



Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Maximilian Friedrichs-Dachale

Establishment of a Digital Interface Between System Definition and System Analysis Models to Optimize the Aircraft Preliminary Sizing Process

*Fakultät Technik und Informatik
Department Fahrzeugtechnik und Flugzeugbau*

*Faculty of Engineering and Computer Science
Department of Automotive and
Aeronautical Engineering*

Maximilian Friedrichs-Dachale

**Establishment of a Digital Interface
Between System Definition and System
Analysis Models to Optimize the Aircraft
Preliminary Sizing Process**

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Flugzeugbau
am Department Fahrzeugtechnik und Flugzeugbau
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Erstprüfer/in: Prof. Dr. Jutta Abulawi
Zweitprüfer/in : Prof. Dr.-Ing. Kay Kochan

Abgabedatum: 04.04.2022

Zusammenfassung

Maximilian Friedrichs-Dachale

Thema der Bachelorthesis

Establishment of a Digital Interface Between System Definition and System Analysis Models to Optimize the Aircraft Preliminary Sizing Process

Stichworte

MBSE, SysML, Schnittstelle, Datenaustausch, Digitale Prozesskette, Prozess automatisierung, Systemanalyse, Systemoptimierung, MDO, Aircraft Preliminary Sizing

Kurzzusammenfassung

Im Rahmen dieser Arbeit werden Möglichkeiten einer digitalen Verknüpfung zwischen Systemsdefinitionsmodellen und Systemanalysemodellen im Flugzeugvorentwurf untersucht. Eine solche digitale Schnittstelle könnte eine digitale Prozesskette ermöglichen, die eine automatisierte multidisziplinäre Systemoptimierung erlaubt. Zu diesem Zweck wird ein digitales Modell eines Flugzeugs mit dem SysML-basierten PTC Integrity Modeler entworfen. Zusätzlich wird ein Berechnungsmodell in MATLAB erstellt, mit dem das optimale Preliminary Design des Flugzeugs auf Basis einer zuvor definierten Zielfunktion berechnet wird. In einem weiteren Schritt werden Methoden für einen automatisierten Parameteraustausch zwischen den beiden Modellen erforscht. Schließlich wird die digitale Verbindung hergestellt und überprüft.

Maximilian Friedrichs-Dachale

Title of the paper

Establishment of a Digital Interface Between System Definition and System Analysis Models to Optimize the Aircraft Preliminary Sizing Process

Keywords

MBSE, SysML, Interface, Data Exchange, Digital Process Chain, Process Automatization, System Analysis, System Optimization, MDO, Aircraft Preliminary Sizing

Abstract

In this work, possibilities of a digital link between system definition models and system analysis models in aircraft preliminary design are investigated. Such a digital interface could enable a digital process chain that allows automated multidisciplinary system optimization. For this purpose, a digital model of an aircraft is designed using the SysML-based PTC Integrity Modeler. In addition, a computational model is created in MATLAB to calculate the optimal preliminary aircraft design based on a previously defined objective function. In a further step, methods for an automated parameter exchange between the two models are explored. Finally, the digital link is established and verified.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Objectives	2
1.3	Literature Review	3
1.4	Structure	3
2	Fundamentals of Model-Based Systems Engineering	4
2.1	Systems Engineering	4
2.1.1	Motivation for Systems Engineering	4
2.1.2	Systems Engineering Definition	5
2.1.3	The Systems Engineering Process	6
2.2	Model-Based Systems Engineering	7
2.2.1	Document-Based Systems Engineering	7
2.2.2	Model-Based Systems Engineering	7
2.3	The SysML	9
2.3.1	The SysML Origins	10
2.3.2	The SysML Syntax	10
2.3.3	The SysML Profile	10
2.3.4	Stereotyping	11
2.3.5	Modelling	11
2.4	SysML Diagrams	12
2.4.1	Overview	12
2.4.2	The Key Diagram Elements	13
2.4.3	Requirements	15
2.4.4	Blocks	16
2.4.5	Block Properties	19
2.4.6	Constraints and Parametric Modelling	22
2.4.7	Packages	27
2.5	Summary	30
3	Fundamentals of Aircraft Preliminary Sizing	31
3.1	Introduction	31
3.2	Aircraft Design Sequence	31
3.3	Aircraft Preliminary Sizing	37
3.3.1	Introduction and Main Idea	37
3.3.2	The Five Flight Phases	37

3.3.3	The Matching Chart	41
3.3.4	Calculation of the Output Parameters	42
3.4	Iterations in Aircraft Preliminary Sizing	46
3.5	Decision Making in Aircraft Preliminary Sizing	46
3.6	Optimization in Aircraft Preliminary Sizing	48
3.7	Systems Engineering in Aircraft Preliminary Sizing	50
3.8	Summary	51
4	The PTC Integrity Modeler	53
4.1	Introduction	53
4.2	The PTC Integrity Modeler	53
4.3	User Interface	53
4.4	Modelling	55
4.5	The PTC Integrity Automation Interface	55
4.6	The PTC Integrity Meta Model	56
4.7	Navigation in a Model With the Automation Interface	57
4.8	Controlling the User Interface	59
4.9	Summary	60
5	Establishment of the Interface	61
5.1	Introduction	61
5.2	Overview	61
5.3	Modelling the System in SysML	66
5.3.1	Loading the SysML Profile	66
5.3.2	Packages	66
5.3.3	Requirements	67
5.3.4	Block Definition	68
5.3.5	Internal Block Structure	69
5.3.6	Constraints	75
5.3.7	Data Types	76
5.3.8	Stereotypes	77
5.3.9	Parametric Diagram	77
5.4	Mathematical Model of the System in MATLAB	81
5.4.1	Overview	81
5.5	Interface Between Model Definition and Model Design	87
5.5.1	Introduction	87
5.5.2	Motivation	88
5.5.3	Possible Technical Solutions	90
5.5.4	Overview of the Visual Basic Interface	94
5.5.5	Required Libraries	96
5.5.6	The Visual Basic Code	97
5.5.7	Used Functions	102
5.6	Summary	109

6	Installation and Testing of the Visual Basic Interface	111
6.1	Installation of the Visual Basic Interface	111
6.1.1	Importing the Aircraft Model Into the PTC Integrity Modeler . .	111
6.1.2	Adding the Executable to the PTC Integrity Modeler Toolbar . .	111
6.2	Testing of the Visual Basic Interface	112
7	Discussion	116
8	Summary	118
9	Conclusions and Recommendations	119

List of Tables

6.1	Fixed input parameters for the two preliminary designs	113
6.2	Variable input parameters for the two preliminary designs	113
6.3	Most important output parameters for the two preliminary designs	114

List of Figures

1.1	Design groups' unique visions and interests (taken from [SAD12, fig. 2.15])	1
2.1	Evolution of electrical power need (gray: short- to medium-range aircraft and black: medium- to long-range aircraft, taken from [VIN18, fig.1]) . . .	4
2.2	V model (taken from [VDI21])	6
2.3	Architecture as an example of a model (taken from [ALT12, fig. 3.1]) . . .	8
2.4	Stereotyping model elements in SysML (taken from [HOL13, fig. 5.5]) . .	11
2.5	Relationship of metaclasses to model elements (taken from [FRI08, fig. 4.5])	12
2.6	SysML diagram taxonomy (taken from [FRI08, fig. 3.1])	13
2.7	A diagram frame (taken from [FRI08, fig. 4.8])	14
2.8	Examples of node symbols (taken from [FRI08, fig. 4.9])	14
2.9	Examples of path symbols (taken from [FRI08, fig. 4.10])	14
2.10	Examples of icon symbols (taken from [FRI08, fig. 4.11])	15
2.11	Exemplary requirement diagram to model the aircraft's requirements (based on [FRI08, fig. 3.2])	16
2.12	Block definition diagram of the aircraft domain to model the aircraft, its external users and the physical environment (based on [FRI08, fig. 3.3]) .	18
2.13	Block definition diagram of the aircraft block composition, modelling the the aircraft and its components (based on [FRI08, fig. 3.10])	19
2.14	Modelling of the association between the <i>Aircraft</i> block and the <i>Physical Environment</i> block	21
2.15	Exemplary internal structure of the <i>Aircraft</i> block, modelled in an internal block diagram	22
2.16	Modelling the compositions of the equation $E = m \cdot c^2$ in a block definition diagram	24
2.17	Constraining the upper and lower limits of a value property in SysML . .	25
2.18	Representation of the parametric equation $E = m \cdot c^2$ with SysML (based on [ALT12, fig. 4.16])	26
2.19	Representation of the parametric equation $E = m \cdot c^2$ with Simulink (taken from [ALT12, fig. 4.17])	27
2.20	Representation of the parametric equation $E = m \cdot c^2$ with SysML	27
2.21	Exemplary package structure of the <i>Aircraft</i> model	28
2.22	Package diagram showing how the <i>Aircraft</i> model is organized into packages (based on [FRI08, fig. 3.19])	29

3.1	Typical mission profiles for three different aircraft: (a) transport aircraft, (b) fighter, and (c) reconnaissance (taken from [SAD12, fig. 4.2])	32
3.2	Relationship among the four major design activities (taken from [SAD12, fig. 2.2])	33
3.3	Formal design reviews (taken from [SAD12, fig. 2.8])	34
3.4	Schematic Design Sequence including preliminary sizing, conceptual design and iteration loops (taken from [SCH19, fig. 2.1])	36
3.5	Matching chart	42
3.6	Two main groups of design activities in aircraft design (taken from [SAD12, fig. 1.2])	47
3.7	Evaluation of three presumptive configuration alternatives (taken from [SAD12, tab. 3.10])	48
3.8	Exemplary MDO with three variable design parameters	49
3.9	Systems engineering and aerospace engineering influence on the design (taken from [SAD12, Fig. 2.3])	51
4.1	PTC Integrity Modeler user interface	54
4.2	Different tabs within the browser pane	54
4.3	How Visual Basic programs interact with the PTC Integrity Modeler (taken from [PTC15])	56
4.4	Accessing the description of a requirement via the PTC Integrity Modeler Automation Interface	58
5.1	Problems arising with different software landscapes during system definition and system analysis	62
5.2	Exchange of parameters between both models	63
5.3	Exchange of parameters between both models (more detailed)	65
5.4	Selecting the SysML profile in the PTC Integrity Modeler	66
5.5	Aircraft model package structure	67
5.6	Section from the requirement diagram (operational requirements)	68
5.7	The <i>Aircraft Domain</i> block definition diagram	69
5.8	The internal block diagram of the <i>Airport</i> block	70
5.9	Section of the internal block diagram of the <i>Aircraft</i> block	71
5.10	Section of the block definition diagram of the <i>Aircraft</i> block	72
5.11	The internal block diagram of the <i>Engine</i> block	73
5.12	The internal block diagram of the <i>FuelTank</i> block	73
5.13	The internal block diagram of the <i>Wing</i> block	74
5.14	The internal block diagram of the <i>MassPrediction</i> block	75
5.15	Modelling of the constraints within an internal block diagram	76
5.16	Used data types for the <i>Aircraft</i> model, modelled with a block definition diagram	76
5.17	Custom stereotypes to label model items as input or output	77
5.18	Section of the parametric diagram, <i>Constraint Property</i> for the landing design constraint	77

5.19	Section of the parametric diagram, <i>Constraint Property</i> for the matching chart	78
5.20	Section of the parametric diagram, value properties and constraints of the variable parameters	79
5.21	Output of the MATLAB program	81
5.22	Structure of the MATLAB code	82
5.23	Example of an infeasible iteration point (first iteration, coloured in red)	87
5.24	Problem with multi-user support for parallel editing and analysis of the model	89
5.25	XMI metadata interchange	90
5.26	Utilization of the database for parameter exchange	91
5.27	Parameter exchange with PTC Integrity Modeler SySim, Visual Basic and Simulink	92
5.28	Example of a system simulation performed with SySim and Visual Basic (taken from [Inc19, fig. 4.19])	92
5.29	XMI metadata interchange	93
5.30	Integration of the Phoenix Model Center with different analysis tools (taken from [Int22])	94
5.31	Parameter transfer between system definition software (PTC Integrity Modeler) and system analysis software (MATLAB) through a Visual Basic program	95
5.32	Structure of the Visual Basic Program	96
5.33	Solution to save and access value parameters within the requirement description	103
5.34	After execution of the script, the calculated values are displayed at the Modeler result pane	108
6.1	How to import the <i>Aircraft</i> model	111
6.2	How to add an executable to the tools ribbon	112
6.3	The executable file is pinned to the the tools ribbon	112
6.4	Matching chart of the first preliminary design	114
6.5	Matching chart of the second preliminary design	115
9.1	Status of various design features during the design process (taken from [SAD12, fig. 1.7])	120

List of Symbols

Latin Symbols

Symbol	Meaning	Unit
A	Wing aspect ratio	[]
$a(h_{Cr})$	Speed of sound at the cruise attitude	$[m \cdot s^{-1}]$
B_{Cr}	Breguet range factor during cruise	$[m]$
B_{Cr}	Breguet range factor during loiter	$[m]$
C_D	Drag coefficient	[]
$C_{D,0}$	Zero-lift drag coefficient in the clean condition	[]
C_L	Lift coefficient	[]
$C_{L,Cr}$	Cruise lift coefficient	[]
$C_{L,max,L}$	Maximum lift coefficient during landing	[]
$C_{L,max,TO}$	Maximum lift coefficient during take-off	[]
DI	Design index	[]
e	Oswald factor	[]
e_{Cr}	Oswald Factor during cruise	[]
E_{2ndS}	Glide ratio during the 2 nd segment	[]
E_{Cr}	Glide ratio during cruise	[]
E_{MA}	Glide ratio during missed approach	[]
f_{OBJ}	Objective function	[]
fun	Function to minimize (MATLAB fmincon)	[]
g	gravitational constant	$[m \cdot s^{-2}]$
h_{Cr}	Cruise attitude	$[m]$
k_e	Auxiliary factor	[]
lb	Lower bound (MATLAB fmincon)	[]
m_{MTO}/S_W	Wing loading at take-off	$[kg \cdot m^{-3}]$
m_{CARGO}	Cargo mass	$[kg]$
M_{Cr}	Cruise Mach number	[]
m_F	Fuel mass	$[kg]$
$M_{ff,Alternate}$	Mission fuel fraction alternate	[]
$M_{ff,CLB}$	Mission fuel fraction climb	[]
$M_{ff,Cr}$	Mission fuel fraction cruise	[]
$M_{ff,DES}$	Mission fuel fraction descend	[]
$M_{ff,L}$	Mission fuel fraction landing	[]
$M_{ff,Loiter}$	Mission fuel fraction Loiter	[]
$M_{ff,TO}$	Mission fuel fraction take-off	[]

$m_{fuel,res}$	Reserve fuel weight	[kg]
m_{ML}	Maximum landing weight	[kg]
m_{MTO}	Maximum take-off weight	[kg]
m_{MZF}	Maximum zero fuel weight	[kg]
m_{OE}	Operational empty weight	[kg]
m_{PAX}	Passenger weight	[kg]
m_{PL}	Payload	[kg]
n_{PAX}	Amount of passengers	[]
n_E	Amount of engines	[]
$p(h_{Cr})$	Air pressure at cruise attitude	[Pa]
R	Design range	[NM]
R_{Alt}	Alternate range	[NM]
SFC_{Cr}	Specific fuel consumption during cruise	[kg·N ⁻¹ ·s ⁻¹]
SFC_T	Specific fuel consumption during loiter	[kg·N ⁻¹ ·s ⁻¹]
s_{LFL}	Landing field length	[m]
s_{TOFL}	Take-off field length	[m]
S_{Wet}/S_W	Relative wetted area	[]
S_W	Wing reference area	[m ²]
t_{loiter}	Loiter time	[s]
$T_{TO}/(m_{MTO} \cdot g)$	Thrust to weight ratio during take-off	[]
T_{TO}	Take-off thrust	[N]
ub	Upper bound (MATLAB fmincon)	[]
V_{Cr}/V_{md}	Ratio between the cruise speed and the minimum drag speed	[]
V_{Cr}	Cruise speed	[m · s ⁻¹]
V_{md}	Minimum drag speed	[m · s ⁻¹]
x_0	Initial point (MATLAB fmincon)	[]

Greek Symbols

Symbol	Meaning	Unit
$\Delta C_{D,f}$	Increments in profile drag coefficient associated with trailing-edge flap deflection	[]
$\Delta C_{D,g}$	Additional drag coefficient by the landing gear	[]
$\Delta C_{D,s}$	Increments in profile drag coefficient associated with leading-edge flap deflection	[]
γ	Heat capacity ratio	[]
γ_{CLB}	Steady gradient of climb during the 2 nd segment	[]
γ_{MA}	Steady gradient of climb during missed approach	[]
μ	Engine by-pass ratio	[]
σ	Ratio of the air stream density at a chosen reference station relative to sea level standard atmospheric conditions	[]

List of Abbreviations

Abbreviation	Meaning
CDR	Conceptual Design Review
CS	Certification Specification
FAR	Federal Aviation Regulation
ibd	Internal Block Diagram
INCOSE	International Council on Systems Engineering
MBSE	Model-Based Systems Engineering
MDO	Multidisciplinary Design Optimization
MOE	Measures of Effectiveness
MIWG	Model Interchange Working Group
OMG	Object Management Group
par	Parametric Diagram
PDR	Preliminary Design Review
pkg	Package Diagram
req	Requirement Diagram
SE	Systems Engineering
SysML	System Modeling Language
TLAR	Top Level Aircraft Requirements
UML	Unified Modeling Language
XMI	XML Metadata Interchange
XML	Extensible Markup Language

1 Introduction

1.1 Motivation

The Wright Brothers would certainly be surprised to see the evolution that aircraft design has undergone since they designed their Wright Flyer. Today's passenger aircraft are extremely technically advanced, complex, and interconnected systems. Consequently, the design process has changed dramatically. Nowadays, thousands of engineering specialists are involved in aircraft development over decades and across continents. The engineers belong to many different design domains, transforming the knowledge that has accumulated over the last century into a finished aircraft. With so many technically highly specialized design groups, it can be challenging to maintain a common understanding of the final product (see fig. 1.1).

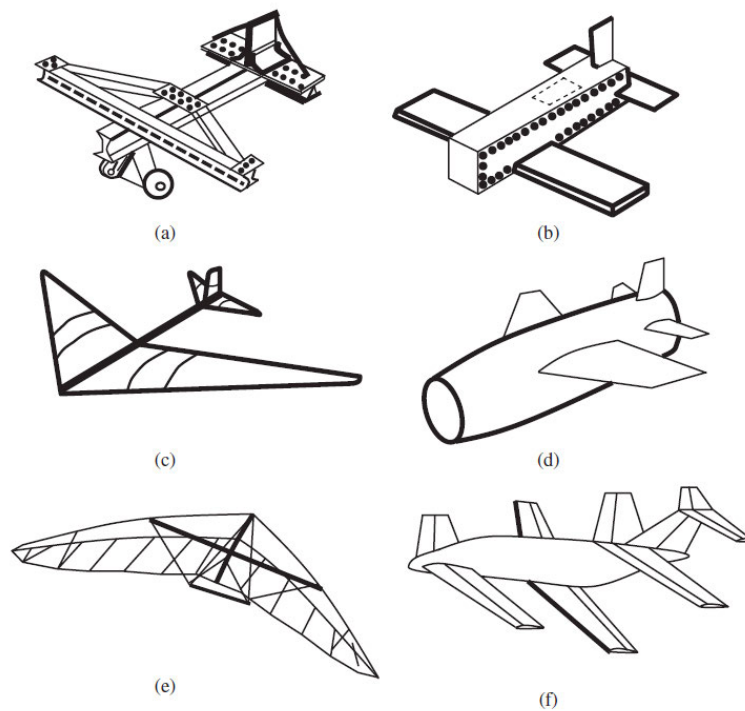


Figure 1.1: Design groups' unique visions and interests (taken from [SAD12, fig. 2.15])

It might then be helpful to take a step back and look at the aircraft as a whole again. This can be done, for example, by creating a digital model of the aircraft, into which the individual design group's visions are then integrated. This approach is called Model-Based Systems Engineering (MBSE).

Besides a common understanding, a digital model also offers other advantages. For instance, the requirements for the system or certain test cases can be easily assigned to the individual system components. This ensures that the aircraft is developed in accordance with the stakeholder requirements. In addition, a digital model that covers the entire system facilitates the modelling of interfaces between different design domains. Moreover, a digital model allows easy variation possibilities. This can be useful to customize the aircraft to the individual customer requirements, without major additional costs.

These and other advantages have made MBSE an essential element of aircraft design. At the same time, aircraft are largely designed with the help of mathematical models. Calculation software, such as MATLAB, offer special optimization functions that can be used for multidisciplinary design optimizations (MDO). They enable to find the combination of several variable parameters that leads to an optimal design. In this way, many different aircraft designs can be computed in a short time, which can then be compared with each other. Also, the computational models are required for the evaluation of the requirements. This assures a continuous design improvement.

However, in an MBSE approach, the input values required for these calculation programs are often stored inside the digital system definition model. In addition, the output parameters of these calculations also need to be stored in the system definition model. A manual transfer of the parameters between the two models takes a lot of time. This can reduce the incentive to evaluate many different designs, resulting in a poorer overall design. Furthermore, data transfer errors or inconsistencies can occur in the digital model.

A digital interface between the system definition model and the system analysis models could provide a solution for that. Especially during the aircraft preliminary design stage, an automatic parameter transfer can be advantageous. That is because design changes, iterations and the calculation of different designs are particularly common at this stage. At the same time, MBSE plays an important role at the beginning of the design process.

1.2 Objectives

In this thesis, the digital linking of system definition software and system analysis software will be investigated. This is carried out using the example of the aircraft preliminary sizing process.

First, a digital model of an aircraft will be developed. For this purpose, the popular Systems Modeling Language (SysML) will be used. The aircraft model will be mod-

elled with the the PTC Integrity Modeler. It will contain the Top Level Requirements (TLAR) and the mission definition required for preliminary sizing. Furthermore, the basic architecture of the aircraft will be modelled.

Additionally, a computational model will be developed in MATLAB that can be used for preliminary sizing of passenger aircraft. Subsequently, methods for digitally linking the two models will be explored. The digital connection will then be constructed and tested in practice.

The digital connection of the two models could form the basis for an end-to-end digital development process. In particular, multidisciplinary system optimization in the aircraft design process could be better integrated into an MBSE approach.

1.3 Literature Review

The fundamentals of MBSE and the SysML were primarily taken from [FRI08] and [HOL13]. The mathematical relationships for aircraft preliminary sizing are based on the method presented in [LOF80]. The application of MBSE for aircraft preliminary sizing was mainly obtained from [SAD12]. Information about the PTC Integrity Modeler was taken from [PTC19a], [PTC19b] and [Inc19]. The establishment of the digital interface was mainly performed using the methods presented in [PTC15].

1.4 Structure

This thesis is divided into a theory part, a modelling and programming part and the practical testing of the digital interface. This is followed by a discussion of the chosen solution approach, a summary and the conclusions and recommendations.

The theory part first covers the fundamentals of model-based systems engineering in chapter 2. Then, the fundamentals of aircraft preliminary sizing are discussed in chapter 3. In particular, the method for preliminary sizing of jet aircraft from [LOF80] is discussed. Chapter 4 then discusses the software used to model the system definition, the PTC Integrity Modeler.

Chapter 5 is the main part of this thesis. First, the modelling of the aircraft model in SysML is described. Then the calculation program for aircraft preliminary sizing, which is programmed in MATLAB, is presented. After discussing different approaches for a digital connection, the connection between both models is presented.

In chapter 6, the selected solution is then tested in practice. The following chapter 7 is devoted to a discussion of the selected solution. The thesis then concludes with a summary (ch. 8) as well as conclusions and recommendations (ch. 9).

2 Fundamentals of Model-Based Systems Engineering

2.1 Systems Engineering

2.1.1 Motivation for Systems Engineering

The past decades have led to many technological innovations. For example, electronics has become an increasingly important part of modern systems. This also applies to modern aircraft. Newly developed aircraft are far more computerized today than they were 50 years ago (see fig. 2.1).

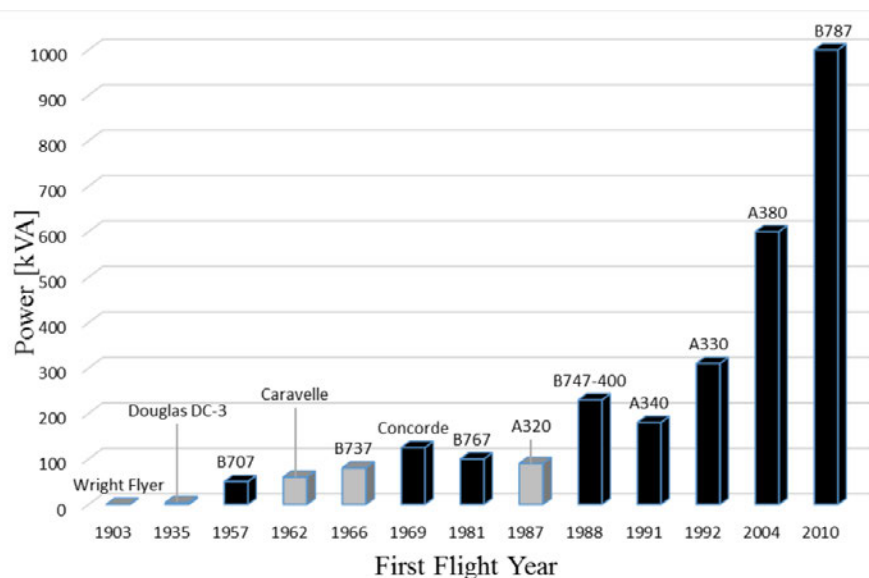


Figure 2.1: Evolution of electrical power need (gray: short- to medium-range aircraft and black: medium- to long-range aircraft, taken from [VIN18, fig.1])

Additionally, the interconnectivity of systems has increased. Today, most systems can no longer be treated as stand-alone, but behave as part of a larger whole [FRI08, p. 3]. It often proves useful to assign a technical resource for different purposes. For instance, a modern aircraft engine does not only generate thrust, but is also required for electric power generation and cabin ventilation. Both the introduction of new technologies as

well as the increasing interconnectivity has led to an increase of system complexity.

Apart from new technologies, new perspectives can lead to an increase of complexity as well. Lately, environmental concerns and resource scarcity made it necessary to consider a product during all phases of its life cycle. This so called life-cycle engineering can lead to additional requirements during the product design, which in turn can increase the complexity of the design process.

Additionally, changes in the way of working have led to new challenges for engineering design teams. Today, the design teams usually consist of people from different companies that work from different locations. All in all, the way of working is much more interdisciplinary, which can lead to communication problems [HOL13, p.82].

Finally, even the most modern technical systems cannot compete if they are not oriented to the stakeholders requirements. Today, products are designed to fit the requirements of many stakeholders through minor modifications. This is made possible by effective variant management, which also saves development time and costs.

In summary, some of the main challenges in engineering are the increase of complexity, communication barriers and new customer requirements such as product variance and life-cycle engineering. In particular, the increase in complexity has two key effects. First, it is more likely that the complexity will be underestimated. That can lead to exploding development time and costs or to a faulty design [HOL13, p.80]. Additionally, the complicated integrated behaviour can lead to more "opaque" errors [BOE11].

Originally developed to help understand and manage complexity, the systems engineering (SE) discipline aims at solving the problems named above. The following chapter will explain the main principles of systems engineering.

2.1.2 Systems Engineering Definition

The International Council on Systems Engineering (INCOSE) defines systems engineering as:

‘An interdisciplinary approach and means to enable the realization of successful systems.’ [INC04, p.12]

Another definition is given by [EIS08, p.5]:

‘Systems engineering is an iterative process of top-down synthesis, development and operation of a real-world system that satisfies, in a near optimal manner, the full range of requirements for the system.’ [EIS08, p.5]

All in all, the application of systems engineering looks at a system as a whole with all its different disciplines. Systems engineering considers the whole life cycle of a system, as well as the needs of every stakeholder in a multidisciplinary fashion. The main goal of systems engineering is to design, produce and manage complex systems over their life cycles [PTC19b, p.1].

2.1.3 The Systems Engineering Process

A systems engineering approach usually follows a development cycle similar to fig. 2.2. The so called V-model was first introduced in 1995 in software development to describe a formal linking of tasks in interdisciplinary product development [VDI21]. The product development usually starts with the decomposition of stakeholder requirements. Stakeholders are individuals, teams, organizations or classes that have an interest in a system [ISO11, p. 2]. The stakeholder requirements are then converted into system requirements. After that, the system architecture is defined. The definition starts at the general system level and will be refined down to the system element level over time. At the same time, the system requirements are refined down to the system element level. After that, the implementation of the system elements can take place (the lower tip of the V). The system elements are then integrated to the overall system. The system is then validated, starting at the system element level and finishing at the whole system level.

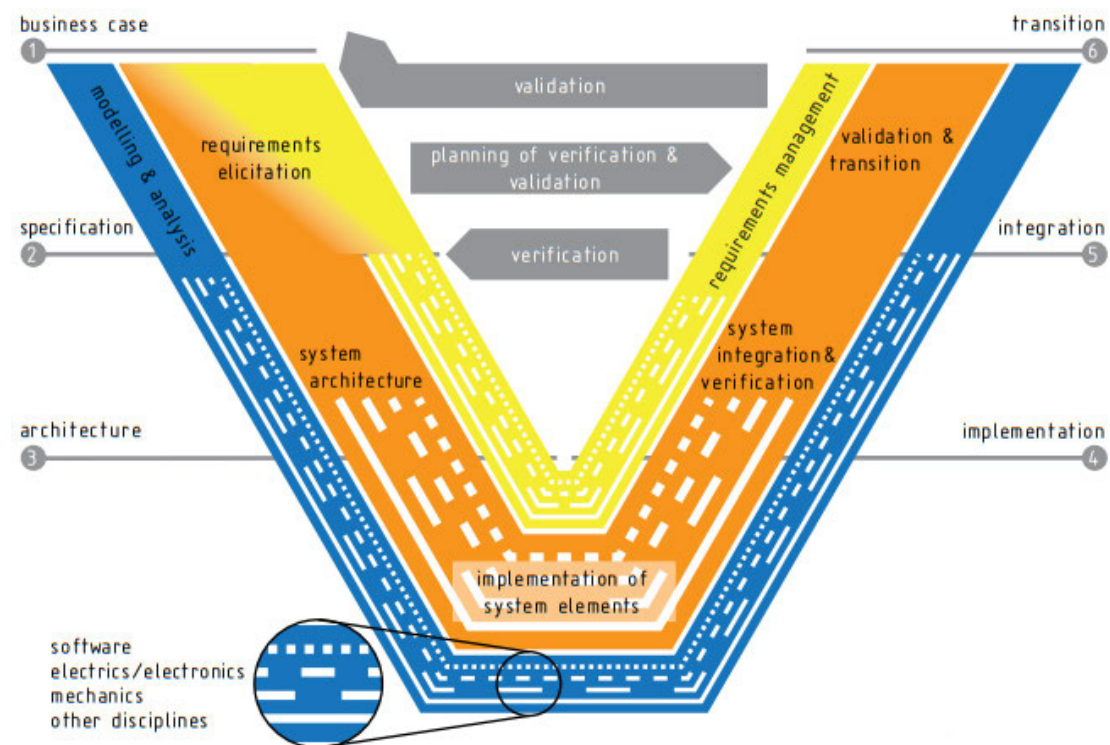


Figure 2.2: V model (taken from [VDI21])

Fig. 2.2 illustrates two important concepts of systems engineering, that are of particular relevance to this thesis. One basic concept of systems engineering is that reference is made to the requirements throughout the whole development cycle. This can lead to an iterative approach if the requirements prove to be infeasible at a later design stage (grey arrows in fig. 2.2). Furthermore, the development is accompanied by modelling

and analysis during the whole design period (blue outer "V" in fig 2.2).

2.2 Model-Based Systems Engineering

2.2.1 Document-Based Systems Engineering

The previous chapter has emphasized the importance of considering the system as a whole. However, this is not always very easy. In the traditional, document based, approach the system requirements are usually spread across different documents, drawings and spreadsheets. This makes it difficult to infer from the final system architecture and system validation back to the original system requirements [ALT12, p. 2]. That however results in inefficiencies and potential quality issues. The quality issues often show up during integration and testing, or worse, after the system is delivered to the customer [FRI08, p.16].

Another disadvantage of document-based systems engineering is that documents rely on the underlying written language. However, written language is inherently flexible in interpretation, which can lead to misunderstandings [BAR16] .

A document-based approach also makes it difficult to maintain or reuse the system requirements and design information for an evolving or variant system design [FRI08, p.16]. An alternative approach is model-based systems engineering (MBSE), which is presented in the next chapter.

2.2.2 Model-Based Systems Engineering

Unlike the document-based approach, in model-based systems engineering a digital model is the primary means of information sharing between engineers [PTC19b, p.1]. This means, that the information that was previously captured in documents is now captured in a so called model repository, which is a collection of all model items. MBSE is intended to facilitate systems engineering activities that have traditionally been performed using the document-based approach and result in enhanced communications, specification and design precision, system design integration and reuse of system artefacts [FRI08, p.17]. One of the most widely accepted definitions for MBSE is:

‘Model-based systems engineering (MBSE) is the formalized application of modeling to support system requirements, design, analysis, verification and validation activities beginning in the conceptual design phase and continuing throughout development and later life cycle phases. ’ [INC07, p.15]

It should be mentioned, however, that both the document-based and model-based approaches represent two extremes. In practice, there is usually a mixture of both approaches, with a move towards more model-based systems engineering. Furthermore, some tools used for a MBSE offer the possibility to automatically create documents

from the models. These documents can then be further used to report the information [ALT12, p. 69]. This thesis is only concerned with MBSE. Therefore, some basic concepts of MBSE will be discussed in the next chapters.

The System Model

In model-based systems engineering, a system model serves as the basis for the entire systems engineering process. According to [ALT12, p. 20], a model is an abstract description of the reality. Abstraction means discarding details and transferring the reality to something more general or simple. The model's level of detail is chosen, so that the model achieves a desired result.

In addition, different abstractions of the same object of observation lead to different views. For example, fig. 2.3 shows two different views of a house with two different levels of abstraction. The concept of views makes it possible for different system participants to see only the system elements and level of abstraction that are relevant to them.

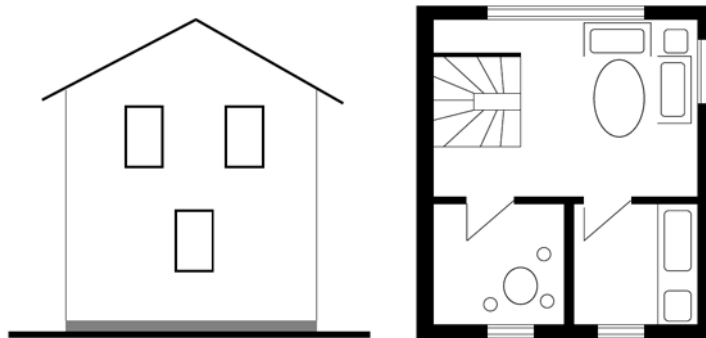


Figure 2.3: Architecture as an example of a model (taken from [ALT12, fig. 3.1])

A model always includes a realisation. The realisation can be created with the help of the information that is contained in the model. For example, a house could be built with the help of a model similar to fig. 2.3 [ALT12, p. 21]).

In MBSE, a system model is the coherent output of all systems engineering activities. It integrates system requirements, design, analysis and verification models to address multiple aspects of the system in a cohesive manner, rather than a collection of individual models [FRI08, p.20]. The emphasis is placed on evolving and refining the model using model-based methods and tools [FRI08, p.17]. A primary use of the system model is to design a system that satisfies system requirements and allocates the requirements to the systems components [FRI08, p. 17]. If supported by an execution environment, the model may also be simulated. This ensures verification of the design at an early stage (see fig. 2.2).

When modelling a system, it must be ensured that the model meets the model's purpose.

The systems engineer must be aware of the required model breadth (which parts of a system should be modelled) and the required model depth (the level of design hierarchy of the system). Furthermore, a good model must be consistent and have the possibility to integrate with other models [FRI08, p.23].

Usually, system models are created with special modelling software (see ch. 4). Different modelling software can be based on different modelling languages. In order to understand a model, the modelling language on which the model is based must be understood. That is, the syntax (notation) and the semantics (meaning) of the chosen modelling language must be known. Otherwise the model remains incomprehensible to the user [ALT12, p. 20]. A frequently used modelling language, the System Modeling Language (SysML) will be presented in ch. 2.3.

Motivation for Model-Based Systems Engineering Approach

Now that the basic concepts of MBSE are introduced, the motivation for an MBSE approach are identified. As already mentioned in ch. 2.1, a MBSE approach considers a system as a whole. This leads to a shared understanding of the system across the development team and other stakeholders. Together with the ability to integrate views of the system from multiple perspectives, the overall communication is enhanced [FRI08, p.20].

When the underlying modelling language is clearly defined, there is less room for misconceptions than in a written document. Therefore, a MBSE approach leads to more complete, unambiguous and verifiable requirements [FRI08, p.20]. Since all the system information is stored in one model, the requirements can be easily traced to other system elements. This improves the overall quality of the design. A MBSE approach also improves the ongoing requirement validation and design verification (see fig. 2.2). That can improve the design and reduce the overall development risk [FRI08, p.20].

Another advantage of MBSE is the increase of productivity. The traceability of requirements enable a faster impact analysis of requirements and design changes [FRI08, p.20]. Furthermore, MBSE allows simultaneous engineering or rapid design iterations [IWA15]. Additionally, a possible reuse of models and the automated document generation can improve the productivity. All in all, a MBSE approach can improve the design quality while reducing costs and development time.

2.3 The SysML

The previous chapters have demonstrated the necessity of MBSE. However, there is a wide variety of different modelling approaches [BOE11]. The choice of the modelling approach has an influence on the success of the model. A widely used modelling language is the Systems Modeling Language (SysML), which is used for modelling in this thesis

(see ch. 5.3). For this, the fundamentals of the SysML are explained in the following chapters.

2.3.1 The SysML Origins

The SysML is a general-purpose graphical system modelling language. In particular, the language provides graphical representations with a semantic foundation for modelling system requirements, behaviour, structure and parametric relationships, which is used to integrate with other engineering analysis models [Gro12]. Most importantly, the language provides a means to capture the system modelling information as part of an MBSE approach without imposing a specific method on how this is performed [FRI08, p.31].

The SysML originates from an initiative between the Object Management Group (OMG) and the International Council on Systems Engineering (INCOSE) in 2003 and was meant to adapt the Unified Modeling Language (UML) for systems engineering applications [HOL13, p.83]. The UML is also a general-purpose graphical modelling language that has its origins in software engineering. Approximately half of the UML language was reused for the SysML [FRI08, p.64]. Nevertheless, some concepts like the parametric diagram or the requirement diagram are only a part of the SysML and are not contained in the UML.

2.3.2 The SysML Syntax

Like all programming languages, the SysML is defined by a certain syntax. The SysML syntax consists of the concrete syntax and the abstract syntax. The concrete syntax is the uniform notation of the language. A uniform notation is the basis to prevent misunderstandings, since all stakeholders have the same understanding of the system [ALT12, p. 30].

The abstract syntax is called meta model and defines the item types (such as classes, attributes and operations) that exist in the model and their properties [PTC15, p. 2]. Additionally, the meta model defines how the modelling elements relate to one another [HOL13, p. 127]. This is achieved through metaclasses that are related to each other using relationships such as generalizations and associations [FRI08, p. 66].

Since the SysML is a graphical modelling language, each element of the abstract syntax has its own graphical representation. To fully comply with the SysML, a SysML based modelling software must implement both the concrete syntax and abstract syntax [Gro12, p.13].

2.3.3 The SysML Profile

An UML profile is the mechanism used to customize the UML language. By adding additional profiles to the UML, the metaclasses from existing meta models can be extended

or modified to adapt for different purposes [Gro12, p.141]. The systems engineering extensions to the UML in SysML are defined by a profile called the *SysML Profile* [FRI08, p. 67]. That means that the SysML meta model is built in the UML meta model.

2.3.4 Stereotyping

Another way to configure the UML to one's specific needs are stereotypes. Stereotypes are a way to extend the UML by providing elements from the meta model by additional meaning [ALT12, p. 26]. Possible applications of this function would be, for example, the assignment of the supplier to certain modelled parts or to differentiate between software and hardware model elements. Fig 2.4 shows how an arbitrary model element (upper box), which belongs to an arbitrary meta class, is assigned a stereotype (bottom box). This way, for example, a model element "CPU" could be assigned a stereotype called "Hardware".

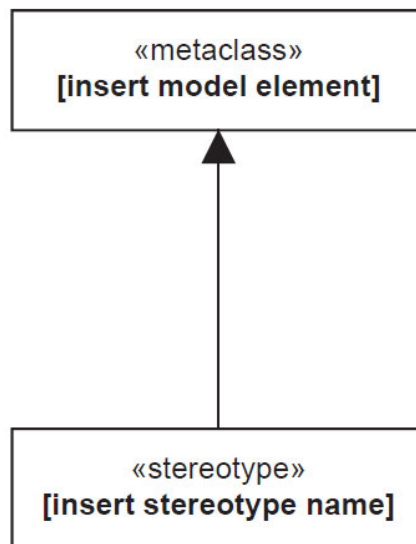


Figure 2.4: Stereotyping model elements in SysML (taken from [HOL13, fig. 5.5])

2.3.5 Modelling

When modelling a system using the SysML, the concrete syntax is mapped to the abstract syntax. That means that model elements become instances of the respective metaclass defined in the meta model. The properties of the metaclass are then inherited by the model elements.

Fig. 2.5 shows the relationship between different model items (in the middle) and the SysML meta model. The model item *Pilot* is an instance of the SysML metaclass *Actor*. Thus, the pilot inherits the properties defined in the metaclass *Actor*. The blocks *Wing*

and *Airplane* are both instances of the SysML stereotype *Block*. As a result, the pilot element has different properties than the *Wing* and the *Airplane* block.

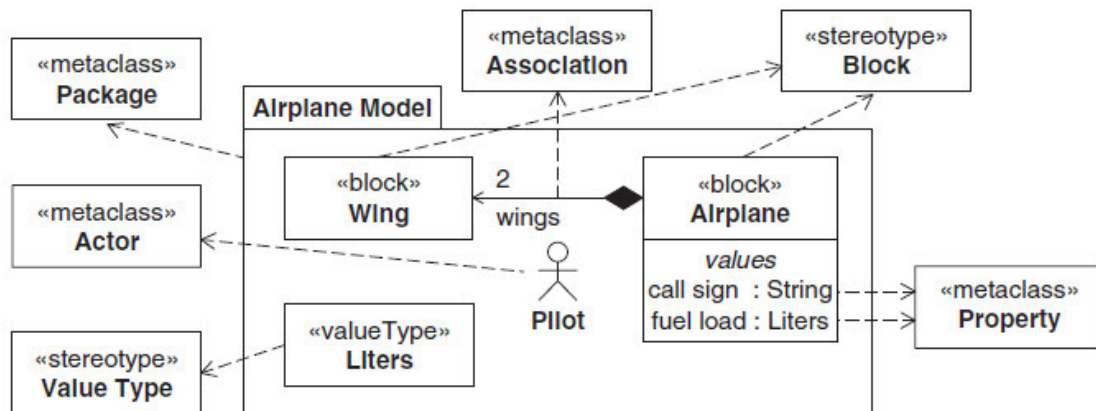


Figure 2.5: Relationship of metaclasses to model elements (taken from [FRI08, fig. 4.5])

A SysML modeling tool that complies with the SysML specification enforces the meta class properties, constraints, and relationships on the information entered or retrieved from the model [FRI08, p.68].

2.4 SysML Diagrams

2.4.1 Overview

As mentioned earlier, the SysML is a graphical modelling language. That implies, that model items are created, modified or deleted by editing diagrams. The items displayed on a diagram can therefore be seen as symbols representing underlying model items. The SysML offers a total of nine different diagrams for different applications of MBSE (see fig. 2.6) [ALT12, p. 40]. In the next chapters, however, only the diagrams that are important for this thesis will be presented (red borders in fig. 2.6).

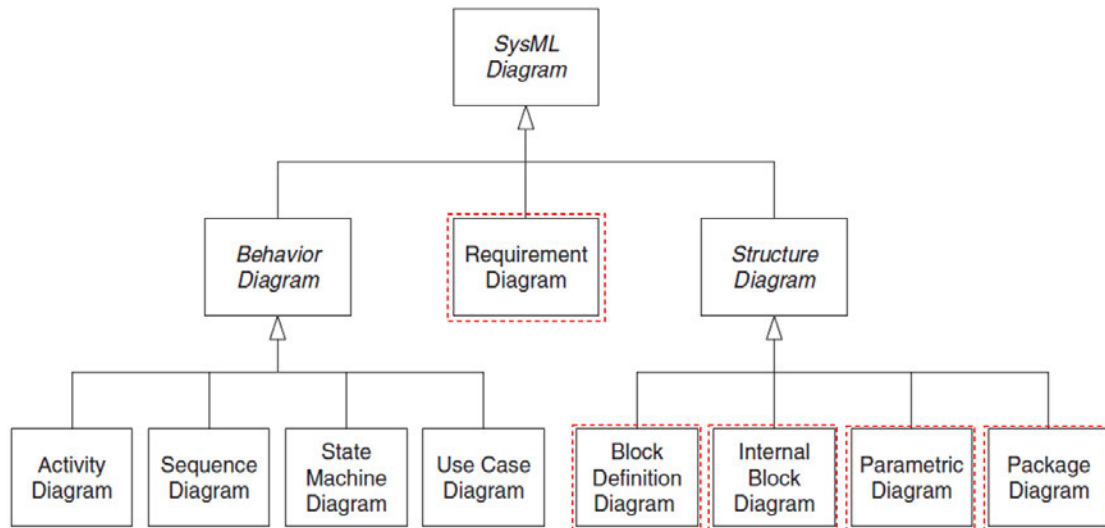


Figure 2.6: SysML diagram taxonomy (taken from [FRI08, fig. 3.1])

A Requirement diagram represents text-based requirements and their relationship with other requirements, design elements and test cases to support requirements traceability [FRI08, p. 30]. It is an extension by SysML and cannot be found in UML. The requirement diagram will be explained in ch. 2.4.3.

A block definition diagram represents structural elements called blocks, and their composition and classification [FRI08, p. 30]. It is a modification of the UML class diagram and will be introduced in ch. 2.4.4.

The internal block diagram is used to model the internal structure of a block. It represents properties, interconnections and interfaces between the parts of a block [FRI08, p. 30]. The internal block diagram is a modification of UML composite structure diagram (see ch. 2.4.5).

The parametric diagram represents constraints on property values used to support engineering analysis [FRI08, p. 30]. It is not a part of the UML and will be explained in ch. 2.4.6.

The package diagram can be used to model the structure of so-called packages. Packages are comparable to file folders and are used to structure model elements. The package diagram will be explained in ch. 2.4.7.

2.4.2 The Key Diagram Elements

Each SysML diagram has a diagram frame consisting of a rectangle with a header. The header contains the standard information like the diagram type and the diagram name. It can also contain additional information like the specialized use or the type of the

model element that the diagram represents. Each diagram can also be extended by an optional note containing additional information (see fig. 2.7).

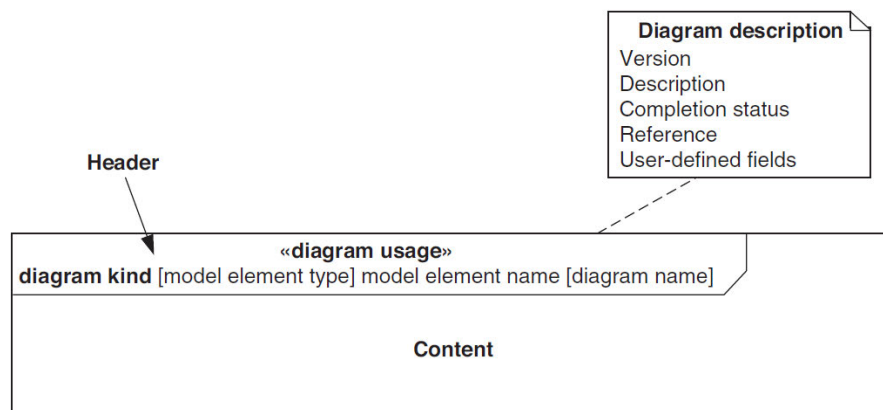


Figure 2.7: A diagram frame (taken from [FRI08, fig. 4.8])

The diagram content area consists of the graphical elements that represent underlying elements in the model. The graphical elements can be divided into three groups: node symbols, path symbols and icon symbols (see fig. 2.8, 2.9 and 2.10). A node is a symbol that can contain text and/or other symbols to represent the internal detail of the represented model element [FRI08, p. 73].

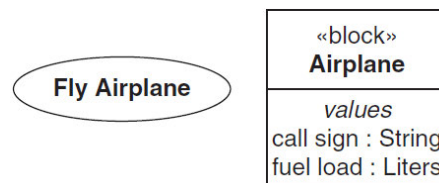


Figure 2.8: Examples of node symbols (taken from [FRI08, fig. 4.9])

Path symbols are lines that may have multiple additional adornments such as arrows and text strings [FRI08, p. 73]. They are normally used to create relationships between model elements.

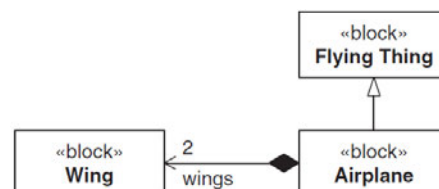


Figure 2.9: Examples of path symbols (taken from [FRI08, fig. 4.10])

Icons are typically used to represent low-level concepts that do not have further internal details [FRI08, p. 74]. If the model element represented by an icon has properties, they are displayed in a text string floating near the object (see fig. 2.10).

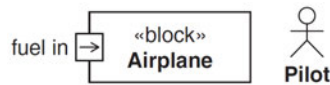


Figure 2.10: Examples of icon symbols (taken from [FRI08, fig. 4.11])

Now that the basic elements of a SysML diagram have been introduced, the next chapters will explain the different diagram types.

2.4.3 Requirements

As stated earlier, requirements form the starting point of every new system development or adjustment. Requirements usually consist of a description and a unique identification. As part of an MBSE approach, requirements can either be created and edited directly in the modelling tool or, alternatively, imported from a requirement management database.

There are different types of requirements. Functional requirements specify the behaviour of the system or the system components [ALT12, p. 10]. They can only be satisfied by blocks or block properties of type part [PTC19a]. Blocks and block properties of type part are primarily used to model a system's internal structure. They will be discussed in more detail in the next chapter.

Performance requirements allow quantitative statements and can be satisfied only by block properties of type value [PTC19a]. Block properties of type value are primarily used to assign value properties to system components (see ch. 2.4.5). A physical requirement specifies the physical characteristics and/or physical constraints of a system or a system part [Inc22]. They may be derived from performance requirements. Requirements can be modelled in so-called requirement diagrams, which are discussed in the following chapter.

The Requirement Diagram (req)

As stated earlier, with MBSE the requirements are managed directly as part of the digital model. A requirement diagram (req) is used to depict the requirements that are typically captured in a text specification [FRI08, p.34]. It is possible to group requirements depending on certain criteria or to model hierarchical relationships among them. In fig. 2.11, the requirement "REQ005" represents a rather vague top-level requirement. The top-level requirement can contain multiple lower-level requirements to further break down the top-level requirement (see REQ016). This can be modelled via a containment relationship (\oplus). The lower-level requirements can also be further specified in the same way. (see fig. 2.11).

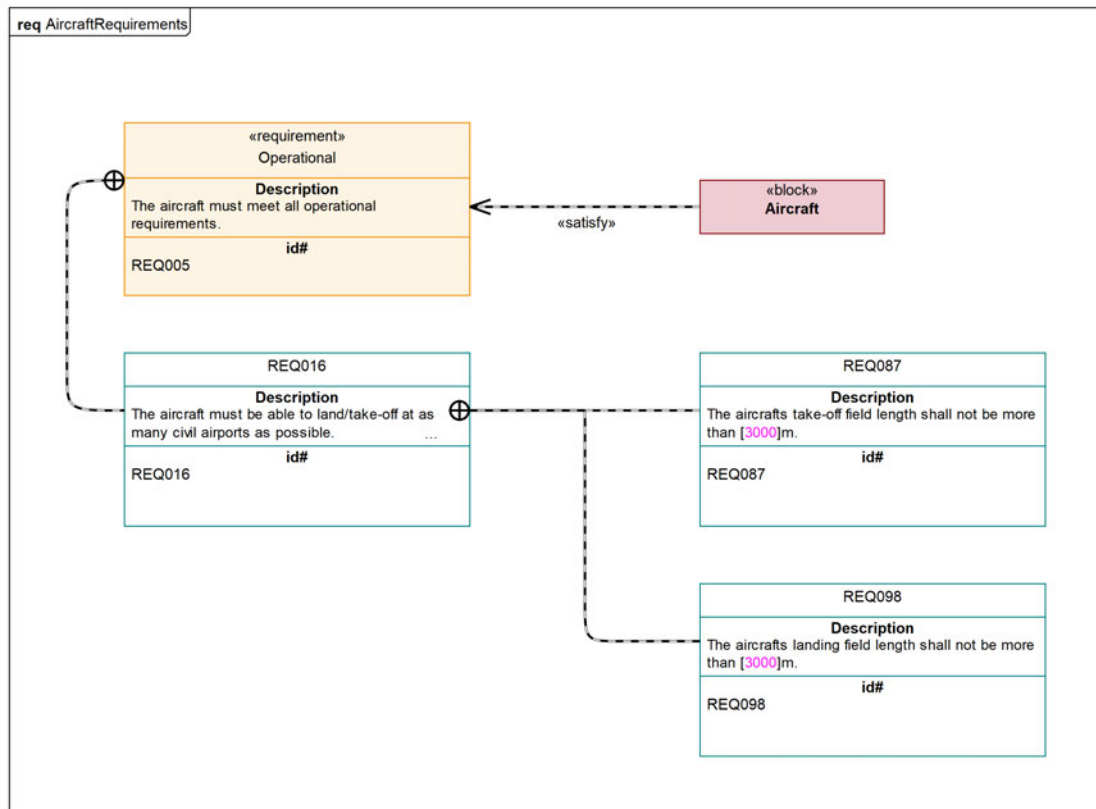


Figure 2.11: Exemplary requirement diagram to model the aircraft's requirements (based on [FRI08, fig. 3.2])

Furthermore, requirements can be linked to other requirements, design elements, and test cases using derive, satisfy, verify, refine, trace, and copy relationships [FRI08, p. 34]. That way, traceability of requirements throughout the overall hierarchical structure of the system is ensured. Fig. 2.11 shows that the aircraft satisfies the requirement *REQ005*. The requirement diagram can also be used to link requirements to the different stakeholders or test cases. That way, it can be ensured that the requirements address the stakeholders needs [FRI08, p.7].

Some modellers represent different types of requirements differently. For example, in fig. 2.11 performance requirements are displayed in blue and requirements of no particular type are displayed in orange.

2.4.4 Blocks

Blocks are one of the key model elements in SysML [PTC19b, p.13]. According to [FRI08], a block is defined as

‘ a very general modeling concept in SysML that is used to model a wide variety of entities that have structure such as systems, hardware, software, physical objects, and abstract entities. That is, a block can represent any real or abstract entity that can be conceptualized as a structural unit with one or more distinguishing features. ’

A block can be further specified by assigning properties, activities, operations or ports to it. That way the properties, behaviour and interfaces can be modelled as desired. Blocks can be modelled in a block definitions diagram, which is presented in the next chapter.

The Block Definition Diagram (bdd)

The block definition diagram (bdd) is used to define blocks in terms of their features and their structural relationships with other blocks [FRI08, p. 96]. The block definition diagram can be used to model external systems, users and other entities that a system may directly or indirectly interact with [FRI08, p. 34].

Fig. 2.12 shows the top level block, the *Aircraft Domain*. It can be seen that the aircraft domain is composed of the aircraft, the physical environment, the cargo and the aircraft occupants. This is represented through the black diamond symbol (◆). Furthermore, it is possible to model subclasses of different properties through the hollow triangle symbol (◁). Subclasses are specializations of classes from more generalized classes [FRI08, p. 37]. Fig. 2.12 shows that the pilot and the passenger are both subclasses of the *Aircraft Occupant*.

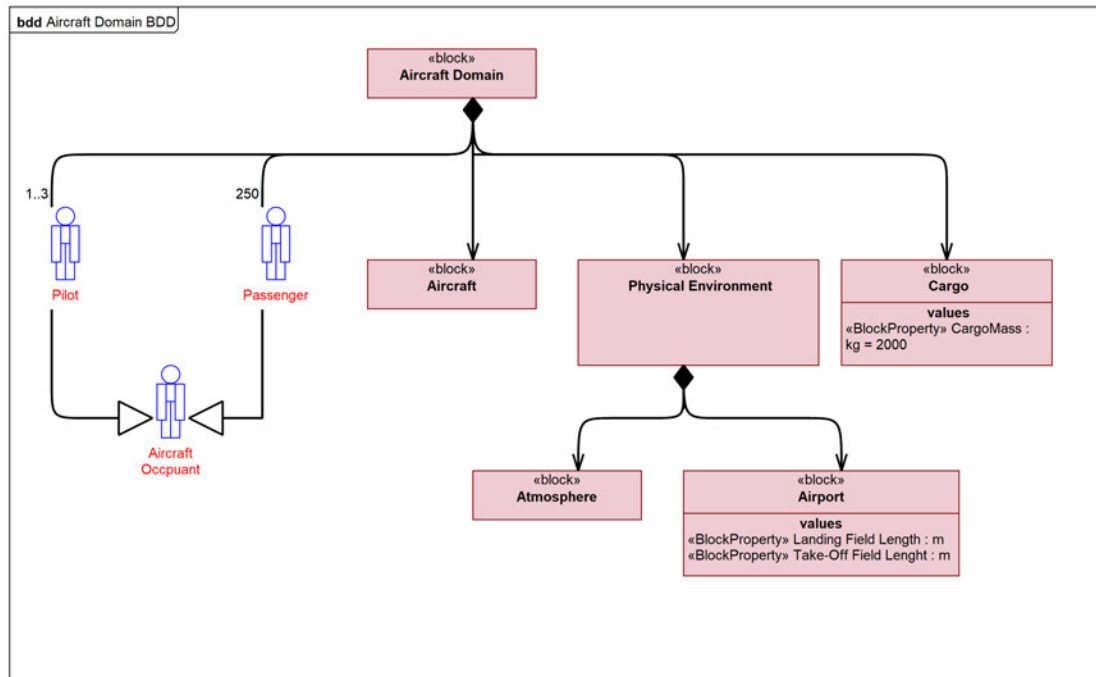


Figure 2.12: Block definition diagram of the aircraft domain to model the aircraft, its external users and the physical environment (based on [FRI08, fig. 3.3])

The block definition diagram can also be used to model the multiplicity of certain model elements. The multiplicity can be defined for the start or end role of a relationship. It can represent an undetermined maximum number of external entities (*), a single number (x) or a range (x..y) [FRI08, p. 37]. Fig. 2.12 shows that the aircraft domain contains 250 passengers and between one to three pilots.

A block definition diagram can also be used to decompose a system into its components. Fig. 2.13 shows the system hierarchy of the *Aircraft* block. It can be seen that the aircraft consists of two engines, at least one wing, one fuel tank and a mass prediction block. Another block definition diagram could then represent the system hierarchy of the engines. That way the system hierarchy can be depicted from the top-level domain block (e.g. the aircraft domain) down to aircraft components (e.g. a single bolt of the engine) [FRI08, p. 37].

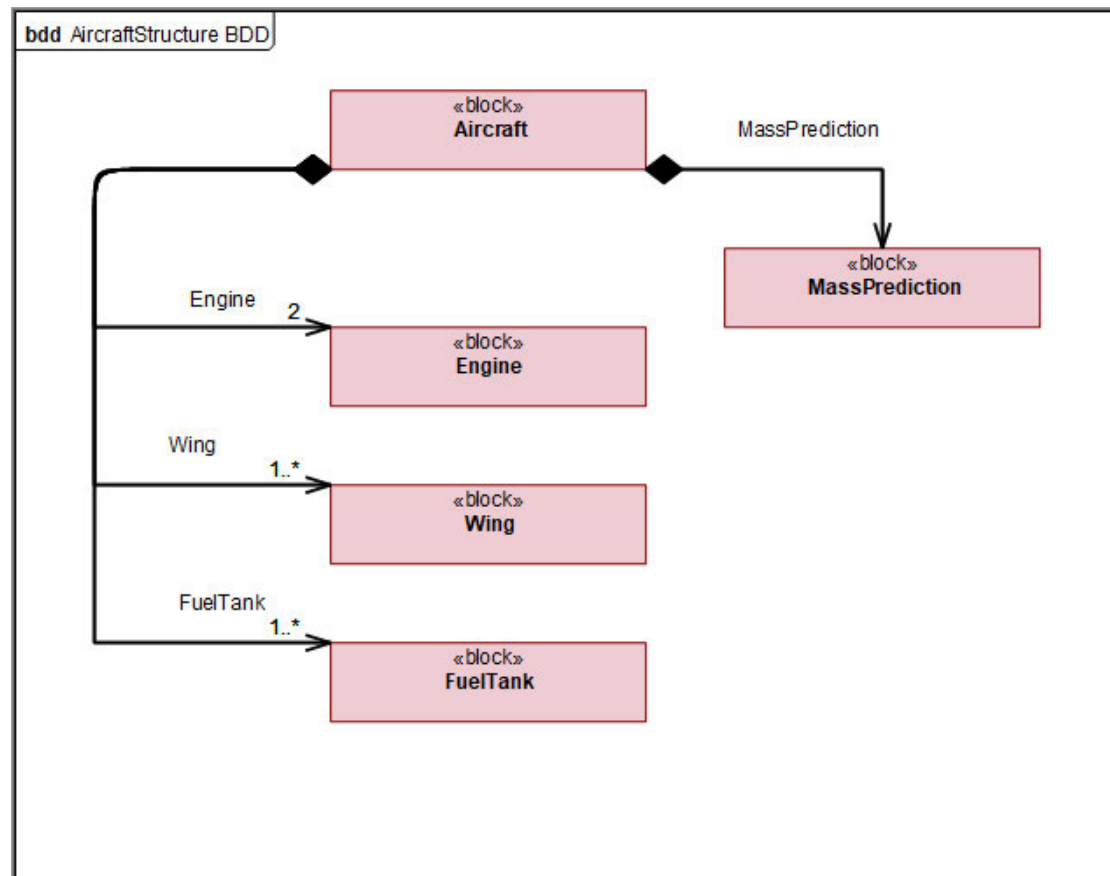


Figure 2.13: Block definition diagram of the aircraft block composition, modelling the the aircraft and its components (based on [FRI08, fig. 3.10])

2.4.5 Block Properties

Until now, blocks have been used to define the internal structure of a system. Now these blocks are to be filled with additional information. For this the SysML model element block property is used.

SysML classifies the block properties into three different types, which can be used for three different applications. Block properties can be either of type value, part and reference property. The individual block property types will be explained in the following.

Part Properties

SysML part properties are the SysML equivalent to the UML part type. An UML part is a property which is contained by a class using composition. That means that all part properties are destroyed when the containing class instance is destroyed [uml20].

Part properties describe the decomposition hierarchy of a block and provide a critical mechanism to define a part in the context of its whole [FRI08, p. 95]. They are always owned by a block. Therefore, they are properties that are intrinsic to the block but which may have their own identity. A composition relationship (◆) creates part properties between the owning block and the blocks that it is composed of [HOL13, p. 94]. This means that every time a composed relationship is modelled in a block definition diagram, a part property is created for the parent block as well. For example, if it is modelled in the block definition diagram that the *Aircraft* block is composed of the *Engine* block (see fig. 2.13), the *Aircraft* block also receives the part property *Engine*. Consequently, part properties allow the same block to be reused in different contexts.

Value Properties

Value properties are used to describe quantifiable physical, performance and other characteristics of a block such as its weight or speed [FRI08, p. 95].

Unlike part properties they are typed by value types. The intent of the value type is to provide a uniform definition of a quantity that can be shared by all value properties. Value type definitions can be reused by typing multiple value properties with the same value type [FRI08, p. 113]. A value type can be based on the fundamental types (e.g. *Integer*, *Boolean*, *Real* or *String*) or can be further characterized by adding a dimension and/or unit.

Block properties of type value can be assigned an initial value. In addition, value properties have the particular feature that they can be related by using parametric constraints (see ch. 2.4.6).

Reference Properties

Reference properties are referenced by a block, but not owned by it. An association between two blocks creates a reference property in the "from block" to the block at the other end of the association [HOL13, p. 94].

Fig. 2.14 shows an exemplary association between the *Aircraft* and the *Physical Environment* block, that was modelled in a block definition diagram. The association represents that the *Aircraft* is operated in the *Physical Environment*. This association does not symbolize a hierarchy as with part properties. Still, the association creates a reference property in each block to the other block.

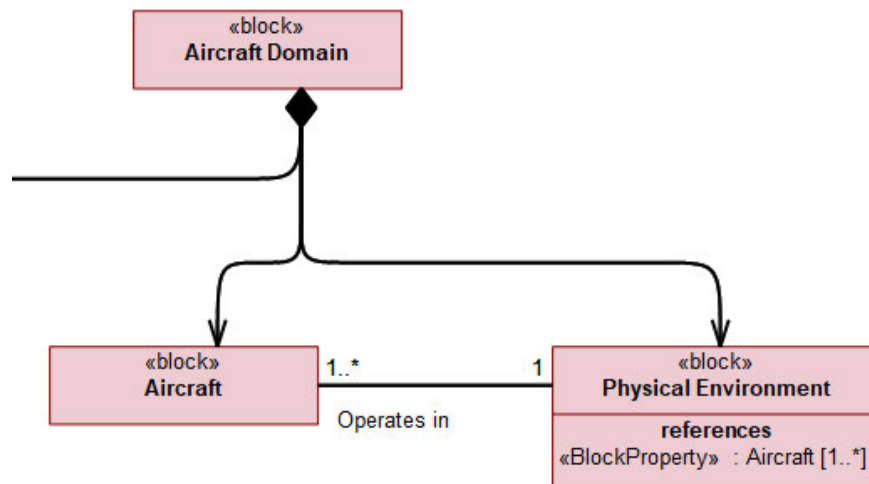


Figure 2.14: Modelling of the association between the *Aircraft* block and the *Physical Environment* block

Block properties can be modelled in so-called internal block diagrams. They will be presented in the next chapter.

The Internal Block Diagram (ibd)

Internal block diagrams (ibd) are used to describe the internal structure of a block in terms of its value properties and how its parts are interconnected.

Fig. 2.15 shows the schematic structure within the aircraft block. It can be seen that the aircraft block has an engine part property with a multiplicity of 2 and at least one wing and one fuel tank. Additionally, the aircraft has two value properties: the cruise Mach number and the cruise attitude.

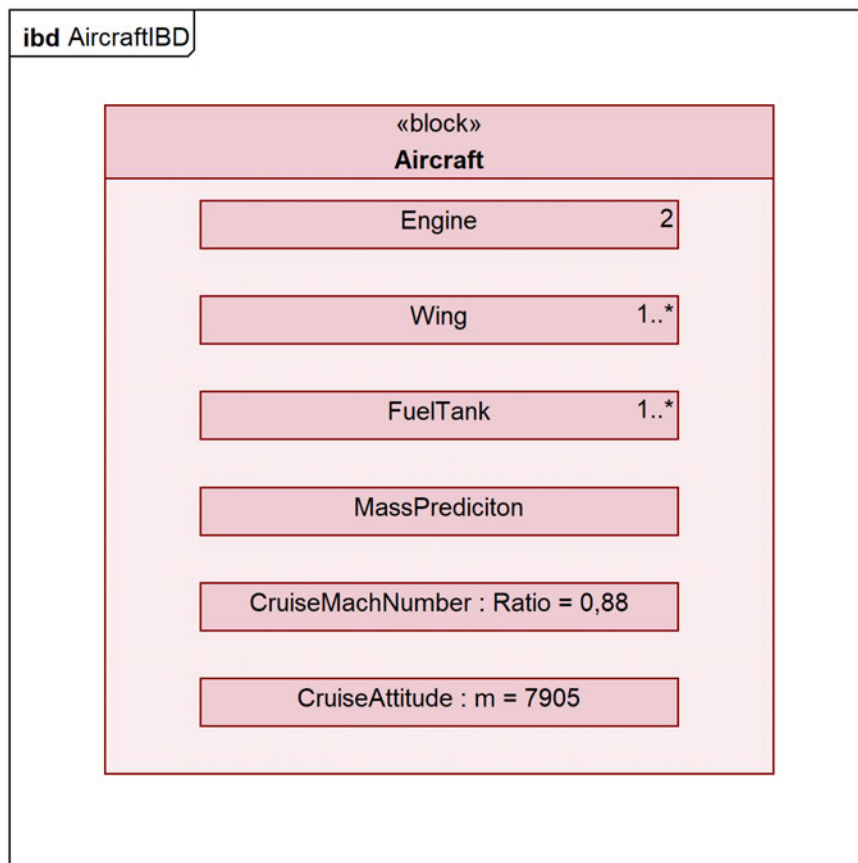


Figure 2.15: Exemplary internal structure of the *Aircraft* block, modelled in an internal block diagram

Even though the information content of the block definition diagram (fig. 2.13) and the internal block diagram (see fig. 2.15) does not differ much, the internal block diagram can be further elaborated from here. For example, the fuel flow from the tank to the engine or a mechanical connection of the engine and the wing could be modelled.

However, the detailed modelling of the internal structure of blocks will be neglected in this thesis. For more information on the different uses of internal block diagrams it is referred to [FRI08] or [HOL13].

2.4.6 Constraints and Parametric Modelling

The previous chapters have introduced ways to model a system in terms of requirements, system structure and system properties. A fundamental part of the system design is also the system analysis. It is beneficial to integrate the underlying mathematical models into the SysML model. By this, it can be defined how the various system components are

related to each other through the underlying mathematical correlations. Furthermore, it might provide a means to integrate the SysML model with analysis models [FRI08, p.53].

The UML, having its roots in software development, does not offer the modelling of mathematical relationships. Therefore, the SysML has extended the UML with concepts for parametric modelling. The fundamental SysML element for parametric modelling is the constraint block, which is described in the following section.

The Constraint Block

Constraint blocks are similar to conventional SysML blocks, only that they define constraints in terms of equations and their parameters instead of systems and their components. Just like the SysML blocks, constraint blocks can be defined in block definitions diagrams. This enables the modelling of hierarchies between different constraint blocks. Thus, complex constraint blocks can be modelled by reusing lower level constraint blocks [FRI08, p. 154].

Fig. 2.16 shows the composition of equation $E = m \cdot c^2$ (the upper yellow rectangle). It can be seen that it is composed of the equation $y = x^2$ and a product equation in the form $out = in_1 \cdot in_2$ (the lower yellow rectangles).

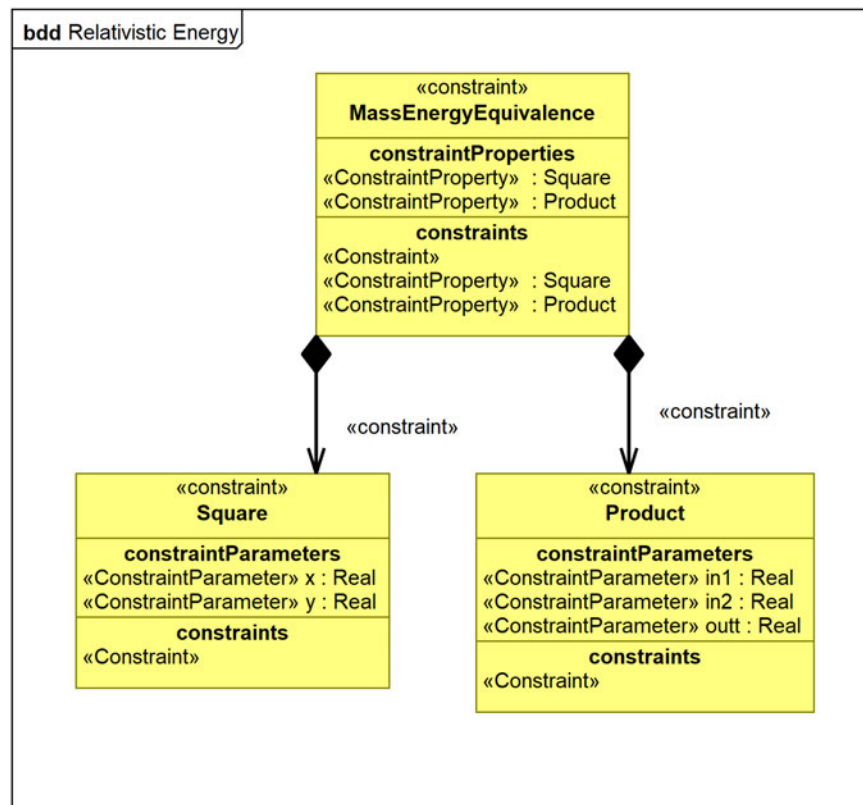


Figure 2.16: Modelling the compositions of the equation $E = m \cdot c^2$ in a block definition diagram

As shown in fig. 2.16, the In SysML does not explicitly define the equations. Instead, only the critical parameters of the equations are defined [FRI08, p. 51]. This has the advantage that complicated mathematical backgrounds do not have to be modelled in detail.

The critical parameters are called constraint parameters. Fig. 2.16 shows, that the *Square* constraint $y = x^2$ contains the two constraint parameters x and y . The *Product* constraints the three constraint parameters out , in_1 and in_2 . Through its type, the parameter can also be constrained to have a specific unit and dimension. For example, all the constraint parameters in fig. 2.16 are typed "Real".

The Constraint

Each constraint block contains a constraint (see fig. 2.16). In parametric modeling, the exact mathematical relationship can be stored in this element in the form of an equation.

Furthermore, constraints can be used to define the permissible conditions of design features and the permissible range of the design and performance parameters [SAD12, p.6]. Fig. 2.17 shows how a constraint is linked to a value property within an internal block diagram. This allows the upper and lower limits of the value property to be defined.

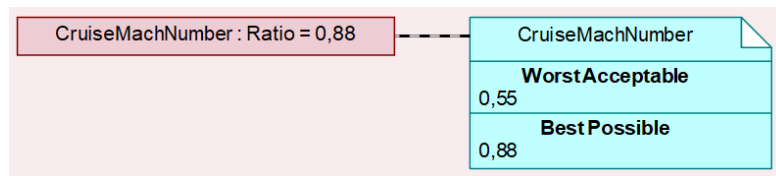


Figure 2.17: Constraining the upper and lower limits of a value property in SysML

The Constraint Property

In ch. 2.4.5, it was mentioned that part properties allow to use the same block in different contexts. Similarly, a constraint block can also be reused in different contexts. The equivalent concept to the part property is called a constraint property and is typed by a constraint block [FRI08, p. 152].

Until now, only the structural composition of the constraint blocks has been modelled. Often, however, one would like to model how the system parameters modelled through value properties are related to each other. For this, the parametric diagram can be used.

The Parametric Diagram (par)

A parametric diagram (par) is used to model the mathematical relationships between various value properties through constraint properties. A constraint property is the use of a constraint block in a specific context, just like a part property is the use of a block in a particular context (see ch. 2.4.5). Constraint properties enable to bind the constraint parameters of a constraint block to block properties of type value on a parametric diagram [PTC19a].

Fig. 2.18 shows the exemplary representation of the parametric equation $E = m \cdot c^2$ in a parametric diagram. The yellow blocks with rounded corners are the constraint properties. They resemble the different components that make up the equation $E = m \cdot c^2$. The *Square* constraint property establishes a relationship between x and y in the form:

$$y = x^2$$

The *Product* constraint property establishes a relationship between in_1 and in_2 and out in the form:

$$out = in_1 * in_2$$

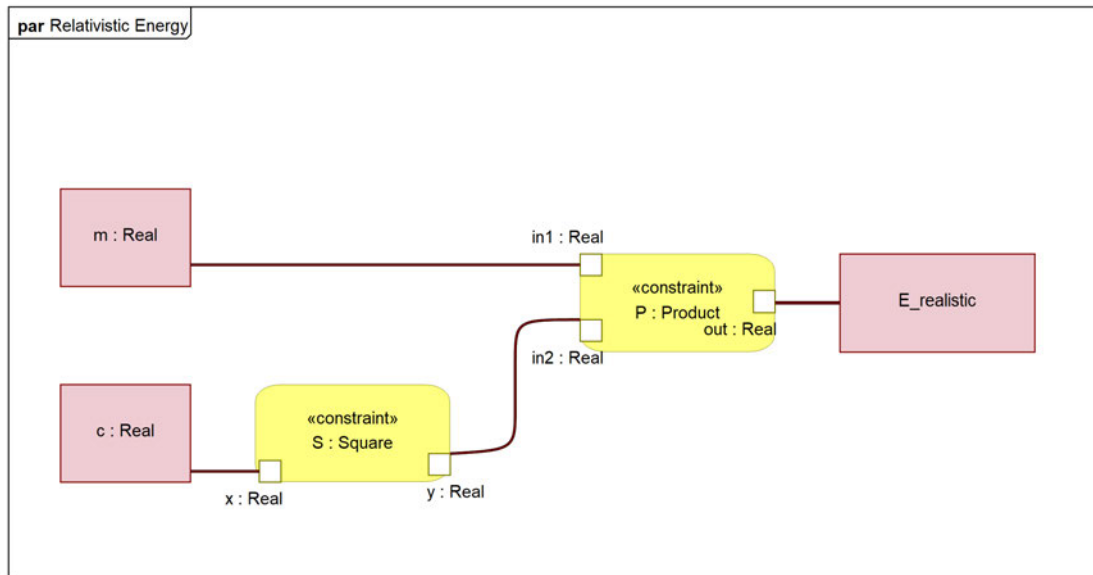


Figure 2.18: Representation of the parametric equation $E = m \cdot c^2$ with SysML (based on [ALT12, fig. 4.16])

As explained above, each constraint property has constraint parameters. They are shown as small rectangles within the inside boundary of the constraint (\square) [FRI08, p.52]. The specific values needed to support the evaluation of the constraints (m , c , $E_{realistic}$) are stored inside the value properties (red blocks with square corners, see ch. 2.4.5). The constraint parameters are connected to the corresponding value properties through binding connectors ($_$). The binding connector symbolizes that the value of one constraint parameter is the same as the connected constraint parameter or value property [FRI08, p.52]. For example, the binding connector on the lower left corner of fig. 2.18 resembles:

$$x = c$$

The component-wise representation of a mathematical relationship, as shown in fig. 2.18, is however not SysML-compliant. Instead, one would model the relationship between e , m , and $E_{realistic}$ by a single constraint (see fig. 2.20). However fig. 2.18 shows the parallels to mathematical modelling with Simulink. Simulink is a commonly used graphical programming environment based on MATLAB for modeling, simulation, and analysis of systems. Some modellers offer the possibilities to synchronize parametric

diagrams with Simulink to avoid duplicate modelling in both tools [PTC19a] (see ch. 5.5.3).

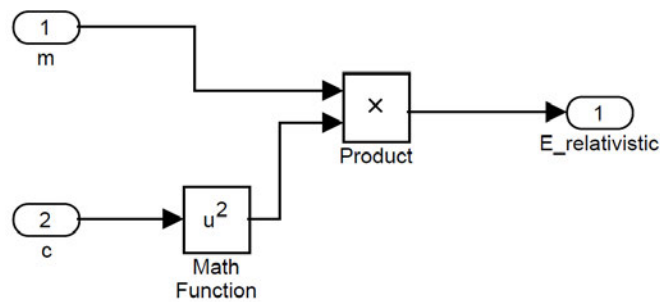


Figure 2.19: Representation of the parametric equation $E = m \cdot c^2$ with Simulink (taken from [ALT12, fig. 4.17])

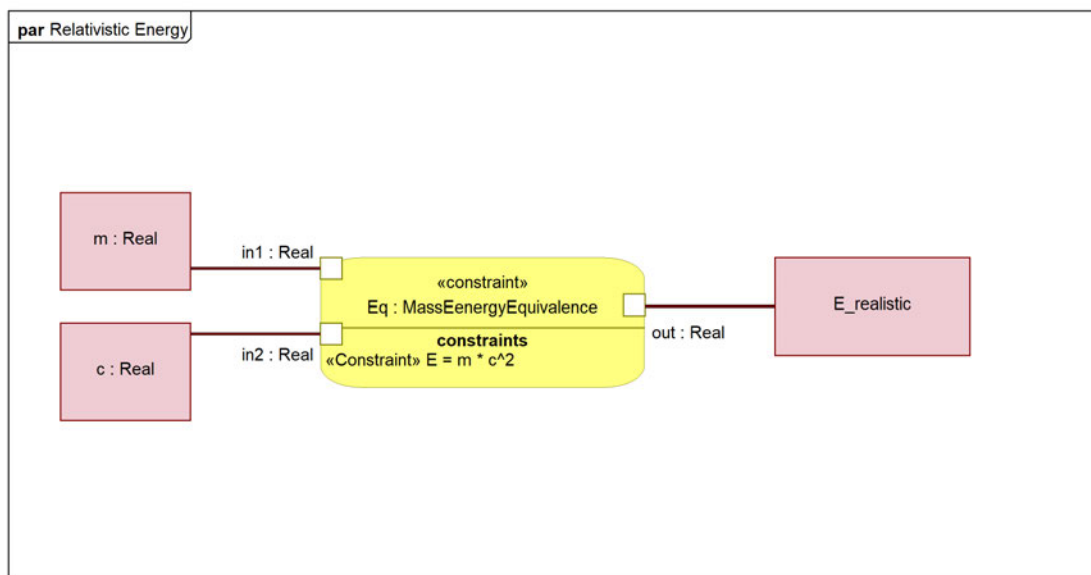


Figure 2.20: Representation of the parametric equation $E = m \cdot c^2$ with SysML

2.4.7 Packages

For larger models with many items, it becomes increasingly difficult to maintain a clear structure. SysML models can therefore be organized in a hierarchical tree of packages, comparable to folders in a Windows directory structure [FRI08, p. 81]. A package is a container for other model elements. Any model element is contained in exactly one container. When that container is deleted or copied, the contained model element is deleted or copied along with it [FRI08, p. 81]. Mostly, model items of the same topic

are grouped together in one package. For example, the packages can be divided into categories such as system structure, system requirements, system behaviour. Fig. 2.21 shows the exemplary package structure of an aircraft model.



Figure 2.21: Exemplary package structure of the *Aircraft* model

The packages can be modelled in a package diagram, which will be explained in the following chapter.

The Package Diagram (pkg)

The package diagram (pkg) can be used to create a comprehensive overview of all model packages and to display the package hierarchy. It can also be used to create, delete or edit packages. Fig. 2.22 shows the folder structure from fig. 2.21 displayed in a package diagram.

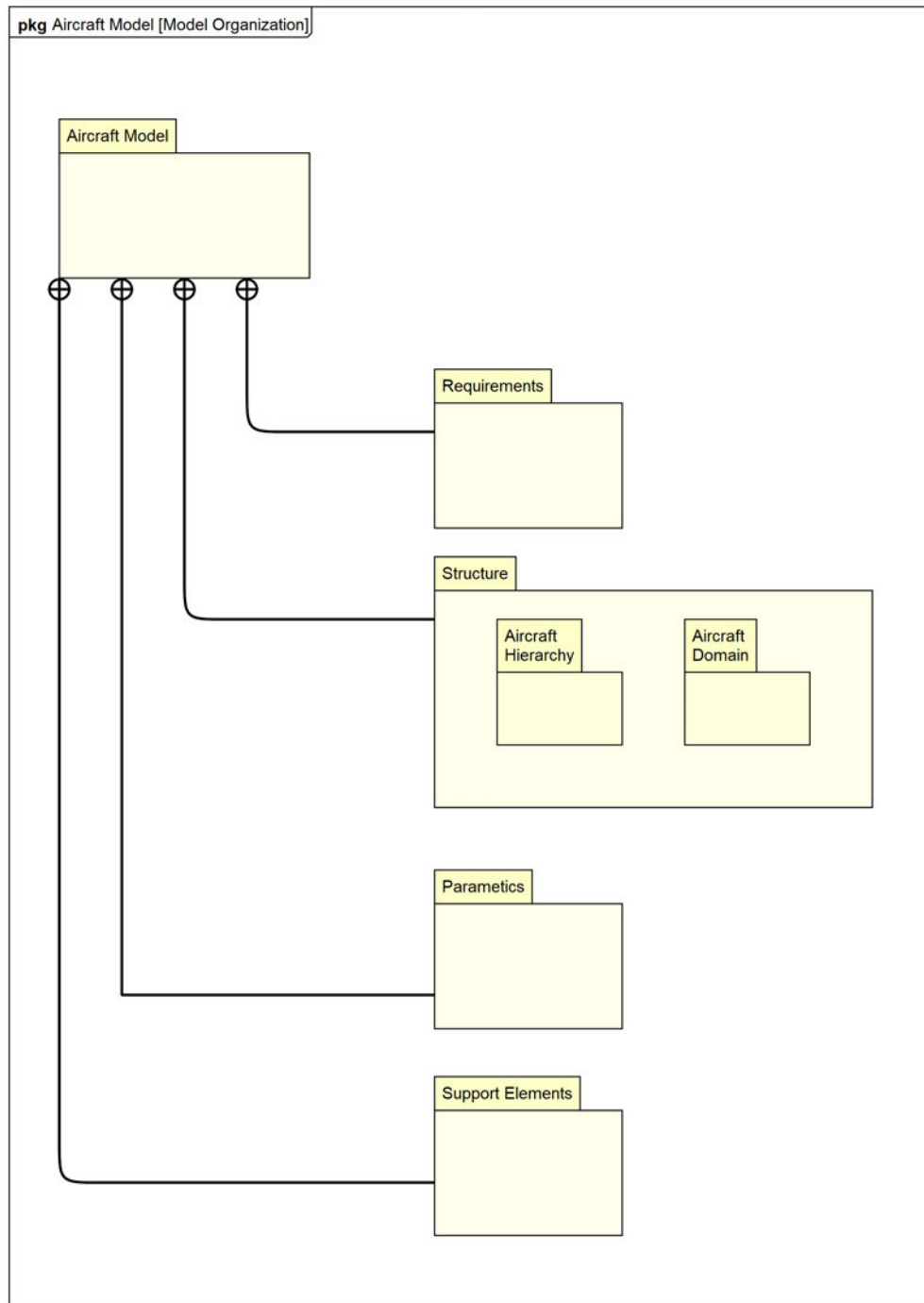


Figure 2.22: Package diagram showing how the *Aircraft* model is organized into packages (based on [FRI08, fig. 3.19])

2.5 Summary

The design challenges in engineering have increased significantly over the past decades. In particular, the increase of complexity and interconnectivity, globalization as well as the demand for custom designs and life-cycle engineering make it necessary to consider other design methods. One method to manage these challenges is provided by model-based systems engineering (MBSE).

In MBSE, a digital model is created that stores the requirements, design, analyses, and verifications at all stages of a system's life cycle in one place. By considering the system as a whole, a common understanding of the system is created among the development team and other stakeholders. In addition, the digital model elements can be easily associated with each other. This allows, for example, easy traceability of requirements or test cases. Furthermore, the reusability of model items is simplified, enabling a simple design customization according to the stakeholder requirements.

There are different approaches for creating the digital model. One modelling method is the Systems Modeling Language (SysML). It is an extension of the United Modeling Language (UML) used in software development. The UML is adaptable by additional profiles and custom stereotypes to the individual needs. This adapts the so-called meta model of the language, which defines the objects, attributes and associations of the different modelling items.

The SysML is a general purpose graphical system modelling language. This means that the digital model can be created and edited from within diagrams. The items displayed on a diagram can be seen as symbols representing underlying model items. These model items are stored in a model repository. The SysML provides nine different types of diagrams for different applications. The requirement diagram (req) represents text-based requirements and their relationship with other requirements, design elements or test cases. A block definition diagram (bdd) represents one of the key elements in SysML, the blocks, and their relationship with other design elements, requirements or test cases. An internal block diagram (ibd) is used to model the internal structure of a block. The parametric diagram (par) can be used to model the mathematical relationships between different parameters specified in the model.

3 Fundamentals of Aircraft Preliminary Sizing

3.1 Introduction

In the previous chapter the main principles of SE, MBSE and the SysML were introduced. This knowledge is important for the system definition model. This chapter is intended to provide the foundation for the analysis model. For this purpose, the main principles of the preliminary sizing of civil passenger jet aircraft are presented.

First, the design sequence leading to the preliminary sizing is explained in ch. 3.2. In ch. 3.3, a preliminary design process based on [LOF80] is introduced. Afterwards, important components of preliminary sizing such as iterations, decision making and optimization are discussed. Finally, ch. 3.7, serves to create a linkage between aircraft preliminary sizing and systems engineering.

3.2 Aircraft Design Sequence

Aircraft design starts with defining the aircraft's intended purpose. The main purpose of a passenger aircraft is to safely transport passengers and cargo at the lowest possible operating cost. This differs, for example, from the purpose of a military aircraft or an ultralight utility aircraft.

The aircraft's purpose has an influence on the aircraft's mission. The mission specification contains assumptions about the expected operation of an aircraft. It is essential for making statements about fuel consumption and the resulting aircraft weight. Fig. 3.1 shows the mission profiles of three different aircraft.

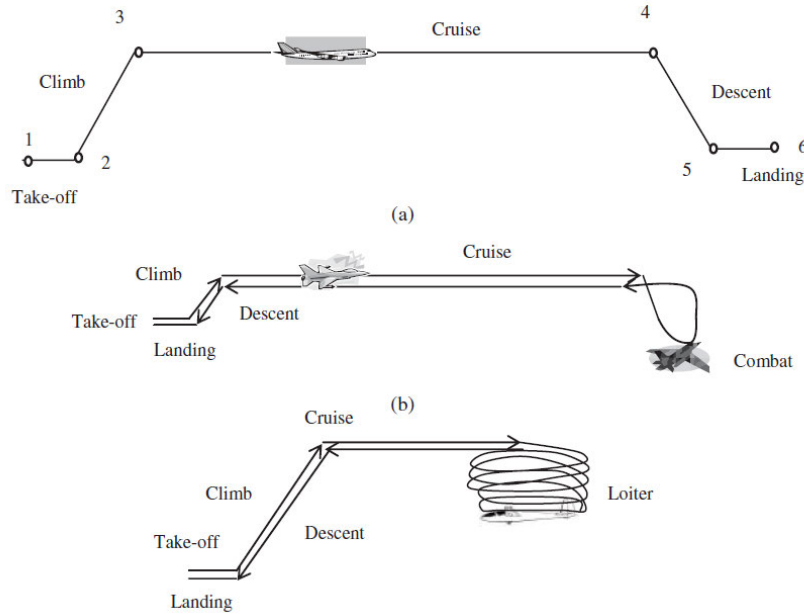


Figure 3.1: Typical mission profiles for three different aircraft: (a) transport aircraft, (b) fighter, and (c) reconnaissance (taken from [SAD12, fig. 4.2])

Some important parameters of the mission specification are the cruise attitude h_{Cr} , the range R and the mission fuel fractions M_{ff} (see ch. 3.3).

Additionally, the Top Level Aircraft Requirements (TLAR) are defined. The TLAR summarize the expected performance of the future aircraft, such as the cruise speed V_{Cr} . Furthermore, basic requirements, such as the number of passengers n_{PAX} , the required runway length s_{LFL} or the cargo mass m_{CARGO} , are specified in the TLAR. Moreover, customer requirements and certification requirements have a great influence on the TLAR.

The mission specification and the TLAR are determined by market analyses, forecasts, experience and strategic issues [PET20]. Together they form the listing of requirements called performance or contract specification. The contract specification forms the starting point of the aircraft design. The following requirements should at least be defined when the aircraft design begins [SCH19, p. 1-1]:

- Payload m_{PL}
- Cruise performance
 - Range R
 - Mach Number M_{Cr}
- Airport performance

- Take-off field length s_{TOFL}
- Landing field length s_{LFL}
- Climb gradient for the 2nd segment γ_{CLB}
- Climb gradient for a missed approach γ_{MA}

Conceptual Design

With the requirements listed above, the conceptual design can be initiated (see fig. 3.2). The conceptual design is intended to define the basic aircraft configuration. Basic design considerations, such as the position of the wings and the type of control surfaces, are defined. The conceptual design usually starts with a brain storming stage. The brain storming aims at finding many different concepts that all meet the requirements. Eventually, the best concept based on certain criteria is selected. This is usually performed through a trade-off analysis (see ch. 3.5). Besides, feasibility studies are performed to discover potential solutions for certain technical requirements. Overall, during the conceptual design phase almost all parameters are determined based on decision-making processes and selection techniques [SAD12, p. 38].

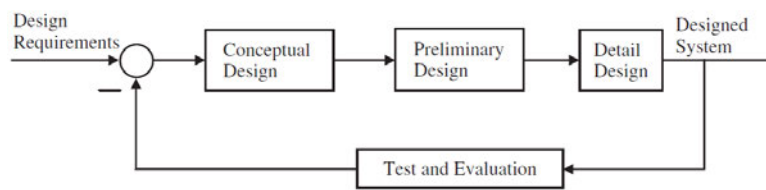


Figure 3.2: Relationship among the four major design activities (taken from [SAD12, fig. 2.2])

Fig. 3.3 shows the aircraft design process with design reviews taking place after each design round. The design reviews serve to compare the current design with the requirements. For this purpose, various tests and evaluations are carried out [SAD12, p. 22]. In case of disappointments, the previous design step must be performed again. This leads to iteration loops (see fig. 3.3 & ch. 3.4).

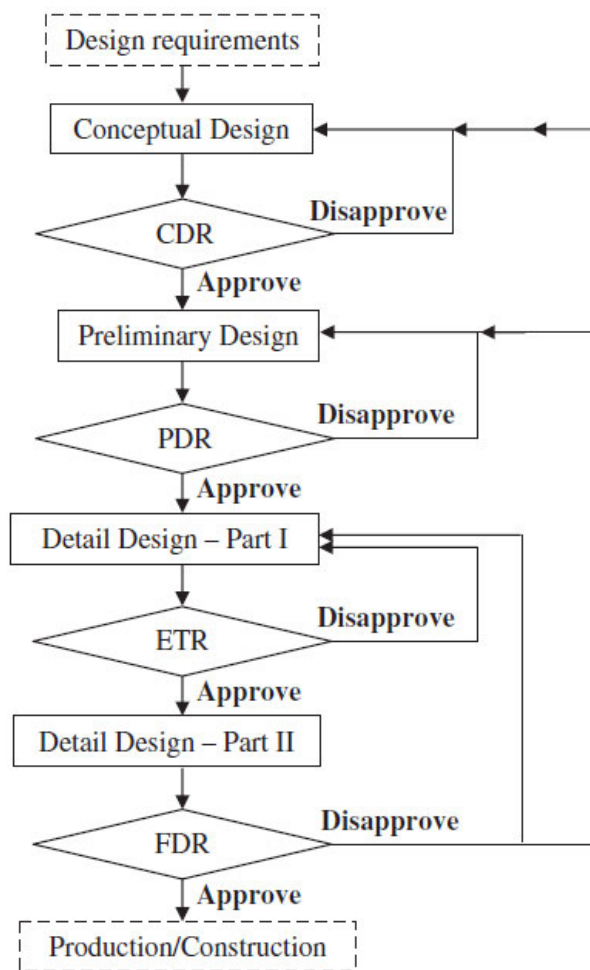


Figure 3.3: Formal design reviews (taken from [SAD12, fig. 2.8])

As illustrated in fig. 3.3, the conceptual design phase is concluded with the Conceptual Design Review (CDR). During the CDR, a formalized check of the proposed system design is provided, major problems are discussed, and corrective actions are taken [SAD12, p. 28].

Preliminary Design

If there are no disapprovals at the CDR, the preliminary design can be initiated (see fig. 3.2 and 3.3). In contrast to the conceptual design phase, where the aircraft is designed in accordance with non-precise results, the aircraft preliminary design phase tends to employ the outcomes of a calculation procedure [SAD12, p. 38]. On top of that, the design requirements for subsystems are developed from system-level requirements. Furthermore, performance technical measures are determined at the subsystem level

[SAD12, p. 29]. These are the major output values that are estimated during the preliminary design (see ch. 3.3):

1. the aircraft maximum take-off weight: m_{MTO}
2. the engine power or thrust: T_{TO}
3. the wing reference area: S_W

As the name implies, the parameters defined during the preliminary design phase are not final and will be altered later. This is due to the iterative nature of aircraft design [SAD12, p. 38]. Fig. 3.4 shows a schematic aircraft design sequence. The associated iteration loops are symbolized by the red and pink arrows in fig. 3.4. The iterations are continued until all design requirements are satisfied or if the cost of one new iteration exceeds the benefits of a the new design.

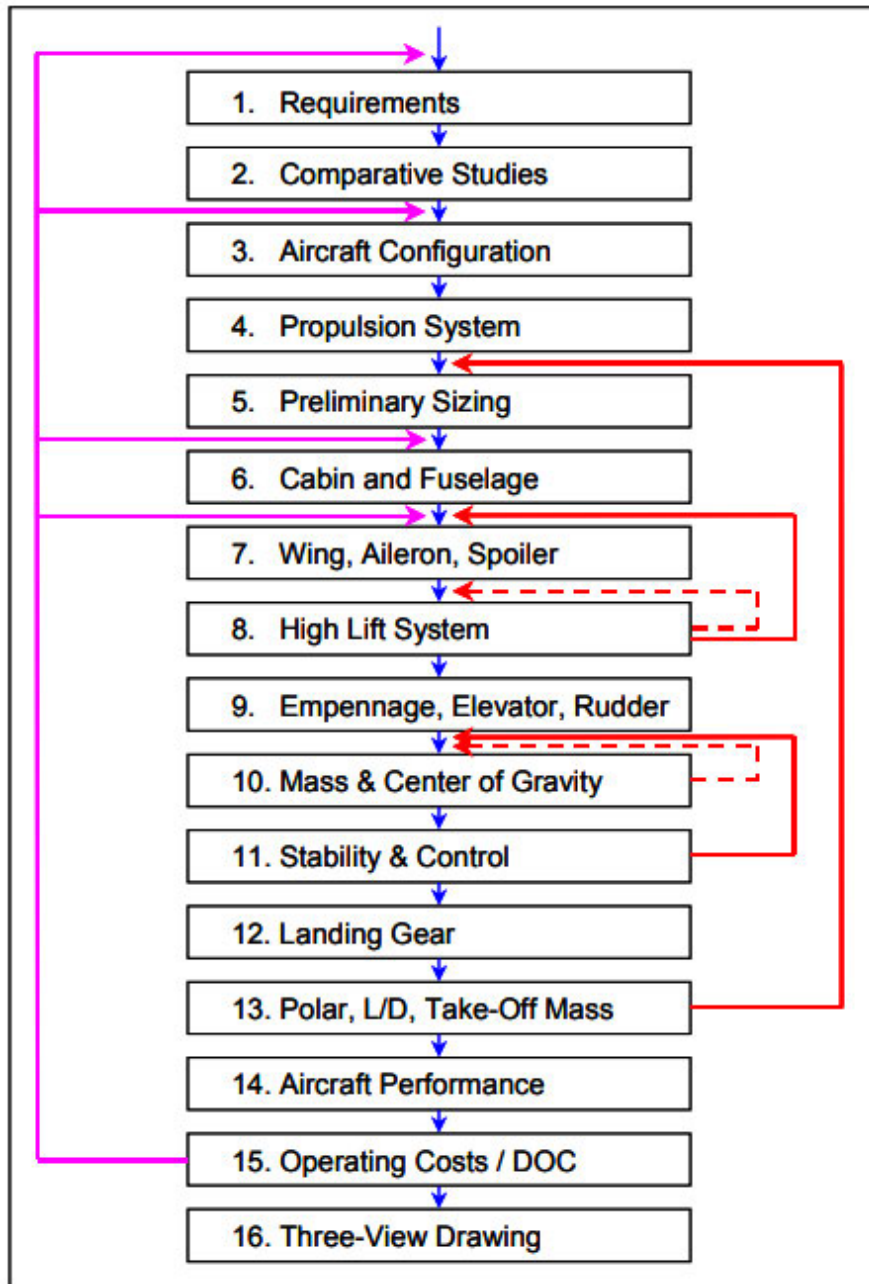


Figure 3.4: Schematic Design Sequence including preliminary sizing, conceptual design and iteration loops (taken from [SCH19, fig. 2.1])

The aircraft design is further refined during the stages following the preliminary design. The decisions made during the conceptual design and preliminary sizing stages have a major impact on the following design process (see fig. 3.4). They will govern the aircraft

size, the manufacturing cost, and the complexity of calculations [SAD12, p. 93]. It is therefore crucial to place a great emphasis on those stages. In the next chapter, a method for the preliminary sizing of civil jet aircraft will be presented.

3.3 Aircraft Preliminary Sizing

3.3.1 Introduction and Main Idea

The approach presented in the following chapters is based on [LOF80]. The design technique is very accurate and the results are considered reliable [SAD12, p. 94]. However, the presented sizing process is limited to civil jet-powered aircraft that are certified according to CS-25 or FAR Part 25 only. A similar approach for propeller aircraft can be found in [LOF80, ch. 6 & 7].

The presented preliminary sizing approach considers the aircraft in its five crucial flight phases. They are

- take-off
- 2nd segment
- cruise
- missed approach
- landing

For each of the flight phases, certain performance criteria have to be met. The main idea is to express the criteria for each flight phase by two design variables: the wing loading during take-off m_{MTO}/S_W and the thrust-to-weight ratio during take-off $T_{TO}/(m_{MTO} \cdot g)$.

The five sets of relationships between the wing loading and the thrust-to weight ratio at take-off lead to a two dimensional optimization problem. The optimization problem can then be solved by plotting the relationships in a matching chart and selecting a feasible design point (see ch. 3.3.3). Based on the selected value pair, a rapid estimation of other design parameters is possible (see ch. 3.3.4). Most importantly, the maximum take-off weight m_{MTO} can be estimated.

3.3.2 The Five Flight Phases

I - Landing

For the landing phase [LOF80] combines a statistical approach with Federal Air Regulations and safety margins to estimate the aircraft performance during landing. Combining [LOF80, p. 105] with [LOF80, p.111] leads to an upper limit for the design wing loading:

$$\frac{m_{MTO}}{S_W} \leq 0.107 \frac{kg}{m^3} \cdot \sigma \cdot C_{L,max,L} \cdot s_{LFL} \cdot \frac{1}{m_{ML}/m_{MTO}} \quad (3.1)$$

The following design parameters are required:

- σ : the landing-field density divided by the sea-level density. It is assumed to be $\sigma = 1$ []
- $C_{L,max,L}$: the maximum lift coefficient during landing []
- s_{LFL} : the landing field length [m]
- m_{ML}/m_{MTO} : the estimated ratio of the maximum landing weight m_{ML} and the maximum take-off weight m_{MTO} [], see [LOF80, p.119]

In the matching chart, eq. (3.1) will appear as a vertical line that may not be exceeded (see ch. (3.3.3)).

II - Take-Off

For the take-off performance, the following constraint between take-off thrust $T_{TO}/(m_{MTO} \cdot g)$ and maximal wing loading at take-off m_{MTO}/S_W can be derived from [LOF80, p.114]:

$$\frac{T_{TO}}{m_{MTO} \cdot g} \geq \frac{2.34 \frac{m^3}{kg}}{s_{TOFL} \cdot \sigma \cdot C_{L,max,TO}} \cdot \frac{m_{MTO}}{S_W} \quad (3.2)$$

The following parameters are required:

- σ : the take-off-field density divided by the sea-level density. It is again assumed to be $\sigma = 1$ []
- $C_{L,max,TO}$: the maximum lift coefficient during take-off []
- s_{TOFL} : the take-off field length [m]

Eq. (3.2) will appear as a line with positive slope that acts as a lower barrier for the design point (see ch. (3.3.3)).

III - Climb Rate during 2nd Segment

The 2nd climb is that portion of the flight path, following take-off, which extends from an altitude of 35 to 400 ft [LOF80, p.117]. The thrust-to-weight ratio must be sufficient to maintain a certain climb gradient in event of an engine failure. The design constraint can be obtained by transforming [LOF80, p.109] to:

$$\frac{T_{TO}}{m_{MTO} \cdot g} \geq \left(\frac{n_E}{n_E - 1} \right) \cdot \left(\frac{1}{E_{2ndS}} + \sin \gamma \right) \quad (3.3)$$

The following values are required:

- n_E : the amount of engines []
- γ : the steady gradient of climb [], as defined in [EAS07, CS 25.121]

The lift-to-drag ratio during the 2nd segment E_{2ndS} can be approximated by [LOF80, p.121]:

$$E_{2ndS} = \frac{C_L}{C_{D,0} + \Delta C_{D,f} + \Delta C_{D,s} + \frac{C_L^2}{\pi \cdot A \cdot e}} \quad (3.4)$$

with:

- C_L : approach lift coefficient, estimated according to [LOF80, p.109] to:
 - $C_L = 1.3$ [] for a 15° flap deflection
 - $C_L = 1.5$ [] for a 25° flap deflection
 - $C_L = 1.7$ [] for a 35° flap deflection
- $C_{D,0}$: zero-lift drag coefficient for the aircraft in the clean condition, assumed to be $C_{D,0} = 0.02$ [] [LOF80, p.110]
- $\Delta C_{D,f}$: increments in profile drag coefficient associated with trailing-edge flap deflection, estimated according to [LOF80, p.110] to:
 - $\Delta C_{D,f} = 0.01$ [] for a 15° flap deflection
 - $\Delta C_{D,f} = 0.02$ [] for a 25° flap deflection
 - $\Delta C_{D,f} = 0.03$ [] for a 35° flap deflection
- $\Delta C_{D,s}$: increments in profile drag coefficient associated with leading-edge flap deflection [], this drag component is negligible [LOF80, p.110]
- A : the wing aspect ratio []
- e : the Oswald efficiency factor, assumed to be $e = 0.7$ [] [LOF80, p.110]

IV - Missed Approach

The thrust-to-weight ratio required to perform a missed approach can be calculated similarly to eq.(3.4). If the aircraft is certified according to FAR-25, an additional drag coefficient for the landing gear must be considered [FAA17, p. 25.121]. The landing

gear drag coefficient is assumed to be $\Delta C_{D,g} = 0.015$ [LOF80, p.114)]. Altogether, the following constraint applies:

$$\frac{T_{TO}}{m_{MTO} \cdot g} \geq \left(\frac{n_E}{n_E - 1} \right) \cdot \left(\frac{1}{E_{MA}} + \sin \gamma \right) \cdot \frac{m_{ML}}{m_{MTO}} \quad (3.5)$$

with the glide ratio during missed approach, E_{MA} :

$$E_{MA} = \frac{C_L}{C_{D,0} + \Delta C_{D,f} + \Delta C_{D,s} + \Delta C_{D,g} + \frac{C_L^2}{\pi \cdot A \cdot e}} \quad (3.6)$$

The design parameters are the same as for the 2nd segment. Only a mass conversion, using the estimated ratio of the maximum landing and take-off weight m_{ML}/m_{MTO} , is added (see [LOF80, p.119]). Eq. (3.3) and (3.5) both appear as horizontal lines that act as a lower barrier in the matching chart (see ch. (3.3.3)).

V - Cruise

By combining equations from [LOF80, p. 137] and [NIT12, p. 22], the following equation can be formed. It states the necessary thrust-to-weight ratio $T_{TO}/(m_{MTO} \cdot g)$ depending on the cruise attitude h_{Cr} :

$$\frac{T_{TO}}{m_{MTO} \cdot g} (h_{Cr}) = \frac{1}{\left((0.0013\mu - 0.0397) \cdot \frac{h_{Cr}}{km} - 0.0248\mu + 0.7125 \right) E_{Cr}} \quad (3.7)$$

- μ : the engine by-pass ratio []
- E_{Cr} : the cruise glide ratio [], calculated with eq.(3.8)

The following equation states the wing loading depending on the cruise attitude [LOF80, p. 138]:

$$\frac{m_{MTO}}{S_W} (h_{Cr}) = \frac{C_{L,Cr} \cdot M_{Cr}^2}{g} \cdot \frac{\gamma}{2} \cdot p(h_{Cr}) \quad (3.8)$$

with:

- $C_{L,Cr}$: the cruise lift coefficient [], calculated with eq.(3.10)
- M_{Cr} : the cruise Mach number []
- γ : heat capacity ratio [], assumed to be $\gamma = 1.4$
- $p(h_{Cr})$ the air pressure at cruise attitude, calculated according to [NAC55, p. 724]

The glide ratio E_{Cr} needed for eq.(3.7) can be estimated by [NIT12, p. 22, 23]:

$$E_{Cr} = \frac{2 \cdot k_E \cdot \sqrt{\frac{A}{(S_{Wet}/S_W)}}}{(V_{Cr}/V_{md})^2 + \frac{1}{(V_{Cr}/V_{md})^2}} \quad (3.9)$$

The cruise lift coefficient $C_{L,Cr}$ needed for eq.(3.8) can be calculated by combining [NIT12, p. 22, 23] and [LOF80, p. 110] to:

$$C_{L,Cr} = \frac{\pi \cdot A \cdot e_{Cr}}{2 \cdot (V_{Cr}/V_{md})^2 \cdot k_E \cdot \sqrt{\frac{A}{(S_{Wet}/S_W)}}} \quad (3.10)$$

The following input parameters are needed for eq.(3.9) and (3.10):

- k_e : assumed to be $k_e = 14.5$ [NIT12, p. 23]
- A : the wing aspect ratio []
- S_{Wet}/S_W : the relative wetted area, assumed to be $S_{Wet}/S_W = 6.1$ [RAY89, p. 21]
- V_{Cr}/V_{md} : the ratio between the cruise speed V_{Cr} and the minimum drag speed V_{md}
- e_{Cr} : the cruise Oswald factor [], assumed to be $e_{Cr} = 0.8$ [NIT12, p. 23]

3.3.3 The Matching Chart

Fig. shows the plot resulting from the design constraints set by eq.(3.1), eq.(3.2), eq.(3.3), eq.(3.5), eq.(3.7) and eq.(3.8) [LOF80, p. 144]. The white area shows feasible value pairs. The grey area marks infeasible designs that violate at least one of the design constraints.

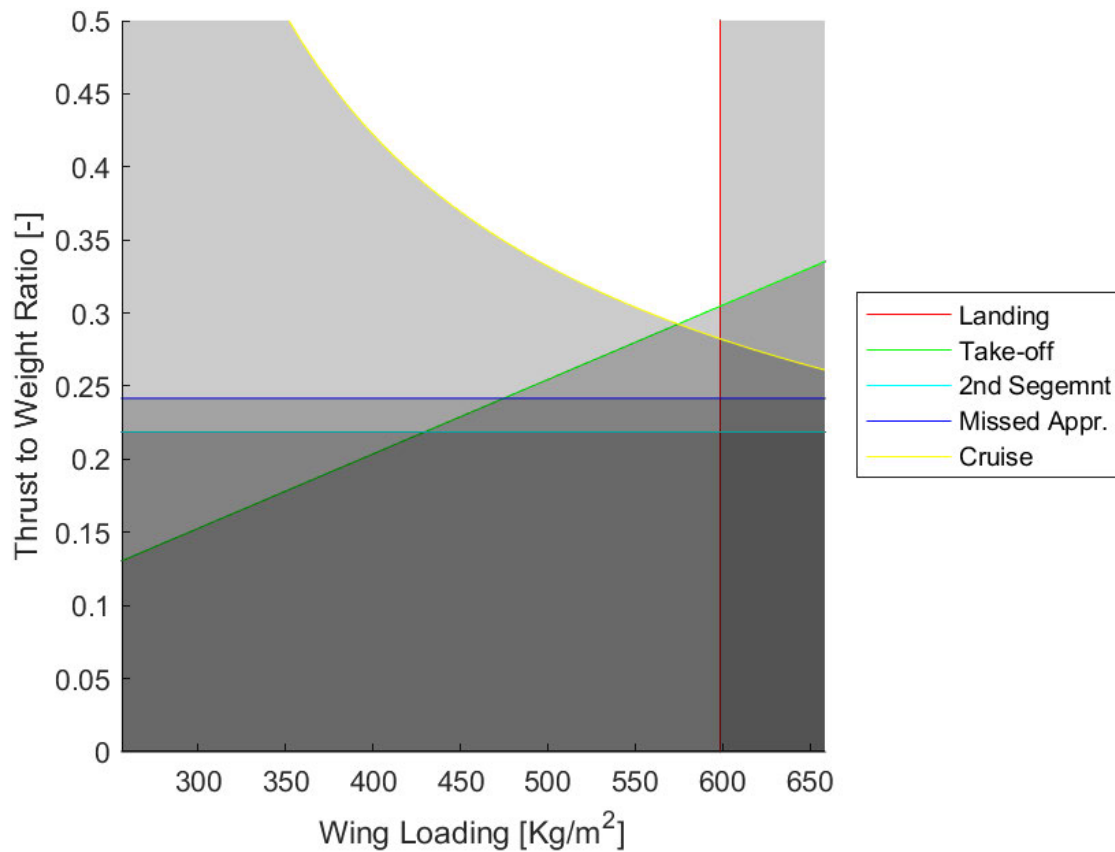


Figure 3.5: Matching chart

The next step is to choose a design point within the feasible region. A common approach is to choose the design point with the highest wing loading and lowest thrust-to-weight ratio. The assumption is that this results in the lowest maximum take-off weight. However, in this thesis an optimization algorithm is used to find the design point. As the optimization objective, both the maximum take-off weight and the thrust-to-weight ratio were minimized (see ch. 5.4).

3.3.4 Calculation of the Output Parameters

For the following chapters it is assumed that the optimization algorithm has found an optimal solution within the permissible region of the matching chart. Now, that the thrust-to-weight ratio and the wing loading is set, the design parameters that depend on them are calculated.

Calculation of the Cruise Parameters

The pressure at cruise altitude $p(h_{Cr})$ can be calculated by rearranging eq.(3.8) to:

$$p(h_{Cr}) = \left(\frac{m_{MTO}}{S_W} \right)_{DES} \cdot \frac{2 \cdot g}{C_{L,Cr} \cdot M_{Cr}^2 \cdot \gamma} \quad (3.11)$$

with:

- $\left(\frac{m_{MTO}}{S_W} \right)_{DES}$: the design wing loading obtained from the matching chart
- $C_{L,Cr}$: the cruise lift coefficient [], calculated with eq.(3.10)
- M_{Cr} : the cruise Mach number []
- γ : heat capacity ratio [], assumed to be $\gamma = 1.4$

Once the pressure $p(h_{Cr})$ is known, the cruise attitude h_{Cr} and the speed of sound at cruise attitude $a(h_{Cr})$ can be calculated according to [NAC55]. The aircraft's cruise speed V_{Cr} can then be calculated with:

$$V_{Cr} = a(h_{Cr}) \cdot M_{Cr} \quad (3.12)$$

Calculation of the Fuel Consumption

The Breguet range factors for cruise and loiter can then be calculated via [LOF80, p. 120]. They are required to estimate the fuel consumption during cruise:

$$B_{Cr} = \frac{E_{Cr} \cdot V_{Cr}}{SFC_{Cr} \cdot g} \quad (3.13)$$

$$B_{Loiter} = \frac{E_{Cr} \cdot V_{Cr}}{SFC_T \cdot g} \quad (3.14)$$

The specific fuel consumption during cruise and loiter are assumed to be $SFC_{Cr} = 14.2 \cdot 10^{-6} Kg/N/s$ and $SFC_T = 11.3 \cdot 10^{-6} Kg/N/s$ [RAY89, p.19]. The glide ratio during cruise E_{Cr} can be calculated with eq.(3.9).

The fuel fraction for cruise can then be calculated with [LOF80, p. 120]. The mission fuel fraction for alternate $M_{ff,Alternate}$ and loiter $M_{ff,Loiter}$ are calculated similarly (see eq.(3.16) & eq.(3.17)).

$$M_{ff,Cr} = e^{-\frac{R}{B_{Cr}}} \quad (3.15)$$

$$M_{ff,Alternate} = e^{-\frac{R_{Alt}}{B_{Cr}}} \quad (3.16)$$

$$M_{ff,Loiter} = e^{-\frac{t_{loiter} \cdot V_{Cr}}{B_{Loiter}}} \quad (3.17)$$

The following input values are required:

- R : the design range [m]
- R_{Alt} : the alternate range according to the applicable certification regulations. For international routes, the alternate range is multiplied by the factor 1.05
- V_{Cr} : the cruise speed, calculated with eq.(3.12)
- t_{loiter} : the loiter time [s], assumed to be $t_{loiter} = 1800 \text{ s}$

The fuel consumption for other flight stages than the cruise are considered by the following fuel fractions (see [ROS85, p. 12]):

- $M_{ff,TO}$: the mission fuel fraction for take-off, assumed to be $M_{ff,TO} = 0.995$
- $M_{ff,CLB}$: the mission fuel fraction for climb, assumed to be $M_{ff,CLB} = 0.980$
- $M_{ff,DES}$: the mission fuel fraction for descend, assumed to be $M_{ff,DES} = 0.990$
- $M_{ff,L}$: the mission fuel fraction for landing, assumed to be $M_{ff,L} = 0.992$

Once the fuel fractions for each flight stage are known, the fuel weight proportion can be calculated according to [ROS85, p. 16]:

$$\begin{aligned} \frac{m_F}{m_{MTO}} &= 1 - M_{ff,std} \cdot M_{ff,res} \\ &= 1 - (M_{ff,TO} \cdot M_{ff,CLB} \cdot M_{ff,Cr} \cdot M_{ff,DES} \cdot M_{ff,L}) \\ &\quad \cdot (M_{ff,Loiter} \cdot M_{ff,CLB} \cdot M_{ff,Alternate} \cdot M_{ff,DES}) \end{aligned} \quad (3.18)$$

Calculation of the Maximum Take-Off Weight

The maximum take-off weight m_{MTO} can be calculated with [ROS85, p. 5]:

$$m_{MTO} = \frac{m_{PL}}{1 - \frac{m_F}{m_{MTO}} - \frac{m_{OE}}{m_{MTO}}} \quad (3.19)$$

The required weight ratio between the operational empty weight m_{OE} and the maximum take-off weight m_{MTO} can be estimated based on previously built aircraft. The payload weight m_{PL} required for eq.(3.19) is calculated with:

$$m_{PL} = n_{PAX} \cdot m_{PAX} + m_{CARGO} \quad (3.20)$$

and:

- n_{PAX} : the amount of passengers []
- m_{PAX} : the passenger mass, assumed to be $m_{PAX} = 93kg$ [SAD12, p. 97]
- m_{CARGO} : the cargo mass [kg]

Calculation of the Wing Area and Take-Off Thrust

Once the maximum take-off weight is known, the wing area S_W and the take-off thrust for all engines T_{TO} can be calculated via:

$$S_W = \frac{m_{MTO}}{(m_{MTO}/S_W)_{DES}} \quad (3.21)$$

$$T_{TO} = m_{MTO} \cdot g \cdot (T_{TO}/(m_{MTO} \cdot g))_{DES} \quad (3.22)$$

Remaining Mass Calculation

Once the maximum take-off weight m_{MTO} is known, the maximum landing weight m_{ML} , the operational empty weight m_{OE} and the fuel weight m_F can be calculated with:

$$m_{ML} = \frac{m_{ML}}{m_{MTO}} \cdot m_{MTO} \quad (3.23)$$

$$m_{OE} = \frac{m_{OE}}{m_{MTO}} \cdot m_{MTO} \quad (3.24)$$

$$m_F = \frac{m_F}{m_{MTO}} \cdot m_{MTO} \quad (3.25)$$

The weight proportions $\frac{m_{ML}}{m_{MTO}}$ and $\frac{m_{OE}}{m_{MTO}}$ were already required for eq.(3.1) and eq.(3.19). The fuel weight proportion $\frac{m_F}{m_{MTO}}$ is calculated with eq.(3.18).

The maximum zero-fuel weight m_{MZF} and the reserve fuel weight $m_{fuel,res}$ can then be calculated with:

$$m_{MZF} = m_{PL} + m_{OE} \quad (3.26)$$

$$m_{fuel,res} = m_{MOT} \cdot (1 - M_{ff,Loiter} \cdot M_{ff,CLB} \cdot M_{ff,Alternate} \cdot M_{ff,DES}) \quad (3.27)$$

Last, it must be verified that the following design constraint applies:

$$m_{ML} \geq m_{MZF} + m_{fuel,res} \quad (3.28)$$

If the design constraint issued by eq.(3.28) applies, the preliminary sizing is finished. Now that a method for preliminary sizing of jet airliners has been presented, other important aspects of aircraft preliminary sizing will be discussed in the next chapters.

3.4 Iterations in Aircraft Preliminary Sizing

As already briefly mentioned in ch. 3.2, iterations play an important role in aircraft design. Especially the preliminary design stage is affected by many iterations, since the parameters defined in it have an influence on all other components of the aircraft and vice versa. Fig. 3.2 shows that the preliminary design is located between the conceptual design and the detailed design. Both design stages may express the need to re-perform the preliminary sizing stage.

On the one hand, design changes can occur further upstream of the design progress. For example, requirements can be changed or the conceptual design is adapted. Changes in the input parameters of the preliminary design stage naturally lead to different output parameters. Since the output parameters of the preliminary design are required for the component design, a change in the preliminary design has a major impact further down the design process. For instance, new data for the maximum take-off weight requires a new round of calculations and new designs for all aircraft components such as wing, tail, and fuselage [SAD12, p. 94].

On the other hand, changes in the design further downstream can also lead to a change in the preliminary design. For example, component developers provide feedback to the systems engineering team and request an adjustment of the component requirements. This is an iterative process throughout development that is often required to achieve a balanced design solution [FRI08, p.9]. This feedback loop also leads to the need for an iterative design approach (see grey arrows in in fig. 2.2).

Finally, iterations also occur within the preliminary design phase, independent of the overall design process. This is due to the fact that the first design rarely is the optimum. By iteratively changing the variable design parameters, the design can be optimized (see ch. 3.6).

3.5 Decision Making in Aircraft Preliminary Sizing

Even though the previous chapters mainly introduced formulas for the mathematical calculation of design parameters, aircraft design always has a second important aspect. Many aircraft design activities involve decision making and the logic-based selection of different design alternatives (see fig 3.6). This chapter is intended to explain the basic principles of decision making in aircraft preliminary sizing.



Figure 3.6: Two main groups of design activities in aircraft design (taken from [SAD12, fig. 1.2])

Especially during the conceptual design, it is important to consider as many different designs as possible. Ultimately, it is necessary to select the optimal design from a variety of design alternatives. Investigating many different designs ensures that the best possible design is realized.

However, some parameters that are worth comparing only arise from the preliminary design. Examples would be the maximum take-off weight m_{MTOW} or the range R . It can therefore be beneficial to develop promising designs further into the preliminary design stage in order to obtain meaningful comparable parameters.

Yet, there are may be difficulties in choosing the best design. For most design problems, there is a trade-off where one attribute is improved and the other is degraded. In addition, the criteria to select the best design and their prioritization have to be defined. This decision-making process can be supported by a so-called trade-off analysis.

A trade-off analysis starts with defining the measures of effectiveness (MOE)). A MOE is used to define a criteria that needs to be evaluated in a trade study (e.g. cost, weight, maintainability). Next, the priority of the different criteria is defined through weighting factors. After that, the different designs are compared in terms of the MOE. Depending on how well a design performs with respect a MOE, different factors are allocated. These factors can either be estimated or calculated based on a mathematical model. Ultimately, the allocated factors are multiplied by the respective weighting factors and summed up. The optimal design can then be determined by comparing the different sums (see fig. 3.7).

This approach can also be taken one step further. The trade-off analysis can also be used to compare different configurations of the same conceptual design. This can be useful during the preliminary design phase, when the concept is already frozen. Fig. 3.7 shows how a trade-off analysis is used to find the optimal configuration within the configurations A, B and C. It can be seen that some design criteria are to be maximized and others minimized. Furthermore, some design criteria are to be maximized and others minimized. It is possible that some of the allocated factors are determined by mathematical calculation programs.

No.	Criteria	Must be	Priority (%)	Configuration		
				A	B	C
1	Cost	Minimized	9	115	183	210
2	Weight	Minimized	4	136	163	94
3	Period of design	Minimized	7	190	176	217
DI_{\min}			20	20.1	35.3	37.8
4	Performance	Maximized	40	210	195	234
5	Flying qualities	Maximized	15	183	87	137
6	Scariness	Maximized	1	87	124	95
7	Maintainability	Maximized	5	95	83	68
8	Producibility	Maximized	6	215	184	164
9	Disposability	Maximized	2	246	254	236
10	Stealth	Maximized	11	65	36	42
DI_{\max}			80	142	116.5	137.7

Figure 3.7: Evaluation of three presumptive configuration alternatives (taken from [SAD12, tab. 3.10])

The MOE and its weighting factors can also be formulated as a so-called objective function. An objective function establishes a mathematical relationship between all MOE. A possible objective function for the example shown in fig. 3.7 could be:

$$f_{OBJ} = \frac{0.04 \cdot DI_4 + 0.15 \cdot DI_5 + 0.01 \cdot DI_6 + 0.05 \cdot DI_7 + 0.06 \cdot DI_8 + 0.02 \cdot DI_9 + 0.11 \cdot DI_{10}}{0.09 \cdot DI_1 + 0.04 \cdot DI_2 + 0.07 \cdot DI_3}$$

The objective function f_{OBJ} is then calculated for all three configurations. The maximum value of f_{OBJ} would provide the optimal configuration.

3.6 Optimization in Aircraft Preliminary Sizing

The trade-off analysis from the previous chapter can be carried out very easily, for example with a spreadsheet. Nevertheless, a configuration that is close to the optimum could only be obtained by calculating the objective function for as many different configurations as possible. In addition, dependencies between the individual design criteria cannot be easily taken into account. Computer-aided design optimization can offer a solution to these problems [FRI08, p. 169].

It is necessary to create a calculation model, that reflects the influence of the variable design parameters on the objective function. In addition, the limits in which the variable parameters vary are defined. With the help of optimization algorithms, the design

parameters can be varied in such a way that the objective function yields an optimal value (see ch. 5.4). This procedure is called Multidisciplinary Design Optimization (MDO). MDO are considered as one of the most effective techniques in trade-off studies. That is because the optimum solution of a simultaneous problem is superior to the design found by optimizing each discipline sequentially, since it can exploit the interactions between the disciplines. [SAD12, p.70].

Fig. 3.8 shows an example of a MDO with three variable parameters. First, the upper and lower limits of the variable parameters are defined. Then, using a mathematical model, various design criteria are calculated (for example the maximum take-off weight m_{MTO}). Together with weighting factors, an objective function can be defined. This objective function can also include design criteria from sources other than the mathematical model. With the aid of an optimization algorithm, the variable parameters are then varied in such a way that the objective function produces an optimum value. This is done in an iterative manner. The values of the variable parameters at the end of the iterations define the optimal design.

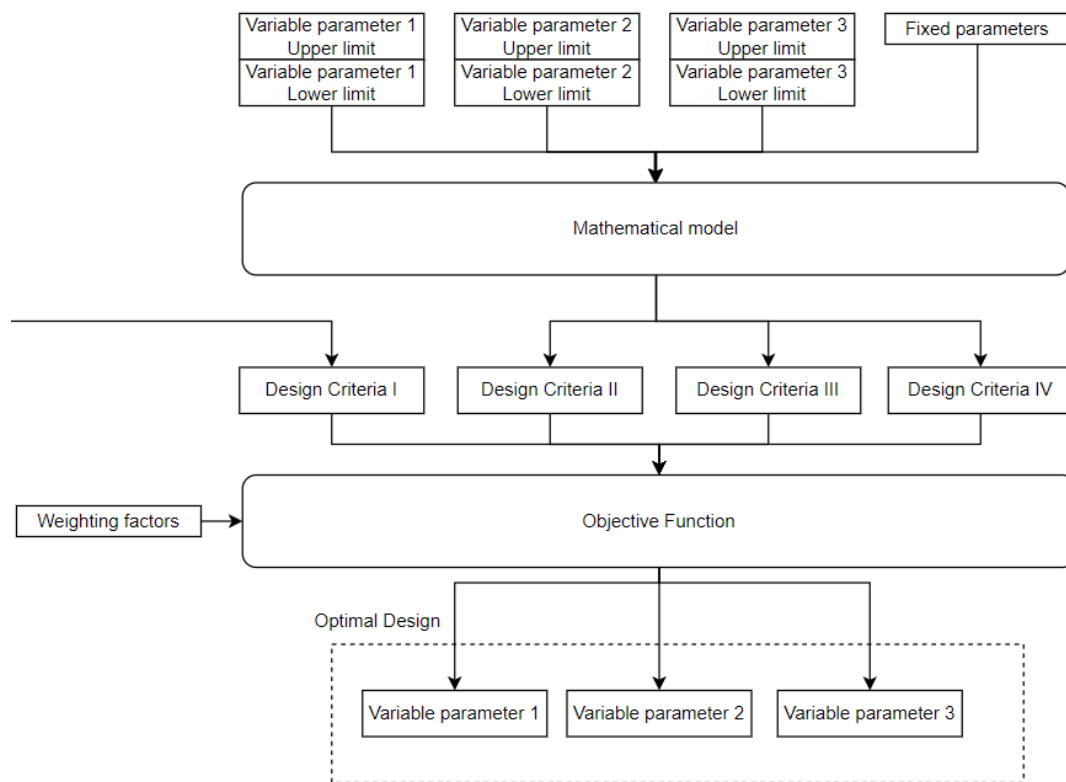


Figure 3.8: Exemplary MDO with three variable design parameters

Historically, the maximum-take off weight m_{MTO} was considered as the objective design criterion. That is because a reduced maximum-take off weight is intended to improve

performance and subsequently lower operating costs, primarily through reduced fuel consumption [SAD12, p. 52].

Therefore, the preliminary sizing method presented in ch. 3.3 can be used as a mathematical model for MDO (see mathematical model in fig. 3.8). Any input values of the preliminary design can then be used as a variable parameter. The output values of the preliminary design, such as the range R or the maximum take-off weight m_{MTO} , could then be considered in the objective function.

3.7 Systems Engineering in Aircraft Preliminary Sizing

In ch. 2, the fundamental principles of MBSE were introduced. In the previous chapters, the fundamentals of preliminary design process for civil passenger aircraft were presented. This chapter serves to establish a link between these two disciplines.

All challenges of system design listed in ch. 2.1.1 also apply to aircraft preliminary sizing. That is because aircraft design projects usually consist of complex, multi-disciplinary design problems with various constraints [SAD12, p. 37]. Besides, the design problems in aircraft design are often technologically challenging and mission-critical [FRI08, p. 4].

Another challenge are the numerous different stakeholders with different requirements involved in the aircraft design. Furthermore, the development teams are widely dispersed geographically and culturally. Many of the components of an aircraft are co-developed and manufactured by suppliers. Finally, almost every developed aircraft is unique. That is because different customer requirements result in many different aircraft configurations.

In order to establish requirement traceability and to follow the stakeholder requirements, it is beneficial to perform a MBSE approach from the very beginning of aircraft design. Especially at the beginning it is important to develop a good set of requirements. This way, a uniform basis from which all lower-level requirements can be developed can be defined [SAD12, p. 23]. Fig. 3.9 shows the influence of SE during the different design phases. It can be seen that the influence of SE decreases as the design becomes more detailed, while the aerospace engineering disciplines gain in importance.

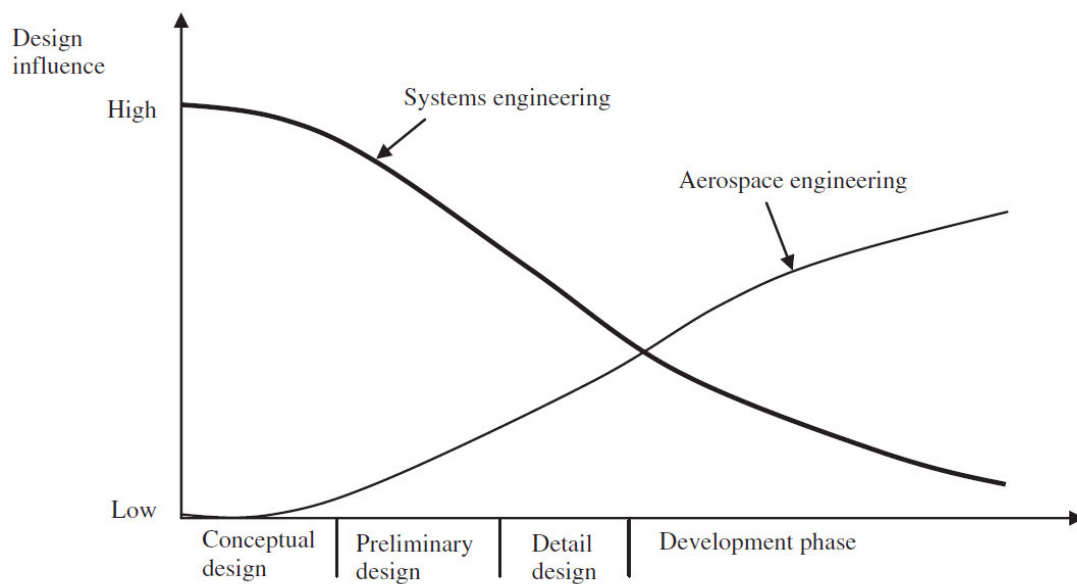


Figure 3.9: Systems engineering and aerospace engineering influence on the design (taken from [SAD12, Fig. 2.3])

Fig. 3.9 also shows that systems engineering still plays a major role in the preliminary design phase. Pursuing a MBSE approach during the preliminary design ensures that the design fulfils the stakeholder requirements and that undesirable designs are identified at an early stage.

MBSE enables the TLAR to be broken down to subsystem level. This permits early distribution of development tasks among different disciplines, which helps to manage the complexity of aircraft development. It also ensures that all the different design domains pursue the same design goal. MBSE also allows different aircraft configurations to be developed simultaneously at an early design stage.

3.8 Summary

The design of an aircraft starts with the determination of the TLAR and the mission definition. After that, the aircraft concept is developed in the conceptual design phase. The design is then further refined in the preliminary design phase. For this purpose, the method described in [LOF80] can be used. It reduces the preliminary design to the selection of two design parameters, that can be plotted in a matching chart. By selecting the two design parameters in the matching chart, important parameters such as the maximum take-off weight can be determined.

The entire design process is structured in a iterative manner and is validated by design

reviews after each design phase. Trade-off analyses can be used to select an optimal design. These can be extended by computer algorithms to perform a multidisciplinary design optimization (MDO). MBSE facilitates many of the challenges encountered in aircraft design. It plays a particularly important role in the early stages of aircraft development.

4 The PTC Integrity Modeler

4.1 Introduction

As already mentioned in chapter 2.2.2, there is different software for system modelling available. Many of the available modelling software is based on the SysML. For this thesis, the SysML-based modeller "PTC Integrity Modeler" was used, which will be briefly introduced in the following chapters. Detailed instructions on how to work with the PTC Integrity Modeler can be found in [PTC19a].

4.2 The PTC Integrity Modeler

The PTC Integrity Modeler is a modelling software that has its origins in the UML-based software development. Over time, the PTC Integrity Modeler has been extended to include SysML constructs. That allows the PTC Integrity Modeler to be used for SysML based MBSE. The PTC Integrity Modeler offers several additional features, such as product line variation modelling or the automatic document generation [PTC19a]. Since the PTC Integrity Modeler runs directly on an active multi-user database, multi-user capability with over 100 simultaneous users is possible.

4.3 User Interface

The PTC Integrity user interface can be divided into four parts: the *Browser Pane* (upper left), the *Diagram Pane* (upper right), the *Output Pane* (lower left) and the *Property Pages* (lower right). In addition, the *Modeler Ribbon* is located at the top end of the user interface. The *Modeler Ribbon* is where most of the Modeler's features can be accessed from.

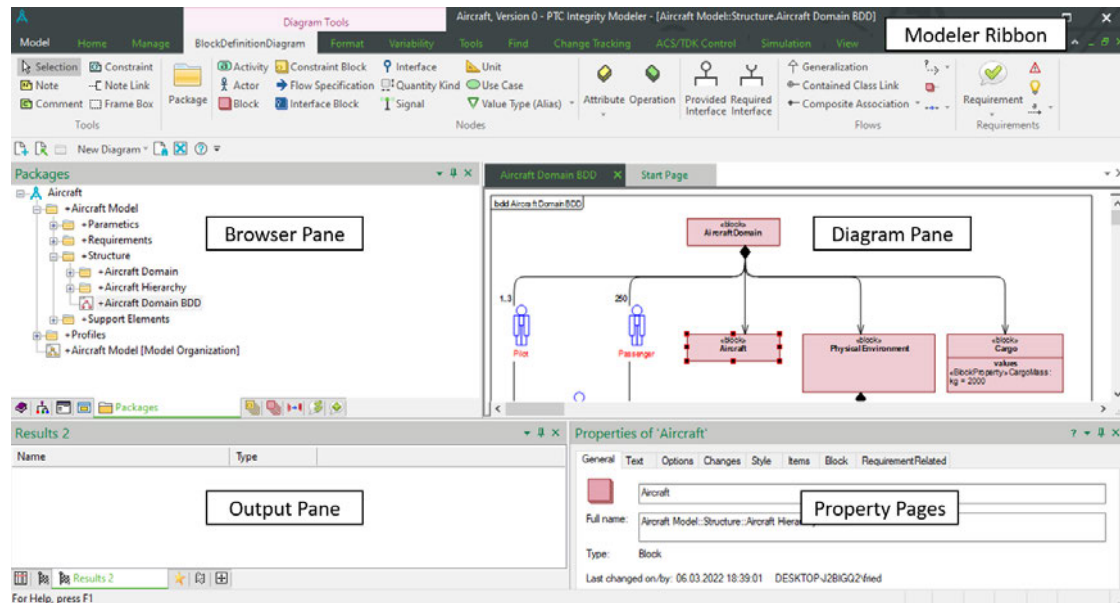


Figure 4.1: PTC Integrity Modeler user interface

In the following chapters, all components of the user interface and their functions will be briefly explained.

The Browser Pane

The *Browser Pane* provides access to the various items, diagrams and relationships defined within a model [PTC19b, p. 9]. It is divided into several tabs to quickly access certain SysML items (see fig. 4.2)



Figure 4.2: Different tabs within the browser pane

For example, the *Packages* tab can be used to access all the model packages (see fig. 4.2 & ch. 2.4.7). Similarly, the *Requirements* tab offers a quick overview of all the requirements specified in the model.

The Diagram Pane

The *Diagram Pane* is used to display and edit the UML/SysML diagrams. As stated earlier, the displayed items are actually symbols representing an underlying model item

in the model database. Consequently, when one wants to display or view the internal structure of the *Aircraft* block, one would have to open the corresponding internal block diagram first. All changes in the internal block diagram are then applied to the model repository and its underlying database.

The Property Pages

The *Property Pages* show the the properties of a selected model item. As already explained in ch. 2.3.5, different model items have different properties depending on their respective meta class.

The Output Pane

Like the browser pane, the output pane is divided into several tabs. The most important tabs and their function are the following:

- Contents tab: Displays the contents of a currently selected item (e.g. a package)
- Results and Results2 tab: Show the results from, for example, a model item search
- Output Tab: Provides a view of the Modeler's output information (e.g. status messages or error messages)

4.4 Modelling

Now that the graphical user interface of the PTC Integrity Modeler has been introduced, a short description on how to create digital system models using the PTC Integrity Modeler will be provided. Since the SysML is a graphical modelling language, the modelling is also done by the graphical representation provided by the SysML diagrams. Depending on which model item is to be modelled, a corresponding diagram is created first. The newly created diagram will then be displayed in the *Diagram Pane*. It can then be edited via the *Modeler Ribbon* (see fig. 4.1). All changes are immediately stored in the database. That way, the controlled multi-user operation is ensured.

4.5 The PTC Integrity Automation Interface

A special feature of PTC Integrity Modeler is the Automation Interface. It is essential for the program presented in ch. 5.5, and will therefore be briefly explained in this chapter. An automation interface is a mechanism for sharing information between applications or to control another application [PTC15, p. 2]. The PTC Automation Interface enables to perform the following operations from within a Visual Basic program:

- Read Modeler Model item structures, attributes and associations

- Write to Modeler model properties
- Create Modeler model items and links
- Control the Modeler user interface

Fig. 4.3 displays how a Visual Basic program interacts with the PTC Integrity Modeler via the PTC Automation Interface. The PTC Automation Interface provides certain functions to access the model objects. These functions allow the modification of model objects (see ch. 5.5). The Automation Interface then communicates the requests or changes with the PTC Modeler. The Modeler in turn controls the access to the underlying database (see fig. 4.3). Therefore, the user never directly interferes with the database. This prevents data corruption or unintentional modifications. In addition, the user only requires knowledge about the Automation Interface functionalities and not about the structure of the database itself.

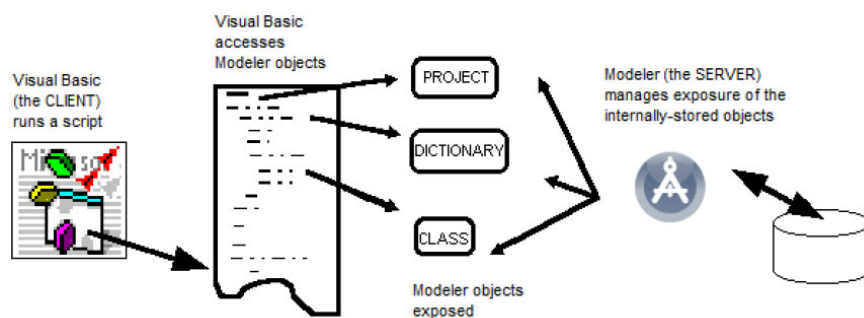


Figure 4.3: How Visual Basic programs interact with the PTC Integrity Modeler (taken from [PTC15])

The Automation Interface is single-threaded. That means that the operations can only be executed one after the other. In order to thesis with the PTC Automation Interface it is important to have an understanding of the Modeler’s meta model. Therefore, the meta model will be explained briefly in the next chapter.

4.6 The PTC Integrity Meta Model

Similar to the SysML meta model, the PTC Integrity meta model defines the item types (such as classes, attributes and operations) that exist in the model, their properties and how they relate to each other [PTC15, p. 2]. All Modeler objects are either of the object type, attribute type or association type. The three different types are presented in the following.

Objects

The object type defines the type of a Modeler item, such as *Class*, an *Actor* or a *Use Case* [PTC15, p. 3].

Attributes

Each object type has attributes that define the properties of the associated Modeler item, such as *Name*, *Visibility* and *Last Change Date* [PTC15, p. 3]. The attribute types can be read/write or read only. Furthermore, almost all meta model attributes are stored as strings. It is therefore necessary to convert the strings to numeric values before they can be mathematically processed.

Associations

Object associations are used to define relationships between model items. Associations are important to access from a given object to the associated objects. For example, the *Package Item* association of the *Package* object defines all items that are scoped to this package [PTC15, p. 258].

4.7 Navigation in a Model With the Automation Interface

In this chapter the information provided in the previous chapters will be applied. It will be illustrated how to navigate through a model in PTC Integrity Modeler using the Automation Interface. As an example, it will be shown how to access and change the description of a requirement.

When a Visual Basic application wants to use the Automation Interface, it typically starts with the following lines of code [PTC15, p. 6].

```
1 Dim Projects As Object
2 Projects = CreateObject("OMTE.Projects")
```

The *Projects* object acts as a container object for the *Projects* (i.e. digital models) that can be accessed. Afterwards, the active project can be accessed via the *ActiveProject* association (see code below). From there, it is possible to navigate to the project's *Dictionary* object via the *Dictionary* association. The *Dictionary* object is a container for all dictionary items in the model [PTC15, p. 6]. From the *Dictionary* object, the requirement can be accessed through a class association. According to the PTC Integrity meta model, requirements are defined as *Class* objects. The requirement description can be read out as an attribute of the *Requirement* object. The description can also be changed in a similar manner (see line 15 of the code below).

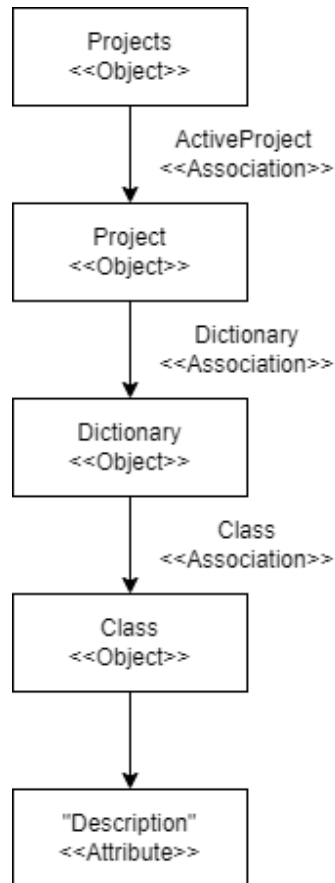


Figure 4.4: Accessing the description of a requirement via the PTC Integrity Modeler Automation Interface

```

1 Dim Projects As Object
2 Dim Project, Dictionary, Requirement As ENT6Lib.CCaseProjects
3 Dim RequirementDescription As String
4
5 'Access the description attribute
6 Projects = CreateObject("OMTE.Projects")
7 Project = Projects.Item("ActiveProject")
8 Dictionary = Project.Item("Dictionary")
9 Requirement = Dictionary.Item("Class", "RequirementName")
10
11 'Read requirement description
12 RequirementDescription = Requirement("Description")
13
14 'Change requirement description
15 Requirement("Description") = "NewDescription"
  
```

Listing 4.1: Visual Basic code to access the description of a requirement and change it

4.8 Controlling the User Interface

As mentioned earlier, the PTC Integrity Automation Interface also provides functions to control the Modeler's user interface. Some of the functions used in the program presented in ch. 5.5 will be briefly introduced in this chapter.

In order to access the Modeler from external applications, a root object of the following form must be created first.

```
1 Dim Studio As Object
2 Studio = CreateObject("Studio.Editor")
```

Controlling the Results Pane

As already mentioned in ch. 4.3, the *Results Pane* is located in the lower left corner of the user interface and displays the model items resulting from certain queries (see fig. 4.1). The *Results Pane* can be cleared from an external application with the following line of code.

```
1 Studio.ClearResultsPane(1)
```

It is also possible to add certain model items to the *Results Pane*. For this, the so called *Object ID* of the object to be added is required. The *Object ID* is a string of characters that uniquely identifies the model object.

```
1 Studio.AddToResultsPane(1, "ObjectId")
```

Controlling the Output Window

The *Output Window* provides the user with information as like status messages or error messages (see fig. 4.1). It can be cleared with the following line of code.

```
1 Studio.ClearOutputWindow
```

Furthermore, it is possible to add output messages to the *Output Window* with the following line of code.

```
1 Studio.DisplayOutputWindowMessage("Message")
```

4.9 Summary

PTC Integrity Modeler is a database-driven modelling tool that supports, among other things, SysML-based MBSE. Via the Automation Interface it is possible to access model objects from external Visual Basic programs. The navigation to specific model items requires knowledge about the Modeler's meta model. In addition, the Modeler itself can be accessed via the Automation Interface. This enables, for example, to adjust the Modeler's user interface.

5 Establishment of the Interface

5.1 Introduction

Ch. 2 has introduced the principles of MBSE and the SysML. In ch. 3, a method for preliminary sizing of civil jet airliners was presented. Furthermore, it was discussed why MBSE is of particular importance for aircraft preliminary sizing. In the previous chapter, a software for SysML-based MBSE was presented.

This chapter aims to create a link between the system modelling and system analysis disciplines. Therefore, an exemplary aircraft is modelled with the SysML. The PTC Integrity Modeler presented in ch. 5.3 is used for this. Furthermore, a MATLAB code for the preliminary sizing of civil jet airliners is presented in ch. 5.4.

After introducing both the SysML based model definition and the MATLAB based computational model, the advantages of digitally linking both models will be mentioned in ch. 5.5.2. After that, different solutions for a digital interface between both models will be discussed in ch. 5.5.3. In ch. 5.5.4, the chosen solution will be presented.

5.2 Overview

The previous chapters have demonstrated the importance of MBSE in the preliminary sizing of aircraft. This can be done with the PTC Integrity Modeler.

At the same time, computer-aided calculation programs such as MATLAB or Simulink are irreplaceable for aircraft design. For example, computer-aided design optimization would not be possible without a powerful mathematics engine such as MATLAB. Consequently, there are two models of the aircraft - one in SysML and one computational model in MATLAB. However, it is not possible to transfer the parameters without further ado (see fig. 5.1). A digital interface for the automatic parameter transfer between both models could be a solution. This could lead to many advantages, which are described in ch. 5.5.2.

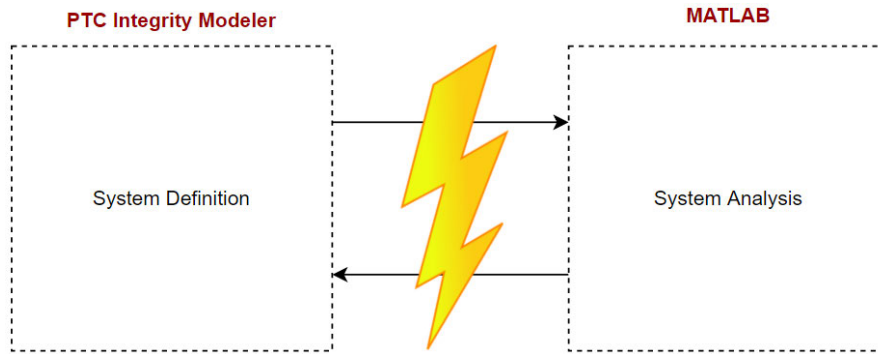


Figure 5.1: Problems arising with different software landscapes during system definition and system analysis

In order to provide a better understanding of the relationship between the models, this chapter will provide a basic overview first. This is intended to clarify the relationships between the two different models and the interface. In the following chapters, the two models and the interface will be explained in more detail.

Fig. 5.2 shows a detailed view of the parameter transfer between the system definition software (PTC Integrity Modeler) and the system analysis software (MATLAB). All of the input parameters required for the preliminary sizing (coloured orange in fig. 5.2) are stored in the PTC Integrity system model. They are provided to MATLAB through a digital interface (red block in fig. 5.2)). After the calculation is performed within MATLAB, the output values (coloured green in fig. 5.2) are provided back to the system model via the interface.

As mentioned earlier, the preliminary sizing process described in ch. 3 leads to five sets of relationships between the wing loading and the thrust-to weight ratio at take-off. The five relationships are represented by the five differently coloured rectangles on the right of fig. 5.2. As previously stated, the five relationships lead to a two dimensional optimization problem that can be displayed in a matching chart (see fig. 5.2). If at least one additional design parameter is left to be variable, a multidimensional design problem arises. The optimal solution can then no longer be read from the matching chart, but must be determined mathematically by an optimization algorithm. This is represented by the "*Further Calculation / Iteration*" block below the matching chart in fig. 5.2.

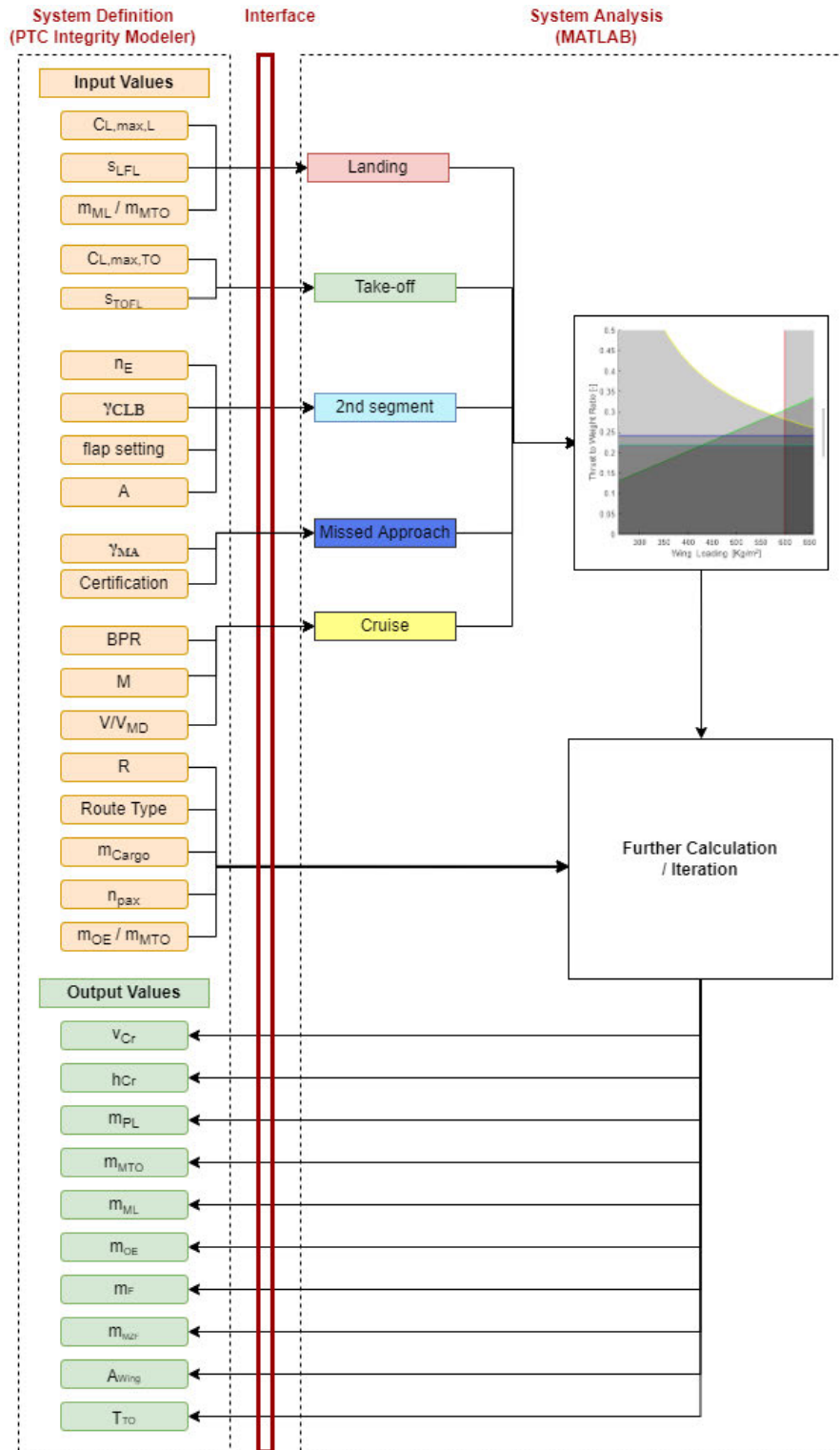


Figure 5.2: Exchange of parameters between both models

The input parameters are distributed in different model items within the aircraft model. Thus, a wide range of input possibilities is demonstrated. The following SysML model items were used to store input parameters:

- Requirements
- Constraints
- Associations
- Block properties

Fig. 5.3 shows where exactly the input parameters are stored in the SysML model. All design parameters representing quantities were modelled as associations (n_E , n_{PAX}). The upper and lower limits of variable parameters are stored as constraints (see fig. 5.3). This is the case for the two parameters displayed in the matching chart, $T_{TO}/(m_{MTO} \cdot g)$ and m_{MTO}/S_W . In order to demonstrate that this approach can also be applied to multidimensional optimization problems, the Mach number M and the ratio between the cruise speed and the minimum drag speed V_{Cr}/V_{md} were left variable as well. Fig. 5.3 also illustrates that the optimal values for the four variable parameters are stored at a block property after the optimization is finished (see bottom left output values in fig. 5.3).

To demonstrate both possibilities, the remaining input values are stored as SysML block properties as well as directly within SysML requirements. The reason for modelling input parameters directly in SysML requirement items will be explained in chapter 5.5.7. For the sake of simplicity, all output parameters are stored as block properties (see fig. 5.3). It would however also be possible to store the output parameters in other model items.

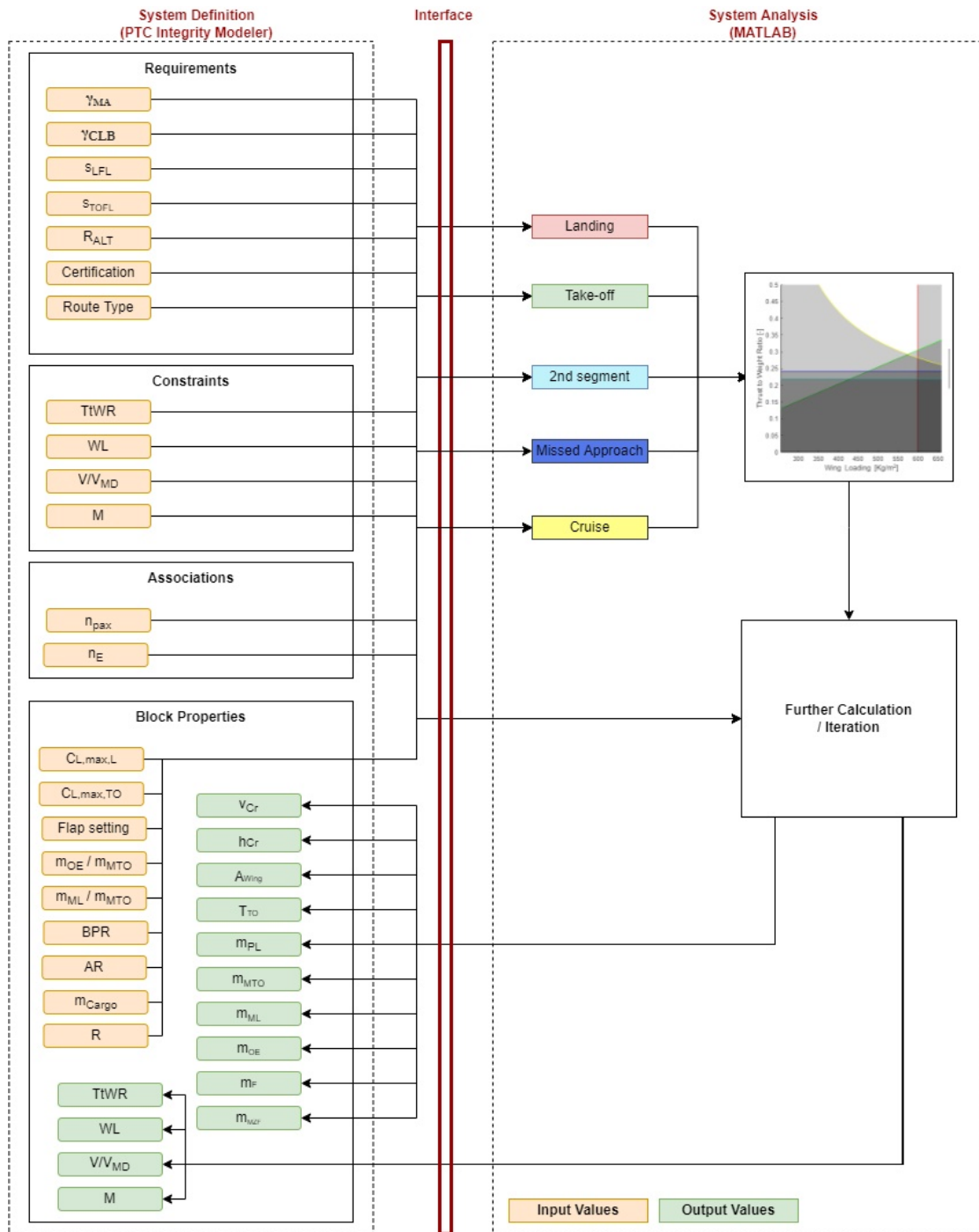


Figure 5.3: Exchange of parameters between both models (more detailed)

5.3 Modelling the System in SysML

Now that the transfer of parameters has been defined, the modelling can begin. First the SysML aircraft model is modelled (left part of fig. 5.2 and 5.3). The PTC Integrity Modeler is used for modelling. For simplicity, only the model items that are either required for preliminary sizing or are important for comprehension are modelled.

5.3.1 Loading the SysML Profile

As already mentioned in chapter 2.3.3, the UML can be extended by profiles for extended functionalities. Since the aircraft model is modelled in SysML, the SysML profile has to be loaded in the Modeler (see fig. 5.4).

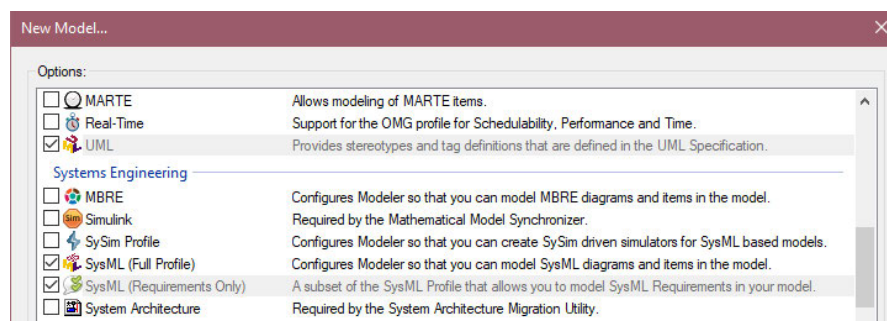


Figure 5.4: Selecting the SysML profile in the PTC Integrity Modeler

5.3.2 Packages

As explained in ch. 2.4.7, packages are used to organize the model items. The requirements (see ch. 5.3.3) are stored in the *Requirements* folder. All blocks and block properties can be found in the *Structure* folder (see ch. 5.3.4). All constraint blocks are stored in the *Parametrics* folder (see ch. 5.3.9). The data types required for the model are stored in the *Aircraft Types* folder (see ch. 5.3.7). The two custom stereotypes are stored in the *Stereotype* folder (see ch. 5.3.8). Fig. 5.5 shows the aircraft model package structure.

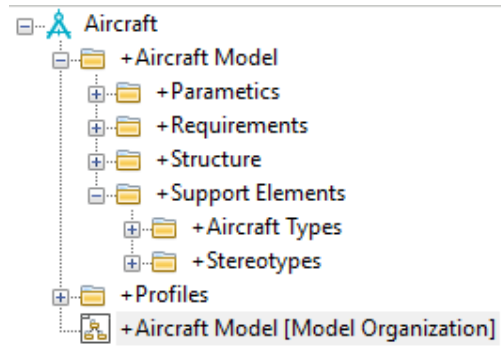


Figure 5.5: Aircraft model package structure

In the following chapters the contents of the individual packages will be described.

5.3.3 Requirements

As shown in fig. 5.3, the following parameters are saved directly within the respective requirement description:

- The landing field length s_{LFL}
- The take-off field length s_{TOFL}
- The alternate range R_{Alt}
- The steady gradient of climb during the 2nd segment γ_{CLB}
- The steady gradient of climb during missed approach γ_{MA} ,
- The certification basis
- The route type

The PTC Integrity Modeler provides the synchronization between the Modeler and certain requirement management tools [PTC19b, p.31]. However, for this thesis it is sufficient to model the requirements directly in the *Aircraft* model. The requirements are divided into the following categories (see [SAD12, p.33]) via a *Requirement Nesting Link* (see ch. 2.4.3):

- Airworthiness requirements
- Performance requirements
- Operational requirements

The nesting association is also used to further break down requirements (see fig. 5.6). The parameters for each requirement (pink in fig. 5.6) are stored between two square brackets. That way, the parameters required for the subsequent calculation can be read correctly (see sec. 5.5.7).

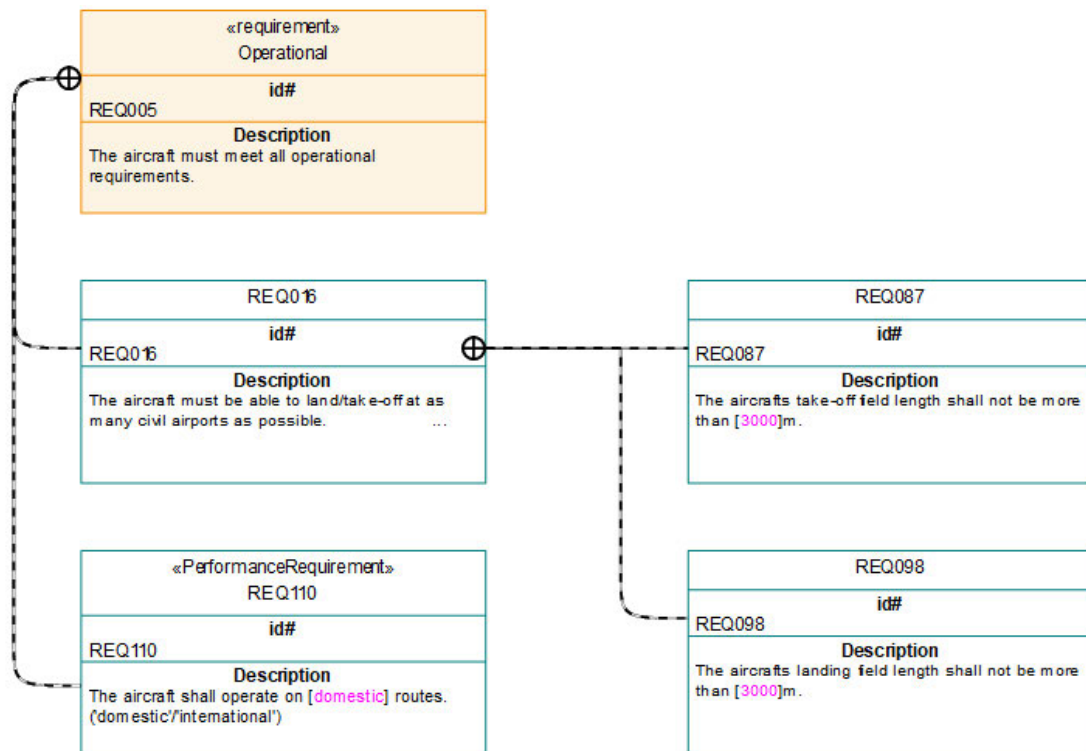


Figure 5.6: Section from the requirement diagram (operational requirements)

5.3.4 Block Definition

The *Aircraft* block is modelled as a part of the *Aircraft Domain* block through a composite association. By modelling the *Aircraft Domain*, additional parameters can be included in the calculation that do not belong to the *Aircraft* block itself. For example, the number of passengers n_{PAX} is modelled as the *Passenger* cardinality (quantity) of the association "*Aircraft Domain* - *Passenger*" (see fig. 5.7). Besides, the cargo weight m_{CARGO} is modelled as a value property of the *Cargo* block, which itself is associated with the *Aircraft Domain* block. Furthermore, the landing field-length s_{LFL} and take-off field length s_{TOFL} are modelled as value properties of the *Airport* block, which is also modelled as a part of the *Aircraft Domain* (see fig. 5.7).

As mentioned in the previous chapter, the values for s_{LFL} and s_{TOFL} are read directly from within the requirements (see fig. 5.6). The value properties seen in fig. 5.7 are modelled for the completeness and are not required for the calculation (see ch. 5.3.5).

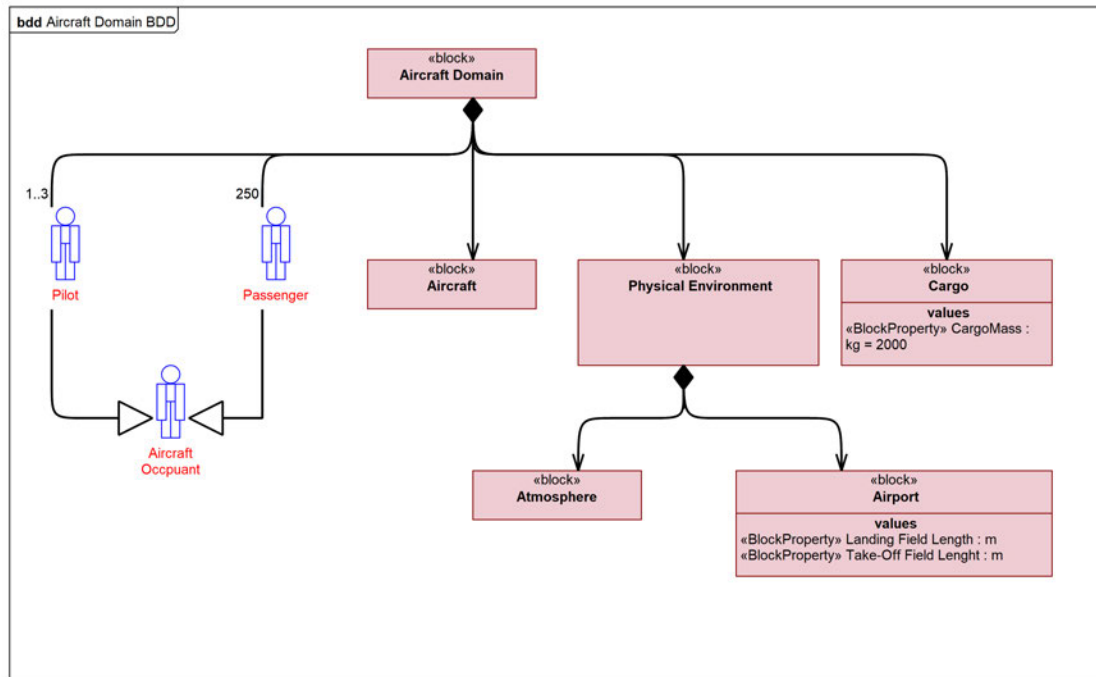


Figure 5.7: The *Aircraft Domain* block definition diagram

5.3.5 Internal Block Structure

Airport Block

The airport block has two value properties, the landing field-length s_{LFL} and the take-off field length s_{TOFL} . The numerical values are saved within requirements that are allocated to the respective value properties (see fig. 5.8). By reading the numerical values directly from the requirements instead of the value properties, additional manual work is avoided (see ch. 5.5.7).

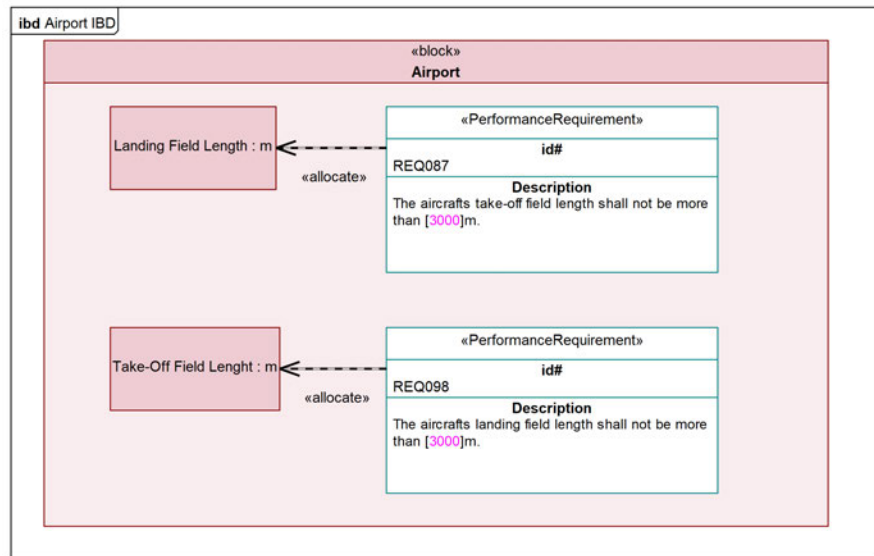


Figure 5.8: The internal block diagram of the *Airport* block

Aircraft Block

All properties of the aircraft itself were modelled as value properties of the aircraft block. The value itself is stored within the allocated requirement for five of the parameters (see ch. 5.3.3). For consistency, the associated value properties are created as well:

- The alternate range R_{Alt}
- The steady gradient of climb during the 2nd segment γ_{CLB}
- The steady gradient of climb during missed approach γ_{MA} ,
- The certification basis
- The route type

The aircraft's design range R serves as an input value for the calculation and is modelled as a value property of the aircraft block. Furthermore, some of the output values are saved as value properties of the aircraft block (see fig. 5.3):

- The cruise speed V_{Cr}
- The cruise attitude h_{Cr}
- The cruise lift coefficient $C_{L,Cr}$
- The glide ratio during cruise E_{Cr}
- The thrust-to-weight ratio during take-off $T_{TO}/(m_{MTO} \cdot g)$
- The wing loading during take off m_{MTO}/S_W

- The cruise Mach number M
- The ratio between the cruise speed and the minimum drag speed V_{Cr}/V_{md}

In addition, the *Aircraft* block contains part properties, such as two engines, a wing, fuel tanks and the *MassPrediction* block. They will be explained in the following chapters. The internal structure of the aircraft block is modelled by the internal block diagram shown in fig. 5.9. The constraints displayed in fig. 5.9 will be explained in ch. 5.3.6.

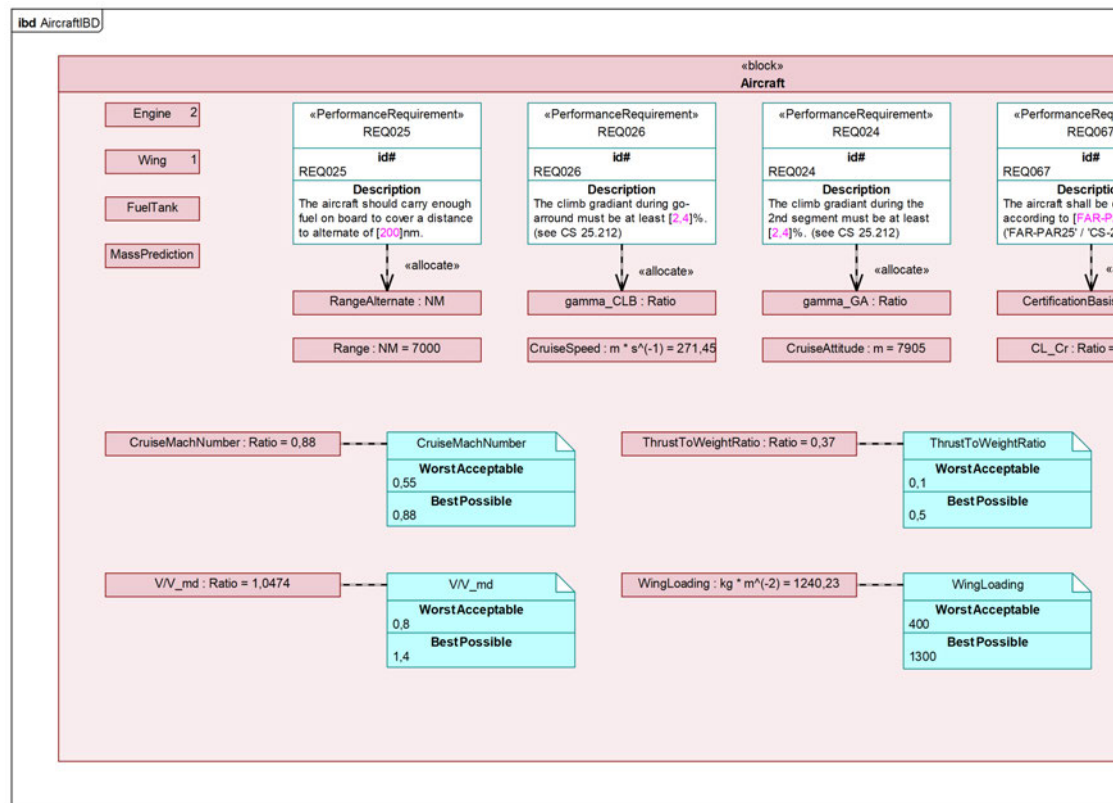


Figure 5.9: Section of the internal block diagram of the *Aircraft* block

The composition of the *Aircraft* block was modelled with a block definition diagram (see fig. 5.10). The value properties defined in the *Aircraft* internal block diagram (fig. 5.9) are displayed within the aircraft block. Furthermore, the *Engine*, *Wing*, *FuelTank* and *MassPrediction* blocks are connected to the *Aircraft* block via a composite association. The amount of engines (n_E) is saved as the *Engine* cardinality (quantity) of the association "*Aircraft* - *Engine*".

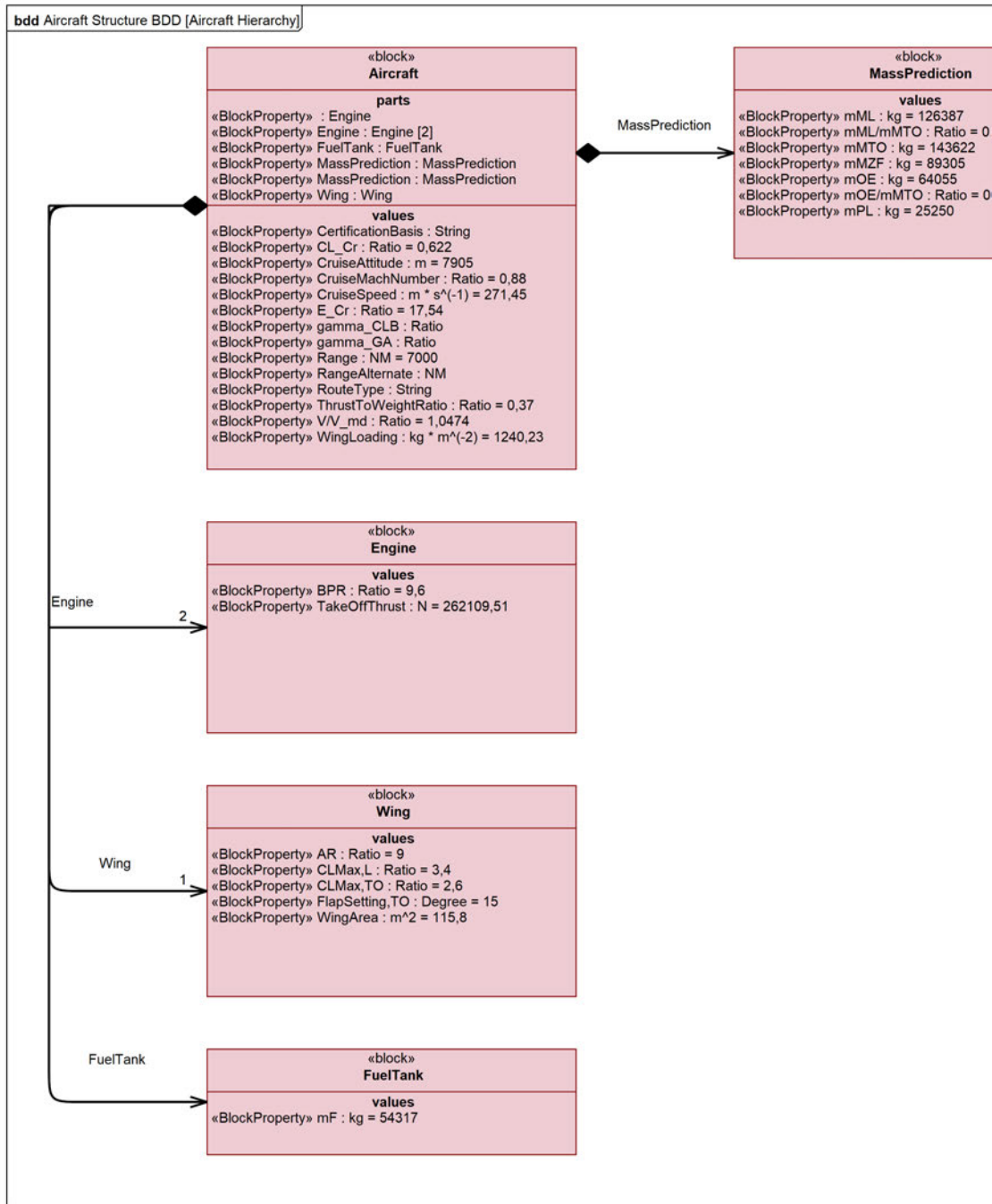


Figure 5.10: Section of the block definition diagram of the *Aircraft* block

Engine Block

The *Engine* block contains the engine by-pass-ratio μ as a value property. The output value for the take-off thrust for one engine T_{TO} is stored as a value property of the *Engine* block (see fig. 5.11):

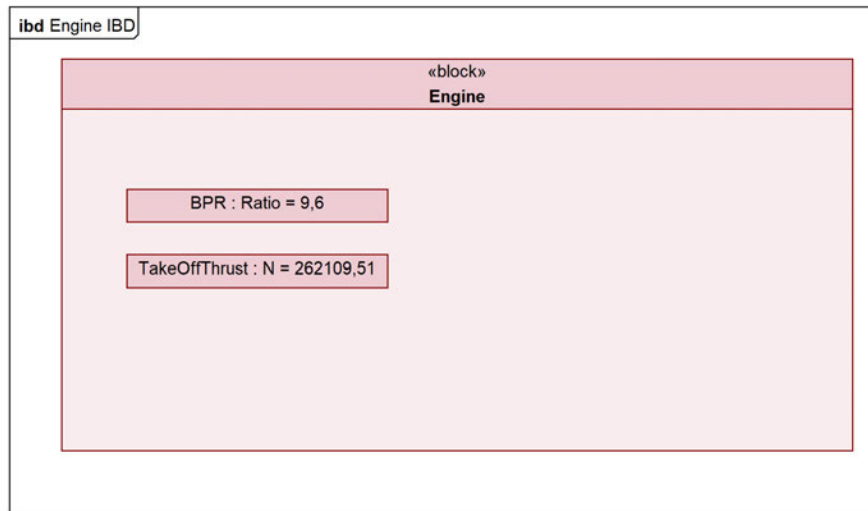


Figure 5.11: The internal block diagram of the *Engine* block

FuelTank Block

The *Fueltank* block contains the value property for the fuel mass m_F , that is returned by the preliminary design as an output value (see fig. 5.2 & fig. 5.12).

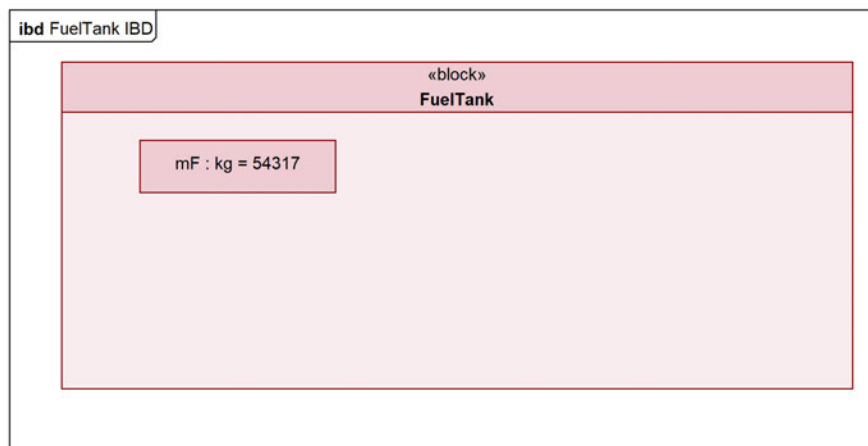


Figure 5.12: The internal block diagram of the *Fueltank* block

Wing Block

The *Wing* block contains the value properties for the maximum lift coefficient during landing $C_{L,max,L}$, the maximum lift coefficient during take-off $C_{L,max,TO}$, the flap setting during take-off and the wing aspect ratio A as input values. They are stored as value properties of the *Wing* block. Furthermore, the wing reference area S_W is stored as a value property of the *Wing* block. Unlike the other value properties, the wing reference area is an output value of the preliminary design (see fig. 5.2).

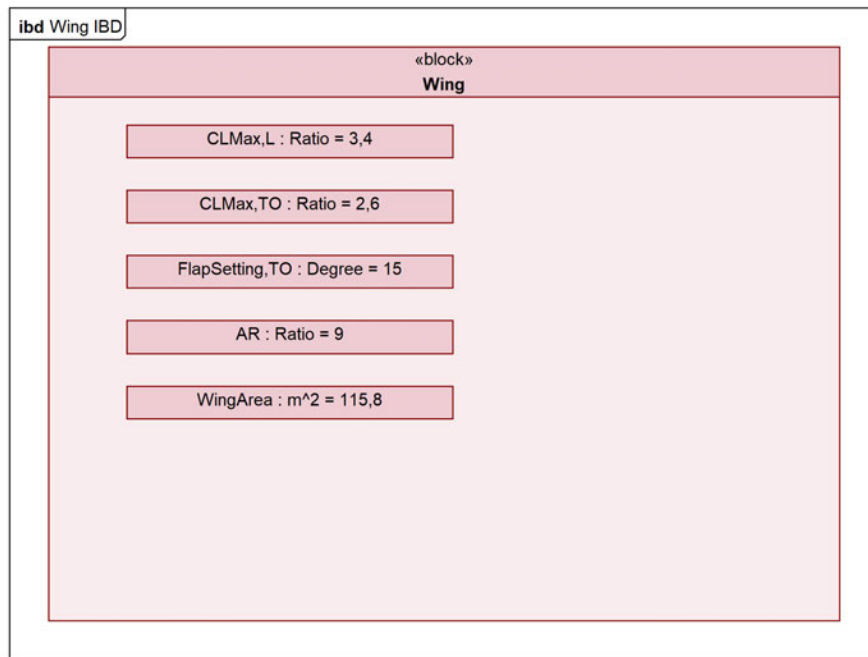


Figure 5.13: The internal block diagram of the *Wing* block

MassPrediction Block

The *MassPrediction* block contains all mass relevant parameters of the aircraft. The two mass ratios m_{OE}/m_{MTO} and m_{ML}/m_{MTO} serve as input values for the preliminary design and are stored as value properties. The other value properties modelled in the *MassPrediction* internal block diagram are all output values:

- The payload m_{PL}
- The maximum take-off weight m_{MTO}
- The maximum landing weight m_{ML}
- The maximum zero fuel weight m_{MZF}
- The operational empty weight m_{OE}

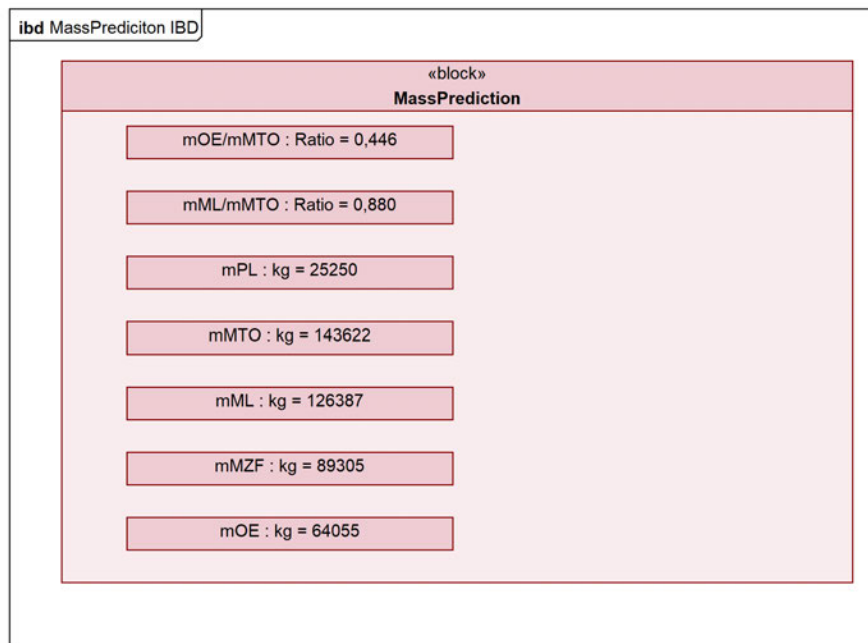


Figure 5.14: The internal block diagram of the *MassPrediction* block

5.3.6 Constraints

As shown in fig. 5.3, four design parameters are left to be variable. They are modelled as constraints that constrain the corresponding value properties (see fig. 5.15). The upper and lower limit of the constraints are later passed to MATLAB. The MATLAB algorithm then calculates an optimal design by changing the parameters within their respective limit (see ch. 3.6). The optimal value is then stored in the corresponding value properties after the optimization.

Fig. 5.15 shows the four SysML constraints containing the lower limits (*Worst Acceptable*) and the upper limits (*Best Possible*). Unfortunately, the PTC Integrity Modeler does not support a neutral indication of the upper and lower limit. In some cases this can lead to confusion. For example, the lower limit of the thrust-to-weight ratio is actually the desired optimum (see ch. 3.3.3). However, the lower limit is stored as "*Worst Acceptable*" (see fig. 5.15). The same applies to variable parameters where it is not possible to define the superior limit of the upper and lower limit. This is the case for the cruise Mach number M or the ratio between the cruise speed and the minimum drag speed V_{Cr}/V_{md} . Therefore, for this thesis, the *Worst Acceptable* value is always considered as the lower limit and the *Best Possible* value as the upper limit of a constraint.

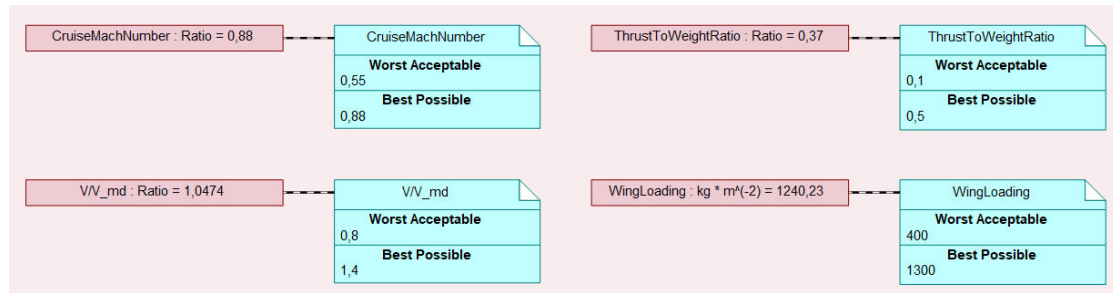


Figure 5.15: Modelling of the constraints within an internal block diagram

Fig. 5.15 shows how the constraints are linked to the block properties where the optimal value is stored later. This way it would be directly noticeable if a calculated value is outside the predefined limits.

5.3.7 Data Types

The data types required to type the value properties of the model are stored in the *Aircraft Types* package (see fig. 5.5). They are modelled with the block definition diagram shown in fig. 5.16.

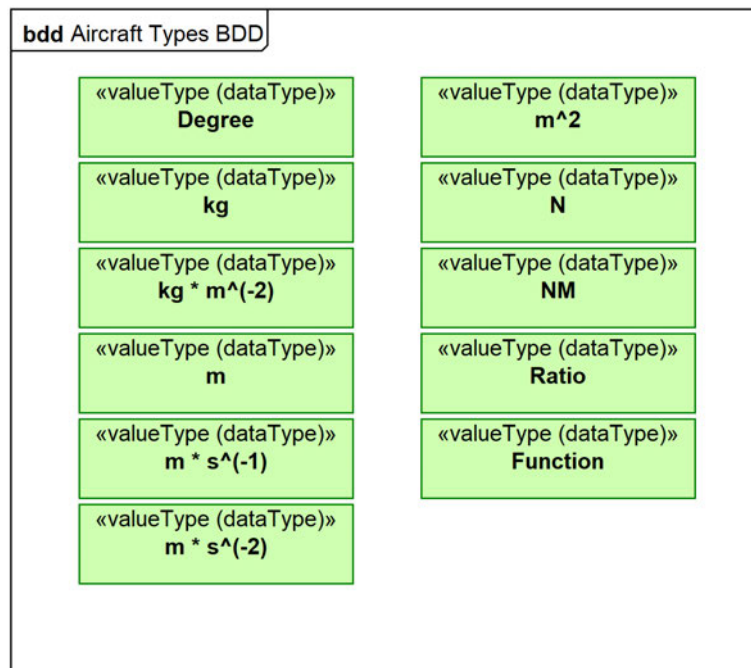


Figure 5.16: Used data types for the *Aircraft* model, modelled with a block definition diagram

5.3.8 Stereotypes

For clarity, all parameters stored in the model are designated with either *MATLABInput* or *MATLABOutput*. Thus, for each parameter it is evident whether it is required as input value for the preliminary design or whether it originates from it. The stereotypes are saved in the *Stereotypes* package (see fig. 5.17).

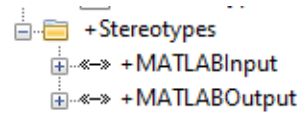


Figure 5.17: Custom stereotypes to label model items as input or output

5.3.9 Parametric Diagram

At this point, all the information required for the preliminary design is stored in the system model (see fig. 5.3). This theoretically completes the modelling of the aircraft model as far as it is relevant for the preliminary design. Therefore, the modelling of the mathematical relationships can theoretically be initiated. Before doing so, however, the diagram specifically intended for modelling the mathematical relationships - the parametric diagram - should be briefly discussed (see ch. 2.4.6).

An effort was made to model the relationships between the parameters shown in fig. 5.2 in a parametric diagram. First, the design constraints resulting from the five different flight phases were modelled as constraint properties. Fig. 5.18 shows the section of the parametric diagram that represents the design constraint for the landing (eq. (3.1) & red curve in fig. 3.5). The arrow directions indicate whether the parameters serve as an input or output of the equation.

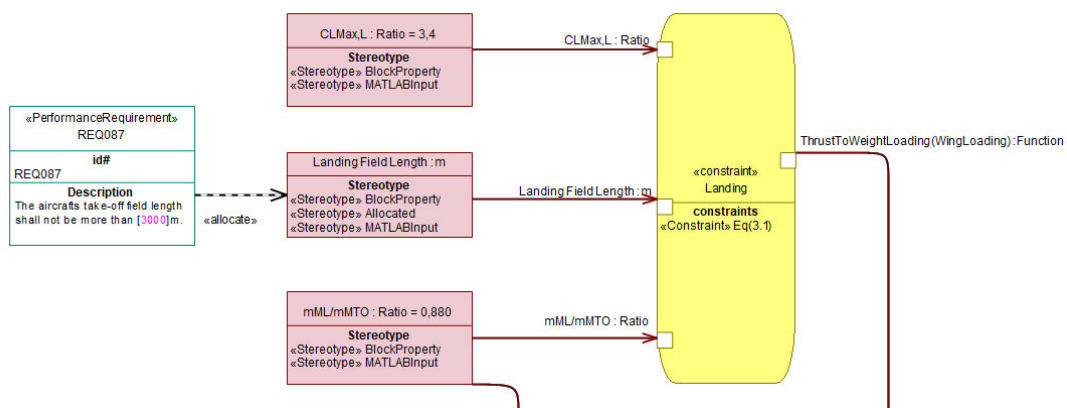


Figure 5.18: Section of the parametric diagram, *Constraint Property* for the landing design constraint

On the left side of the five constraint properties, the input values are modelled as constraint parameters. The respective constraint parameters are connected to the respective value properties via a binding connector (see ch. 2.4.6). On the right side of each of the five constraints, a functional relationship between the thrust-to weight-ratio during take-off $T_{TO}/(m_{MTO} \cdot g)$ and the wing loading at take-off m_{MTO}/S_W is modelled. This functional relationship is displayed in the matching chart (see fig. 3.5).

Next to the five constraint blocks, another constraint block resembling the matching chart itself is modelled (see fig. 5.19). Again, all input parameters are recognizable by the arrow direction. Among others, the functional relationship between the thrust-to-weight ratio and the wing loading during take-off is modelled as a constraint parameter. It is connected to the other constraint property via a binding connector.

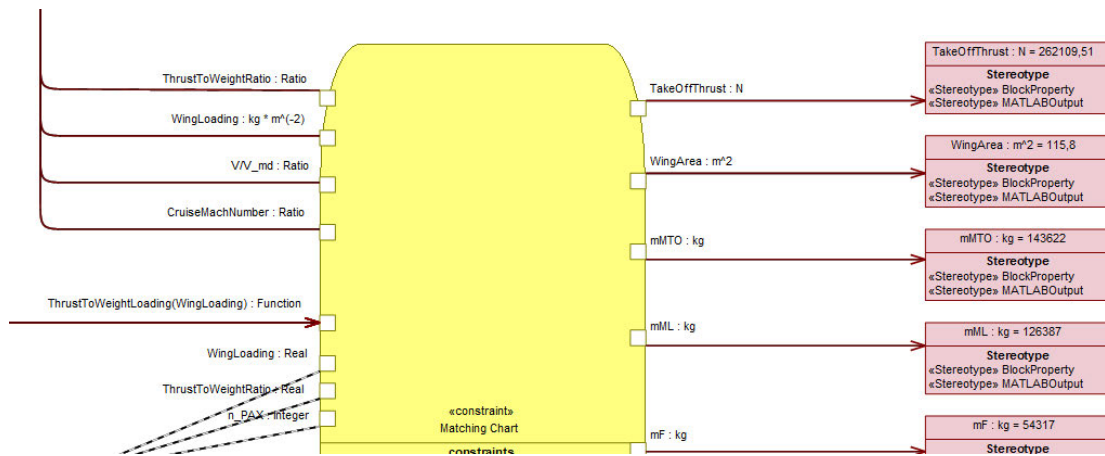


Figure 5.19: Section of the parametric diagram, *Constraint Property* for the matching chart

The value properties of the output parameters are located to the right of the *Matching Chart* constraint property. They are also connected to the *Matching Chart* constraint property with binding connectors. In addition, the optimized values for the four variable parameters are modelled as constraint parameters (upper left corner of the *Matching Chart* constraint property). Again, the constraint parameters are connected to the respective value properties with a binding connector (see fig. 5.20). As for the *Aircraft* internal block diagram, the value properties are connected to the corresponding constraints via a note link. This means that input values (upper and lower limits) are displayed directly next to the optimized output values.

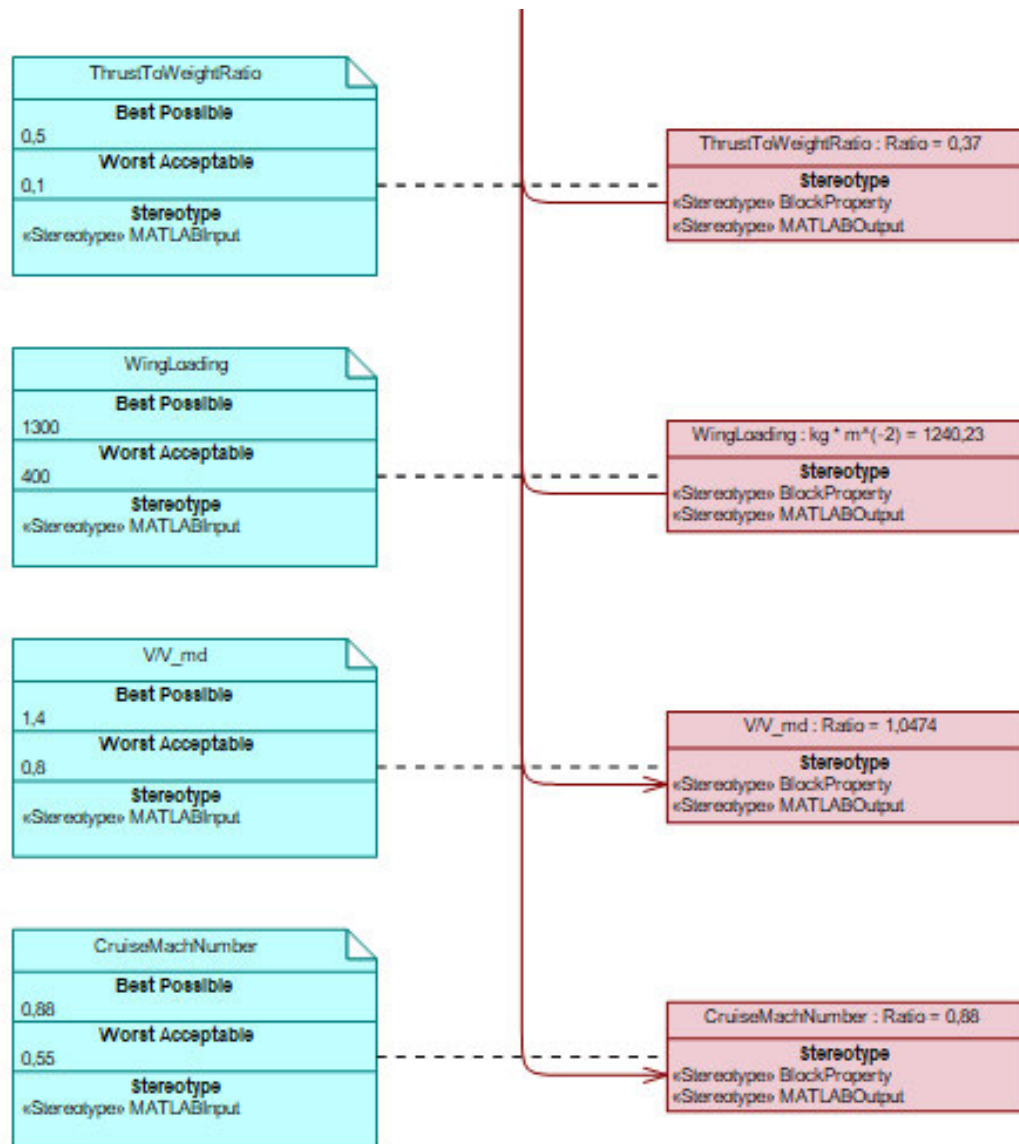


Figure 5.20: Section of the parametric diagram, value properties and constraints of the variable parameters

However, there are a few discrepancies between fig. 5.3 and the parametric diagram that cannot be avoided. The calculation model contains some input parameters, that are stored in model items, which cannot be represented in a parametric diagram. All parameters defined in composite associations cannot be displayed. This concerns the amount of engines n_E and the amount of passengers n_{pax} . Certainly these parameters could be additionally modelled as value properties, in order to display them in the parametric diagram. However, the same information would then be available twice in

the model, which could lead to inconsistencies.

Furthermore, it could not be modelled that the lower and upper limits of the variable parameters are also included in the calculation (see fig. 5.20). Also, the iterative procedure of the calculation could not be modelled.

It is important to emphasize that the parametric diagram was primarily created to simplify understanding and to model the relationships between parameters directly in the SysML model. Due to the modelling difficulties mentioned above, the SysML model does not fully comply with the mathematical model presented in the next chapter. Also, the computational model is not connected to the parametric diagram in any way.

5.4 Mathematical Model of the System in MATLAB

5.4.1 Overview

Now that the system has been modelled using the SysML (left part in fig. 5.2 & 5.2), the next step is to analyse the system (right part in fig. 5.2 & 5.3). The analysis is performed by a MATLAB code, which will be presented in this chapter.

The MATLAB code uses the input parameters modelled in the previous chapter (coloured orange in fig. 5.2 & 5.3). The input parameters are required to preliminary design an aircraft with minimum maximum take-off weight m_{MTO} and minimum thrust-to-weight ratio during take-off $T_{TO}/(m_{MTO} \cdot g)$. For this, the code makes use of MATLAB's abilities to find a minimum of a constrained non-linear multi-variable function. The output values of the MATLAB calculation are the output values of the preliminary design method presented in chapter 3.3, that are required for the following design stages (coloured green in fig. 5.2 & 5.3).

Additionally, the MATLAB program plots the matching chart as well as the iteration progress. Fig. 5.21 shows the window that opens when the calculation is finished. The upper plot shows the matching chart introduced in ch. 3.3. The large black circle shows the design point, i.e. the optimal design at the end of the optimization.

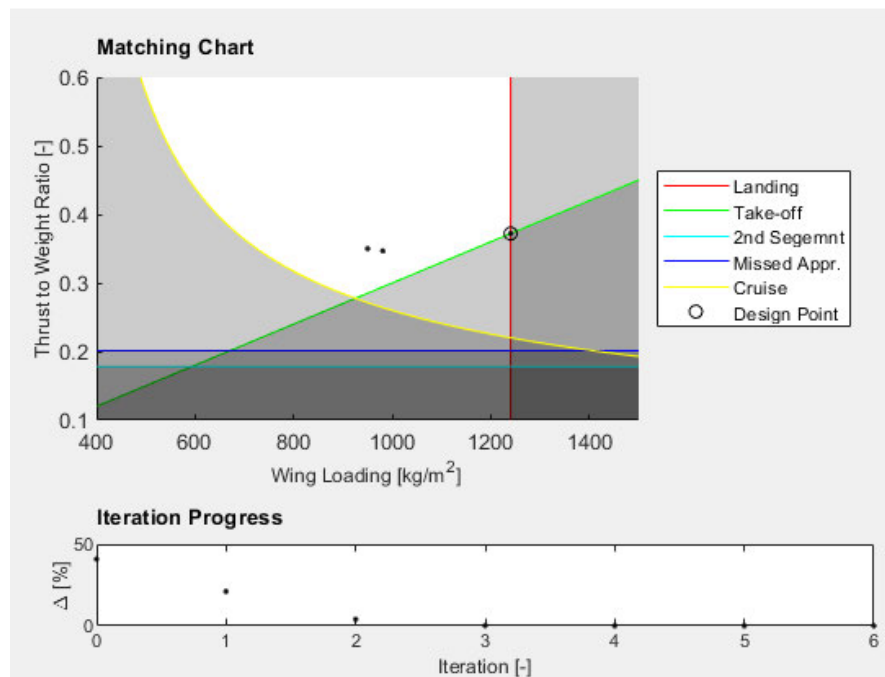


Figure 5.21: Output of the MATLAB program

The small black dots show the design points of each iteration. The lower plot shows the percentage reduction at each iteration of the objective function. The objective function

is defined as the the sum of the maximum take-off weight m_{MTO} and the wing loading at take-off m_{MTO}/S_W . Therefore, the optimal solution is the minimum of the objective function.

In fact, is it not necessary to plot the matching chart, since the optimal design is calculated in the background and not determined from the matching chart. Nevertheless, the chart is more illustrative. For instance, it shows which of the five flight phases significantly constrains the design. In addition, the iteration progress in the matching chart indicates whether the MDO has led to a realistic outcome.

The structure of the MATLAB program is shown in fig 5.22. The individual elements of the code will be explained in detail in the following chapters.

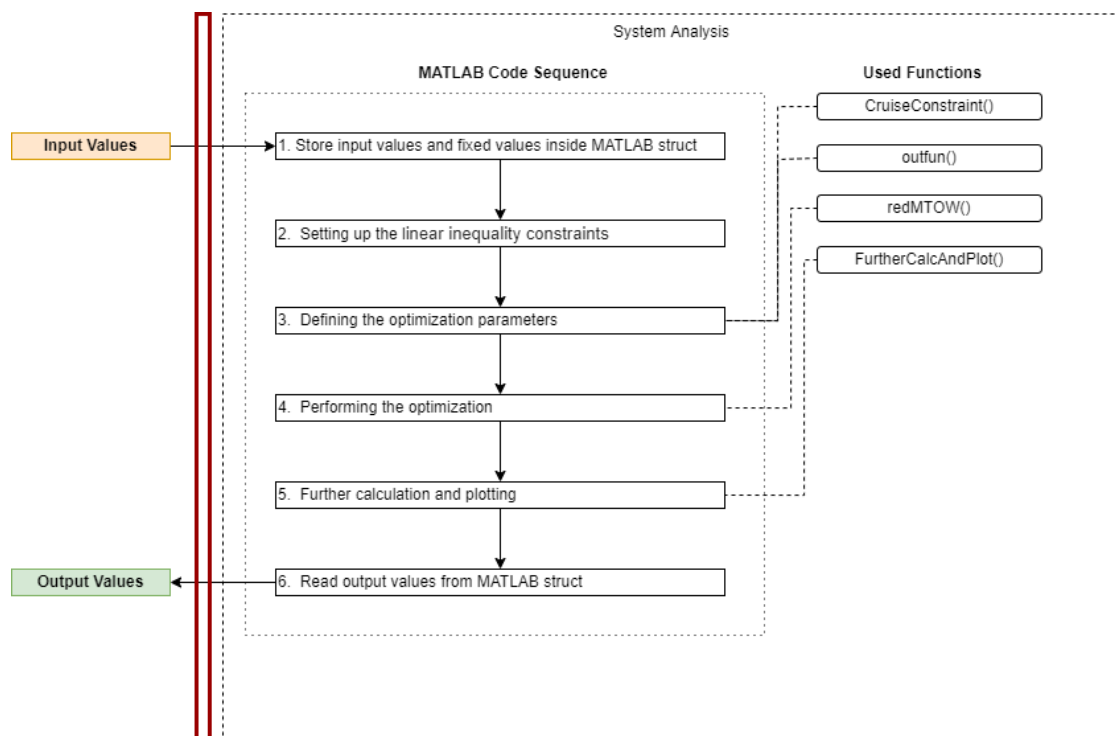


Figure 5.22: Structure of the MATLAB code

1. Store Input Values and Fixed Values Inside MATLAB Struct

The first step is to save all input parameters and the fixed, predefined, values in a MATLAB structure array (see step 1 in fig. 5.22). This is done within the lines 1 - 41 in App. B. A structure array is a data type that groups related data using data containers called fields. Each field can contain any type of data [Mat22]. Consequently, all parameters are stored in one array only. Thereby the transfer to and the modification of parameters by other functions are simplified.

2. Setting up the Linear Inequality Constraints

In the second step the linear inequality constraints, that must be fulfilled by the optimal solution, are set up (see step 2 in fig. 5.22). This is done within the lines 44 - 73 in App. B . The linear inequality constraint defined in line 46 corresponds to eq.(3.1). The constraint defined in lines 49 - 64 corresponds to eq.(3.3). Likewise, the constraint defined in lines 66 - 87 corresponds to eq.(3.5).

3. Defining the Optimization Parameters

The next step is to define the optimization parameters (step 3 in fig. 5.22). The MATLAB function *fmincon()* is used for the optimization. The *fmincon()* function requires the linear inequality constraints in the form [Mat22]:

$$A \cdot x \leq b \quad (5.1)$$

Bringing the the equations eq.(3.1) - (3.5) in the form of eq.(5.1) leads to the following matrices:

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \left(\frac{2.34}{s_{TOFL} \cdot C_{L,max,TO}} \right) & -1 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & -1 & 0 & 0 \end{bmatrix} \quad (5.2)$$

$$b = \begin{bmatrix} \left(\frac{m_{MTO}}{S_W} \right)_1 \\ 0 \\ \left(\frac{T_{TO}}{m_{MTO} \cdot g} \right)_3 \\ \left(\frac{T_{TO}}{m_{MTO} \cdot g} \right)_4 \end{bmatrix} \quad (5.3)$$

$$x = \begin{bmatrix} \left(\frac{m_{MTO}}{S_W} \right)_{DES} \\ \left(\frac{T_{TO}}{m_{MTO} \cdot g} \right)_{DES} \\ \left(\frac{V_{Cr}}{V_{md}} \right)_{DES} \\ M_{DES} \end{bmatrix} \quad (5.4)$$

with:

$$\left(\frac{m_{MTO}}{S_W} \right)_1 = 0.107 \frac{kg}{m^3} \cdot \sigma \cdot C_{L,max,L} \cdot s_{LFL} \cdot \frac{1}{m_{ML}/m_{MTO}} \quad (5.5)$$

$$\left(\frac{T_{TO}}{m_{MTO} \cdot g}\right)_3 = \left(\frac{n_E}{n_E - 1}\right) \cdot \left(\frac{1}{E_{2ndS}} + \sin \gamma\right) \quad (5.6)$$

$$\left(\frac{T_{TO}}{m_{MTO} \cdot g}\right)_4 = \left(\frac{n_E}{n_E - 1}\right) \cdot \left(\frac{1}{E_{MA}} + \sin \gamma\right) \cdot \frac{m_{ML}}{m_{MTO}} \quad (5.7)$$

Eq. (5.5), eq.(5.6) and eq.(5.7) correspond to eq.(3.1), eq.(3.3) and eq. (3.5) respectively. In summary, the system of equations defined by eq. (5.2) - (5.4) defines the matching chart from ch. 3.3.3 without the cruise line (yellow line in fig. 3.5). The cruise constraint is considered separately as a non-linear constraint by the *CruiseConstraint()* function (see App. C). The *CruiseConstraint()* function converts the equations eq.(3.7) and eq.(3.8) into the right form required by the *fmincon()* function. The required form for non-linear constraints for the *fmincon()* function is [Mat22]:

$$c(x) \leq 0$$

The line 82 in App. B is required to define an output function. The output function provides outputs at each iteration step of the optimization problem (see App. F). The outputs are required to plot the iteration progress (see fig. 5.21). Line 83 defines the options for the MATLAB *fmincon()* function. The options include that the "active-set" algorithm is used, the previously defined output function is applied and every iteration satisfies the defined constraints. More information about the options for the *fmincon()* function can be found in [Mat22].

4. Performing the Optimization

After all input arguments for the *fmincon()* have been defined, the optimization can be performed (see step 4 in fig. 5.22). The optimization can be executed with the following MATLAB command:

$$x = \text{fmincon}(\text{fun}, x_0, A, b, Aeq, beq, lb, ub, \text{nonlcon}, \text{options})$$

The following input arguments are required:

- *fun* = Function to minimize (objective function)
- *x₀* = Initial point
- *lb* = Lower bounds
- *ub* = Upper bounds

- *nonlcon* = Nonlinear constraints
- *options* = Optimization options

In the following, the input arguments will be dealt with in more detail.

Function to minimize (fun)

Line 87 of App. B shows that the function to minimize *fun* is the *RedMTOW()* function (see App. D). The function is to be explained in the following.

First, the variable parameters are read out from the *x* vector of the respective iteration (line 2 - 5 in App. D). In the following step, the cruise coefficient $C_{L,Cr}$ and the glide ratio during cruise E_{Cr} are calculated according to eq.(3.10) and eq.(3.9) (line 9 - 10). The air pressure at cruise attitude can then be calculated according to eq.(3.11). Once the air pressure is known, the cruise attitude h_{Cr} and the speed of sound at cruise attitude $a(h_{Cr})$ can be calculated with the MATLAB "*atmosisa()*" function (line 13). The cruise speed is then calculated according to eq.(3.12).

The lines 17 and 18 are required to convert the design range R and the alternate range R_{Alt} from NM to m . As mentioned in ch. 3.3.4, the alternate range is multiplied by a factor of 1.05 for international routes (line 21 - 25 in App. D). The lines 31 and 32 calculate the Breguet range factors according to eq.(3.13) and eq.(3.14). Afterwards, the mission fuel fractions for cruise, alternate and loiter are calculated according to eq.(3.15) - eq.(3.17). The maximum take-off weight is then calculated according to eq.(3.18) - eq.(3.20) within the lines 37 - 44. Finally, the objective function is defined as the sum of maximum take-off weight m_{MTO} and thrust-to-weight ratio during take-off $T_{TO}/(m_{MTO} \cdot g)$ (see line 44 of App. D). This value also serves as the output parameter of the *RedMTOW()* function.

Remaining Inputs for the *fmincon()* Function

The initial point for the optimization x_0 is chosen as the middle point between the upper and lower bounds (see line 87 in App. B). The upper bounds *ub* and the lower bounds *lb* are saved as constraints in the SysML model (see ch. 5.3.6). They are converted into vectors in line 21 and 22 of App. B. As non-linear constraint only the *CruiseConstraint()* function must be considered. As already mentioned, the optimization options are defined in line 83 of App. B.

After the optimization has been performed the output message is displayed (line 89). The *DATA* structure array is filled with the optimum values by re-executing the *REDMTOW()* function with the optimum *x* values (line 90). That way, the first preliminary design parameters for the optimal design are known.

5. Further Calculation and Plotting

Once the optimization has been performed, the remaining output parameters can be calculated (see step 5 in fig. 5.22). The equations presented in ch. 3.3.4 are used to achieve this. Furthermore, the matching chart is plotted.

For this, the *FurtherCalcAndPlot()* function is executed (line 94 in App. B).

The *FurtherCalcAndPlot()* function first recalculates the take-off constraint (eq. (3.2)) and the cruise constraint (eq.(3.7) and eq.(3.8)) resulting in a point set (line 3 - 6 in App. E). This point set is required so that the two curves can be plotted afterwards. Moreover, the *FurtherCalcAndPlot()* function calculates the additional output parameters as presented in ch. (3.3.4) (see lines 12 - 28 in App. E). The remaining lines of App. E are required to plot the matching chart.

Besides, the optimization progress is plotted (lines 96 - 126 in App. B). In order to make statements about whether an iteration step is feasible, the *constrviolation* value of the *output* function is used. For feasible iteration steps, the *constrviolation* value is close to 0. For infeasible iteration steps, the *constrviolation* value is much greater than 0 [Mat22]. The *constrviolation* value is assigned to the second column of the *IterationOutput()* matrix (see line 23 in App. F).

For both iteration progress plots, the feasible iteration steps are plotted in black and the infeasible iteration steps are plotted in red. Fig. 5.23 shows that the first iteration point is infeasible (coloured in red), because the cruise constraint (yellow line) is violated. The following iteration steps are all feasible (coloured in black).

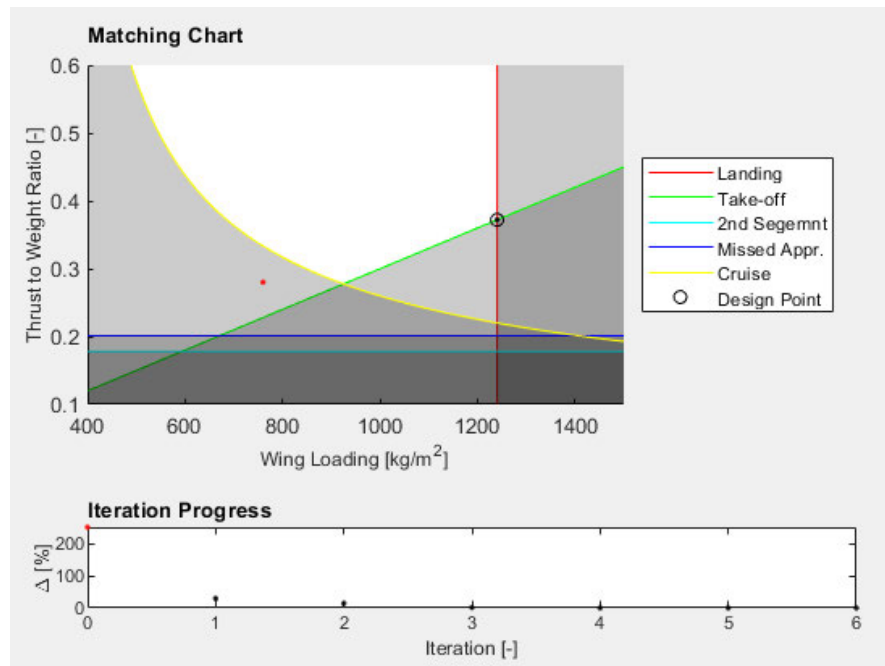


Figure 5.23: Example of an infeasible iteration point (first iteration, coloured in red)

The lines 96 - 108 in App. B plot the iteration steps in the upper chart. For this, the wing loading at take-off m_{MTO}/S_W and the thrust-to-weight ratio at take-off $T_{TO}/(m_{MTO} \cdot g)$ of each iteration step are stored in the output function (see line 23 in App. F). The lower plot shows the percentage deviation of the objective function at each iteration step from the final value of the objective function (line 110 - 126 in App. B).

6. Read Output Values From MATLAB Struct

In the last step (step 6 in fig. 5.22), the calculated output parameters are read from the *DATA* structure array and stored in the MATLAB workspace. This is necessary so that they can be accessed from an external program later (see ch. 5.5.6). In addition, the values are rounded to a desired scale (see lines 128 - 144 in App. B).

5.5 Interface Between Model Definition and Model Design

5.5.1 Introduction

At this point it is useful to take a step back and look at the current situation. In ch. 5.3, the system to be analysed was modelled using the SysML in the PTC Integrity Modeler. In chapter 5.4, a mathematical model was then developed for the preliminary design of civil jet airliners.

The problem is that the input parameters required by MATLAB for the calculation are stored in the SysML model. In addition, the output parameters from the calculation are stored in the MATLAB workspace only. This means that there is still no connection between the system definition landscape and the system analysis landscape (see fig. 5.1). An interface could automatically transfer the parameters between the two software landscapes. The next chapters will discuss the motivation for such an interface and possible technical solutions.

5.5.2 Motivation

Of course it would be possible to transfer the parameters manually from the SysML model to the system analysis program. After the analysis, the calculated values can be manually transferred back into the SysML model. However, this procedure has some significant disadvantages.

First of all, the manual transfer of the values is much more vulnerable to errors than the automatic transfer. For instance, it may happen that a calculated value is assigned to the wrong model item or that the calculated value is transferred incorrectly. Besides, an adjustment of the decimal separator may have to be taken into account.

In addition, the manual transfer of parameters requires significantly more time. As already mentioned, the preliminary design stage of aircraft is subject to frequent changes (see ch. 3.4). Therefore, especially this stage, a lot of time can be lost. Every time an input parameter of the preliminary design changes or an output parameter turns out to be infeasible, the calculation has to be re-performed. With manual data transfer, this can have a negative impact on the overall development costs.

Another problem with manual data transfer is multi-user support. As mentioned in ch. 4, the PTC Integrity Modeler uses a database to enable multi-user capabilities. However it may happen, that a design parameter is changed by another user at the same time as the manual data transfer from the system model to the system analysis program. If now the system calculation is performed with the old parameter, inconsistencies may occur within the system model (see fig. 5.24). The tool users may end up competing over various design parameters or characteristics [FRI08, p. 501]. One of the reasons for MBSE, namely the consistent storage of all system information in one place, would be lost.

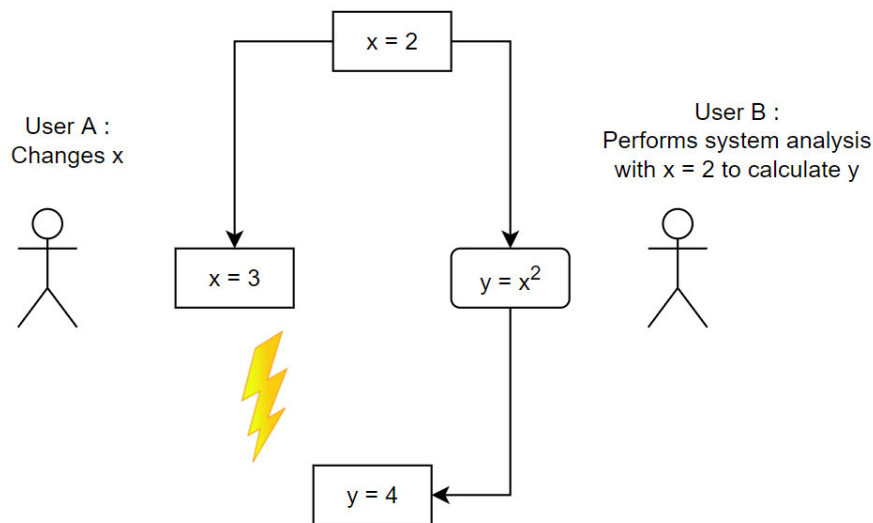


Figure 5.24: Problem with multi-user support for parallel editing and analysis of the model

This problem can be solved by locking the input parameters during the system analysis. However, a locking of certain items would only be possible with automatic data transfer. This is because a manual data transfer takes more time and therefore also keeps the model elements locked. Another solution using automatic data transfer would be to synchronize the input parameters used between the system definition program and the system analysis program before providing the calculated values back in the system model. If differences should arise after the system analysis, then a storage of the output parameters in the system model can be prevented. It can be concluded that only a complete integration of the analysis tool allows collaborative engineering to be realized at all [FRI08, p.489]. This is not achievable with manual data transfer.

Besides, the automatic data transfer between system definition and system analysis is of particular importance for the preliminary design stage of aircraft, since it enables a fast iterative design optimization (see ch. 3.4). Due to the automatic parameter transfer, a design change can be simplified. This increases the incentive to perform more design iteration, leading to a better final design.

The automatic data transfer also allows different configurations to be quickly designed and compared. As mentioned in ch. 3.5, it is particularly important to explore as many design variants as possible in the early design phases of aircraft design. The digital interface allows a quick assessment of whether and how certain configurations are technically feasible.

Finally, automatic parameter transfer allows the integration of additional features, such as warnings if certain model elements have been deleted or automatic verifications when requirements are violated.

5.5.3 Possible Technical Solutions

The last chapter emphasized the need for a digital and automated parameter transfer between the SysML model in PTC Integrity Modeler and the mathematical model in MATLAB. In this chapter different solution approaches for the interface will be discussed.

XMI Metadata Interchange

As mentioned in chapter 4, the PTC Integrity Modeler runs on a database. Since accessing a database with respect to user rights may be difficult, it is advisable to investigate a file-based data exchange first. PTC Integrity Modeler allows ModelerOMG Model Interchange Working Group (MIWG) compliant XMI import [PTC19a]. The XMI is an Object Management Group (OMG) standard for exchanging metadata information via Extensible Markup Language (XML) [OMG15].

First, an XMI file extract would be generated within the Modeler (see fig. 5.25). This XMI file could then be accessed through MATLAB directly or through an additional interface program. The input parameters could then be extracted from the XMI file and written back into the file after the MATLAB calculation. However, this approach was not pursued further because exporting and importing files would require too much time. In addition, the user would still have to perform the import and export manually, which should be prevented, if possible.

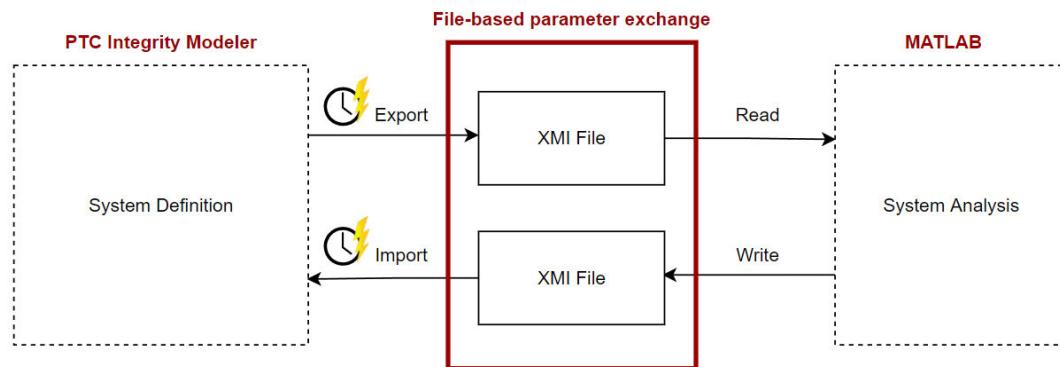


Figure 5.25: XMI metadata interchange

Utilization of the Database for Parameter Exchange

If a file-based data exchange takes too long and involves manual work, the next approach would be to access the underlying database directly. After the required input parameters are read from the database, they could be written back into the database after the calculation (see fig. 5.26).

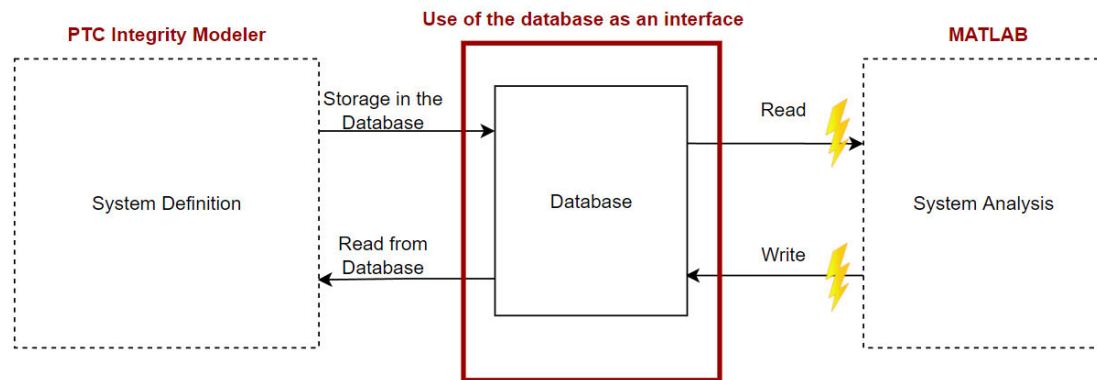


Figure 5.26: Utilization of the database for parameter exchange

However, this approach also involves potential problems. First of all, it is difficult to find the desired input parameters in the database. The data structure of the database proved to be complex to understand and bears little resemblance to the model structure. In addition, not every user who has access to a model in the Modeler automatically has write access to the underlying database [PTC19a]. This makes it difficult to return the parameters after calculation. Furthermore, the improper entry of the calculated values into the database would most likely corrupt the data or make it impossible to retrieve them later in the Modeler. Therefore, only interface approaches provided by the PTC Integrity Modeler have been investigated from this point.

PTC Integrity Modeler SySim

The Modeler SySim provides a mechanism for validating complex system behaviour defined within a Modeler SysML model [Inc19]. To do so, the analysis context is modeled in a internal block diagram. Using the SySim feature, a simulation model can be generated from the parent SysML block. This simulation can then be executed in Visual Basic.

SySim also allows the simulation model to be executed in Simulink. Parameters could then be transferred from Simulink to the MATLAB calculation engine. After the calculation in MATLAB, the calculated values could be transferred from MATLAB back to Simulink and from there back to the SySim simulation (see fig. 5.27). Own tests have confirmed this parameter transfer between SySim and MATLAB.

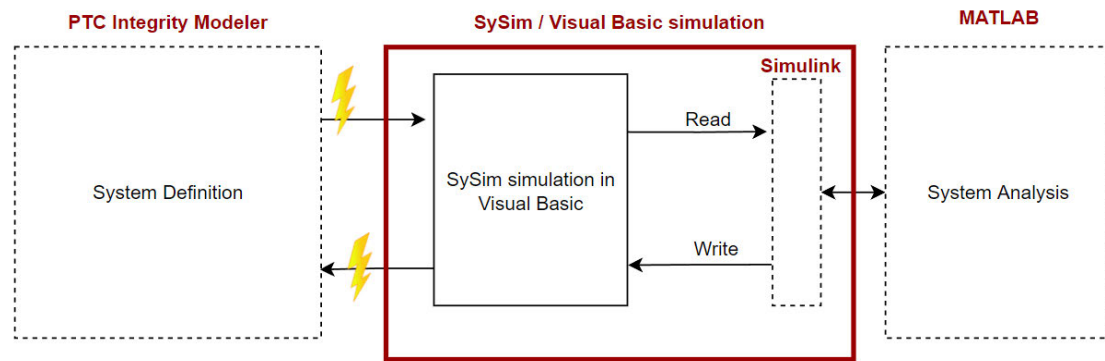


Figure 5.27: Parameter exchange with PTC Integrity Modeler SySim, Visual Basic and Simulink

However, it proved to be impossible to pass values stored within a SysML value property to the simulation (see fig. 5.27). All SySim simulation examples mentioned in [Inc19] provide only possibilities to change the input values of the simulation within the Visual Basic application itself. This can be done through the use of sliders (see fig. 5.28). The same applies to a saving of output parameters of the simulation into value properties. Again, there was no satisfactory result. For all examples mentioned in [Inc19], the output values of the simulation are displayed within the Visual Basic simulation only. For this, special visual indicators are provided by the SySim feature (see fig. 5.28). Fig. 5.28 shows an exemplary SySim simulation. The sliders on the left side change the input values for the simulation. The indicators on the right display the simulations outputs.

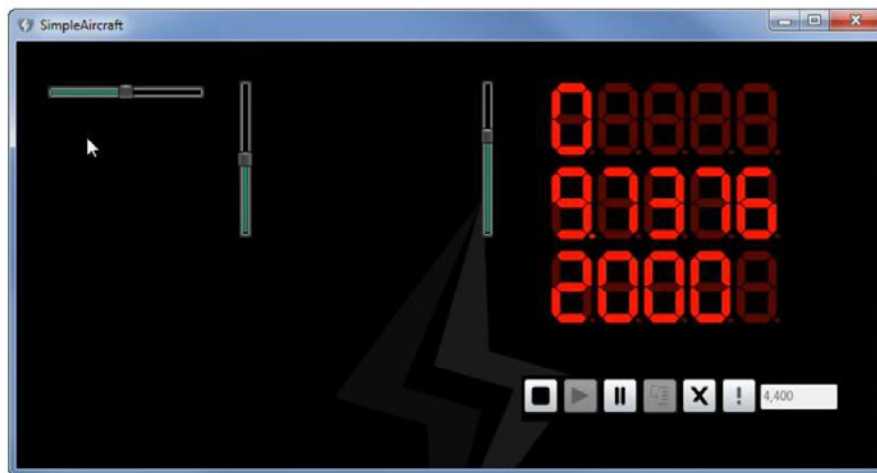


Figure 5.28: Example of a system simulation performed with SySim and Visual Basic (taken from [Inc19, fig. 4.19])

Since no solution was found to have the SySim simulation interact with values stored

within the model, this solution approach was discarded.

Simulink Synchronizer

The Simulink Synchronizer allows to link SysML diagram contents with Simulink diagrams. For example, it is possible to automatically generate Simulink block diagrams from parametric diagrams or internal block diagrams. Once the Simulink model is linked to the respective SysML diagram, the synchronization can be performed rapidly by pressing one single button in the Modeler.

Fig. 5.29 shows the synchronization of the SysML diagram contents with Simulink. Again, the Simulink diagram could be modified to run the calculation in the MATLAB calculation engine. Own tests confirmed the synchronization of the SysML diagram components to the respective Simulink model. However, the parameters assigned to the objects were not synchronized (see fig. 5.29). This approach was then discarded.

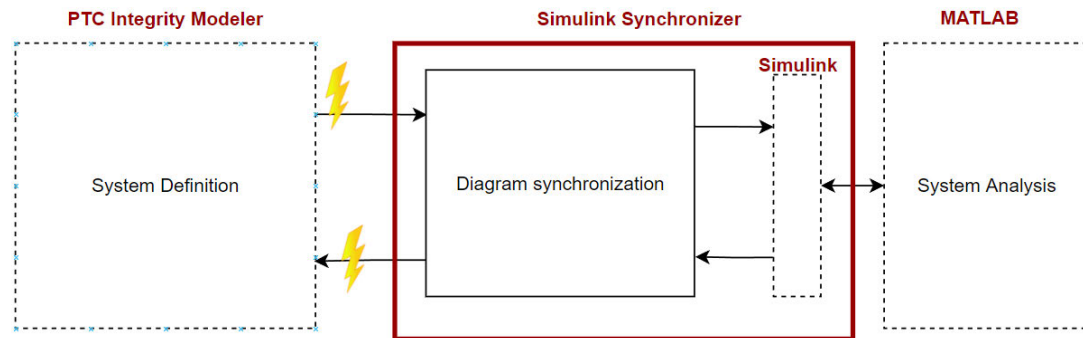


Figure 5.29: XMI metadata interchange

Phoenix Model Center

The Phoenix Model Center is a software package developed by Phoenix Integration Inc. that provides users with tools and methods that allow them to automate the execution of almost any modelling and simulation tool [Int22]. The Phoenix Model Center also provides additional features to the PTC Integrity Modeler. For example, requirements are supplemented by a numerical property, as well as an upper and a lower margin [PTC19a]. In addition, constraint properties can be linked directly to calculations that are stored in the Phoenix Model Center. The Model Center in turn links the calculations to a variety of analysis tools (see fig. 5.30). Moreover, the Phoenix Model Center enables to check whether calculated values lie within the margins of the corresponding requirement.



Figure 5.30: Integration of the Phoenix Model Center with different analysis tools (taken from [Int22])

Since the Phoenix Model center was developed specifically for automatic parameter exchange with analysis models, it offers a solution for connecting the two models. However, special licenses are required. To link the PTC Integrity Modeler with the Phoenix Model center, a Phoenix Model Center 13 licence and a Analysis Server 13 license is compulsory [PTC19a]. Besides, there is a strong dependence on the solutions developed by Phoenix Integration. If the parameter exchange has to be adapted to company-specific needs, high additional costs may be charged. Therefore, the next chapters present the development of an automated parameter exchange between the PTC Integrity Modeler and MATLAB without the need for a Phoenix Model Center or Analysis Server license.

5.5.4 Overview of the Visual Basic Interface

In chapter 4, it was already mentioned that it is possible to access and edit model items of the Modeler using the PTC Integrity Automation Interface. This approach is now further pursued.

For this purpose, a Visual Basic program is created that can automatically exchange parameters between the PTC Integrity Modeler and MATLAB via the Automation Interface. The Visual Basic program is required as an intermediate step because it allows the interaction with both the PTC Integrity Modeler as well as MATLAB. This is achieved by adding two libraries to the extend the Visual Basic functionality. The "*Enterprise OLE Automation 1.0 Type Library*" library is required for the interaction with the PTC Integrity Modeler. The "*Matlab Automation Server Type Library*" is added for the interaction with MATLAB. This chapter provides an explanation of the structure and function of the Visual Basics program created to exchange parameters between system definition software (PTC Integrity Modeler) and system analysis software (MATLAB).

The visual basic program has to perform two main tasks. First, the values stored in the PTC Integrity system model must be transferred to MATLAB. After the calculation in MATLAB, the newly calculated values must then be transferred back to the system model (see fig. 5.2). In addition, the user should be informed about the status of data exchange and calculation. On top of that, the user should be informed about any occurring errors. Parameter exchange and system analysis should be performed as conveniently as possible. Ideally, the system analysis is performed directly from within

the PTC Integrity Modeler window.

Fig. 5.31 shows the parameter transfer between system definition software (PTC Integrity Modeler) and system analysis software (MATLAB). The black arrows symbolize the parameter transfer. It can be seen that the Visual Basic program interacts with the Modeler using the *PTC Automation Interface* and accesses MATLAB using the *MATLAB Automation Server*.

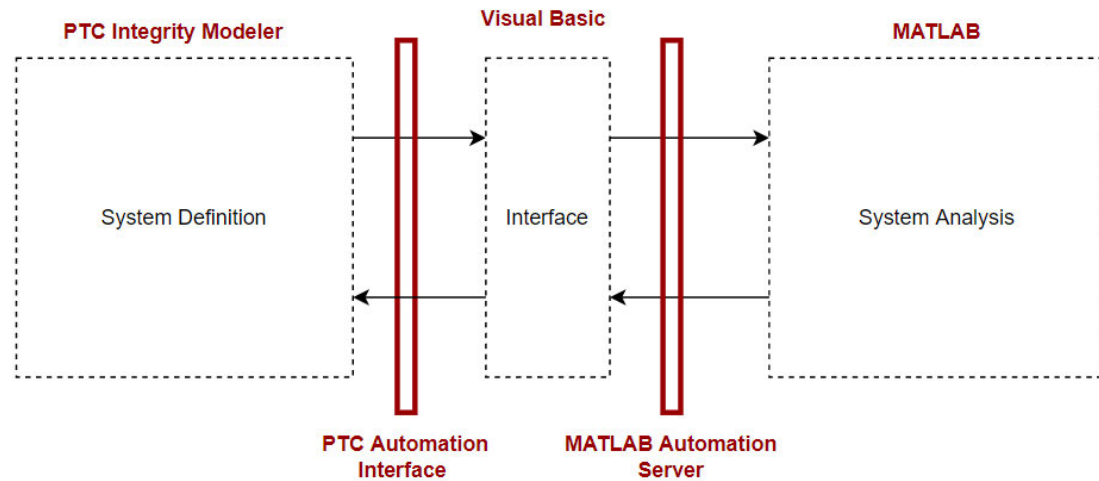


Figure 5.31: Parameter transfer between system definition software (PTC Integrity Modeler) and system analysis software (MATLAB) through a Visual Basic program

Fig. 5.32 shows the detailed structure of the Visual Basic program that is used to exchange the parameters (center box in fig. 5.31). The left side is connected to the SysML model stored inside the PTC Integrity Modeler repository (see ch. 5.3). The right side is connected to the MATLAB code introduced in ch. 5.4 (see fig. 5.22).

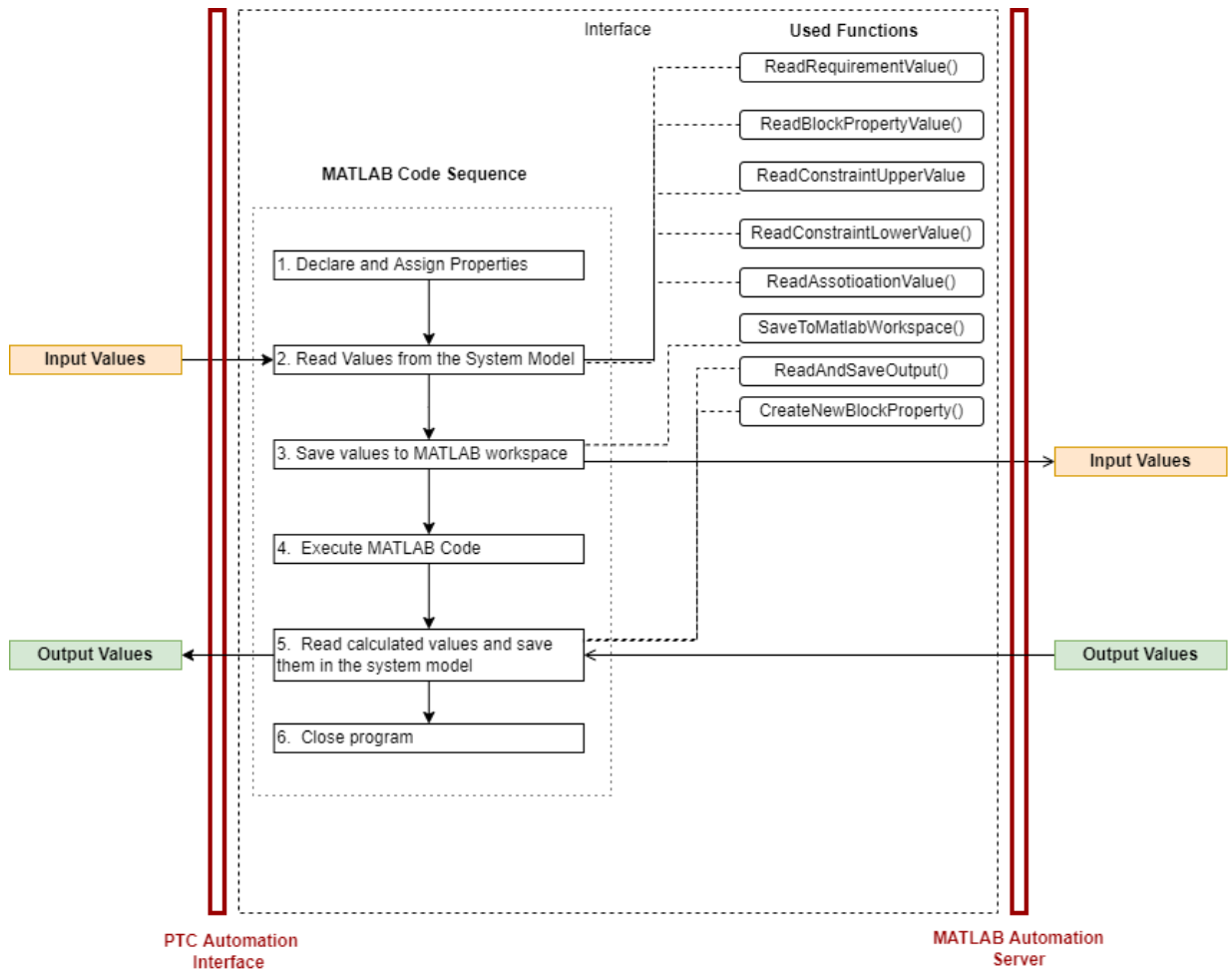


Figure 5.32: Structure of the Visual Basic Program

5.5.5 Required Libraries

In order for the visual basic interface to work, a couple precautions must be taken. First, a working and licensed version of PTC Integrity must be installed. In addition, a license for the *PTC Integrity Automation Interface* and the *PTC Integrity API Access License* must be registered [PTC15, p. 1]. Furthermore, a working MATLAB version containing the "*MATLAB Optimization Toolbox*" must be installed. The installation of the "*Automation Toolbox*" can be verified with the following MATLAB function:

```
1 ver -support
```

In order to execute MATLAB commands from Visual Basic, MATLAB must be registered as a "*COM server*". By default, MATLAB is automatically registered as a "*COM*"

server" during installation. If that is not the case, it can be accomplished with the following MATLAB command:

```
1 comserver('register','User','current')
2 state = enableservice('AutomationServer',true);
3 enableservice('AutomationServer')
```

5.5.6 The Visual Basic Code

In this chapter the Visual Basic code will be explained. The programmed Visual Basic code is displayed below. In the following chapters, the code is explained using the sequence illustrated in fig. 5.32.

```
1 Imports System.Text.RegularExpressions
2 Module MatLabAPI
3     '1. Declare and assign properties
4     Private Declare Auto Function ShowWindow Lib "user32.dll" (ByVal hWnd As ...
5         IntPtr, ByVal nCmdShow As Integer) As Boolean
6     Private Declare Auto Function GetConsoleWindow Lib "kernel32.dll" () As ...
7         IntPtr
8     Private Const SW_HIDE As Integer = 0
9
10    Dim Studio, Models, MatLab As Object
11    Dim Model, Dictionary, Requirement, BlockProperty, Constraint, ...
12        VariantObj, DecisionSet, VariantParameter,
13        DecisionSetVariantParameters, DecisionSetVariantParameter, ...
14        BlockPropertyType, Block, Parts, Part As ENT6Lib.CCCaseProjects
15
16    Dim Assotiation As ENT6Lib.CCCaseProjects
17
18    Dim RequirementName, BlockPropertyName, ConstraintName, DecisionSetName, ...
19        VariantName, VariantParameterType, OutputMessage As String
20    Dim OutputValue
21
22    Sub Main()
23        'Minimize Console Window
24        Dim hWndConsole As IntPtr
25        hWndConsole = GetConsoleWindow()
26        ShowWindow(hWndConsole, SW_HIDE)
27
28        'Assign Properties
29        Studio = CreateObject("Studio.Editor")
30        Models = CreateObject("OMTE.Projects")
31        Model = Models.Item("ActiveProject")
32        Dictionary = Model.Item("Dictionary")
33
34        Studio.ClearResultsPane(1)
35        Studio.ClearOutputWindow
36        Studio.DisplayOutputWindowMessage("Preliminary Sizing Tool [" + ...
37            TimeOfDay + "]: Reading values:" + vbCrLf)
38
39        '2. Read Values from the System Model
40        '2.1. Read Values from Requirements
41        Dim climb_grad_MA As Double = ReadRequirementValue("REQ026")
42        Dim climb_grad_CLB As Double = ReadRequirementValue("REQ024")
```

```

37     Dim s_LFL As Double = ReadRequirementValue("REQ098")
38     Dim s_TOFL As Double = ReadRequirementValue("REQ087")
39     Dim R_Alt As Double = ReadRequirementValue("REQ025")
40     Dim Certification As String = ReadRequirementValue("REQ067")
41     Dim RouteType As String = ReadRequirementValue("REQ110")
42
43
44     '2.2. Read Values from Block Properties
45     Dim CL_max_L As Double = ReadBlockPropertyValue("CLMax,L")
46     Dim CL_max_TO As Double = ReadBlockPropertyValue("CLMax,TO")
47     Dim FlapSetting As Double = ReadBlockPropertyValue("FlapSetting,TO")
48     Dim m_ML_DIV_m_MTO As Double = ReadBlockPropertyValue("mML/mMTO")
49     Dim m_OE_DIV_m_MTO As Double = ReadBlockPropertyValue("mOE/mMTO")
50     Dim BPR As Double = ReadBlockPropertyValue("BPR")
51     Dim AR As Double = ReadBlockPropertyValue("AR")
52     Dim m_CARGO As Double = ReadBlockPropertyValue("CargoMass")
53     Dim R As Double = ReadBlockPropertyValue("Range")
54
55
56     '2.3. Read Values from Constraints
57     Dim TtWR_UPPER As Double = ...
58         ReadConstraintUpperValue("ThrustToWeightRatio")
59     Dim TtWR_LOWER As Double = ...
60         ReadConstraintLowerValue("ThrustToWeightRatio")
61     Dim WL_UPPER As Double = ReadConstraintUpperValue("WingLoading")
62     Dim WL_LOWER As Double = ReadConstraintLowerValue("WingLoading")
63     Dim M_UPPER As Double = ReadConstraintUpperValue("CruiseMachNumber")
64     Dim M_LOWER As Double = ReadConstraintLowerValue("CruiseMachNumber")
65     Dim V_DIV_V_md_UPPER As Double = ReadConstraintUpperValue("V/V_md")
66     Dim V_DIV_V_md_LOWER As Double = ReadConstraintLowerValue("V/V_md")
67
68     '2.4 Read Values from Assotiations
69     Dim n_E As Double = ReadAssotiationValue("AircraftEngine", "StartRole")
70     Dim n_PAX As Double = ...
71         ReadAssotiationValue("AircraftDomainPassenger", "StartRole")
72
73     '3. Saving values to MATLAB workspace
74     Dim MatLabFileLocation As String
75     MatLabFileLocation = My.Application.Info.DirectoryPath + "\Main.m"
76     Studio.DisplayOutputWindowMessage("Preliminary Sizing Tool [" + ...
77         TimeOfDay + "]: Starting the MATLAB application" & vbCrLf)
78     MatLab = CreateObject("Matlab.Application")
79
80     SaveToMatlabWorkspace(MatLab, climb_grad_MA, "climb_grad_MA")
81     SaveToMatlabWorkspace(MatLab, climb_grad_CLB, "climb_grad_CLB")
82     SaveToMatlabWorkspace(MatLab, s_LFL, "s_LFL")
83     SaveToMatlabWorkspace(MatLab, s_TOFL, "s_TOFL")
84     SaveToMatlabWorkspace(MatLab, R_Alt, "R_Alt")
85     SaveToMatlabWorkspace(MatLab, Certification, "Certification")
86     SaveToMatlabWorkspace(MatLab, RouteType, "RouteType")
87
88     SaveToMatlabWorkspace(MatLab, CL_max_L, "CL_max_L")
89     SaveToMatlabWorkspace(MatLab, CL_max_TO, "CL_max_TO")
90     SaveToMatlabWorkspace(MatLab, FlapSetting, "FlapSetting")
91     SaveToMatlabWorkspace(MatLab, m_ML_DIV_m_MTO, "m_ML_DIV_m_MTO")
92     SaveToMatlabWorkspace(MatLab, m_OE_DIV_m_MTO, "m_OE_DIV_m_MTO")
93     SaveToMatlabWorkspace(MatLab, BPR, "BPR")
94     SaveToMatlabWorkspace(MatLab, AR, "AR")
95     SaveToMatlabWorkspace(MatLab, m_CARGO, "m_CARGO")
96     SaveToMatlabWorkspace(MatLab, R, "R")

```



```

95     SaveToMatlabWorkspace(MatLab, TtWR_UPPER, "TtWR_UPPER")
96     SaveToMatlabWorkspace(MatLab, TtWR_LOWER, "TtWR_LOWER")
97     SaveToMatlabWorkspace(MatLab, WL_UPPER, "WL_UPPER")
98     SaveToMatlabWorkspace(MatLab, WL_LOWER, "WL_LOWER")
99     SaveToMatlabWorkspace(MatLab, M_UPPER, "M_UPPER")
100    SaveToMatlabWorkspace(MatLab, M_LOWER, "M_LOWER")
101    SaveToMatlabWorkspace(MatLab, V_DIV_V_md_UPPER, "V_DIV_V_md_UPPER")
102    SaveToMatlabWorkspace(MatLab, V_DIV_V_md_LOWER, "V_DIV_V_md_LOWER")
103
104    SaveToMatlabWorkspace(MatLab, n_PAX, "n_PAX")
105    SaveToMatlabWorkspace(MatLab, n_E, "n_E")
106
107
108    Studio.DisplayOutputWindowMessage("Preliminary Sizing Tool [" + ...
        TimeOfDay + "]: Executing the MATLAB script: " & ...
        MatLabFileLocation & vbCrLf)
109
110    '4. Execute MATLAB Code
111    MatLab.Execute("cd '" & My.Application.Info.DirectoryPath & "'")
112    Dim result As String = MatLab.Execute("run('" & MatLabFileLocation & ...
        "')")
113
114    If result.Contains("cannot execute the file") Then
115        MsgBox("Error: MATLAB file could not be opened!" + ...
            Environment.NewLine + "Please make sure that the MATLAB file ...
            is located in the same folder as the executable file.")
116        Environment.Exit(0)
117    End If
118
119    'Make sure that the calculation is finished before further calculation
120    If result.Contains("Calculation has finished") Then
121
122        '5. Reading and saving the calculated values in the Model
123        Studio.DisplayOutputWindowMessage("Preliminary Sizing Tool [" + ...
            TimeOfDay + "]: Saving calculated values in the model:" + ...
            vbCrLf)
124
125        ReadAndSaveOutput(MatLab, "h_Cr", "Aircraft", "CruiseAttitude")
126        ReadAndSaveOutput(MatLab, "V_Cr", "Aircraft", "CruiseSpeed")
127        ReadAndSaveOutput(MatLab, "C_L_Cr", "Aircraft", "CL_Cr")
128        ReadAndSaveOutput(MatLab, "E_Cr", "Aircraft", "E_Cr")
129        ReadAndSaveOutput(MatLab, "M", "Aircraft", "CruiseMachNumber")
130        ReadAndSaveOutput(MatLab, "V_DIV_V_md", "Aircraft", "V/V_md")
131        ReadAndSaveOutput(MatLab, "ThrustToWeightRatioDesign", ...
            "Aircraft", "ThrustToWeightRatio")
132        ReadAndSaveOutput(MatLab, "WingLoadingDesign", "Aircraft", ...
            "WingLoading")
133        ReadAndSaveOutput(MatLab, "m_PL", "MassPrediction", "mPL")
134        ReadAndSaveOutput(MatLab, "m_MTO", "MassPrediction", "mMTO")
135        ReadAndSaveOutput(MatLab, "m_ML", "MassPrediction", "mML")
136        ReadAndSaveOutput(MatLab, "m_MZF", "MassPrediction", "mMZF")
137        ReadAndSaveOutput(MatLab, "m_OE", "MassPrediction", "mOE")
138        ReadAndSaveOutput(MatLab, "m_F", "FuelTank", "mF")
139        ReadAndSaveOutput(MatLab, "T_TO_per_engine", "Engine", ...
            "TakeOffThrust")
140        ReadAndSaveOutput(MatLab, "A_Wing", "Wing", "WingArea")
141
142
143    '6. Close Program
144    Studio.DisplayOutputWindowMessage("Preliminary Sizing Tool [" + ...
        TimeOfDay + "]: Refreshing model" & vbCrLf)
145    Studio.Refresh

```

```

146         Studio.DisplayOutputWindowMessage("Preliminary Sizing Tool [" + ...
           TimeOfDay + "]: Calculation finished" & vbCrLf)
147
148     End If
149
150
151     Do While True
152         Dim IsHandle As String = MatLab.Execute("ishandle(1)")
153         If IsHandle.Contains("0") Then
154             Exit Do
155         End If
156     Loop
157
158     Studio.DisplayOutputWindowMessage("Preliminary Sizing Tool [" + ...
           TimeOfDay + "]: Closing the MATLAB application" & vbCrLf)
159     MatLab.Execute("quit")
160
161
162
163 End Sub

```

1. Declaring and Assigning Properties

First of all, the variables are declared and assigned (see step 1 in fig. 5.32). All model items accessed through the PTC Integrity Automation Interface are declared as "*ENT6Lib.CCcaseProjects*".

Line 4 - 6 in the Visual Basic Code are required to hide the console window while the program is executed. This allows the application to run solely in the background. Afterwards, the variables used in the program are declared.

Main() Sub between line 17 to line is executed as soon as the program is started. The sub starts by hiding the console window in line 21. After that, a *Studio* object is created. The *Studio* object is required to control the PTC Integrity Modeler user interface (see ch. 4.8). The Visual Basic program uses the PTC Integrity Modeler user interface to notify the user of the progress of the system analysis.

The lines 25 - 27 are required to access the *Dictionary* object of the active model. The *Dictionary* object is a container for all dictionary items in the Model (see ch. 4.7). It is required to read and modify the parameters for the preliminary design. After that, the result pane and the output window of the PTC Integrity user interface are cleared (lines 29 - 30).

2. Reading Values From the System Model

After all properties have been declared or assigned, the parameters can be read from the SysML model (see step 2 in fig. 5.32). This happens between line 33 and 108. All parameters can be accessed via the previously defined *Dictionary* object through the PTC Automation Interface (see fig. 5.31). However, the parameters are distributed over different model items within the model (see fig. 5.3). Therefore, different access methods

must be used. To keep the program clear and to ensure flexibility and reusability, a function for accessing each model item was developed. The functions are presented in ch. 5.5.7.

First, the requirement values are accessed via the *ReadRequirementValue()* function (lines 34 - 41). After that, the values stored in value properties are accessed via the *ReadBlockPropertyValue()* function (lines 44 - 35). The upper and lower limits of the constraints are accessed through the *ReadConstraintUpperValue()* and *ReadConstraintLowerValue()* functions (lines 56 - 64). Finally, the two parameters stored inside associations are accessed by using the *ReadAssociationValue()* function (lines 66 - 58).

3. Saving Values to the MATLAB Workspace

At the beginning of the third step, all the input parameters needed for the MATLAB calculation are cached inside the Visual Basic program as variables. In the next step, they must be passed on to MATLAB (see fig. 5.31). This step corresponds to step 3 in fig. 5.32. To transfer the parameters to MATLAB, first the MATLAB application has to be executed (line 75). After that, all parameters are transferred to the MATLAB workspace by using the *SaveToMatlabWorkspace()* function (lines 77 - 105).

4. Executing the MATLAB Code

All parameters are now stored in the MATLAB workspace. The MATLAB script presented in ch. 5.4 can now be executed (step 4 in fig. 5.32). To achieve this, the MATLAB folder is changed to the folder in which the MATLAB script is stored (line 111). Afterwards, the MATLAB script is executed (line 112). If the MATLAB script is not found, an error message appears and the Visual Basic program is closed (line 115). If the calculation was successful, the return string of the MATLAB calculation contains the string "*Calculation has finished*" (line 120). Then the return of the calculated parameters to the SysML model can be initiated.

5. Reading Calculated Values and Storing Them in the System Model

The fifth step consists of reading the calculated values from MATLAB and saving them in the SysML model (lower arrows in fig. 5.31). Both the import of the output values to the Visual Basic program as well as the storage in the SysML model is performed one step. The *ReadAndSaveOutput()* function is used for this. The *ReadAndSaveOutput()* function stores all calculated parameters as value properties of the respective block (lines 125 - 140). If the value property is not found in the SysML model, it is possible to create it using the *CreateNewBlockProperty()* function.

6. Closing the Program

After all parameters have been transferred back to the SysML model, the model is refreshed (line 145). Theoretically, the work of the Visual Basic program is now complete. The input parameters were read from the SysML model, transferred to MATLAB, the calculation was performed and the calculated values were finally transferred back to the SysML model (see fig 5.32). The only remaining step is to close the MATLAB application. However, the MATLAB plot would then also be closed immediately. The loop between lines 161 and 156 is used to check if the MATLAB plot is still open. As soon as the plot is closed, the MATLAB application can also be closed (line 159).

5.5.7 Used Functions

This chapter is intended to explain the functions used in the Visual Basic Code in more detail (see fig. 5.32).

ReadRequirementValue()

To read requirement values from the Automation Interface the *ReadRequirementValue()* function was used. The function is called as follows:

```
1 Dim <RequirementValue> As Double = ReadRequirementValue("<RequirementID>")
```

```
1 Function ReadRequirementValue(RequirementName) As String
2   'Get Requirement Object
3   Requirement = Dictionary.Item("Class", RequirementName)
4
5   'Read Requirement Value
6   If Requirement Is Nothing Then
7     MsgBox("Error: Requirement '" & RequirementName & "' was not found!")
8     Environment.Exit(0)
9   Else
10    ReadRequirementValue = ...
11    CStr(Regex.Match(Requirement("Description"), ...
12    "(?<=\[ (.*) (?=\])").Value)
13    OutputMessage = "Preliminary Sizing Tool [" + CStr(TimeOfDay) + ...
14    "]:" + vbTab + "Requirement: " + vbTab + vbTab + ...
15    RequirementName + " = " + ReadRequirementValue + vbCrLf
16    Studio.DisplayOutputWindowMessage(OutputMessage)
17   End If
18 End Function
```

First, the requirement is accessed from the dictionary. As demonstrated earlier, requirements can be accessed through *Class* objects from the Automation Interface Dictionary object (see line 3) [PTC15, p. 187]. Afterwards, it is checked whether the desired requirement exists. If not, the application closes (line 8). If the requirement exists, the *Description* attribute is accessed. In order to access the desired input parameter, the value inside the square bracket of the requirement description is read out (line 10). This

is necessary because there is no separate input field to store the parameters within the PTC Integrity Modeler. The numerical value must therefore be entered between the two square brackets (see fig. 5.33).

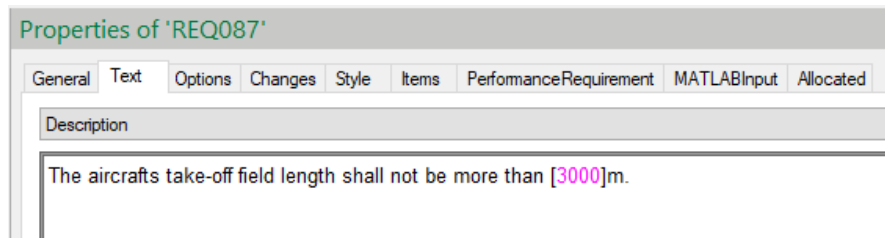


Figure 5.33: Solution to save and access value parameters within the requirement description

A conventional modelling technique would be to save the value parameter inside a value property and link the value property to the respective requirement [PTC19b]. The value can then be read from the value property similarly to the function presented below. However, in case the requirements change, the value within the value property would also have to be changed manually. Therefore, it was decided to read the values directly from the requirements to increase productivity and minimize possible errors.

ReadBlockPropertyValue()

The *ReadBlockPropertyValue()* function can be used to read the default value of a block property.

```
1 Dim <BlockPropertyValue> As Double = ...
   ReadBlockPropertyValue("<BlockPropertyName>")
```

Just like the function presented above, first the desired block property is accessed. This is done by accessing all objects that are typed *Role* and have the name of the block property through the PTC Automation Interface *Dictionary* object [PTC19b, p. 272]. Again, the application is closed if the desired *Role* object is not found (line 6 - 8). Once the object for the desired block property is accessed, the default value can be read out through the "*Default Value*" attribute (line 10).

```
1 Function ReadBlockPropertyValue(BlockPropertyName) As Double
2     'Get Block Property Object
3     BlockProperty = Dictionary.Item("Role", BlockPropertyName)
4
5     'Read Requirement Value
6     If BlockProperty Is Nothing Then
7         MsgBox("Error: Block property '" & BlockPropertyName & "' was not ...
8             found!")
9         Environment.Exit(0)
```

```

9     Else
10        ReadBlockPropertyValue = CDb1(BlockProperty("Default Value"))
11        OutputMessage = "Preliminary Sizing Tool [" + CStr(TimeOfDay) + "]:" ...
            + vbTab + "Block Property: " + vbTab + vbTab + ...
            BlockPropertyName + " = " + CStr(ReadBlockPropertyValue) + vbCrLf
12        Studio.DisplayOutputWindowMessage(OutputMessage)
13    End If
14 End Function

```

ReadConstraintUpperValue() and ReadConstraintLowerValue()

The constraints consist of an upper and a lower value. Both values are read separately with the *ReadConstraintUpperValue()* and the *ReadConstraintLowerValue()* function. The functions can be called the following way:

```

1 Dim <UpperValue> As Double = ReadConstraintUpperValue("<ConstraintName>")
2 Dim <LowerValue> As Double = ReadConstraintLowerValue("<ConstraintName>")

```

Both functions are constructed the same. First, the desired *Constraint* object is accessed from the Automation Interface *Dictionary* object [PTC15, p. 199]. Just like before, the application closes if the *Constraint* object does not exist. Depending on which of the two functions is called, the upper value (*Best Possible* attribute) or lower value (*Worst Acceptable* attribute) is read (line 10).

```

1 Function ReadConstraintUpperValue(ConstraintName) As Double
2     'Get Constraint Object
3     Constraint = Dictionary.Item("Constraint", ConstraintName)
4
5     'Read Constraint Upper (=BestPossible) Value
6     If Constraint Is Nothing Then
7         MsgBox("Error: Constraint '" & ConstraintName & "' was not found!")
8         Environment.Exit(0)
9     Else
10        ReadConstraintUpperValue = CDb1(Constraint("Best Possible"))
11        OutputMessage = "Preliminary Sizing Tool [" + CStr(TimeOfDay) + "]:" ...
            + vbTab + "Constraint Upper Value: " + vbTab + ConstraintName + ...
            " = " + CStr(ReadConstraintUpperValue) + vbCrLf
12        Studio.DisplayOutputWindowMessage(OutputMessage)
13    End If
14 End Function

```

```

1 Function ReadConstraintLowerValue(ConstraintName) As Double
2     'Get Constraint Object
3     Constraint = Dictionary.Item("Constraint", ConstraintName)
4
5     'Read Constraint Lower (=Worst Acceptable) Value
6     If Constraint Is Nothing Then
7         MsgBox("Error: Constraint '" & ConstraintName & "' was not found!")
8         Environment.Exit(0)
9     Else
10        ReadConstraintLowerValue = CDb1(Constraint("Worst Acceptable"))

```

```

11     OutputMessage = "Preliminary Sizing Tool [" + CStr(TimeOfDay) + "]:" ...
        + vbTab + "Constraint Lower Value: " + vbTab + ConstraintName + ...
        " = " + CStr(ReadConstraintLowerValue) + vbCrLf
12     Studio.DisplayOutputWindowMessage(OutputMessage)
13     End If
14 End Function

```

ReadAssocioationValue()

The amount of engines and the amount of passengers is directly read from the association multiplicities. For this, the *ReadAssocioationValue()* function was used. It can be called the following way:

```

1 Dim <MultiplicityValue> As Double = ...
    ReadAssocioationValue("<AssociationName>", "<StartRole>/<EndRole>")

```

The association values can be accessed via the *Association* object from the Automation Interface *Dictionary* object (line 3) [PTC15, p. 166]. If the requested *Association* object does not exist, the application is closed (line 7-8). Depending on which role is requested, the *Start Multiplicity* attribute or *End Multiplicity* attribute is read (line 11).

```

1 Function ReadAssocioationValue(AssociationName, Type) As Double
2     'Get Association Object
3     Association = Dictionary.Item("Association", AssociationName)
4
5     'Read Requirement Value
6     If Association Is Nothing Then
7         MsgBox("Error: Association '" & AssociationName & "' was not found!")
8         Environment.Exit(0)
9     Else
10        If Type = "StartRole" Then
11            ReadAssocioationValue = Cint(Association("Start Multiplicity UML"))
12            OutputMessage = "Preliminary Sizing Tool [" + CStr(TimeOfDay) + ...
                "]:" + vbTab + "Association Multiplicity (Start): " + vbTab + ...
                AssociationName + " = " + CStr(ReadAssocioationValue) + vbCrLf
13        ElseIf Type = "EndRole" Then
14            ReadAssocioationValue = Cint(Association("End Multiplicity UML"))
15            OutputMessage = "Preliminary Sizing Tool [" + CStr(TimeOfDay) + ...
                "]:" + vbTab + "Association Multiplicity (End): " + vbTab + ...
                AssociationName + " = " + CStr(ReadAssocioationValue) + vbCrLf
16        End If
17        Studio.DisplayOutputWindowMessage(OutputMessage)
18    End If
19 End Function

```

SaveToMatlabWorkspace()

Once all required input parameters are read from the Automation Interface item dictionary they have to be saved in the MATLAB base workspace. This is achieved by using

the *SaveToMatlabWorkspace()* function. Before calling the function, the MATLAB application has to be started. As already mentioned, the Matlab Automation Server Type Library must be referenced.

```
1 Dim MatLab As Object
2 Studio.DisplayOutputWindowMessage("Preliminary Sizing Tool [" + TimeOfDay + ...
   "]: Starting the MATLAB application" & vbCrLf)
3 MatLab = CreateObject("Matlab.Application")
```

The *SaveToMatlabWorkspace()* function can then be called the following way:

```
1 SaveToMatlabWorkspace(MatLab, <Parameter>, "<ParameterName>")
```

The parameters can be saved to the MATLAB workspace using the *.PutWorkspaceData* method (line 2) [Mat22].

```
1 Function SaveToMatlabWorkspace(MatLab, Parameter, ParameterName)
2     MatLab.PutWorkspaceData(ParameterName, "base", Parameter)
3 End Function
```

ReadAndSaveOutput()

The function *ReadAndSaveOutput()* is used to read the calculated values from the MATLAB base workspace and stores them at the respective value property. The function can be called as follows:

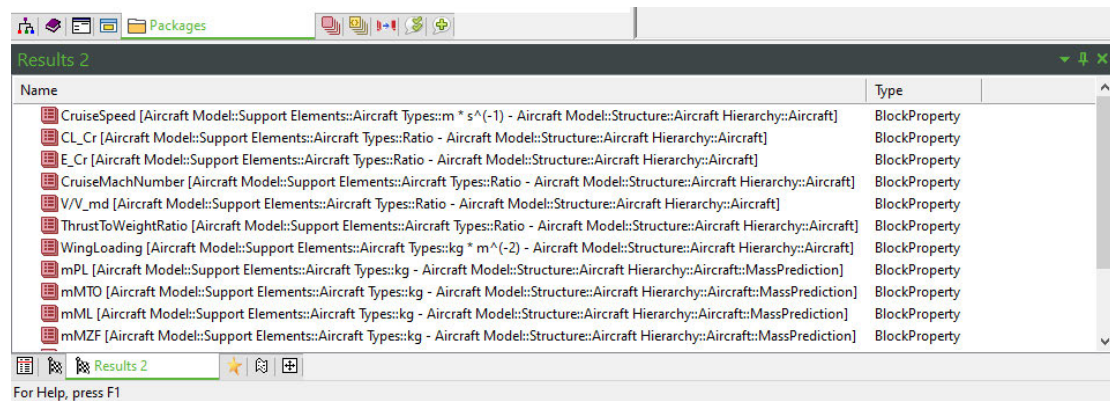
```
1 ReadAndSaveOutput(MatLab, "<MATLABParameterName>", "<BlockName>", ...
   "<BlockPropertyName>")
```

First, the parameter is read from the MATLAB workspace. For this, the *.GetWorkspaceData* method is used (line 5) [Mat22]. If the requested parameter is not found in the workspace the function is exited (line 9). Otherwise the script will be continued.

The next step is to store the calculated parameters within the respective block property. Therefore, the block property is accessed similarly to ch.5.5.7. It can happen that desired block property does not exist. In this case a dialogue window is opened, asking if the block property should be created (line 15). If it is declined, the function is exited (line 20). If it is accepted, the new block property is created, using the *CreateNewBlockProperty()* function (see ch. 5.5.7).

After that, the calculated parameter is assigned to the "Default Value" attribute of the respective block property (line 27). Furthermore, the parameter is displayed at the results pane of the Modeler (line 29, see fig. 5.34).


```
1 Function ReadAndSaveOutput(MatLab, VarName, BlockName, BlockPropertyName)
2
3 'read value from MATLAB: saves value of MATLAB var=Varname to OutputValue
4 Dim OutputValue
5 MatLab.GetWorkspaceData(CStr(VarName), "base", OutputValue)
6
7 If OutputValue Is Nothing Then
8     MsgBox("Error: MATLAB value '" + VarName + "' was not found!")
9     GoTo Line1
10 End If
11
12 BlockProperty = Dictionary.Item("Role", BlockPropertyName)
13
14 If BlockProperty Is Nothing Then
15     Dim answer As Integer = MsgBox("The block property '" & ...
16     BlockPropertyName & "' was not found." + Environment.NewLine + ...
17     "Do you want to create it?", vbQuestion + vbYesNo + ...
18     vbDefaultButton2, "Error: Block property was not found!")
19
20     If answer = vbYes Then
21         CreateNewBlockProperty(BlockName, BlockPropertyName)
22     ElseIf answer = vbNo Then
23         GoTo Line1
24     End If
25 End If
26
27 'Save output value as default value of the block property
28 BlockProperty = Dictionary.Item("Role", BlockPropertyName)
29 BlockProperty("Default Value") = CStr(OutputValue)
30 'Add output to the Resultspane
31 Studio.AddToResultsPane(1, BlockProperty.Property("Id"))
32
33 OutputMessage = "Preliminary Sizing Tool [" + CStr(TimeOfDay) + "]:" + ...
34 vbTab + " Saved calculated value '" + VarName + "' as default value ...
35 of block property '" + BlockPropertyName + "' = " + ...
36 CStr(OutputValue) + vbCrLf
37 Studio.DisplayOutputWindowMessage(OutputMessage)
38
39 Line1:
40
41 End Function
```



Name	Type
CruiseSpeed [Aircraft Model::Support Elements::Aircraft Types:m * s ⁻¹ - Aircraft Model::Structure::Aircraft Hierarchy::Aircraft]	BlockProperty
CL_Cr [Aircraft Model::Support Elements::Aircraft Types:Ratio - Aircraft Model::Structure::Aircraft Hierarchy::Aircraft]	BlockProperty
E_Cr [Aircraft Model::Support Elements::Aircraft Types:Ratio - Aircraft Model::Structure::Aircraft Hierarchy::Aircraft]	BlockProperty
CruiseMachNumber [Aircraft Model::Support Elements::Aircraft Types:Ratio - Aircraft Model::Structure::Aircraft Hierarchy::Aircraft]	BlockProperty
V/V_md [Aircraft Model::Support Elements::Aircraft Types:Ratio - Aircraft Model::Structure::Aircraft Hierarchy::Aircraft]	BlockProperty
ThrustToWeightRatio [Aircraft Model::Support Elements::Aircraft Types:Ratio - Aircraft Model::Structure::Aircraft Hierarchy::Aircraft]	BlockProperty
WingLoading [Aircraft Model::Support Elements::Aircraft Types:kg * m ⁻² - Aircraft Model::Structure::Aircraft Hierarchy::Aircraft]	BlockProperty
mPL [Aircraft Model::Support Elements::Aircraft Types:kg - Aircraft Model::Structure::Aircraft Hierarchy::Aircraft::MassPrediction]	BlockProperty
mMTO [Aircraft Model::Support Elements::Aircraft Types:kg - Aircraft Model::Structure::Aircraft Hierarchy::Aircraft::MassPrediction]	BlockProperty
mML [Aircraft Model::Support Elements::Aircraft Types:kg - Aircraft Model::Structure::Aircraft Hierarchy::Aircraft::MassPrediction]	BlockProperty
mMZf [Aircraft Model::Support Elements::Aircraft Types:kg - Aircraft Model::Structure::Aircraft Hierarchy::Aircraft::MassPrediction]	BlockProperty

Figure 5.34: After execution of the script, the calculated values are displayed at the Modeler result pane

CreateNewBlockProperty()

The *CreateNewBlockProperty()* function is required when a value calculated in MATLAB is to be stored back in the Modeler, but the corresponding block property does not exist. It can be called the following way (see ch. 5.5.7):

```
1 CreateNewBlockProperty("<BlockName>", "<BlockPropertyName>")
```

First the block is accessed under which the block property will be created. SysML blocks can be accessed through the PTC Automation Interface via UML *Class* objects [PTC15, p. 187]. If the block does not exist, the application is closed (line 7).

Once the block is accessed through the *Class* object, a new block property for this block has to be created. However, the PTC Automation Interface meta model does not allow the direct creation of block properties [PTC15]. Therefore, the following workaround must be performed.

First, a new *Part* object is created for the block (line 11 - 12). As mentioned in ch. Ch.SysMLBlockProperties, parts are the UML equivalent to SysML part properties.

In the next step the newly created *Part* object should be accessed via the *Name* attribute, as was explained earlier. However, it is not possible to specify the name of the *Part* object when creating it (see line 11 - 12). In order to access the newly created part, it is looped through all *Parts* object of the *Class* object (line 14 - 22). If one of the parts has no name, it must be the newly created part (line 19). It is therefore renamed to the desired block property name (line 20).

Now, the newly created part can be accessed through the block property name (line 25). This is necessary to assign the part as a block property. The newly created *Part* object

is assigned with the SysML block property stereotype (line 25 - 26). This turns an UML *Part* object into a SysML block property (see ch. 2.3.4).

```

1  Function CreateNewBlockProperty(BlockName, BlockPropertyName)
2
3  'Access block object
4  Block = Dictionary.Item("Class", BlockName)
5  If Block Is Nothing Then
6      MsgBox("Error: Block '" & BlockName & "' was not found!")
7      Environment.Exit(0)
8  End If
9
10 'Add a new part (UML equivalent for SysML block property of type Real)
11 BlockPropertyType = Dictionary.Item("Class", "Real")
12 Block.AddDirected("Part", BlockPropertyType)
13
14 'Loop through the parts of the block object until the newly created part ...
   is found (has no name)
15 Parts = Block.Items("Part")
16 Parts.ResetQueryItems()
17 Do While Parts.MoreItems
18     Part = Parts.NextItem
19     If Part("Name") = "" Then
20         Part("Name") = BlockPropertyName
21     End If
22 Loop
23
24 'Assigning the block property stereotype
25 BlockProperty = Dictionary.Item("Role", BlockPropertyName)
26 Dictionary.Item("Stereotype", "BlockProperty").Add("Model Object", ...
   BlockProperty)
27
28 OutputMessage = "Preliminary Sizing Tool [" + CStr(TimeOfDay) + "]:" + ...
   vbTab + "Created new block property " + BlockName + "/" + ...
   BlockPropertyName + vbCrLf
29 Studio.DisplayOutputWindowMessage(OutputMessage)
30
31 End Function

```

5.6 Summary

In this chapter, the SysML model in PTC Integrity, the calculation model in MATLAB and an interface for digitally linking the two models were introduced. In order to demonstrate many different input possibilities, the input parameters required for preliminary sizing are distributed in different SysML objects. The input parameters are stored in requirements, constraints, associations and block properties.

The objects can be accessed through an external Visual Basic program via the Automation Interface. Custom Visual Basic functions were developed to access the individual objects. The parameters are then passed from the Visual Basic program to MATLAB via the MATLAB Automation Server library. An optimization algorithm then calculates the minimum of a previously defined objective function. This minimum corresponds to

an optimal design according to the defined criteria. The calculated parameters are then fed back to the SysML model via the Visual Basic interface.

The digital interface offers many advantages. In particular, it allows time and cost savings, minimization of manual data transfer errors and collaborative engineering. In addition, the interface allows a rapid preliminary design of different configurations. Lastly, the digital interface can improve the overall aircraft design by minimizing the effort for additional design iterations.

6 Installation and Testing of the Visual Basic Interface

6.1 Installation of the Visual Basic Interface

The folder for the digital interface contains the SysML Aircraft model, the MATLAB calculation program and the Visual Basic application (see ch. MBSE). First, the SysML Aircraft model must be imported into the PTC Integrity Modeler. Afterwards, it can be configured that the Visual Basic application can be executed directly from the PTC Integrity Modeler.

6.1.1 Importing the Aircraft Model Into the PTC Integrity Modeler

First, the *Aircraft* model must be imported into the PTC Integrity Modeler. For this, the *Open Model* button located in the *Home* tab is pressed. In the opening pop-up window, the *Import Model* option is clicked (see fig. 6.1). The *Aircraft* model can then be selected and imported.

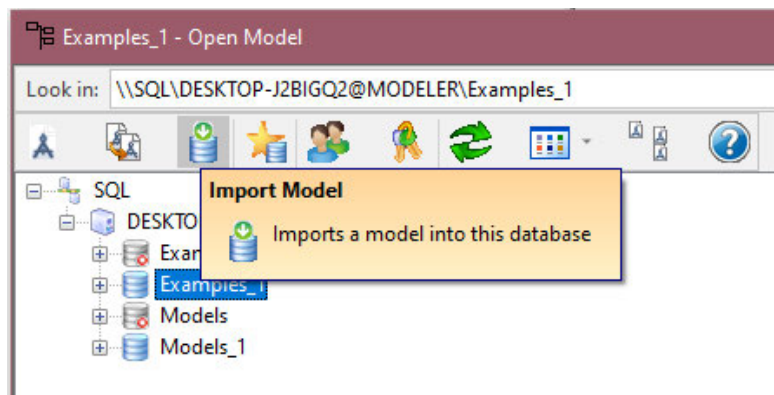



Figure 6.1: How to import the *Aircraft* model

6.1.2 Adding the Executable to the PTC Integrity Modeler Toolbar

It is possible to launch the executable directly from the tool bar of the Modeler ribbon in the PTC Integrity Modeler user interface. This allows the preliminary design to be easily performed directly from the Modeler.

To do this, the *Model* button in the upper right-hand corner of the Modeler must be pressed. The *Customize Tools* button opens a new window. A new tool can then be added by pressing the button in the upper right corner () (see fig. 6.2). After that, the executable file can be selected with the help of the *Browse* button.

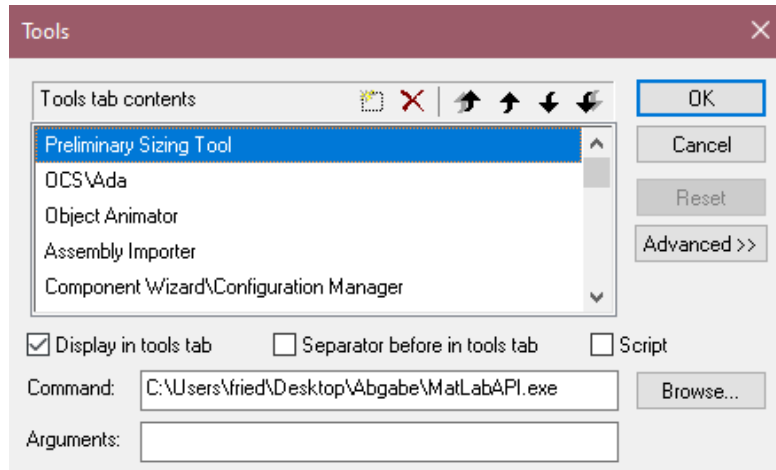


Figure 6.2: How to add an executable to the tools ribbon

After completion, a link to the Visual Basic program appears in the tool bar (see fig. 6.3). By pressing the "*Preliminary Sizing Tool*" button, the preliminary sizing can be performed quickly and easily from within the Modeler.

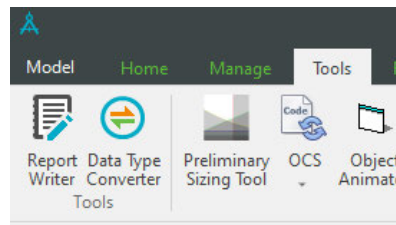


Figure 6.3: The executable file is pinned to the the tools ribbon

6.2 Testing of the Visual Basic Interface

In this chapter, the preliminary sizing tool in MATLAB together with the Visual Basic interface will be tested in practice. For this purpose, the calculation is performed with two different values for the landing field length s_{LFL} and the take-off field length s_{TOFL} . The input parameters for both calculations are listed in tab. 6.1 and tab. 6.2. The matching chart resulting from the first design is displayed in fig. 6.4.

Symbol	Unit	First design	Second design
γ_{CLB}	[]	0.0024	0.0024
γ_{MA}	[]	0.0024	0.0024
s_{LFL}	[m]	3000	2500
s_{TOFL}	[m]	3000	2500
R_{Alt}	[NM]	200	200
Route type	[]	domestic	domestic
Certification	[]	FAR-PAR25	FAR-PAR25
$C_{L,max,L}$	[]	3.4	3.4
$C_{L,max,TO}$	[]	2.6	2.6
Flap setting TO	[°]	15	15
m_{ML}/m_{MTO}	[]	0.88	0.88
m_{OE}/m_{MTO}	[]	0.446	0.446
μ	[]	9.6	9.6
A	[]	9	9
m_{CARGO}	[kg]	2000	2000
R	[NM]	7000	7000
n_{PAX}	[]	250	250
n_E	[]	2	2

Table 6.1: Fixed input parameters for the two preliminary designs

Symbol	Unit	Both designs
m_{MTO}/S_W	[]	$400 \leq m_{MTO}/S_W \leq 1300$
$T_{TO}/(m_{MTO} \cdot g)$	[]	$0.1 \leq T_{TO}/(m_{MTO} \cdot g) \leq 0.5$
V_{md}	[]	$0.8 \leq V_{md} \leq 1.4$
M_{Cr}	[]	$0.55 \leq M_{Cr} \leq 0.88$

Table 6.2: Variable input parameters for the two preliminary designs

Symbol	Unit	First design	Second design	Δ
m_{MTO}	[kg]	143622	147163	+3541
T_{TO}	[N]	262110	268572	+6462
S_W	[m ²]	115,8	142,4	+26,6

Table 6.3: Most important output parameters for the two preliminary designs

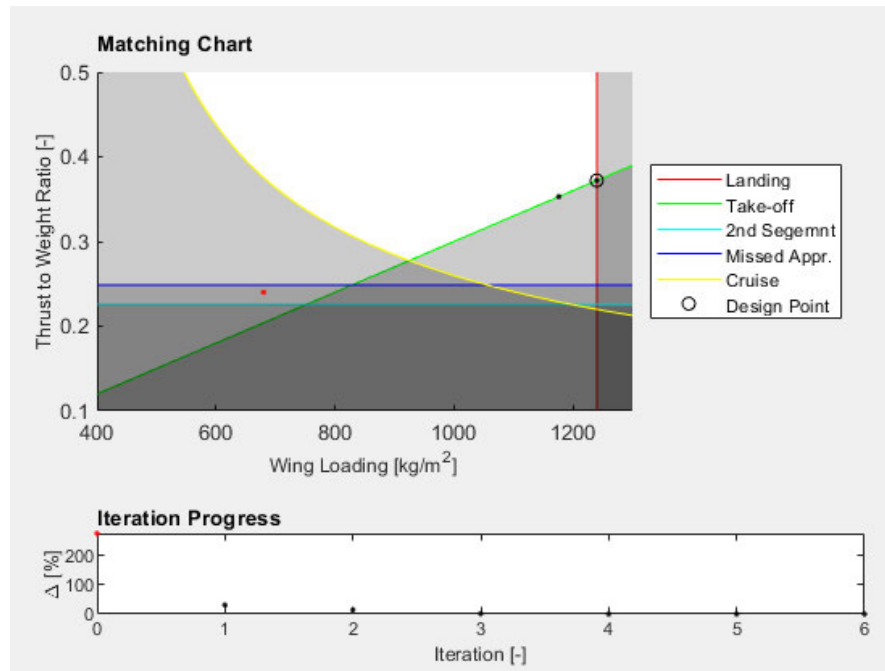


Figure 6.4: Matching chart of the first preliminary design

After the first preliminary design, both the landing field length and the take-off field length are reduced by 500 m. It could be, for example, that the customer's requirements have changed or that the preliminary design is to be carried out for a different configuration of the aircraft. The resulting matching chart is shown in fig. 6.5. It can be seen that the two curves for the take-off phase and the landing phase have shifted. Consequently, also the the design point was shifted. As a result, all output parameters of the preliminary sizing have changed. Some of the most important output parameters are shown in tab. 6.3.

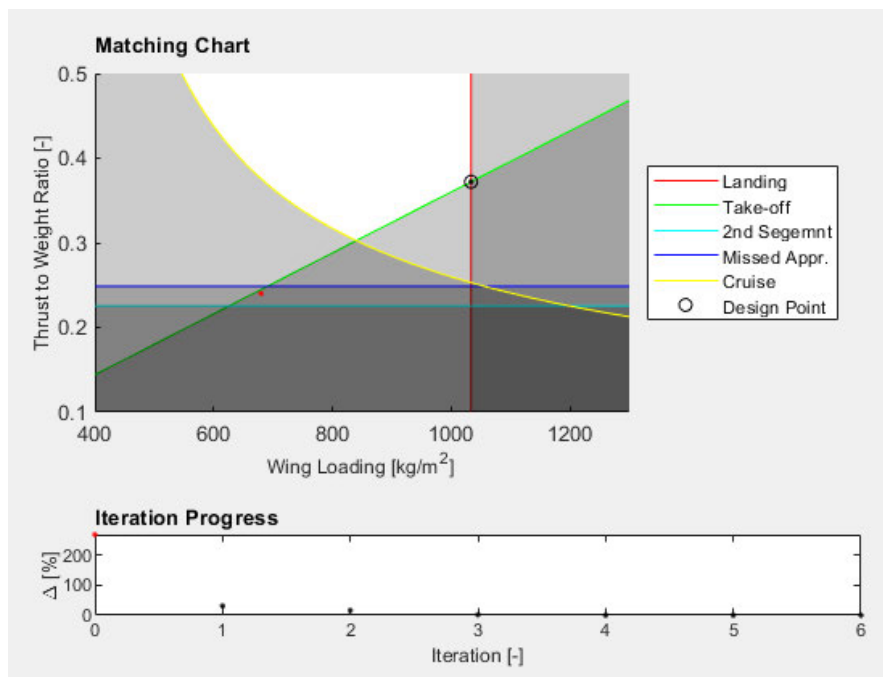


Figure 6.5: Matching chart of the second preliminary design

7 Discussion

The previous practical example has shown that an integration of the MATLAB analysis tool allows a rapid estimation of the effects of certain parameter changes from within the PTC Integrity Modeler. By eliminating the need for manual data transfer, the user only has to press one button and a few seconds later the preliminary design is complete.

The fact that the input parameters are first provided to Visual Basic before they are passed on to MATLAB provides a high degree of flexibility. Thus, other analysis programs could also be accessed from Visual Basic. For instance, it would be possible to access analysis programs other than MATLAB from Visual Basic. Furthermore, it is possible to access several different SysML models at the same. Especially in aircraft design, where many suppliers work together with different tools and models, PTC Integrity Modeler's compatibility with Visual Basic can be advantageous.

In addition, the Visual Basic interface is flexibly adaptable. For example, custom warnings have been implemented if a model item is not found. If the requirements for the interface change, they can be quickly and easily implemented. Another advantage of the chosen solution is the independence from software manufacturers such as *Phoenix Integration* (see ch. 5.5.3). This enables the integration of internal analysis software without having to wait for a solution from a different software manufacturer. Moreover, license fees can be reduced.

The mathematical calculation is done in the background. This means that the user does not need to have a knowledge about the mathematical relationships in aircraft preliminary sizing. The advantage is that this creates a special viewpoint on the system that initially the mathematical relationships. For example, a systems engineer could use the tool to generate requirements for sub-systems, e.g. the wings, from the TLAR. An aircraft designer would then have to ensure that the mathematical relationships of the tool are correct. To do so, he/she needs a different viewpoint on the system than the systems engineer, more precisely the view on the MATLAB model.

However, performing the calculation in the background can also be interpreted as a disadvantage. For example, it is possible that the user relies too much on the automatic parameter transfer. Thus, it errors during data transmission, for example due to programming errors in the Visual Basic code, might remain undetected.

In addition, the information about the structure of the mathematical analysis model is not stored inside the SysML model. When trying to understand the underlying calculation, one would have to first study the Visual Basic Code and then the MATLAB code in

detail. However, a MBSE approach would also integrate the mathematical relationships in the digital model (see ch. 2.4.6).

In ch. 5.3.9 it was attempted, to model the mathematical relationships of the MATLAB code in a parametric diagram. However, the parametric diagram is not linked to the MATLAB calculation model. As a result, the parametric diagram is also no longer up to date, should the MATLAB code change. A solution for this could be an even better integration of the analysis models into the SysML itself, rather than only the transfer of parameters between both models (see ch. 9).

Another disadvantage of the chosen solution is that it is difficult to perform changes in the interface. For instance, if an input parameter is moved from a value property to an association, the Visual Basic code of the interface would have to be modified. Then a new executable file would have to be generated from the updated Visual Basic code. This executable file would then have to be distributed among the aircraft designers. This is inefficient and impractical. Yet, a solution for this is described in ch. 9.

An additional disadvantage of digital linking only becomes apparent for very computationally intensive analysis models. In ch. 5.5.2, it was mentioned that it may be necessary to lock certain model items during the system analysis to ensure a consistent model. For time-consuming or frequently performed system analysis, integration of the analysis models into the system definition models could lead to inconsistencies within the SysML model.

Finally, it has been found that the start-up of MATLAB usually takes a long time when running the preliminary sizing tool for the first time. A solution would be to run the MATLAB Automation Server permanently in the background.

8 Summary

Model-based systems engineering (MBSE) is gaining in significance in aircraft design. This is mainly due to the increasing complexity, interconnectivity and the customer's demand for custom designs. In MBSE, all information such as the requirements, the design or the behaviour of a system is modelled in a digital model throughout all phases of its life cycle. The system model thus acts as a single source of truth. At the same time, aircraft design consists to a large extent of system analysis using special computation software. In an MBSE approach, the input values for these calculations are distributed within the digital model. In addition, the output values of the analyses must also be stored in the model. This thesis focused on the digital linking of system definition models and system analysis models in order to replace the manual parameter exchange between both models. The aim of this thesis is to investigate whether an automated digital parameter transfer between both models is technically feasible.

The research was carried out using the SysML-based modelling software PTC Integrity Modeler for the system definition and MATLAB for the system analysis. A first step was to create a model of the passenger aircraft, using the SysML. The model contains the requirements specified in the theTLAR and the mission definition, which are needed for preliminary sizing. Moreover, the relevant aircraft baseline architecture was modelled. Furthermore, a MATLAB calculation model was developed, which computes the optimal preliminary design with multiple variable input parameters. To connect the two models, a digital interface was programmed in Visual Basic. The Visual Basic program provides the input values to MATLAB, performs the preliminary design and feeds the calculated values back to the SysML model.

The preliminary sizing can be performed directly within the system definition software. This allows the impact of requirement changes on individual subsystems to be estimated and implemented quickly and efficiently. This is particularly important for the iterative approach of the aircraft design process. In addition, the preliminary design can be performed quickly for multiple different configurations, so they can be easily investigated and compared. The digital interface can save development time, development cost and can improve the final design. However, the digital link also has disadvantages. For instance, the digital interface is difficult to customize, there may be a lack of understanding of the underlying mathematical calculations and errors in parameter transfer may remain unnoticed. For time-consuming calculations, there may be problems with multi-user capability of the modelling software or inconsistencies in the model.

9 Conclusions and Recommendations

In summary, it can be concluded that the research goal of the thesis has been achieved. The developed Visual Basic program allows an automatic transfer of the input parameters to the MATLAB calculation model, performs the calculation automatically and feeds the calculated values back to the SysML model. The time required for the preliminary design could be significantly reduced by the digital connection, which provides all the advantages mentioned earlier.

However, this thesis only provides a pure parameter exchange between two models. The MATLAB calculation model is not integrated into the SysML model beyond the parameter exchange. This means that the structure of the calculation model is not apparent from within the SysML Modeler.

A solution would be an even stronger integration of the computation model into the SysML. For example, an attempt could be made to redesign the Visual Basic interface so that constraint properties always remain automatically synchronized with a MATLAB or Simulink calculation model (see [FRI08, p. 169])). To do so, a program using the Automation Interface would have to loop through all constraint parameters of the respective constraint property. For each constraint parameter, it would need to automatically recognize whether the constraint parameters are input or output parameters (see arrow direction in fig. 5.18). Additionally, the associated value property would have to be accessed via the binding connector relationship. This would have the advantage that the interface is not fixed but flexible. If new constraint parameters were added to the constraint property in the parametric diagram, they would automatically be available in Visual Basic program as well. Therefore, one would not have to edit the Visual Basic interface when adding new constraint parameters.

However, this approach also has some disadvantages. First, the storage of the parameters used in the analysis is limited to value properties. As a result, parameters stored associations or constraints cannot be included in system analysis. In addition, the input parameters could no longer be read directly from the requirements. If a requirement is changed, the associated value property in the parametric diagram would need to be manually modified. Furthermore, the parallel access to different models or the additional access to other programs from the Visual Basic program would be difficult. Besides, it should be reconsidered whether the programming effort for an adaptive interface is appropriate or whether the software solutions available on the market (see ch. 5.5.3) should be used.

Another possible enhancement would be to model the objective function within the

PTC Integrity Modeler as described in [FRI08, ch. 7.11]. So far, the objective function is defined within the MATLAB code only. Defining it within the model and transferring it to MATLAB would result in more variation potential when examining different aircraft configurations from within the Modeler.

A further developed would be to test the digital connection with other programs via the Automation Interface (see [HOL13, p. 639]). For example, a connection to CAD software might be possible. This way, a CAD model that is digitally linked to the SysML model could be developed. Furthermore, a connection to a database would be imaginable, in order to store the exchanged parameters of each calculation. This database could be used to collect company internal data for future developments.

In addition, further work could be done to refine the existing aircraft model with additional computational models. For example, the output parameters of the preliminary sizing could be further processed by other analysis models. If these calculation models were to remain interconnected deep into the design process, the ease of change in design could be reduced later in the design process (see fig. 9.1). On the one hand, this could reduce development costs. In addition, the design could be significantly improved, since the more product-specific knowledge increases within the design process (see fig. 9.1).

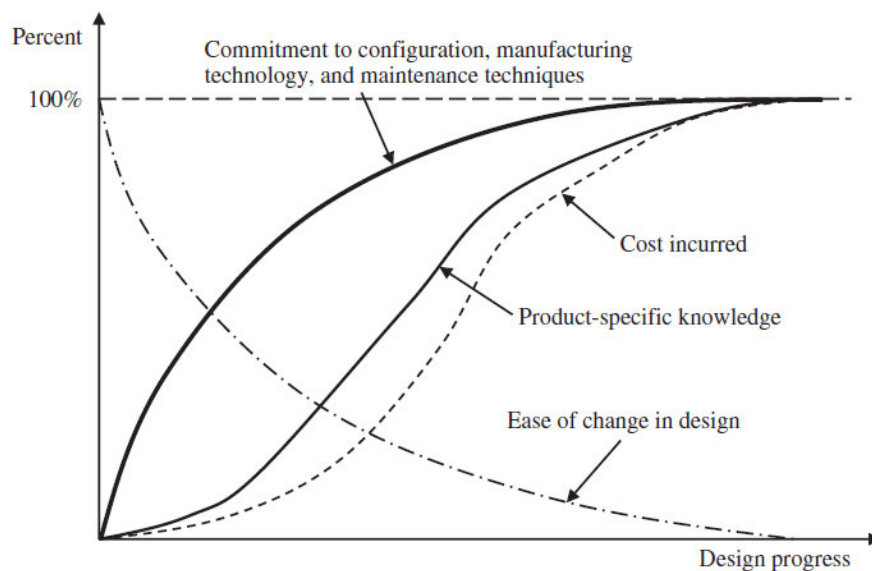


Figure 9.1: Status of various design features during the design process (taken from [SAD12, fig. 1.7])

Furthermore, the aircraft model could be extended to include preliminary design methods for other types of aircraft such as propellers or hydrogen powered aircraft. This would allow even more different aircraft configurations to be compared with each other.

Another improvement would be the development of a program that checks the SysML

model for changes in the background. Thanks to the multi-user capability of PTC Integrity Modeler, such a program could run on a dedicated server and access the SysML model from there. If changes are made to the input values for certain analysis tools, the analysis can be executed immediately. This would keep all parameters stored in the model up to date. Furthermore, the system analysis would be faster because the MATLAB application does not have to be started and closed each time. However, this approach would only be conceivable for minor analysis models (see fig. 5.24).

Lastly, the way in which the calculation is carried out can be developed further. So far, the calculation has been performed in a direction dependent way. The input parameters were passed to MATLAB, the calculation was performed and the calculated values were fed back to the SysML model. This is due to the causal, that is direction-dependent, nature of MATLAB. The direction dependence is however not determined by the SysML. If an acausal, direction-independent, computation program is used, direction-independent computations could be implemented. An example for an acausal calculation program is the object-oriented language Modelica [Ass21]. An acausal calculation software can be particularly useful for aircraft preliminary design, since it is sometimes not possible to define which parameters are fixed and which parameters are to be calculated.

Nevertheless, this thesis provided a good basis for further investigations. In particular, the research on the PTC Integrity Automation Interface and the developed Visual Basic program provide a good starting point. Furthermore, the MDO methods used in MATLAB can be applied to related design problems. Similarly, the Visual Basic functions presented in ch. 5.5.7 can be easily reused for other applications.

Bibliography

- [ALT09] Oliver ALT. *Car Multimedia Systeme Modell-basiert testen mit SysML*. Vieweg + Teubner, 2009.
- [ALT12] Oliver ALT. *Modellbasierte Systementwicklung mit SysML*. HANSER, 2012.
- [Ass21] Modelica Association. *Modelica Language*. Online. Available from <https://modelica.org/modelicalanguage.html> [Accessed March 2022]. 2021.
- [BAR16] Eric BARNHART. *How a Document-Based Approach Differs from Model-Based*. Online. Available from <https://vmcse.com/2016/03/20/document-based-is-not-model-based/> [Accessed February 2022]. Mar. 2016.
- [BOE11] John PALMER (BOEING). “Model-Based Systems Engineering without SysML”. In: *National Defense Industrial Association Systems Engineering Conference* (2011).
- [BOO98] Grady BOOCH. *The Unified Modeling Language User Guide*. Ed. by 1st. Addison Wesley, 1998.
- [CHA18] Bassim CHABIBI. “Model Integration Approach from SysML to MATLAB / Simulink”. In: *Journal of Digital Information Management* (2018).
- [EAS07] EASA. *Certification Specifications for Large Aeroplanes CS-25*. European Aviation Safety Agency, 2007.
- [EIS08] Howard EISNER. *Essentials of Project and Systems Engineering Management*. Ed. by Third. John Wiley & Sons, 2008.
- [FAA17] FAA. *Airworthiness Standards: Transport Category Airplanes (Part 25)*. Federal Aviation Administration, 2017.
- [FRI08] Sanford FRIEDENTHAL. *A Practical Guide to SysML: The Systems Modeling Language*. Morgan Kaufmann OMG Press, 2008.
- [Gro10] Object Management Group. “OMG Systems Modeling Language Specification, Version 1.2”. In: (2010). Accessed from <https://www.omg.org/spec/SysML/1.2/>.
- [Gro12] Object Management Group. *What is OMG SysML*. [Online]. Available from <https://www.omg-systemsml.org> [Accessed February 2022]. 2012.
- [HEI12] Joeri HEINEMANN. “Preliminary Sizing of FAR Part 23 and Part 25 Aircraft”. MA thesis. Hochschule für Angewandte Wissenschaften Hamburg, 2012.
- [HOL13] Jon HOLT. *SysML for Systems Engineering*. Ed. by 2nd. The Institution of Engineering and Technology, 2013.
- [INC04] INCOSE. *Systems Engineering Handbook*. International Council on Systems Engineering, 2004.

- [INC07] INCOSE. *Systems Engineering Vision 2020 (INCOSE-TP-2004-004-02)*. Tech. rep. International Council on Systems Engineering, 2007.
- [Inc19] PTC Inc. *SySim Tutorial for PTC Integrity Modeler 9.2*. 2019.
- [Inc22] No Magic Incorporated. *Cameo Requirements Modeler Plugin 2021x Documentation*. Online. Available from <https://docs.nomagic.com/> [Accessed March 2022]. 2022.
- [Int22] Phoenix Integration. *ModelCenter Integrate*. Available from <https://www.phoenix-int.com/product/modelcenter-integrate/> [Accessed March 2022]. 2022.
- [ISO11] ISO. *42010:2011(E), Systems and software engineering*. ISO/IEC/IEEE, 2011.
- [IWA15] Curtis IWATA. *Model-Based Systems Engineering in Concurrent Engineering Centers*. AIAA SPACE 2015 Conference and Exposition. Aug. 2015.
- [LOF80] Laurence K. LOFTIN. *Subsonic Aircraft Evolution and the Matching of Size to Performance*. National Aeronautics and Space Administration, 1980.
- [Mat22] MathWorks. *MATLAB Help Center*. Online. Available from <https://de.mathworks.com/help/matlab> [Accessed March 2022]. 2022.
- [MOO09] Alan MOORE. *Practical Guide to SysML*. Revised. The MK/OMG Press. Morgan Kaufmann, 2009.
- [NAC55] NACA. *Report 1235 - Standard Atmosphere*. National Advisory Committee for Aeronautics, 1955.
- [NIT12] Mihaela NITA. *Contributions to Aircraft Preliminary Design and Optimization*. Dr. Hut Verlag, 2012.
- [OMG15] OMG. *XML Metadata Interchange (XMI) Specification*. Object Management Group, June 2015. URL: <https://www.omg.org/spec/XMI/2.5.1>.
- [PET20] Nicolas PETEILH. *Challenging Top Level Aircraft Requirements based on operations analysis and data-driven models, application to takeoff performance design requirements*. Online. Available from <https://arc.aiaa.org/doi/10.2514/6.2020-3171> [Accessed March 2022]. June 2020.
- [PTC15] PTC. *Automation Interface User's Guide (Version 8.2)*. Document version 8.2.0. PTC Inc. 2015.
- [PTC19a] PTC. *Modeler Help*. Online. Available from <https://support.ptc.com/help/modeler/r9.2/en/> [Accessed March 2022]. 2019.
- [PTC19b] PTC. *PTC Model-Based Systems Engineering Tutorial for Integrity Modeler 9.2*. PTC Inc. 140 Kendrick Street, Needham, MA 02494 USA, 2019.
- [RAY89] Daniel P. RAYMER. *Aircraft Design: A Conceptual Approach*. AIAA Education Series, 1989.
- [ROS85] Jan ROSKAM. *Part I: Preliminary Sizing of Airplanes*. Roskam Aviation and Engineering Corporation, 1985.
- [SAD12] Mohammad H. SADRAEY. *Aircraft Design: A Systems Engineering Approach*. 2012.
- [SCH19] Dieter SCHOLZ. *Aircraft Design Lecture Notes*. Lecture Notes. Available from <https://www.fzt.haw-hamburg.de/pers/Scholz/H00U/> [Accessed February 2022]. 2019.

-
- [uml20] uml-diagrams.org. Online. Available from: <https://www.uml-diagrams.org/> [Accessed February 2022]. 2020.
- [VDI21] VDI. *VDI/VDE 2206*. Online. Available from: <http://www.vdi.de/2206> [Accessed February 2022]. Nov. 2021.
- [VIN18] Madonna VINCENZO. “Electrical Power Generation in Aircraft: Review, Challenges, and Opportunities”. In: *IEEE Transactions on Transportation Electrification*, Vol. 4, No. 3 (2018).

Appendix

Appendix A: ReadVariantParameterValue() Function

```
1 Dim <VariantParameterValue> As Double = ...
   ReadVariantParameterValue("<DecisionSetName>", "<VariantParametername>", ...
   "<Long>/<String>")
```

```
1 Function ReadVariantParameterValue(DecisionSetName, VariantName, ...
   VariantParameterType) As String
2
3   'Get Decision Set Object
4   DecisionSet = Dictionary.Item("Decision Set", DecisionSetName)
5   VariantObj = Dictionary.Item("Variant", VariantName)
6
7   'Loop through Decision Set Variant Parameters
8   If DecisionSet Is Nothing Then
9       MsgBox("Error: Decision Set '" & DecisionSetName & "' was not found!")
10      Environment.Exit(0)
11  ElseIf VariantObj Is Nothing Then
12      MsgBox("Error: Variant '" & VariantName & "' was not found!")
13      Environment.Exit(0)
14  Else
15
16      'First save the default value for the variable
17      VariantParameter = VariantObj.Item("Parameter")
18      ReadVariantParameterValue = VariantParameter("Default " & ...
   VariantParameterType)
19
20      'If the default value is changed in the decision set, overwrite it
21      DecisionSetVariantParameters = DecisionSet.Items("Actual Variant ...
   Parameter")
22      DecisionSetVariantParameters.ResetQueryItems()
23
24      Do While DecisionSetVariantParameters.MoreItems
25          DecisionSetVariantParameter = DecisionSetVariantParameters.NextItem
26          VariantParameter = DecisionSetVariantParameter.Item("Formal")
27          If VariantParameter("Variant") = VariantName Then
28              ReadVariantParameterValue = ...
   DecisionSetVariantParameter(VariantParameterType & " Value")
29          End If
30      Loop
31
32      OutputMessage = "Preliminary Sizing Tool [" & CStr(TimeOfDay) & "]:" ...
   + vbTab + "Variant Parameter: " + vbTab + vbTab + VariantName + ...
   " = " + CStr(ReadVariantParameterValue) + vbCrLf
33      Studio.DisplayOutputWindowMessage(OutputMessage)
34
35  End If
```

```

36
37 End Function

```

Appendix B: The MATLAB Code

```

1 % 1. INPUT VALUES ARE STORED IN THE STRUCT
2     %Requirements
3     DATA.INP.climb_grad_MA = climb_grad_MA/100;
4     DATA.INP.climb_grad_CLB = climb_grad_CLB/100;
5     DATA.INP.s_LFL = s_LFL;
6     DATA.INP.s_TOFL = s_TOFL;
7     DATA.INP.R_Alt = R_Alt;
8     DATA.INP.Certification = convertCharsToStrings(Certification);
9     DATA.INP.RouteType = convertCharsToStrings(RouteType);
10    %Block Properties
11    DATA.INP.CL_max_L = CL_max_L;
12    DATA.INP.CL_max_TO = CL_max_TO;
13    DATA.INP.FlapSetting = FlapSetting;
14    DATA.INP.m_ML_DIV_m_MTO = m_ML_DIV_m_MTO;
15    DATA.INP.m_OE_DIV_m_MTO = m_OE_DIV_m_MTO;
16    DATA.INP.BPR = BPR;
17    DATA.INP.AR = AR;
18    DATA.INP.m_CARGO = m_CARGO;
19    DATA.INP.R = R;
20    %Constraints
21    DATA.INP.lb = [WL_LOWER, TtWR_LOWER, V_DIV_V_md_LOWER, M_LOWER];
22    DATA.INP.ub = [WL_UPPER, TtWR_UPPER, V_DIV_V_md_UPPER, M_UPPER];
23    %Assotiations
24    DATA.INP.n_PAX = n_PAX;
25    DATA.INP.n_E = n_E;
26
27    %Fixed Values
28    DATA.INP.g = 9.81;
29    DATA.INP.C_D0 = 0.02;
30    DATA.INP.Delta_C_D_slat = 0;
31    DATA.INP.e = 0.7;
32    DATA.INP.e_Cr = 0.85;
33    DATA.INP.k_E = 14.5;
34    DATA.INP.S_wet_DIV_S_W = 6.1;
35    DATA.INP.m_PAX = 93;
36    DATA.INP.SFC_Cr = 14.2 * 10^-6;
37    DATA.INP.SFC_Loiter = 11.3 * 10^-6;
38    DATA.INP.t_loiter = 1800;
39    DATA.INP.M_ff_TO = 0.995;
40    DATA.INP.M_ff_CLB = 0.980;
41    DATA.INP.M_ff_DES = 0.990;
42    DATA.INP.M_ff_L = 0.992;
43
44 %2. SETTING UP THE LINEAR INEQUALITY CONSTRAINTS
45     % 1 - Landing

```

```

46     DATA.CONST.MaxWingLoading1 = ( 0.107 * DATA.INP.CL_max_L * ...
        DATA.INP.s_LFL ) / DATA.INP.m_ML_DIV_m_MTO;
47
48     % 3 - Climb-Rate in the second segment
49     if DATA.INP.FlapSetting == 15
50         Delta_C_D_flap = 0.01;
51         C_L = 1.3;
52     else if DATA.INP.FlapSetting == 25
53         Delta_C_D_flap = 0.02;
54         C_L = 1.5;
55     else if DATA.INP.FlapSetting == 35
56         Delta_C_D_flap = 0.03;
57         C_L = 1.7;
58     else
59         disp('Error: Flap setting is not <15>, <25> or <35> ...
            degrees')
60     end
61 end
62 end
63 DATA.CONST.E_2nd = C_L / ( ( DATA.INP.C_D0 + Delta_C_D_flap + ...
        DATA.INP.Delta_C_D_slat ) + (C_L^2 / (pi * DATA.INP.AR * ...
        DATA.INP.e)));
64 DATA.CONST.MinThrustToWeightRatio3 = (DATA.INP.n_E/(DATA.INP.n_E ...
        - 1)) * ((1/DATA.CONST.E_2nd) + sin(DATA.INP.climb_grad_CLB));
65
66 %4 - Climb rate during the missed approach
67 if DATA.INP.Certification == 'FAR-PAR25'
68     DATA.CONST.Delta_C_D_gear = 0.015 ;% (for FAR Part 25 ...
        Certification)
69 else
70     DATA.CONST.Delta_C_D_gear = 0 ;
71 end
72 DATA.CONST.E_MA = C_L / ( ( DATA.INP.C_D0 + Delta_C_D_flap + ...
        DATA.INP.Delta_C_D_slat + DATA.CONST.Delta_C_D_gear ) + ...
        (C_L^2 / (pi * DATA.INP.AR * DATA.INP.e)));
73 DATA.CONST.MinThrustToWeightRatio4 = (DATA.INP.n_E/(DATA.INP.n_E ...
        - 1)) * ((1/DATA.CONST.E_MA) + sin(DATA.INP.climb_grad_MA));
74
75 % 3. DEFINING THE OPTIMIIZATION PARAMETERS
76 A = [1,0,0,0;( 2.34 / (DATA.INP.s_TOFL * DATA.INP.CL_max_TO) ...
        ),-1,0,0;0,-1,0,0;0,-1,0,0];
77
78 b = [DATA.CONST.MaxWingLoading1,0,
79     -DATA.CONST.MinThrustToWeightRatio3,
80     -DATA.CONST.MinThrustToWeightRatio4];
81
82 outputfn = @(x,optimValues,state)outfun(x,optimValues,state);
83 opts = optimoptions('fmincon',"Algorithm","active-set",'OutputFcn',
84     outputfn,'HonorBounds',true);
85
86 % 4. PERFORMING THE OPTIMIZATION
87 [x,OptValue,exitflag,output] = fmincon(@(x) RedMTOW(x,DATA),0.5 * ...
        (DATA.INP.lb + ...
        DATA.INP.ub),A,b,[],[],DATA.INP.lb,DATA.INP.ub,@(x) ...

```

```

        CruiseConstraint(x,DATA),opts);
88     clc;
89     disp(extractBefore(output.message, '.'))
90     [OptValue,DATA] = RedMTOW(x,DATA);
91     disp(['MTOW = ',num2str(round(DATA.OUT.m_MTO,0)), ' kg'])
92
93     % 5. FURTHER CALCULATION AND PLOTTING
94     DATA = FurtherCalcAndPlot(DATA);
95
96     %plotting the optimization progress in the upper chart
97     figure(1)
98     hold on
99     for i = 1:size(IterationOutput,1)
100         if IterationOutput(i,2) ≤ 10-6
101             plot(IterationOutput(i,4),IterationOutput(i,5),'k. ');
102             hold on
103         else
104             plot(IterationOutput(i,4),IterationOutput(i,5) , 'r. ');
105             hold on
106         end
107     end
108     legend('','Landing','','Take-off','','2nd Segemnt','','Missed ...
        Appr.','','Cruise','Design Point','Location','eastoutside')
109
110     %plotting the optimization progress in the lower chart
111     p = subplot('Position',[0.1 0.1 0.85 0.12]);
112     for i = 1:size(IterationOutput,1)
113         if IterationOutput(i,2) ≤ 10-6
114             plot(IterationOutput(i,1),(IterationOutput(i,3) - OptValue) ...
                * 100/OptValue,'k. ');
115             hold on
116         else
117             plot(IterationOutput(i,1),(IterationOutput(i,3) - OptValue) ...
                * 100/OptValue,'r. ');
118             hold on
119         end
120     end
121
122     xlabel('Iteration [-]','FontSize', 9);
123     ylabel('\Delta [%]');
124     title('Iteration Progress','FontSize', 10);
125     p.TitleHorizontalAlignment='left';
126     hold off
127
128     %6. OUTPUT VALUES ARE READ FROM STRUCT
129     h_Cr = round(DATA.OUT.h_Cr,0);
130     V_Cr = round(DATA.OUT.V_Cr,2);
131     C_L_Cr = round(DATA.OUT.C_L_Cr,3);
132     E_Cr = round(DATA.OUT.E_Cr,2);
133     M = round(DATA.OUT.M,3);
134     V_DIV_V_md = round(DATA.OUT.V_DIV_V_md,4);
135     ThrustToWeightRatioDesign = round(DATA.OUT.ThrustToWeightRatioDesign,2);
136     WingLoadingDesign = round(DATA.OUT.WingLoadingDesign,2);
137     m_PL = round(DATA.OUT.m_PL,0);

```

```

138     m_MTO = round(DATA.OUT.m_MTO,0);
139     m_ML = round(DATA.OUT.m_ML,0);
140     m_MZF = round(DATA.OUT.m_MZF,0);
141     m_OE = round(DATA.OUT.m_OE,0);
142     m_F = round(DATA.OUT.m_F,0);
143     T_TO_per_engine = round(DATA.OUT.T_TO_per_engine,2);
144     A_Wing = round(DATA.OUT.A_Wing,2);
145
146
147     disp('Calculation has finished')

```

Appendix C: The CruiseConstraint() Function

```

1 function [c,ceq] = CruiseConstraint(x,DATA)
2     C_L_Cr= (pi * DATA.INP.AR * DATA.INP.e_Cr) / ( 2 * x(3)^2 * ...
3         DATA.INP.k_E * sqrt(DATA.INP.AR / DATA.INP.S_wet_DIV_S_W ) );
4     E_Cr = (2 * DATA.INP.k_E * sqrt(DATA.INP.AR/DATA.INP.S_wet_DIV_S_W)) ...
5         / ((x(3)^2) + (1 / x(3)^2));
6     p_Cr = ( x(1) * 2 * DATA.INP.g ) / ( 1.4 * C_L_Cr* x(4) );
7     h_Cr = atmspalt(p_Cr);
8     MinThrustToWeightRatio5 = 1 / ( ( (0.0013 * DATA.INP.BPR - 0.0397) * ...
9         (h_Cr / 1000) - 0.0248 * DATA.INP.BPR + 0.7125 ) * E_Cr);
10    c = MinThrustToWeightRatio5 - x(2);
11    ceq = [];
12 end

```

Appendix D: The RedMTOW() Function

```

1 function [OptValue,DATA] = RedMTOW(x,DATA)
2     DATA.OUT.WingLoadingDesign = x(1);
3     DATA.OUT.ThrustToWeightRatioDesign = x(2);
4     DATA.OUT.V_DIV_V_md = x(3);
5     DATA.OUT.M = x(4);
6
7
8     %Calculation of V_Cr
9     DATA.OUT.C_L_Cr= (pi * DATA.INP.AR * DATA.INP.e_Cr) / ( 2 * ...
10        DATA.OUT.V_DIV_V_md^2 * DATA.INP.k_E * sqrt(DATA.INP.AR / ...
11        DATA.INP.S_wet_DIV_S_W ) );
12    DATA.OUT.E_Cr = (2 * DATA.INP.k_E * ...
13        sqrt(DATA.INP.AR/DATA.INP.S_wet_DIV_S_W)) / ...
14        ((DATA.OUT.V_DIV_V_md^2) + (1 / DATA.OUT.V_DIV_V_md^2));

```

```

11     DATA.OUT.p_Cr = ( DATA.OUT.WingLoadingDesign * 2 * DATA.INP.g ) ...
12         / ( 1.4 * DATA.OUT.C_L_Cr * DATA.OUT.M^2 );
13     DATA.OUT.h_Cr = atmospalt(DATA.OUT.p_Cr);
14     [null, DATA.OUT.a_h_Cr, null, null] = atmosisa(DATA.OUT.h_Cr);
15     DATA.OUT.V_Cr = DATA.OUT.a_h_Cr * DATA.OUT.M;
16     % Calculation of MTOW
17     DATA.INP.R = DATA.INP.R * 1852;
18     DATA.INP.R_Alt = DATA.INP.R_Alt * 1852;
19
20
21     if DATA.INP.RouteType == 'international'
22         DATA.INP.R_Alt = DATA.INP.R_Alt * 1.05;
23     else if DATA.INP.RouteType == 'domestic'
24         DATA.INP.R_Alt = DATA.INP.R_Alt * 1.00;
25     else
26         disp('Error: Route type not <international> or <domestic>')
27     end
28 end
29
30
31     DATA.OUT.B_Cr = ( DATA.OUT.E_Cr * DATA.OUT.V_Cr) / ( DATA.INP.g ...
32         * DATA.INP.SFC_Cr);
33     DATA.OUT.B_Loiter = ( DATA.OUT.E_Cr * DATA.OUT.V_Cr) / ( ...
34         DATA.INP.g * DATA.INP.SFC_Loiter);
35     DATA.OUT.M_ff_CR = exp( -(DATA.INP.R./DATA.OUT.B_Cr));
36     DATA.OUT.M_ff_Alt = exp( -(DATA.INP.R_Alt./DATA.OUT.B_Cr));
37     DATA.OUT.M_ff_LOITER = exp( -(DATA.INP.t_loiter * ...
38         DATA.OUT.V_Cr)/DATA.OUT.B_Loiter));
39
40     DATA.OUT.M_ff_std = DATA.OUT.M_ff_CR * DATA.INP.M_ff_TO * ...
41         DATA.INP.M_ff_CLB * DATA.INP.M_ff_DES * DATA.INP.M_ff_L;
42     DATA.OUT.M_ff_res = DATA.OUT.M_ff_Alt * DATA.OUT.M_ff_LOITER * ...
43         DATA.INP.M_ff_CLB * DATA.INP.M_ff_DES;
44     DATA.OUT.M_ff = DATA.OUT.M_ff_std * DATA.OUT.M_ff_res;
45
46     DATA.OUT.m_PL = DATA.INP.n_PAX * DATA.INP.m_PAX + ...
47         DATA.INP.m_CARGO ;
48     DATA.OUT.m_F_DIV_m_MTO = 1 - DATA.OUT.M_ff ;
49     DATA.OUT.m_MTO = DATA.OUT.m_PL / (1 - ( DATA.OUT.m_F_DIV_m_MTO ) ...
50         - ( DATA.INP.m_OE_DIV_m_MTO ));
51     OptValue = DATA.OUT.m_MTO + DATA.OUT.ThrustToWeightRatioDesign;
52 end

```

Appendix E: The FurtherCalcAndPlot() Function

```

1 function DATA = FurtherCalcAndPlot(DATA)
2
3     % Curve for plotting - Take Off

```



```

4      DATA.CONST.MinThrustToWeightRatio2 = ( 2.34 / (DATA.INP.s_TOFL * ...
          DATA.INP.CL_max_TO) ) * DATA.OUT.WingLoadingDesign;
5  %5 Curve for plotting - Cruise
6      h = [0:100:18000]';%m
7      T_CR_DIV_T_TO = (0.0013 * DATA.INP.BPR - 0.0397) * (h / 1000) - ...
          0.0248 * DATA.INP.BPR + 0.7125;
8      ThrustToWeightRatio5 = 1 ./ (T_CR_DIV_T_TO * DATA.OUT.E_Cr);
9      [null, null, p, null] = atmosisa(h);
10     WingLoading5 = ( DATA.OUT.C_L_Cr* DATA.OUT.M^2 * 1.4 * p ) / (2 ...
          *DATA.INP.g);
11
12     %Additional Calculation
13     DATA.OUT.m_ML = DATA.INP.m_ML_DIV_m_MTO * DATA.OUT.m_MTO;
14     DATA.OUT.m_OE = DATA.INP.m_OE_DIV_m_MTO * DATA.OUT.m_MTO;
15     DATA.OUT.m_F = DATA.OUT.m_F_DIV_m_MTO * DATA.OUT.m_MTO;
16
17     DATA.OUT.m_MZF = DATA.OUT.m_PL + DATA.OUT.m_OE;
18     DATA.OUT.m_fuel_res = DATA.OUT.m_MTO * (1 - DATA.OUT.M_ff_res);
19
20     if DATA.OUT.m_ML ≥ DATA.OUT.m_MZF + DATA.OUT.m_fuel_res
21         disp('Sizing finished')
22     else
23         disp('Error: Payload must be reduced!')
24     end
25
26     DATA.OUT.A_Wing = (DATA.OUT.m_MTO) / DATA.OUT.WingLoadingDesign;
27     DATA.OUT.T_TO = DATA.OUT.m_MTO * DATA.INP.g * ...
          DATA.OUT.ThrustToWeightRatioDesign;
28     DATA.OUT.T_TO_per_engine = DATA.OUT.T_TO / DATA.INP.n_E;
29
30
31     %Plot
32     figure(1)
33     p = subplot('Position',[0.1 0.4 0.85 0.5]);
34     ShadeAlpha = 0.2 ;
35     ShadeColor = 'black' ;%uint8([224 224 224])
36     hold on
37     axis([DATA.INP.lb(1) DATA.INP.ub(1) DATA.INP.lb(2) ...
          DATA.INP.ub(2)])
38     xlabel('Wing Loading [kg/m^2]','FontSize', 9);
39     ylabel('Thrust to Weight Ratio [-]','FontSize', 9);
40     title('Matching Chart','FontSize', 10);
41     p.TitleHorizontalAlignment='left';
42
43     %1 - Landing Distance
44     ShadeArea = ...
          patch([DATA.CONST.MaxWingLoading1,DATA.CONST.MaxWingLoading1,
45             DATA.CONST.MaxWingLoading1*100,
46             DATA.CONST.MaxWingLoading1*100], [-100,100,100,-100], 'black');
47     ShadeArea.FaceAlpha = ShadeAlpha;
48     ShadeArea.FaceColor = ShadeColor;
49     plot([DATA.CONST.MaxWingLoading1,
50         DATA.CONST.MaxWingLoading1], [-100,100], 'r-');
51

```

```

52     %2 - Take-Off Distance
53     ShadeArea = patch([0,DATA.OUT.WingLoadingDesign*10,
54     DATA.OUT.WingLoadingDesign*10],
55     [0,DATA.CONST.MinThrustToWeightRatio2*10,0], 'black');
56     ShadeArea.FaceAlpha = ShadeAlpha;
57     ShadeArea.FaceColor = ShadeColor;
58     plot([0,DATA.OUT.WingLoadingDesign*10],
59     [0,DATA.CONST.MinThrustToWeightRatio2*10], 'g-');
60
61     %3 - Climb rate in the second segment
62     ShadeArea = ...
63         patch([0,10000,10000,0], [DATA.CONST.MinThrustToWeightRatio3,
64     DATA.CONST.MinThrustToWeightRatio3,-100,-100], 'black');
65     ShadeArea.FaceAlpha = ShadeAlpha;
66     ShadeArea.FaceColor = ShadeColor;
67     plot([0,10000], [DATA.CONST.MinThrustToWeightRatio3,
68     DATA.CONST.MinThrustToWeightRatio3], 'c-')
69
70     %4 - Climb rate during the missed approach
71     ShadeArea = ...
72         patch([0,10000,10000,0], [DATA.CONST.MinThrustToWeightRatio4,
73     DATA.CONST.MinThrustToWeightRatio4,-100,-100], 'black');
74     ShadeArea.FaceAlpha = ShadeAlpha;
75     ShadeArea.FaceColor = ShadeColor;
76     plot([0,10000], [DATA.CONST.MinThrustToWeightRatio4,
77     DATA.CONST.MinThrustToWeightRatio4], 'b-')
78
79     %5 - Cruise
80     ShadeArea = area(WingLoading5,ThrustToWeightRatio5);
81     ShadeArea.FaceAlpha = ShadeAlpha;
82     ShadeArea.FaceColor = ShadeColor;
83     plot(WingLoading5,ThrustToWeightRatio5, 'y-');
84
85     %Design Point
86     scatter(DATA.OUT.WingLoadingDesign,
87     DATA.OUT.ThrustToWeightRatioDesign, 'ko');
88
89     hold off
90 end

```

Appendix F: The outfun() Function

```

1 function stop = outfun(x,optimValues,state)
2     persistent Iteration constrviolation fval WingLoadingDesign ...
3     ThrustToWeightRatioDesign
4     stop = false;
5     switch state
6     case 'init'

```

```
7         Iteration = [];  
8         constrviolation = [];  
9         fval = [];  
10        WingLoadingDesign = [];  
11        ThrustToWeightRatioDesign = [];  
12  
13  
14        case 'iter'  
15            Iteration = [Iteration; optimValues.iteration];  
16            constrviolation = [constrviolation; ...  
17                optimValues.constrviolation];  
18            fval = [fval; optimValues.fval];  
19            WingLoadingDesign = [WingLoadingDesign; x(1)];  
20            ThrustToWeightRatioDesign = [ThrustToWeightRatioDesign; x(2)];  
21  
22        case 'done'  
23            assignin('base', 'IterationOutput'  
24                [Iteration, constrviolation, fval, WingLoadingDesign,  
25                ThrustToWeightRatioDesign]);  
26        otherwise  
27        end  
end
```

CD Contents

The attached CD includes the following contents:

- The *Files* folder contains:
 - The *Aircraft.zip* file, which contains the digital SysML of model the aircraft
 - The *Main.m* file, which contains the created MATLAB script for aircraft preliminary sizing
 - The *MatLabAPI.exe* file. This is the compiled Visual Basic program that performs the automatic parameter exchange between the two models
- The *PDF* folder contains:
 - A *pdf* version of this work without visible hyper links
 - A *pdf* version of this work with visible hyper links
 - A *pdf* version of the abstract



Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Gemäß der Allgemeinen Prüfungs- und Studienordnung ist zusammen mit der Abschlussarbeit eine schriftliche Erklärung abzugeben, in der der Studierende bestätigt, dass die Abschlussarbeit „– bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit [(§ 18 Abs. 1 APSO-TI-BM bzw. § 21 Abs. 1 APSO-INGI)] – ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt wurden. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich zu machen.“

Quelle: § 16 Abs. 5 APSO-TI-BM bzw. § 15 Abs. 6 APSO-INGI

Dieses Blatt, mit der folgenden Erklärung, ist nach Fertigstellung der Abschlussarbeit durch den Studierenden auszufüllen und jeweils mit Originalunterschrift als letztes Blatt in das Prüfungsexemplar der Abschlussarbeit einzubinden.

Eine unrichtig abgegebene Erklärung kann -auch nachträglich- zur Ungültigkeit des Studienabschlusses führen.

Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,

Name: Friedrichs-Dachale

Vorname: Maximilian

dass ich die vorliegende Bachelorarbeit bzw. bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit – mit dem Thema:

Establishment of a Digital Interface Between System Definition and System Analysis Models to Optimize the Aircraft Preliminary Sizing Process

ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

- die folgende Aussage ist bei Gruppenarbeiten auszufüllen und entfällt bei Einzelarbeiten -


Die Kennzeichnung der von mir erstellten und verantworteten Teile der -bitte auswählen- ist erfolgt durch:

Hamburg

Ort

04.04.2022

Datum


Unterschrift im Original