pump} = 7.81e^{-4}T_i + 3.37e^{-5}\sqrt{\frac{A_e}{A_g}} + 59 \tag{6.12}$$

where T_i is the thrust of only one engine. The total pumps mass (M_{pumps}) of a stage will be given by the product between the previous result and the number of engines.

Trajectory

This discipline performs all the necessary calculations regarding the rocket trajectory. The first step necessary to perform these calculations it to determine the rocket drag coefficient (C_D). It is going to be assumed at this conceptual design phase that the vehicle drag coefficient is determined only by the head shape drag coefficient.

For the case of a semi spherical shape, the theoretical value of the C_D is known an equal to 0.42 (Hoerner, 1965). For the case of a cone, the C_D depends on the cone angle, as shown in figure 6.21. Using that graph as reference, the following numerical approximation can be used to calculate the cone drag coefficient:

$$C_D = 0.0112\epsilon + 0.162 \tag{6.13}$$

For the case of the semi-elliptical head shape, it is supposed that the drag coefficient depends only on the ratio between the semi-ellipse length and the rocket total length. The following linear interpolation has been built, based on the experimental data shown in Fedaravičius et al., 2012:

$$C_D = 0.305 - \frac{R_e - 0.1}{1500} \tag{6.14}$$

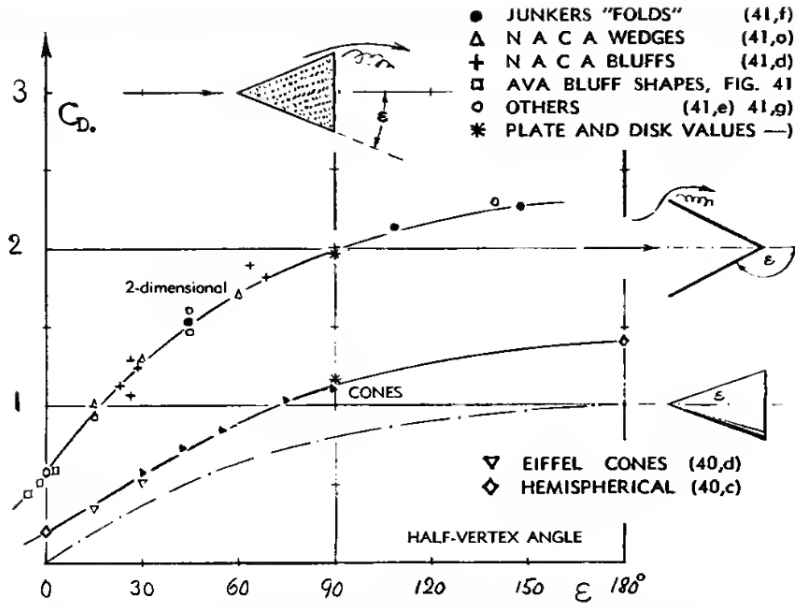


Figure 6.21: Drag coefficient of a 3D cone as a function of the cone angle ϵ. Figure taken from Hoerner, 1965.

With the data proceeding from the other disciplines and the drag coefficient, it is possible to calculate the rocket maximum circular orbit for a certain payload mass. To do so, it is assumed that the trajectory can be divided into three different segments. The first segment is going to be vertical flight. The equation of movement, assuming a zero angle of attack, is determined by the following differential equation:

$$\frac{\partial^2 x}{\partial t^2} = \frac{T - 0.5\rho(h)SV^2C_D}{(m_0 + m_{pay}) - \dot{m} * t} \tag{6.15}$$

where T is the rocket thrust , C_D is the drag coefficient, m_{pay} is the payload mass, S is the rocket frontal surface, $\rho(h)$ is the density [3] and m_0 is the rocket initial mass without considering the payload (the addition of the propellant and structure mass outputs). This equation can be solved numerically assuming a certain payload mass , allowing to obtain the rocket speed for each altitude with the initial conditions $h_0 = 0$ and $v_0 = 0$.

This first segment of vertical flight is used to get rid of atmospheric drag as fast as possible. It usually lasts until the maximum dynamic pressure point (or max Q) is achieved, which is the point in the whole mission where the rocket experiences the maximum structural loads. The altitude of this point varies depending of the rocket. A value of 10 km of altitude is going to be assumed, using this point for an instantaneous transition to the next trajectory stage, the turn maneuver.

To enter into orbit, the rocket needs to gain horizontal speed with respect to the Earth. This is why during this second trajectory stage the rocket gradually starts to change its initial vertical flight ($\gamma = 90°$) to an horizontal flight ($\gamma = 180°$), where γ is the flight angle, defined with respect to the horizontal.

To simplify the calculations, it is going to be assumed that the rocket climbs at a constant angle of 135 degrees[4]. With this assumption, and assuming that the lift forces are negligible, the rocket movement according to flight dynamics (Yaylali, 2018) is given by:

$$\frac{\partial^2 x}{\partial t^2} = \frac{Tcos(\alpha) - 0.5\rho SV^2C_D}{(m_0 + m_{pay}) - \dot{m} * t} - gsin(\gamma) \tag{6.16}$$

[3] ICAO international atmospheric model is used to estimate the density for each altitude.

[4] The rocket will tend to vary its flight angle naturally, so it assumed that some complementary technology allows to fix the flight angle, such as thrust vectoring.

with the initial conditions $h_0 = 10000m$ and $v_0 = v_1$, where v_1 is the rocket velocity at 10 km of altitude. An angle of attack (α) of 5 degrees will be assumed too for the calculations.

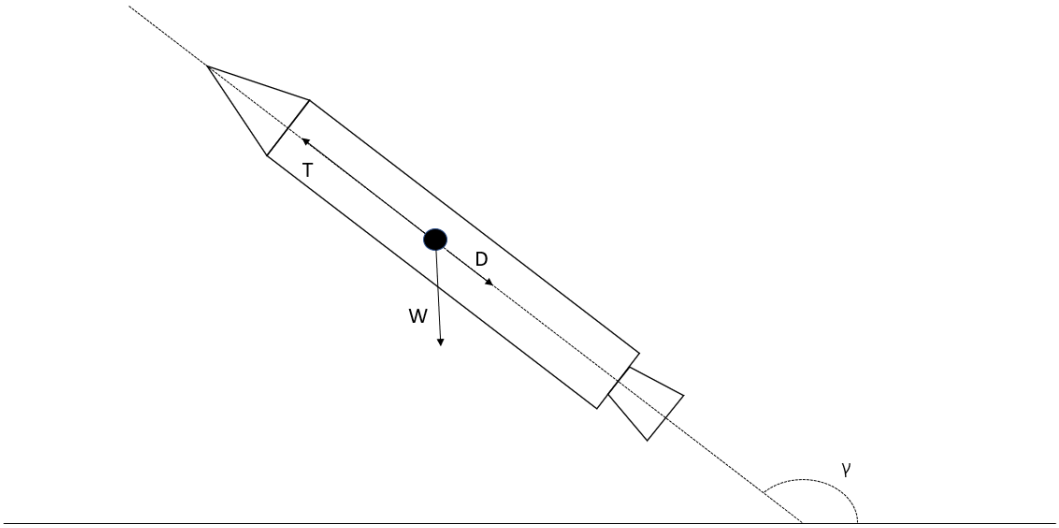


Figure 6.22: Forces actuating on a rocket climbing at a flight angle γ.

This trajectory phase will be maintained until the rocket arrives to the Karman line (100 kilometers of altitude) with a certain speed v_2, and an horizontal speed equal to half the previous velocity. If the rocket still has propellant left, it can further increase its velocity. Assuming that the flight angle transitions instantly from 135 degrees to an horizontal orientation, the maximum speed that the rocket could gain with respect its original horizontal speed ($v_2/2$) would be governed by :

$$\frac{\partial^2 x}{\partial t^2} = \frac{T}{(m_0 + m_{pay}) - \dot{m} * t} \tag{6.17}$$

This equation can be solved numerically with initial conditions $v_0 = v_2/2$ until the rocket runs out of propellant, obtaining a certain final maximum velocity (v_3). Then, knowing that each circular orbit of a height h_{orbit} is attached to a certain minimum speed ΔV (equation 6.18), it can be determined if the rocket is able to arrive to the desired orbit of 400 km or not for the supposed payload mass.

$$\Delta V = \sqrt{\frac{\mu}{h_{orbit} + R_{Earth}}} \tag{6.18}$$

It is important to take into account when solving numerically the trajectory equations that the rocket could have multiple stages. To implement this, at the end of each trajectory phase it is necessary to check if the stage propellant had finished before that trajectory phase ended. If this is the case, the same trajectory phase will be solved again, but taking as initial condition the point where the stage propellant ended and subtracting the mass of the previous stage which already ended.

As a summary, the trajectory discipline will iterate the payload mass until it finds the maximum payload that can be lifted by the rocket to an orbit of 400 km. Apart from the payload mass, the trajectory discipline also provides as output two vectors containing all the altitude and velocity points of the first and second phases of the trajectory (h_{vector} and v_{vector}).

Cost

This discipline calculates the second objective of the optimization problem, which is the total cost of the rocket. As in the previous disciplines, simple numerical expressions will be used to estimate the cost, which is valid for a conceptual design. Three different components are assumed to contribute to the

total cost, which are the engines, the rocket head structure and the propellant.

In the case of the rocket head structure cost, it is obtained calculating the total cost of the material used for its manufacturing. This can be done multiplying its mass (M_s) by the price for aluminium 2024. assumed to be of 2.45 dollars per kilogram (Jagdishmetalindia, 2023). A similar procedure is followed for the case of propellants. Using the mass of propellant calculated before and using the price per kilograms shown in table 6.8, the propellant cost can be calculated.

Table 6.8: Propellants cost (Urban, 2023b).

Propellant	Cost ($\$/kg$)
Solid	5
LOX	0.27
H2	6.1

The third component, which is the engine cost, is calculated using the TRANSCOST model (Koelle, 2007). The expressions used to calculate it depends on the type of propellant:

$$Cost_{engine,solid} = 2.3(M_{casing} + M_p)^{0.399} \tag{6.19}$$

$$Cost_{engine,liquid} = 5.16(M_{tanks} + M_{insulation} + M_{pumps})^{0.45} \tag{6.20}$$

These equations were obtained from the statistical data shown in figure 6.23.

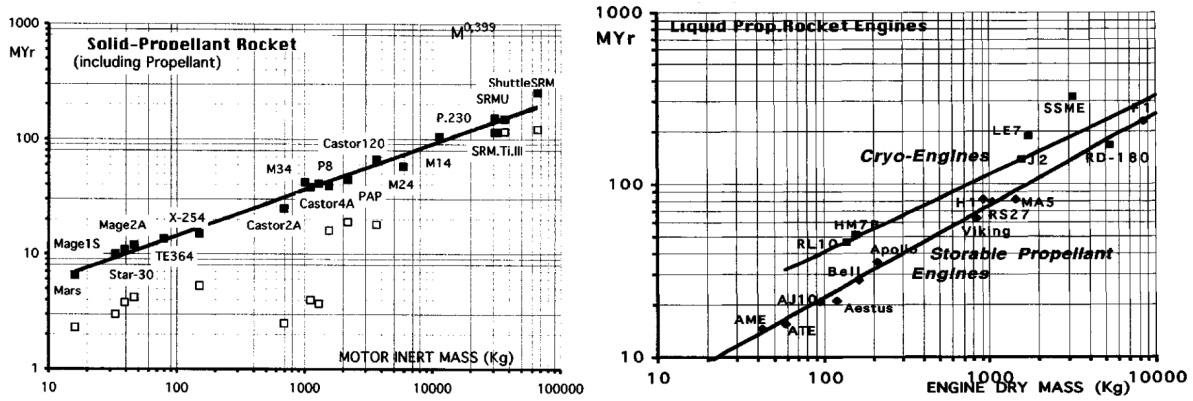


Figure 6.23: Statistical analyses use to estimate the engine production cost for solid and liquid propulsion engines. Figures taken from Koelle, 2007.

It is important to remark that the production cost will be multiplied by the number of engines presented (both for liquid and solid engines). Also, these graphs show the cost for producing the first engine, but this cost is reduced when multiple engines are built during the year. A reducing factor of 0.85 (taken from Koelle, 2007) will be considered.

The cost in all these previous expressions is calculated in an unit introduced in the TRANSCOST model called Man Year Cost (Myr). This was done to avoid the data to be modified because of inflation. However, a conversion can be done to the actual dollar assuming that $1Myr = 366518\$$

Constraints

Two different constraints have been added to the optimization problem to ensure that the design is feasible. These are:

- **Structural constraint:** This constraint checks if the loads during the trajectory exceeded at any point the maximum load capacity of the rocket (assumed to be 50 KPa, Frank et al., 2016). To do so, it receives as inputs the velocity as each altitude (h_{vector} and v_{vector}), and it calculates the dynamic pressure (q) at each altitude. Then the highest value is taken, it is subtracted the maximum load capacity (50 kPa) and is given as an output (Structural_difference). The constraint will be considered to be satisfied if the previous result is negative.
- **Payload constraint:** This constraint ensures that the payload mass calculated by the trajectory discipline fits inside the rocked head. To do so, it receives as input the head volume and the mass of payload. Then, assuming that the payload is mainly made of aluminium (with a density of 2.81 kg/m^3), it can be calculated the volume that the payload occupies. Finally, the payload volume is subtracted the head volume and given as output (Payload_difference). The constraint is satisfied if the result is negative.

Architectural decisions

All the disciplines and the connections between them have already been introduced. Before showing the mathematical formulation of the problem, the different architectural decisions are introduced. The first one, which is going to hugely influence the whole performance of the rocket, is the number of stages (N_s). Then for each stage there are two additional architectural decisions, which are the type of engines and the number of nozzles ($N_{E,i}$), which are combined into one array called $Engines_i$. Finally, for the last stage there is an architectural decision determining the head shape (H_s). As in the case of the mathematical problem, the architectural design space for this problem can be observed in more detail in appendix C.

Mathematical formulation and XDSM

This section includes the system architecture optimization problem formulation for the multistage rocket design. The XDSM is also included in the next page.

$$
\begin{aligned}
\text{minimize :} \quad & cost \\
\text{maximize :} \quad & m_{pay} \\
\text{with respect to :} \quad & N_s = [1,2,3] \\
& H_s = [Semi\_sphere, Cone, Elliptical] \\
& Engines_i = [SRB(N_{E,i}), P80(N_{E,i}), GEM60(N_{E,i}), \\
& VULCAIN(N_{E,i}), RS68(N_{E,i}), SIVB(N_{E,i})] && i = [1, N_s] \\
& N_{E,i} = [1,2,3] && i = [1, N_s] \\
& 0 \leq l_i \leq 20 && i = [1, N_s] \\
& 10 \leq L_t\_D_t \leq 20 \\
& 15 \leq \epsilon \leq 30 \\
& 0.1 \leq R_e \leq 0.25 \\
\text{subject to :} \quad & Structural\_difference \leq 0 \\
& Payload\_difference \leq 0 \\
\text{given :} \quad & h_{orbit} = 400km \\
& maxQ = 50kPa \\
& \rho_{pay} = 2.81kg/m^3
\end{aligned}
$$

With these architectural decisions, the possible number of system architectures (only considering discrete variables) would be of 18522. However, the real number of possible rocket designs (called later design points) would be much higher, as each architecture has additional numerical continuous design variables attached that the optimizer can control.

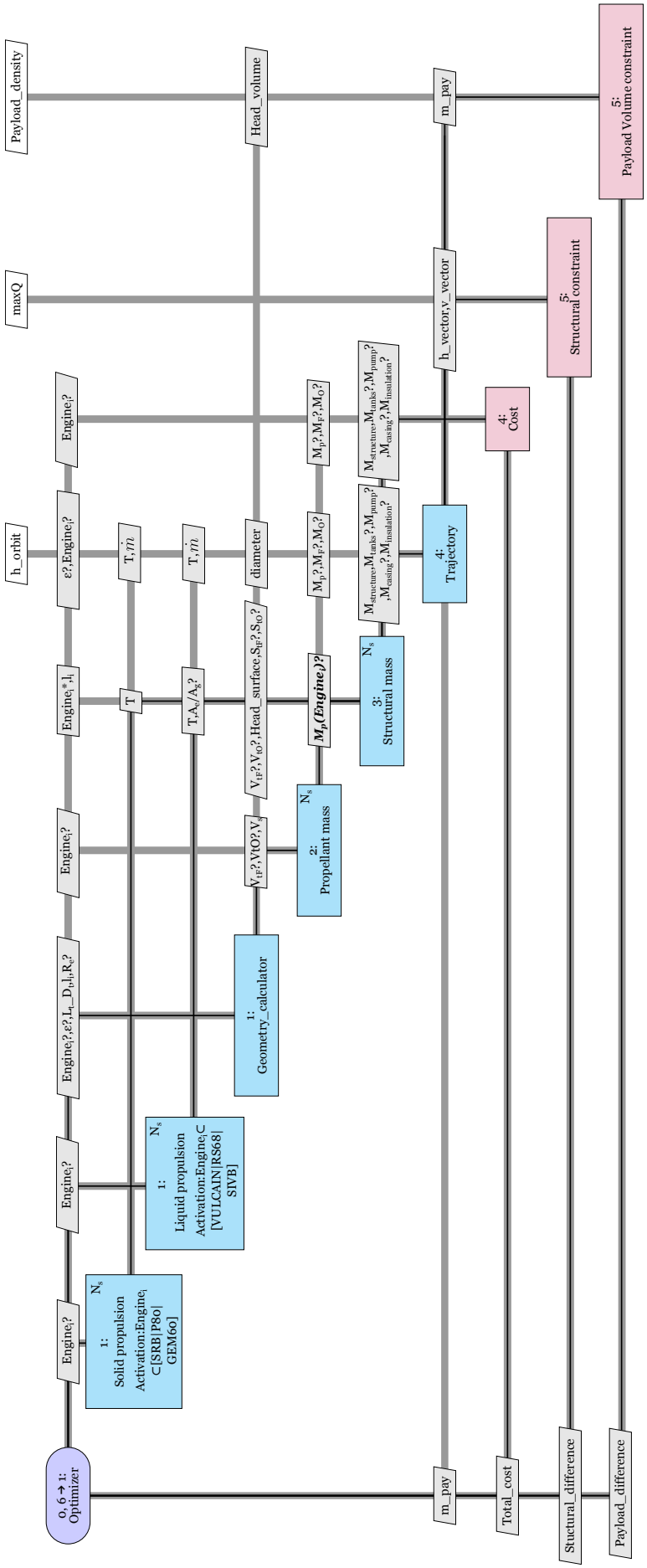


Figure 6.24: XDSM of the space rocket benchmark problem.

6.2.2. Architectural influences

The four different architectural influences can be also found in the space benchmark problem. **Conditional variables (I1)** are found everywhere in the problem, mainly attached to the type of propellant used at each stage. An example could be the mass of the tanks (M_{tanks}). Also the case of variable length arrays is included in the "$Engine_i$" variable, where the number of components depends on an architectural decision ($N_{E,i}$).

Data connection (I2) is found between the Propellant and the Structure mass disciplines. The propellant mass (m_p) will only be exchanged between both disciplines when solid propulsion is used. These disciplines also include **Discipline repetition (I3)**, as they are repeated for each stage of the rocket. Finally, **Activation logic (I4)** can be found in the propulsion disciplines.

6.2.3. Space problem results

In this section, the results obtained for the multistage rocket design problem are going to be showed and briefly analysed. To solve the problem, NSGA-II algorithm was used. A population of 150 points was chosen, with a total of 30 generations, leading to a total of 4500 design points. With these optimization parameters, the results shown in figure 6.25 were obtained.

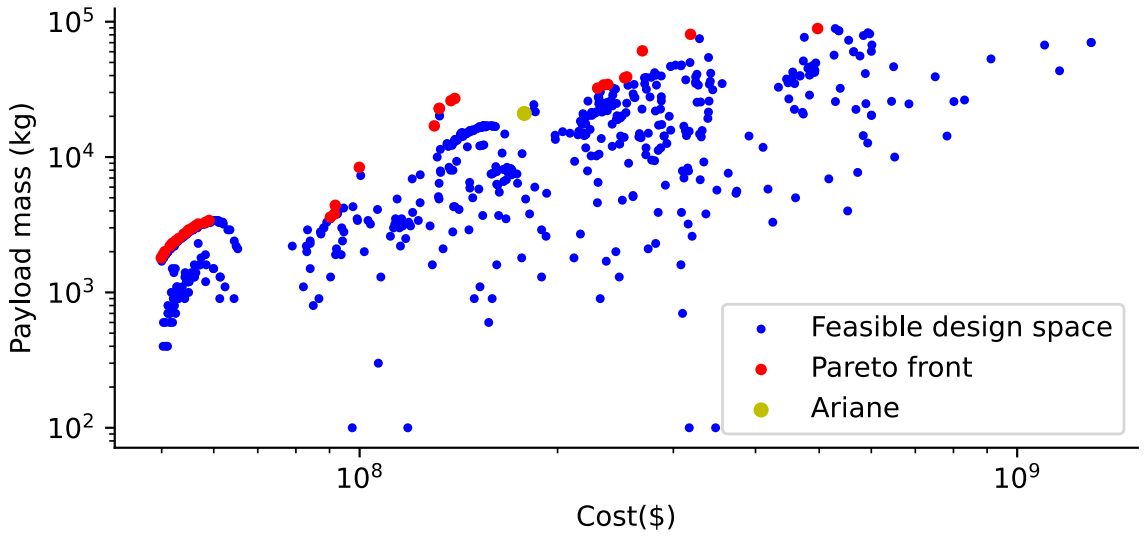


Figure 6.25: This figure shows the feasible design space of the space benchmark problem (blue) and the Pareto front (red). Each of them has associated a certain maximum payload mass (y-axis), and a cost (x-axis). The Ariane V data is in yellow.

More details about these results can be found in table 6.9. These results first confirm the importance of considering all the different design disciplines at the same time, as only then feasible design can be obtained. It also shows that it is possible to obtain optimized results without the necessity of exploring all the possible system architectures of the system. These results are also in the order of magnitude of existing rockets, such as the Ariane V. This has a payload mass capacity for LEO in the range of 21 tonnes, and an estimated cost of 178 million dollars (Urban, 2023a).

Table 6.9: Results space benchmark problem.

Number of possible architectures	18522
Number of architectures analysed	1031 (5.56 %)
Design points analysed	4500
Feasible design points	763 (18 %)
Pareto front design points	32
Pareto front architectures	10

Multiple different architectures can be found in the Pareto front. For example, regarding the number of stages, rockets with only one stage are found on the bottom left of the Pareto front. Of course, they are the cheapest, but the amount of payload they can lift to the orbit is low. If heavier quantities are wanted to be lifted, more stages will be needed, but the cost will be increased (figure 6.26)

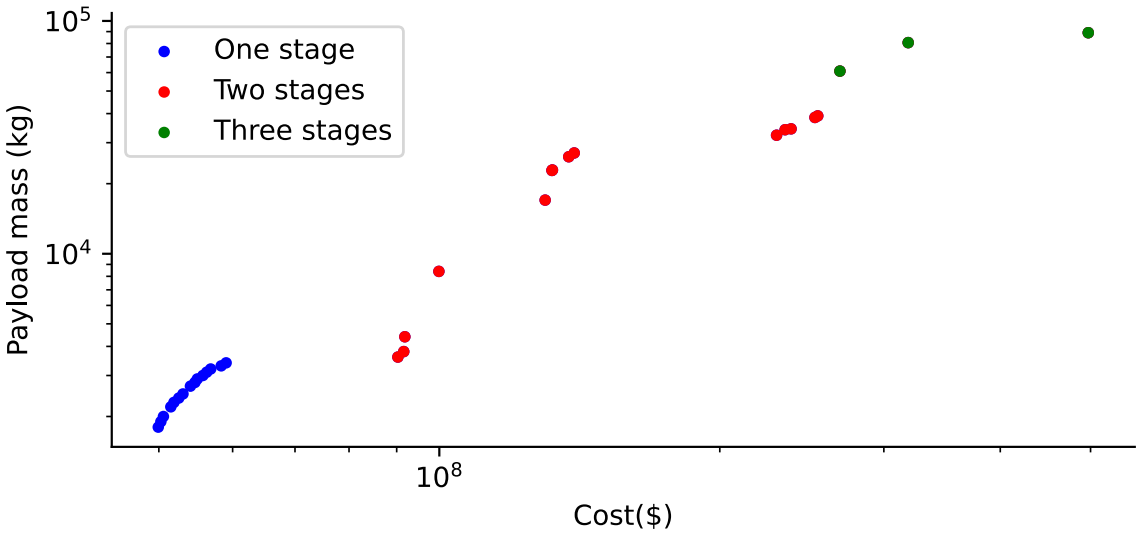


Figure 6.26: When the number of stages increases, the rocket payload mass capability increases, but the cost increases too.

Regarding the rocked head shape, the results obtained were also expected. Usually, with low payload masses, semispherical heads are being chosen. The semisphere is the shape from the possible sources that maximizes the amount of volume for a given surface (and therefore for a given cost). However, the semisphere is the worst shape according to aerodynamics, as it has the highest drag coefficient. When heavier payloads have to be lifted, aerodynamics plays a more important role, and that justifies why elliptical head shapes are being chosen, although they are usually the most expensive ones. In the intermediate cases, usually conical heads appear, as they are a compromise solution between aerodynamics and cost.

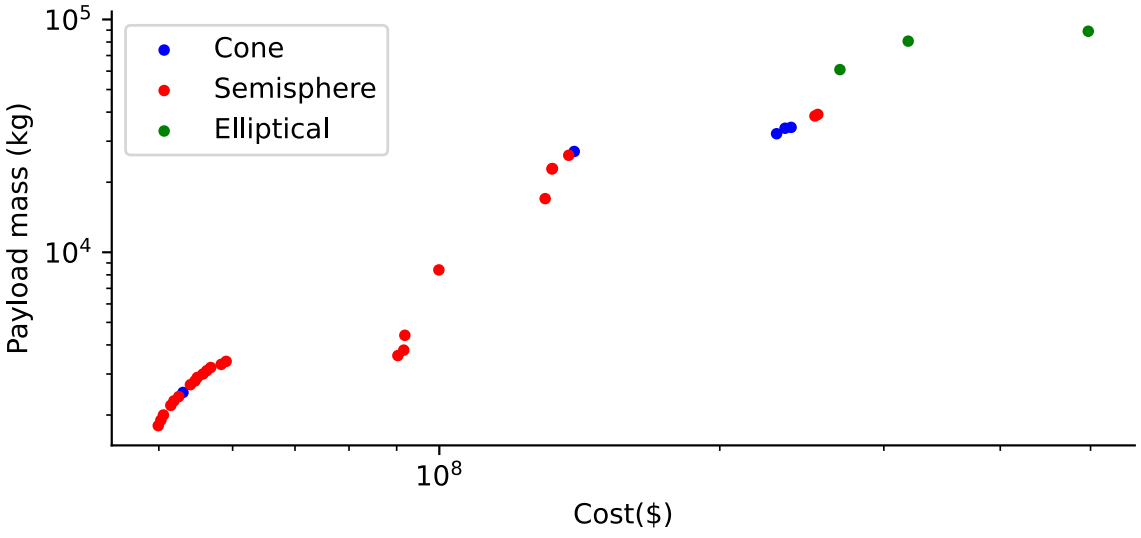


Figure 6.27: Different architectures found in the Pareto front according to the head shape.

It is also interesting to take a look at the type of propellant chosen for the first stage of feasible design points. As it can be observed in figure 6.28, there are two distinctive parts in the Pareto front. Solid propellant is chosen for the right part of the Pareto, while liquid propellant is chosen when lower payload masses are wanted to be lifted. The reason for this are the two key differences between these propellants.

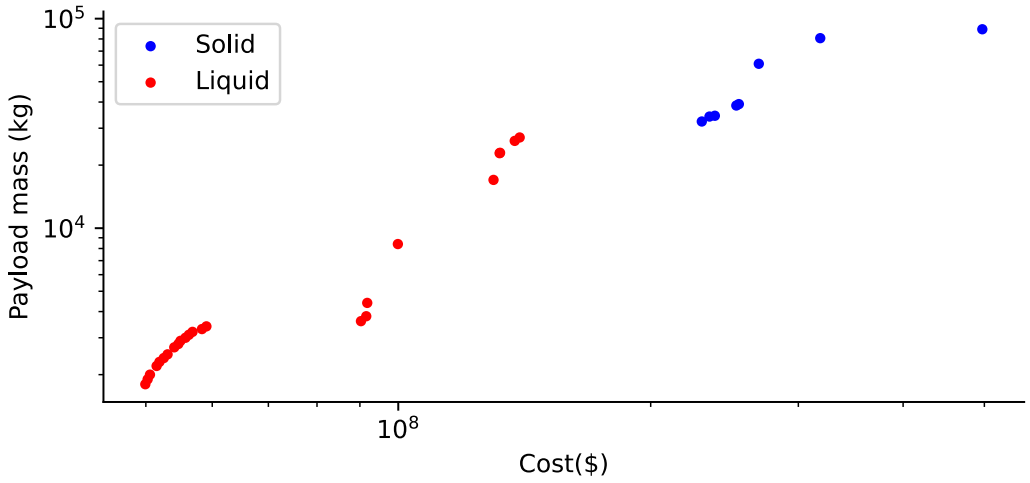


Figure 6.28: This figure shows the propellant type (solid or liquid) for the rockets first stages. As it can be observed, there are two clear clusters inside the Pareto front.

Solid propellant allows to generate more power (and more acceleration). However, their density is higher compared to liquid propellants. Therefore, solid propellant rockets are really powerful, but also have a low working time (as they have usually attached a higher mass flow rate) and the propellant mass associated is also much higher. Usually, liquid propellant is chosen for medium size rockets, as they are really effective, as well as lighter. When the payload mass wanted to be lifted is higher, and a bigger rocket is needed, solid propellant is chosen usually for the first stage. This allows to generate the necessary power to lift the rocket in the first moment of the flight and getting rid of the atmospheric drag as soon as possible, as bigger rockets suffer more the effects of drag. All this can be observed in figure 6.29.

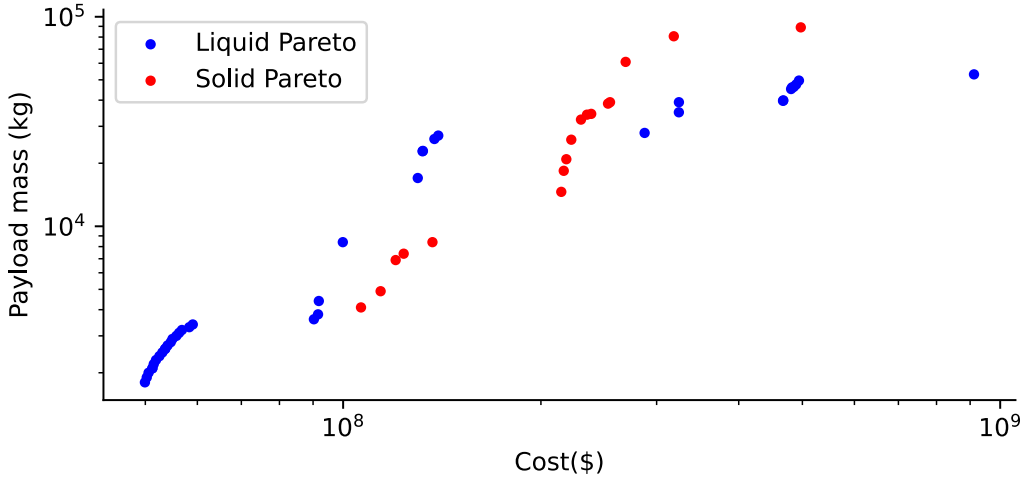


Figure 6.29: This figure shows the two Pareto fronts collision for the rocket propellant first stage.

Finally, it has also been obtained that for multistage rockets found in the Pareto front, all architectures prefer liquid propellant for the second and third stages. This confirms the previous explanations regarding the propellant type and also corresponds to the examples found in existing launcher designs.

6.2.4. Comparison to traditional approach

As mentioned in this research, there were already some examples of system architecture optimization problems using MDO for the evaluation. In most of these problems, Python code was used to formulate and execute these MDO problems. In this section a comparison is going to be made between this previous approach and the methodology adapted to collaborative MDO that has been presented in this research. To do so, an adaptation of the space benchmark problem in Python [5] will be used as a base.

The first difference is the amount of **coding**. Right now, the only coding needed in MDAx is mainly to import the inputs and outputs of the tools and to call the exporter function to RCE. However, in a close future no coding will be necessary at all due to the soon inclusion of architectural influences in the GUI. In the case of the Python implementation, all the problem has to be coded (including the workflow), demanding hundreds of coding lines. This problem is even worse when elements like convergers are needed, as it makes the code really complex. Another disadvantage of this traditional approach is that it has an **important opening barrier**. The user has to be familiar with the coding language being used. This, linked to the extensive coding needed, also leads to considerably **longer times to formulate the MDO problem**.

Another main problem of this approach is related with the **MDO problem implementation**. The traditional method follows an **imperative approach**, opposite to the **declarative approach** of this new methodology. This means that the user has to describe/determine how the process has to be performed, including all necessary instructions. In the declarative approach, the user just states what is wanted to be achieved, but not how it has to be done. This leads to important differences.

First, regarding the inclusion of **modifications in the MDO problem**. Usually, the final MDO problem is formulated after some iterations. This will involve trying different MDO architectures or including/excluding different disciplines in the problem (and their respective connections). This in MDAx can be achieved immediately. However, this is not true for the traditional imperative approach, as the user has to change manually the code for any possible tried formulation. This shows one of the main reasons of why the traditional approach should not be used for collaborative MDO, as it makes this process tedious and time consuming.

Another remarkable difference, also related with the imperative/declarative approach, is the **generation of connections** between the different disciplines. In MDAx, once inputs and outputs are defined, connection are made automatically thanks to the CDS. However, in this traditional approach the designer has to keep track of all possible connections existing in the MDO problem and implement them manually, which is not desired specially for problems with a high number of connections.

Apart from the higher difficulty and set up time that this traditional coding approach involves, it also has another main downside, and it is the **reusability**. As an example, consider the architectural influences. Custom Python code has been added to implement influences like discipline activation (if statements) or discipline repetition (for loops). However, if a new problem is wanted to be formulated, only part of this implementation process might be reusable. This is a great difference with MDAx, where all the methodology necessary to deal with architectural influences is already coded and the user just has to indicate the conditions for each of them (declarative over imperative).

There are also differences in this new methodology regarding the **declaration of inputs and outputs**. In MDAx, XML files given by tools developers are usually used to define inputs and outputs for each discipline. In the traditional approach, all this definition has to be done in a manual manner, again

[5] https://github.com/jbussemaker/SBArchOpt/blob/dev/sb_arch_opt/problems/rocket_eval.py

increasing considerable the implementation time. Finally, this coding approach does not include usual **tools necessary to formulate complex MDO problems**, such as collision detection/resolution, automatic ordering of disciplines, etc...

As a conclusion, the coding approach can be used when there are not many participants in the project, these participants have experience with the coding language and know all the possible elements in the MDO problem. However, even in that case, the setup time would be much longer due to considerably additional coding needed and the lack of tools to detect errors in the formulation. Also most part of the developed methodology could not be easily reused for other problems.

7

Conclusions & Recommendations

This last chapter provides the main conclusions and outputs from this research. Then some recommendations are given to achieve the desired implementation of system architecture optimization in the industry.

7.1. Conclusions

Determining the architecture of the system is one of the key steps of the design process, as it hugely influences its performance, and consequently the project success. Traditionally, several possible architectural candidates were chosen manually based on experts knowledge, leading to bias and conservatism. System architecture optimization allows to objectively look for the best architectures by formalizing and solving an optimization problem. To do so, first an architecture generator is used to formalize the architectural design space. Then, an architecture evaluator is needed to provide numerical feedback to the optimizer for each architecture being proposed.

For this latest task, MDO could be used in the aerospace industry, as it considers the couplings existing between the multiple design disciplines. If it is desired to implement MDO in real industry system architecture optimization problems, the MDO platform used to formulate and execute the MDO problem has to be adapted to the methodologies encompassed by collaborative MDO, allowing to coordinate the different experts necessary in the process.

The MDO platform used has to satisfy some additional requirements too, such as dealing with mixed-discrete variables or incorporating/being able to connect to complex optimization algorithms to solve these problems. However, the main challenge is readjusting the MDO problem automatically for each system architecture, as the variables, connections and disciplines vary for each one.

When performing the literature study, it was determined that there was not any MDO platform able to deal with the previous requirements while adapted to collaborative MDO. The MDO platform based on the joining of MDAx and RCE to formulate and execute the MDO problem (respectively) was already adapted to collaborative MDO. It also satisfied all the previous requirements except the automatic readjustment of the MDO problem. To fill the existing technological gap, the objective of the thesis was to adapt MDAx to be used as an architecture evaluator, by extending its backend code to model and execute MDO problems that can be readjusted automatically during the optimization process, depending on the architecture being analysed.

To achieve this, the first step was to determine the possible modifications that the different system architectures can produce in the MDO problem formulation/execution, called architectural influences. A total of four different architectural influences have been discovered (conditional variables, data connection, discipline repetition and discipline activation). These **architectural influences** are the first outcome of this research, as there was no formal study about the impact system architecture optimization has in

the MDO problem formulation/execution.

Once the different architectural influences were determined, the next step was to extend MDAx back-end code to deal with them. By achieving this, a **platform for evaluating system architectures using MDO** has been obtained, allowing to include all the benefits of MDO in the system architecture optimization process.

Another main outcome of this thesis has been **the development of a guide for the design of tools adapted to system architecture optimization**. Tools to be used for system architecture optimization should own several properties. These include the possibility of receiving different inputs depending on the system architecture, being able to deal with complex data structures (such as arrays) or following standardized schema for the variables. All these requirements have been discussed in this research, specially regarding architectural influences, including also an example of the implementation process.

An additional outcome of this research, related with this implementation process, is the **numerical optimization problem based on Fourier series**. This is a simple problem that contains all the architectural influences and that can be used by developers to verify the implementation of architectural influences in any MDO platform. Finally, a **space multistage rocket design problem** has been solved using system architecture optimization. This problem has shown the utility of this methodology to obtain the best possible architectures of a system by exploring automatically the design space.

Regarding the benchmark problems, it has been identified the necessity of developing an **standardized notation for MDO problems with architectural influences**. This notation should be able to indicate in a simple manner the different architectural influences in the XDSM. It should also be adapted for a representation of the problem in physical media, such as paper. A **first possible notation** has been suggested in this research.

To **contribute to the promotion of system architecture optimization**, a repository has been created on Github [1][2] for each of the benchmark problems, so that any researcher/student can execute in RCE (which is open source) some possible architectures, observing the effects of the different architectural influences in the problem. This is aimed to expand the concept of architectural influences and the importance of dealing with them to use system architecture optimization.

As a summary, this research has allowed to **reduce the existing gap between MDO and system architecture optimization** , being a forward step in the ultimate goal of **including system architecture optimization in real engineering design processes**. However, there are still important steps to be taken to finally close this gap, such as the development of a methodology to translate the architectures given by the architecture generator into inputs that can be understood by the tools or the formalization of the nomenclature suggested previously. Also new functionalities should be included, such as the possibility to modify the order of execution of the MDO problem disciplines depending on the system architecture.

7.2. Recommendations

This section includes multiple recommendations to achieve the inclusion of system architecture optimization in the industry. First, regarding the specific case of MDAx, it is recommended:

- **Incorporation of architectural influences to the GUI**: Right now, the only way to formulate MDO problems including architectural influences in MDAx is through the back end. If MDAx is desired to be used for real engineering cases with architectural influences, all the new code implementations regarding architectural influences have to be extended to the front end (the GUI).

[1] https://github.com/raul7gs/Fourier-benchmark-problem
[2] https://github.com/raul7gs/Space_launcher_benchmark_problem

- **Simplification of RCE export**: Right now, multiple blocks are necessary to be exported to RCE for each discipline. Although this allows to differentiate the different steps during each tool execution, it can lead to really complex workflows when multiple tools are included. It can also make it difficult to handle them and to make the necessary modifications for the workflow execution.

  The original idea was to include only one block before and after the tool block, so that all the procedures regarding inputs and outputs (including architectural influences) were gathered together. This idea was discarded because of time reasons, as it would involve modifying a great part of the code. However, the RCE export should be simplified in the future in order to be used for real design cases.

- **Implementation of architectural influences for tool groups**: It is common in complex MDO problems to be used for system architecture optimization that several disciplines/tools share a common architectural influence. Therefore, allowing to attach a certain architectural influence to multiple disciplines at the same time, without repeating it for each of them individually, would reduce the problem formulation time.

- **Extension of CMDOWS to deal with architectural influences:** Right now, if architectural influences are found in the problem, only the RCE export is available in MDAx. It is suggested to adapt CMDOWs to deal with the different architectural influences. Then, by adapting MDAx export to CMDOWs, it would be possible to generate executable workflows for other MDO platforms, such as OpenMDAO.

Then, regarding system architecture optimization in a more generic manner, the following steps are recommended:

- **Promotion of system architecture optimization:** The idea of system architecture optimization is still not widely spread in the industry. To enjoy its benefits in real case engineering design problems, first it is necessary to promote this methodology. To do so, more system architecture optimization problems should be solved, so that the potential of this methodology is shown. Also workshops should be organized for this purpose. The development of open source platforms able to formulate and execute MDO problems for system architecture optimization would also be an important boost for the promotion of system architecture optimization.

- **Formulation and execution of a complex system architecture optimization problem:** Most of the existing system architecture optimization problems are focused on a specific subsystem (such as the propulsion system of an aircraft). More complex problems regarding systems of systems should be developed, with hundreds of variables and multiple architectural influences. There are two reasons for this.

  First, it would show to the different experts that system architecture optimization can be used for real complex design problems, allowing to further promote this methodology. The second reason is that it would also serve as a test for all the different tools used in the optimization process (from the optimizer to the evaluator, passing through the generator), as they have not been tested with complex real case scenarios yet.

This research has discussed how MDO platforms can be adapted to be used for system architecture optimization. The different requirements to be satisfied, as well as possible validation problems, have been discussed. It is aimed that this research allows to promote the potential of system architecture optimization and serves as a step forward in the inclusion of this methodology in the industry.

References

Abedini, A., Bataleblu, A. A., & Roshanian, J. (2022). Co-design optimization of a novel multi-identity drone helicopter (MICOPTER). *Journal of Intelligent & Robotic Systems*, *106*(3), 56.

Aïello, O., Poitou, O., Chaudemar, J.-C., & De Saqui-Sannes, P. (2022). Sizing a drone battery by coupling MBSE and MDAO. *11th European Congress Embedded Real Time Systems (ERTS)*.

Aigner, B., van Gent, I., La Rocca, G., Stumpf, E., & Veldhuis, L. L. (2018). Graph-based algorithms and data-driven documents for formulation and visualization of large MDO systems. *CEAS Aeronautical Journal*, *9*, 695–709.

Akin, D. L. (2016). *Mass estimating relations. ENAE 791 - Launch and Entry Vehicle Design* (tech. rep.). https://spacecraft.ssl.umd.edu/academics/791S16/791S16L08.MERsx.pdf

Alder, M., Moerland, E., Jepsen, J., & Nagel, B. (2020). Recent advances in establishing a common language for aircraft design with CPACS.

Ali, J. S. M., Embong, W. M. H. W., & Aabid, A. (2021). Effect of cut-out shape on the stresses in aircraft wing ribs under aerodynamic load. *CFD Letters*, *13*(11), 87–94.

Anibal, J. L., Mader, C., & Martins, J. R. (2020). Aerothermal optimization of X-57 high-lift motor nacelle. *AIAA Scitech 2020 Forum*.

Apaza, G., & Selva, D. (2021). Automatic composition of encoding scheme and search operators in system architecture optimization. *International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, *85376*, V002T02A071.

Astronautix. (1997). *J-2 engine*. Retrieved October 26, 2023, from http://www.astronautix.com/j/j-2.html

ATK. (2020). *Gem-60 solid rocket strap-on booster*. Retrieved October 26, 2023, from https://gandalfddi.z19.web.core.windows.net/Shuttle/Misc_Space_Non-Shuttle/GEM-60.pdf

Baalbergen, E., Kos, J., Louriou, C., Campguilhem, C., & Barron, J. (2017). Streamlining cross-organisation product design in aeronautics. *Proceedings of the Institution of Mechanical Engineers, Part G: Journal of Aerospace Engineering*, *231*(12), 2192–2202.

Baalbergen, E., Vankan, J., Boggero, L., Bussemaker, J. H., Lefèbvre, T., Beijer, B., Bruggeman, A.-L., & Mandorino, M. (2022). Advancing cross-organizational collaboration in aircraft development. *AIAA AVIATION 2022 Forum*, 4052.

Boden, B., Flink, J., Mischke, R., Schaffert, K., Weinert, A., Wohlan, A., Ilic, C., Wunderlich, T., Liersch, C. M., Goertz, S., et al. (2019). Distributed multidisciplinary optimization and collaborative process development using RCE. *AIAA aviation 2019 forum*, 2989.

Boggero, L., Ciampa, P. D., & Nagel, B. (2021). An MBSE architectural framework for the agile definition of system stakeholders, needs and requirements. *AIAA Aviation 2021 Forum*, 3076.

Borghi, R., & Spinozzi, T. M. (2017). Space shuttle's liftoff: A didactical model. *European Journal of Physics*, *38*(4), 045006.

Braeunig, R. A. (1996). *Rocket propellants*. Retrieved October 26, 2023, from https://www.scss.tcd.ie/Stephen.Farrell/ipn/background/Braeunig/propel1.htm

Brelje, B. J., & Martins, J. R. (2018). Development of a conceptual design model for aircraft electric propulsion with efficient gradients. *2018 AIAA/IEEE electric aircraft technologies symposium (EATS)*, 1–25.

Broy, M. (2013). Modellbasierte Virtuelle Produktentwicklung auf einer Plattform für System Lifecyle Management. *Industrie 4.0: Beherrschung der industriellen Komplexität mit SysLM*, 91–110.

Bruggeman, A.-L., van Manen, B., van der Laan, T., van den Berg, T., & La Rocca, G. (2022). An mbse-based requirement verification framework to support the mdao process. *AIAA AVIATION 2022 Forum*, 3722.

Bruggeman, A.-L. M., & La Rocca, G. (2023). From requirements to product: An mbse approach for the digitalization of the aircraft design process. *INCOSE International Symposium*, *33*(1), 1688–1706.

Bussemaker, J. H. (2023). SBArchOpt: Surrogate-Based Architecture Optimization. *Journal of Open Source Software*, *8*(89), 5564.

Bussemaker, J. H., Bartoli, N., Lefebvre, T., Ciampa, P. D., & Nagel, B. (2021). Effectiveness of surrogate-based optimization algorithms for system architecture optimization. *AIAA AVIATION 2021 FORUM*, 3095.

Bussemaker, J. H., & Boggero, L. (2022). Technologies for enabling system architecture optimization. *ONERA-DLR Aerospace Symposium (ODAS)*.

Bussemaker, J. H., Boggero, L., & Ciampa, P. D. (2022). From system architecting to system design and optimization: A link between MBSE and MDAO. *INCOSE International Symposium*, *32*(1), 343–359.

Bussemaker, J. H., & Ciampa, P. D. (2022). MBSE in architecture design space exploration. In *Handbook of model-based systems engineering* (pp. 1–41). Springer.

Bussemaker, J. H., Ciampa, P. D., & Nagel, B. (2020). System architecture design space exploration: An approach to modeling and optimization. *AIAA Aviation 2020 Forum*, 3172.

Bussemaker, J. H., De Smedt, T., La Rocca, G., Ciampa, P. D., & Nagel, B. (2021). System architecture optimization: An open source multidisciplinary aircraft jet engine architecting problem. *AIAA Aviation 2021 Forum*, 3078.

Bussemaker, J. H., Garcia Sanchez, R., Fouda, M., Boggero, L., & Björn, N. (2023). Function-based architecture optimization: An application to hybrid-electric propulsion systems. *33rd Annual INCOSE international symposium (INCOSE)*.

Bussemaker, J. H., Garg, S., & Boggero, L. (2022). The influence of architectural design decisions on the formulation of MDAO problems. *3rd European Workshop on MDO*.

Caliò, E., Di Giorgio, F., & Pasquinelli, M. (2016). Deploying model-based systems engineering in Thales Alenia Space Italia. *CIISE*, 112–118.

Chaudemar, J.-C., & de Saqui-Sannes, P. (2021). MBSE and MDAO for early validation of design decisions: A bibliography survey. *2021 IEEE International Systems Conference (SysCon)*, 1–8.

Chemeurope. (2005). *Vulcain (rocket engine)*. Retrieved October 26, 2023, from https://www.chemeurope.com/en/encyclopedia/Vulcain_%28rocket_engine%29.html

Chen, H., Li, W., Cui, W., Yang, P., & Chen, L. (2021). Multi-objective multidisciplinary design optimization of a robotic fish system. *Journal of marine science and engineering*, *9*(5), 478.

Chepko, A., de Weck, O., Crossley, W., Santiago-Maldonado, E., & Linne, D. (2008). A modeling framework for applying discrete optimization to system architecture selection and application to in-situ resource utilization. *12th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, 6058.

Ciampa, P. D., La Rocca, G., & Nagel, B. (2020). A MBSE approach to MDAO systems for the development of complex products. *AIAA Aviation 2020 Forum*.

Ciampa, P. D., & Nagel, B. (2016). Towards the 3rd generation MDO collaborative environment. *30th ICAS*, 1–12.

Ciampa, P. D., & Nagel, B. (2020). Agile paradigm: The next generation collaborative MDO for the development of aeronautical systems. *Progress in Aerospace Sciences*, *119*, 100643.

Crawley, E., Cameron, B., & Selva, D. (2015). *System architecture: Strategy and product development for complex systems*. Prentice Hall Press.

Crea, L. (2022). *Multidisciplinary design optimization of an electric race car*. McGill University (Canada).

Creech, S., Taylor, J., Bellamy, S., & Kuck, F. (2008). *Ares V and RS-68B* (tech. rep.). https://ntrs.nasa.gov/api/citations/20090014109/downloads/20090014109.pdf

DAU. (1993). Committed Life Cycle Cost against Time.

Deb, K., Pratap, A., Agarwal, S., & Meyarivan, T. (2002). A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE transactions on evolutionary computation*, *6*(2), 182–197.

DeTurris, D., & Palmer, A. (2018). Perspectives on managing emergent risk due to rising complexity in aerospace systems. *INSIGHT*, *21*(3), 80–86.

de Vries, D., van Gent, I., La Rocca, G., & Binder, S. (2017). OpenLEGO demonstration: A link between AGILE and OpenMDAO. *First European OpenMDAO workshop*.

Dincer, I. (2018). *Comprehensive energy systems*. Elsevier.

Eigner, M., Huwig, C., Dickopf, T., et al. (2015). Cost-benefit analysis in model-based systems engineering: State of the art and future potentials. *DS 80-7 Proceedings of the 20th International Conference on Engineering Design (ICED 15) Vol 7: Product Modularisation, Product Architecture, systems Engineering, Product Service Systems, Milan, Italy, 27-30.07. 15*, 227–236.

ESA. (2002). Vega qualification flight vv01. https://www.esa.int/esapub/br/br196/br196.pdf

Fedaravičius, A., Kilikevičius, S., & Survila, A. (2012). Optimization of the rocket's nose and nozzle design parameters in respect to its aerodynamic characteristics. *Journal of vibroengineering*, *14*(4), 1885–1891.

Frank, C., Atanian, M. F., Pinon-Fischer, O. J., & Mavris, D. N. (2016). A conceptual design framework for performance, life-cycle cost, and safety evaluation of suborbital vehicles. *54th AIAA Aerospace Sciences Meeting*, 1276.

Friedenthal, S., Griego, R., & Sampson, M. (2007). INCOSE model based systems engineering (MBSE) initiative. *INCOSE 2007 symposium*, *11*.

Gallard, F., Vanaret, C., Guénot, D., Gachelin, V., Lafage, R., Pauwels, B., Barjhoux, P.-J., & Gazaix, A. (2018). GEMS: A python library for automation of multidisciplinary design optimization process generation. *2018 AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*, 0657.

Gonzalez, L. F., Periaux, J., Srinivas, K., & Whitney, E. J. (2004). Evolutionary optimization tools for multi objective design in aerospace engineering: From theory to MDO applications. evolutionary algorithms and intelligent tools in engineering optimization. *CIMNE*.

Gordon, S., & McBride, B. J. (1959). *Theoretical performance of liquid hydrogen with liquid oxygen as a rocket propellant* (tech. rep.).

Gray, J. S., Hwang, J. T., Martins, J. R., Moore, K. T., & Naylor, B. A. (2019). OpenMDAO: An open-source framework for multidisciplinary design, analysis, and optimization. *Structural and Multidisciplinary Optimization*, *59*, 1075–1104.

Grumman, N. (2023). GEM motor series. https://wpcontent.ot5o9s93syrb.net/wp-content/uploads/gem-motor-series.pdf

Gur, O., & Rosen, A. (2009). Design of quiet propeller for an electric mini unmanned air vehicle. *Journal of propulsion and power*, *25*(3), 717–728.

Haberfellner, R., De Weck, O., Fricke, E., & Vössner, S. (2019). *Systems engineering: Fundamentals and applications*. Springer.

Hedayat, A., Knight, K., & Champion Jr, R. (1998). Transient analysis of X-34 pressurization system. *PERC Annual Propulsion Symposium*.

Hegseth, J. M., Bachynski, E. E., & Martins, J. R. (2020). Integrated design optimization of spar floating wind turbines. *Marine Structures*, *72*, 102771.

Hirshorn, S. R., Voss, L. D., & Bromley, L. K. (2017). *NASA systems engineering handbook* (tech. rep.).

Hoerner, S. F. (1965). Fluid-dynamic drag. theoretical, experimental and statistical information. *Copyright by: SF Hoerner Fluid Dynamics, Vancouver, Printed in the USA, Card Number 64-19666*.

Hwang, J. T., & Ning, A. (2018). Large-scale multidisciplinary optimization of an electric aircraft for on-demand mobility. *2018 AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*, 1384.

Iacobucci, J. V. (2012). *Rapid Architecture Alternative Modeling (RAAM): A framework for capability-based analysis of system of systems architectures*. Georgia Institute of Technology.

Jagdishmetalindia. (2023). *Aluminium 2024 sheet price list in india*. Retrieved October 26, 2023, from https://www.jagdishmetalindia.com/aluminium-2024-sheet-bar-price-list.html

Jeong, J., Shi, H., Lee, K., & Kang, B. (2020). Improvement of electric propulsion system model for performance analysis of large-size multicopter UAVs. *Appl. Sci. (Basel)*, *10*(22), 8080.

Kanner, H. S., Freeland, D. M., Olson, D. T., Wood, T. D., & Vaccaro, M. V. (2011). Solid Rocket Booster (SRB)-Evolution and Lessons Learned During the Shuttle Program. *AIAA Space 2011 Conference and Exposition*, (M11-1026).

Koelle, D. E. (2007). Handbook of cost engineering for space transportation systems with Transcost 7.2: Statistical-analytical model for cost estimation and economical optimization of launch vehicles.

Lafage, R., Defoort, S., & Lefebvre, T. (2019). WhatsOpt: A web application for multidisciplinary design analysis and optimization. *AIAA Aviation 2019 Forum*, 2990.

Lambe, A. B., & Martins, J. R. (2012). Extensions to the design structure matrix for the description of multidisciplinary design, analysis, and optimization processes. *Structural and Multidisciplinary Optimization*, *46*, 273–284.

Le Digabel, S., & Wild, S. M. (2023). A taxonomy of constraints in black-box simulation-based optimization. *Optimization and Engineering*, 1–19.

Lin, S.-H. E., & Gerber, D. J. (2014). Designing-in performance: A framework for evolutionary energy performance feedback in early stage design. *Automation in Construction*, *38*, 59–73.

Long, T., Liu, L., Zhou, S., Wang, J., & Meng, L. (2008). Multi-objective multidisciplinary optimization of long-endurance UAV wing using surrogate model in modelcenter. *12th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, 5918.

Madni, A. M., & Purohit, S. (2019). Economic analysis of model-based systems engineering. *Systems*, *7*(1), 12.

Mahiddini, B., Chettibi, T., Benfriha, K., & Aoussat, A. (2019). Multidisciplinary design optimization of a gear train transmission. *Concurrent Engineering*, *27*(3), 268–281.

Martins, J. R., & Lambe, A. B. (2013). Multidisciplinary design optimization: A survey of architectures. *AIAA journal*, *51*(9), 2049–2075.

Martins, J. R., & Ning, A. (2021). *Engineering design optimization*. Cambridge University Press.

Mesmer, B., Mckinney, D., Watson, M., & Madni, A. M. (2022). Transdisciplinary systems engineering approaches. In *Recent trends and advances in model based systems engineering* (pp. 579–590). Springer.

Moerland, E., Ciampa, P. D., Zur, S., Baalbergen, E., Noskov, N., D'Ippolito, R., & Lombardi, R. (2020). Collaborative architecture supporting the next generation of MDAO within the AGILE paradigm. *Progress in Aerospace Sciences*, *119*, 100637.

Müller, J., & Day, M. (2019). Surrogate optimization of computationally expensive black-box problems with hidden constraints. *INFORMS Journal on Computing*, *31*(4), 689–702.

Obstbaum, M., Wurstbauer, U., König, C., Wagner, T., Kübler, C., & Fäßler, V. (2017). From a graph to a development cycle: MBSE as an approach to reduce development efforts. *Deutsche Gesellschaft für Luft-und Raumfahrt-Lilienthal-Oberth eV*.

Onel, A.-I., & Chelaru, T.-V. (2022). Parametric analysis of a two-stage small launcher using a MDO approach. *INCAS Bulletin*, *14*(1), 79–96.

OpenMDAO. (2019). *OpenMDAO website*. Retrieved April 24, 2023, from https://openmdao.org/

Page Risueño, A., Bussemaker, J. H., Ciampa, P. D., & Nagel, B. (2020). MDAx: Agile generation of collaborative MDAO workflows for complex systems. *AIAA Aviation 2020 Forum*, 3133.

Pandi Perumal, R., Voos, H., Dalla Vedova, F., & Moser, H. (2020). Comparison of multidisciplinary design optimization architectures for the design of distributed space systems. *Proceedings of the 71st International Astronautical Congress 2020*.

Pate, D. J., Gray, J., & German, B. J. (2014). A graph theoretic approach to problem formulation for multidisciplinary design analysis and optimization. *Structural and Multidisciplinary Optimization*, *49*, 743–760.

Planès, T., Delbecq, S., Pommier-Budinger, V., & Bénard, E. (2023). Modeling and design optimization of an electric environmental control system for commercial passenger aircraft. *Aerospace*, *10*(3), 260.

Pokhrel, M., Sarojini, D., & Mavris, D. N. (2023). Conceptual aero-structural design of a fan stage under distorted flow. *Proceedings of the Institution of Mechanical Engineers, Part G: Journal of Aerospace Engineering*, *237*(11), 2545–2560.

Roelofs, M., & Vos, R. (2018). Technology evaluation and uncertainty-based design optimization: A review. *2018 AIAA Aerospace Sciences Meeting*, 1–21.

Sakaki, K., Kakudo, H., Nakaya, S., Tsue, M., Suzuki, K., Kanai, R., Inagawa, T., & Hiraiwa, T. (2017). Combustion characteristics of ethanol/liquid-oxygen rocket-engine combustor with planar pintle injector. *Journal of Propulsion and Power*, *33*(2), 514–521.

Sellar, R., Batill, S., & Renaud, J. (1996). Response surface based, concurrent subspace optimization for multidisciplinary system design. *34th aerospace sciences meeting and exhibit*, 714.

Selva, D., Cameron, B., & Crawley, E. (2016). Patterns in system architecture decisions. *Systems Engineering*, *19*(6), 477–497.

Sinsay, J. D., & Alonso, J. J. (2015). Optimization of a lift-offset compound helicopter in a multidisciplinary analysis environment. *AHS 71st Annual Forum and Technology Display*.

Sobieszczanski-Sobieski, J., Agte, J., & Sandusky, R., Jr. (1998). Bi-level integrated system synthesis (bliss). *7th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, 4916.

Sobieszczanski-Sobieski, J., Morris, A., & Van Tooren, M. (2015). *Multidisciplinary design optimization supported by knowledge based engineering*. John Wiley & Sons.

SpaceX. (2021). *Falcon user´s guide* (tech. rep.). https://www.spacex.com/media/falcon-users-guide-2021-09.pdf

Sumrall, J. P., & McArthur, J. C. (2007). Foundation for heavy lift: Early developments in the Ares V cargo launch vehicle. *AIAA Joint Propulsion Conference*, (MSFC-334).

Thomas, J. C. (2018). Mixed HTPB/paraffin fuels and metallic additives for hybrid rocket applications. *phD, Texas AM University*.

Tyree, J., & Akerman, A. (2005). Architecture decisions: Demystifying architecture. *IEEE software*, *22*(2), 19–27.

University, P. (1998). *Boeing rocketdyne RS-68*. Retrieved October 26, 2023, from https://engineering.purdue.edu/~propulsi/propulsion/rockets/liquids/rs68.html

Urban, R. (2023a). *How much does it cost to launch a rocket? [by type size]*. Retrieved December 4, 2023, from https://spaceimpulse.com/2023/08/16/how-much-does-it-cost-to-launch-a-rocket/#Cost_of_a_Heavy-Lift_Rocket_Launch

Urban, R. (2023b). *How much does rocket fuel really cost?* Retrieved October 26, 2023, from https://spaceimpulse.com/2023/06/13/how-much-does-rocket-fuel-cost/

van Gent, I. (2019). Agile MDAO systems: A graph-based methodology to enhance collaborative multidisciplinary design. *phD TU Delft*.

van Gent, I., & La Rocca, G. (2019). Formulation and integration of MDAO systems for collaborative design: A graph-based methodological approach. *Aerospace Science and Technology*, *90*, 410–433.

van Gent, I., La Rocca, G., & Hoogreef, M. F. (2018). CMDOWS: A proposed new standard to store and exchange MDO systems. *CEAS Aeronautical Journal*, *9*, 607–627.

van Gent, I., La Rocca, G., & Veldhuis, L. L. (2017). Composing MDAO symphonies: Graph-based generation and manipulation of large multidisciplinary systems. *18th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, 3663.

Verfondern, K., Cirrone, D., Molkov, V., Makarov, D., Coldrick, S., Ren, Z., Wen, J., Proust, C., Friedrich, A., Jordan, T., et al. (2021). Handbook of hydrogen safety: Chapter on LH2 safety.

Vidner, O., Pettersson, R., Persson, J. A., & Ölvander, J. (2021). Multidisciplinary design optimization of a mobile miner using the OpenMDAO platform. *Proceedings of the Design Society*, *1*, 2207–2216.

Walden, D. D., et al. (2015). Systems engineering handbook: A guide for system life cycle processes and activities. *Wiley*.

Wang, X., Li, M., Liu, Y., Sun, W., Song, X., & Zhang, J. (2017). Surrogate based multidisciplinary design optimization of lithium-ion battery thermal management system in electric vehicles. *Struct. Multidiscipl. Optim.*, *56*(6), 1555–1570.

Wichmann, A., Jäger, S., Jungebloud, T., Maschotta, R., & Zimmermann, A. (2015). System architecture optimization with runtime reconfiguration of simulation models. *2015 Annual IEEE Systems Conference (SysCon) Proceedings*, 660–667.

Wingborg, N., Skarstind, M., Sjöblom, M., Lindborg, A., Brantlind, M., Johansson, J., Ek, S., Liljedahl, M., & Kjellberg, J. (2017). Grail: Green solid propellants for launchers. *7th European Conf. for Aeronautics and Space Sciences (EUCASS), Milan, Italy*.

Woehler, S., Atanasov, G., Silberhorn, D., Fröhler, B., & Zill, T. (2020). Preliminary aircraft design within a multidisciplinary and multifidelity design environment. *Aerospace Europe Conference 2020*.

Yaylali, D. (2018). *Launch dynamics and gravity turn maneuvers*. Retrieved October 26, 2023, from https://www.asthecroworbits.com/Files/LaunchDynamics.pdf

A

Architectural influences in the sample MDO problems

In this appendix, the architectural decisions considered for the different problems in the sample are included, as well as the influence they originated in the MDO problem. As explaining each of them thoroughly would be too extensive, a few words are used to represent each of them, as a clue. There are many more architectural choices that could have been considered, finding in this table the ones that came as more ”natural” to the author.

Table A.1: Architectural influences found in the MDO problems sample.

Problem	I1	I2	I3	I4	Reference
Mining machine	Cutter shape	Batteries connections	Cutters number	Cutters material	Vidner et al., 2021
Wing aerostructural	Ribs number	-	-	Landing Gear Inclusion	Long et al., 2008
Robotic fish	Sections shape	Batt/Motors connections	Motors number	Motors energy source	Chen et al., 2021
Wind turbine	Stiffeners number	Networks connections	-	Material	Hegseth et al., 2020
Car MDO	Rear wing types	Batteries placement	-	Regenerative breaking	Crea, 2022
Electric aircraft	Engines type	Prop/batt connections	Engines number	Engines energy source	Hwang and Ning, 2018
Convertible UAV	Winglet geometry	Batt/Motors connections	Batteries number	Mobile wings rotors	Abedini et al., 2022
Modified SSBJ	Winglet geometry	Landing Gear placement	Engines number	Material	Sobieszczanski-Sobieski et al., 1998

Satellite	Solar panel types	-	Satellites number	Material	Pandi Perumal et al., 2020
Launcher	Nose shape	-	Stage number	Propellant type	Onel and Chelaru, 2022
Multi-objective UAV	High lift system	Batt/prop connections	Batteries number	Material	Gonzalez et al., 2004
ECS design	Inlet shapes	Flow connections	Inlets number	Exchanger type	Planès et al., 2023
Nacelle heat exchanger	Nacelle geometry	-	Nacelles number	-	Anibal et al., 2020
Fan Aerostructural	Blade types	-	Blades number	Blades material	Pokhrel et al., 2023
Building construction	Floors geometry	-	Floors number	Floor purpose	Lin and Gerber, 2014
Gear design	Teeth type	-	-	Gear material	Mahiddini et al., 2019
Thermal management	Batteries shape	-	Fans number	Fan type	Wang et al., 2017
Propeller	Blade sections	-	Blades number	Blades material	Gur and Rosen, 2009
Helicopter	Fuselage shape	-	Rotors number	Blades material	Sinsay and Alonso, 2015
Electric propulsion	Electric source	Batt/Motor connections	Rotors number	Battery type	Jeong et al., 2020

B

Mathematical problem ADSG

This appendix shows a possible architectural design space graph implemented in ADORE for the mathematical benchmark problem (figure B.1). All the design variables (including architectural) can be found in the ADSG. Although in this case there is not a real system, architectures can still be modelled from an abstract point of view. The boundary function is to approximate the objective function f(x). To determine how close the architecture chosen is from the objective, a quantity of interest (QOI) called "Approximation error" is used, which is wanted to be minimized .

To approximate the boundary function, an approximation based on fourier series is used, being necessary three functions to determine its terms. First, in the middle, it is determined what are going to be the trigonometric terms. The first step to do this is to decide what type of functions are going to be used (cosines, sines or both) or what is the same, if N_A and N_B are going to be higher than 0. Then, if they are, the actual number of each term is chosen. For each instance it is determined the value of the coefficient (A_i and B_i) and the correspondent harmonics (w). All this is done in the big grey subsystems blocks.

The next step is to determine the constant term A_0. The architectural decisions determining the calculation method for this term are found on the left of the ADORE model. First it is decided if the volume or the surface output of the C disciplines is going to be considered (x_c). Then, the terms that are going to be included for the calculation in case there are both sines and cosines is determined in the port called "Constant term connections" (x_{con}).

Finally, on the right, there is a port called "Number of terms" which ensures that the number of trigonometric terms in the architecture is always lower or equal to two.

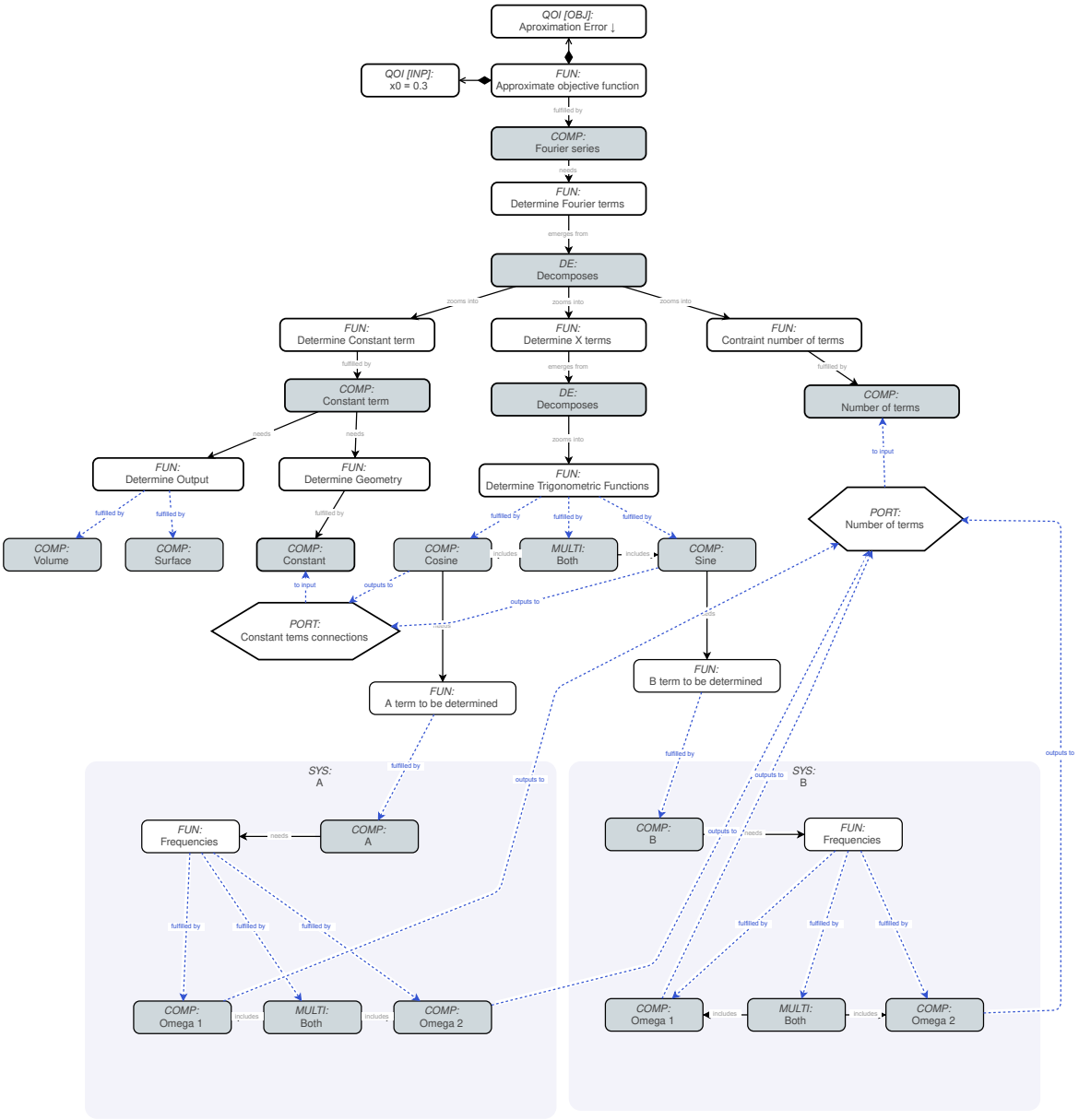


Figure B.1: Mathematical benchmark problem ADSG implemented in ADORE.

C

Space problem ADSG

This appendix includes a possible implementation in ADORE of the rocket architectural design space graph (figure C.1). The boundary function of the system is to carry payload to space. Two objectives are modelled as QOIs to measure the rocket performance. These are the maximum payload mass (wanted to be maximized) and the rocket cost (wanted to be minimized).

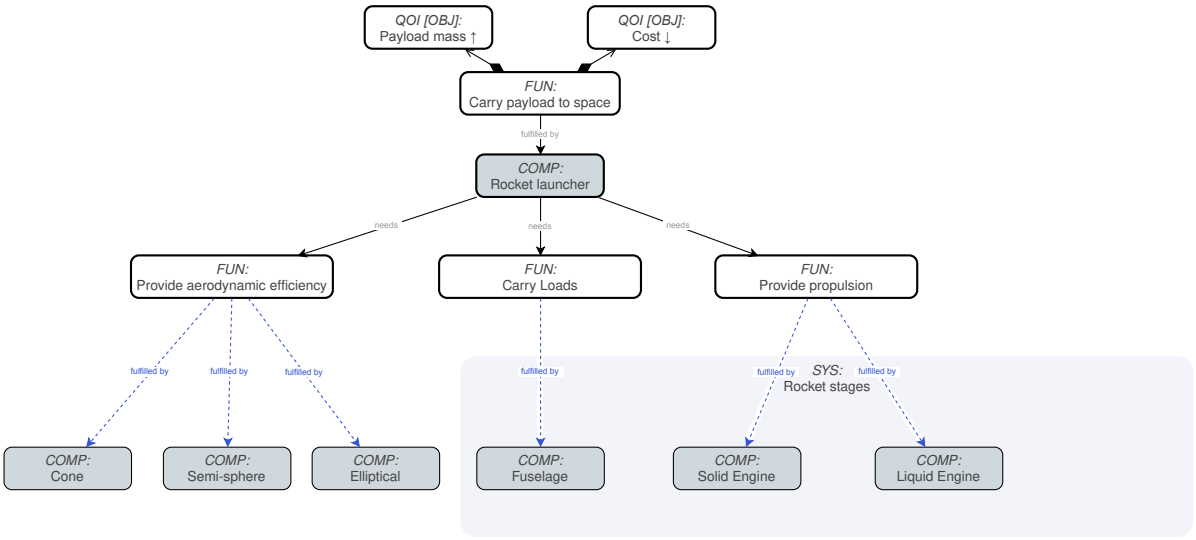


Figure C.1: Space benchmark problem ADSG implemented in ADORE.

Three main functions have to be performed by the rocket to achieve its mission. The first one is to be aerodynamically efficient. This is achieved adding an aerodynamic head (H_s), existing three different possible shapes. The second function is to carry structural loads, performed by the fuselage. Finally, it is necessary to provide propulsion for each stage (N_s), existing two types of possible propellant, and three possible engines for each of them ($Engine_i$).

Figure C.2 shows the implementation of all the problem inputs and constraints, which are modelled in the rocket launcher component. The same methodology is carried out with design variables regarding the possible head shapes, the fuselage or the possible engines to be chosen.

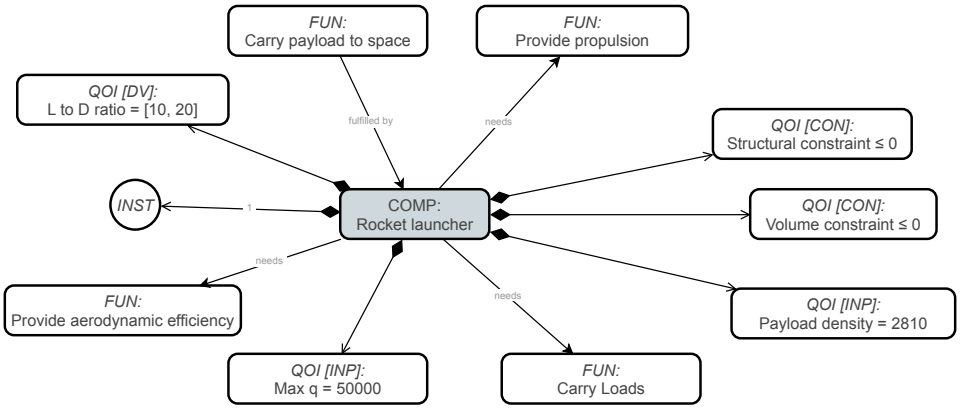


Figure C.2: Space benchmark problem constraints and fixed inputs modelled at the rocket launcher component level. Also the length to diameter variable is included.