Development of a Software Architecture for a Reconfigurable Aircraft Design System

Lukas Müller

Development of a Software Architecture for a Reconfigurable Aircraft Design System

Thesis report

by

Lukas Müller

in partial fulfillment of the requirements

for the degree of Master of Science

in Aerospace Engineering

at the Delft University of Technology,

to be defended publicly on 24th November 2023 at 14:00.

Thesis committee:

Chair:	Dr. ir. M.F.M. Hoogreef
Supervisor:	Dr. ir. G. la Rocca
External examiner:	Dr. ir. M.M. van Paassen

TU Delft Delft University of Technology

This report is based on the TU Delft report template and was created with the LaTeX typesetting system. For this report the AIAA reference style was adopted. The cover picture shows an Airbus A320neo and consists only of ASCII characters.
An electronic version of this report is available at `https://repository.tudelft.nl/`.

Preface

I am pleased to present this Master's thesis report on aircraft design systems. Positioned at the intersection of aerospace engineering and software engineering, these systems have captivated my interest since the beginning of this thesis project. This report aims to provide both interesting and practical insights for individuals working with or being curious about the intricacies of these systems. My hope is that the information presented here contributes to the ongoing improvement of aircraft design systems, making them even more effective tools for exploring innovative aircraft concepts and technologies that hold the potential to shape the future of aviation.

I want to express my sincere gratitude to Reno Elmendorp, Gianfranco la Rocca, and Maurice Hoogreef, for their enduring support, constructive feedback, and the critical discussions that have significantly influenced the development of this thesis. I also extend my deepest thanks to my entire family for their steadfast support and their continuous encouragement. A special acknowledgment is reserved for my brother Elias, whose invaluable support was particularly crucial during the most demanding phase of this project. Lastly, I extend my thanks to everyone who contributed to this endeavor – whether through reviewing drafts, providing feedback, or cheering me up – your collective support has been truly appreciated.

Lukas Müller
Delft, November 2023

Summary

Aircraft design systems are frequently used to synthesize aircraft designs. However, it is inherently difficult to reconfigure these software systems to facilitate a broader range of design studies (e.g. optimization or sensitivity studies) and to address follow-up questions about the synthesized aircraft designs. This report presents an investigation into the feasibility of developing a reconfigurable aircraft design system.

Firstly, the issues preventing current aircraft design systems from being used in a reconfigurable way are identified. These issues are closely tied to the intricate and tightly integrated nature of the source code that underpins these systems. This is unsurprising, given that these systems have typically been devised by experts in aircraft design (with varying expertise in software design) who are primarily interested in solving concrete design problems rather than creating sophisticated source code. In particular, the extensive design logic, characterized by high cyclomatic complexity, combined with cluttered and ambiguous design data structures, makes it challenging to comprehend and adapt the functioning of these systems.

An iterative development methodology is employed to come up with a software architecture aimed at mitigating the identified issues. A number of distinct architectural elements are devised that leverage a centralized semantic data management approach and a standardized interface for the formulation of modular analysis & sizing methods. Furthermore, a prototype aircraft design system that serves as a reference implementation for this architecture is developed. The prototype incorporates a graph database, a self-explanatory ontology (defining the semantics of the data stored in the database), and several abstract base classes for encapsulating the computation logic contained within typical analysis & sizing methods used during aircraft design studies. Concrete instances of these base classes are supposed to interact with the database through a well-defined endpoint interface, which includes extensive logging capabilities.

The prototype aircraft design system exhibits promising characteristics: The semantic data management approach facilitates the creation of a genuinely unambiguous and flexible data model. Simultaneously, it helps uncover inconsistencies and limitations in the employed analysis & sizing methods. Treating analysis & sizing methods as modular and nested instances of standardized classes appears to be the key to achieving reconfigurability. In addition to the unit-testability of these classes, the self-visualization features incorporated within can significantly enhance the comprehensibility and transparency of the analysis & sizing methods.

The architecture development, repository configuration, ontology formulation, and interface generation took a significant amount of time. Furthermore, some critical challenges surfaced, necessitating further investigation. Specifically, some of the employed analysis & sizing methods feature limitations and implicit assumptions that are required for a synthesis system but may need to be revised before being used in a reconfigurable design system based on the proposed architecture. In the end, there was insufficient time left for implementing a comprehensive set of analysis & sizing methods essential for materializing a thorough aircraft design loop. Therefore, achieving a fully functional aircraft design system prototype proved unattainable. Consequently, it was not possible to demonstrate that adopting the proposed architecture yields an aircraft design system that can be used in a reconfigurable way, despite indications that this could very well be the case.

This thesis distinguishes itself by examining aircraft design systems beyond the scope of solving a specific aircraft design problem. It systematically addresses source code issues prevalent in current aircraft design systems and introduces a software architecture designed to rectify these issues. Although time constraints prevented conclusive validation of the proposed architecture, the proposed architectural elements maintain their relevance. These elements can not only be applied during the development of future aircraft design systems but can also be used selectively to enhance current aircraft design systems.

Contents

List of Abbreviations

ADS	Aircraft Design System
AE	Aerospace Engineering
AEDsysDP	Aircraft Engine Design System Analysis Software
AGILE	Aircraft 3rd Generation MDO for Innovative coLlaboration of heterogeneous teams of Experts
AI	Artificial Intelligence
AIAA	American Institute of Aeronautics and Astronautics
API	Application Programming Interface
APU	Auxiliary Power Unit
ASCII	American Standard Code for Information Interchange
AVL	Athena Vortex Lattice
BLISS	Bi-Level Integrated System Synthesis
BPR	Bypass Ratio
CAD	Computer-Aided Design
CD	Continuous Deployment
CFD	Computational Fluid Dynamics
CI	Continuous Integration
CO	Collaborative Optimization
CommonKADS	Common Knowledge Acquisition and Documentation Structuring
CPACS	Common Parametric Aircraft Configuration Schema
CS	Certification Specifications
CSV	Comma-Separated Values
DATCOM	Data Compendium
DBMS	Database Management System
DCTerms	DCMI Metadata Terms
DEE	Design and Engineering Engine
DEM	Delivery Empty Mass
DOC	Direct Operating Cost
DOE	Design of Experiments
DSM	Design Structure Matrix
EASA	European Union Aviation Safety Agency
FAA	Federal Aviation Administration
FAR	Federal Aviation Regulations
FL	Fidelity Level
FPP	Flight Performance and Propulsion
GS	Gauss–Seidel
GUI	Graphical User Interface
HTML	HyperText Markup Language

IDF	Individual-Disciplinary Feasible
IGES	Initial Graphics Exchange Specification
InFoRMA	Integration, Formalization and Recommendation of MDO Architectures
IRI	Internationalized Resource Identifier
JSON	JavaScript Object Notation
KADMOS	Knowledge- and graph-based Agile Design for Multidisciplinary Optimization Systems
KBE	Knowledge-Based Engineering
KNOMAD	Knowledge Nurture for Optimal Multidisciplinary Analysis and Design
KOMPRESSA	Knowledge-Oriented Methodology for the Planning and Rapid Engineering of Small-Scale Applications
KPI	Key Performance Indicator
MDAO	Multi-Disciplinary Analysis and Optimization
MDF	Multi-Disciplinary Feasible
MFZM	Maximum Zero-Fuel Mass
MMG	Multi-Model Generator
MOKA	Methodology and software tools Oriented to Knowledge-Based Engineering Applications
MTOM	Maximum Take-Off Mass
MTOW	Maximum Take-Off Weight
OEM	Original Equipment Manufacturer
OEW	Operating Empty Weight
OOP	Object-Oriented Programming
OOPS!	OntOlogy Pitfall Scanner!
ORM	Object–Relational Mapping
OWL	Web Ontology Language
PANTHER	Propulsion Airframe iNTegration for Hybrid Electric Research
PIDO	Process Integration and Design Optimization
PLM	Product Lifecycle Management
RDF	Resource Description Framework
RDFS	RDF Schema
SAND	Simultaneous ANalysis ad Design
SE	Software Engineering
SHACL	Shapes Constraint Language
SMART	Specific, Measurable, Achievable, Relevant, Time-Bound
SPARQL	SPARQL Protocol and RDF Query Language
SQL	Structured Query Language
STEP	Standard for The Exchange of Product model data
TLAR	Top-Level Aircraft Requirement
TRL	Technology Readiness Level
TTL	Terse RDF Triple Language
TUD	Delft University of Technology
UML	Unified Modelling Language

URI	Uniform Resource Identifier
VLM	Vortex Lattice Method
XDSM	Extended Design Structure Matrix
XLSX	Excel Office Open XML Spreadsheet Extensions
XML	Extensible Markup Language
XSD	XML Schema Definition

List of Symbols

Λ	Sweep	deg
θ	Relative atmospheric temperature	-
a_{sl}	Speed of sound at sea-level	m/s
b	Wing span	m
c	Chord length	m
c_r	Root chord length	m
C_T	Thrust-specific fuel consumption	kg/s/N
c_t	Tip chord length	m
g	Gravitational acceleration	m/s^2
H	Lower calorific value	J/kg
h_f	Mean fuselage height	m
l_c	Cabin length	m
M	Mach number	-
m_{APU}	APU mass	kg
m_{asg}	Airframe structure group mass	kg
m_{eqg}	Equipment group mass	kg
m_e	Dry engine mass	kg
m_f	Fuselage mass	kg
m_{ht}	Horizontal tail mass	kg
m_{lg}	Landing gear mass	kg
m_{ng}	Nacelle group mass	kg
m_{oig}	Operational items group mass	kg
m_{pg}	Propulsion group mass	kg
m_{scg}	Surface control group mass	kg
m_{vt}	Vertical tail mass	kg
m_w	Wing mass	kg
n_{crew}	Number of crew	-
n_{pax}	Number of passengers	-
n_{lim}	Limit load factor	-
n_{ult}	Ultimate load factor	-
R	Range	km
S	Gross wing area	m^2
T	Thrust	N
t	Thickness	m
V_{DD}	Drag divergence speed	m/s
v_c	Cabin volume	m^3
w_f	Mean fuselage width	m

1
Introduction

Aircraft design has come a long way since the Wright brothers developed their powered *Flyer* in 1903. The early design efforts were characterized by trial and error, subsequent ones by theory development, and later ones by methodological advancements. The more recent design processes are shaped by information technology.

Comprehensive software systems have been developed to generate and analyze aircraft designs both rapidly and consistently. These systems are used in the early design phases and contain design knowledge from multiple engineering disciplines. They are employed both in industrial and in academic settings. Anemaat refers to them as "Airplane Design Systems" [1] and Torenbeek calls them "Automated Design Synthesis Systems" [2]. In this report, they are termed Aircraft Design Systems or simply ADSs. Since the 1980s, more than 35 ADSs have been devised. Non-exhaustive overviews can be found in references [1] to [3] and in appendix A.

The ADS which has been developed at the section of Flight Performance and Propulsion (FPP) at the faculty of Aerospace Engineering (AE) of the Delft University of Technology (TUD) is called the *Aircraft Design Initiator* or simply (and more frequently) the *Initiator* [4]. This ADS has been especially valuable for synthesizing and assessing innovative aircraft configurations (incl. blended-wing-body aircraft [5] and box-wing aircraft [6]) as well as novel aircraft technologies (incl. distributed propulsion concepts [7] and kerosene alternatives [8]). The results of studies carried out with *Initiator* have been documented in over 40 journal articles, conference proceedings, and theses. An overview of the *Initiator* and these studies is provided in appendix B.

The *Initiator* is normally used to generate an aircraft design (i.e. the aircraft geometry and several aircraft key performance parameters (KPIs)) based on a fixed set of aircraft requirements (i.e. the aircraft configuration and additional top-level aircraft requirements). At the core of the *Initiator* there are a number of analysis & sizing modules. They are executed iteratively until one aircraft parameter (i.e. the maximum take-off weight (MTOW)) is converged sufficiently. This process is termed aircraft synthesis and is illustrated in figure 1.1.

However, the *Initiator* is rarely used to perform other aircraft design studies. For example, it is difficult to use the *Initiator* to:

- Perform studies where a number of design variables are automatically varied in order to find an optimized aircraft design (e.g. an aircraft design with minimal fuel consumption)
- Come up with an aircraft design that is based on a different set of aircraft requirements than the standard ones (e.g. where a current/existing engine is to be used instead of a novel/virtual engine)

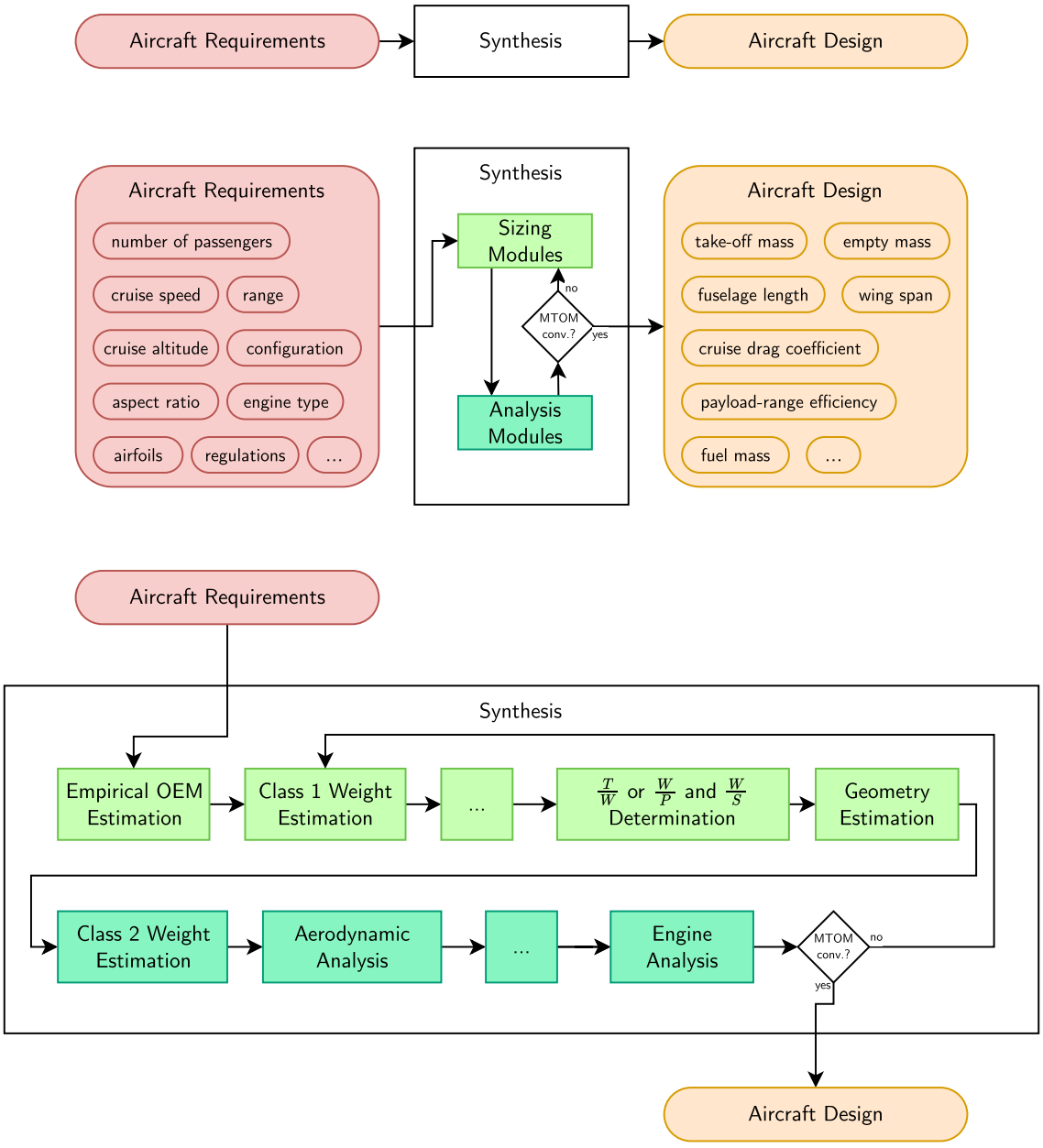


Figure 1.1: The *Initiator* synthesis process at different abstraction levels

- Establish a higher-fidelity synthesis process where modified or additional analysis & sizing modules are employed (e.g. a wing sizing method that is based on a load distribution instead of a total force)
- Determine the importance of aircraft parameters (e.g. by checking how sensitive the empty mass is to changes in certain requirements)

These examples are illustrated schematically in figure 1.2. It is essential to emphasize that conducting these studies constitutes a fundamental step in a comprehensive aircraft design project. They serve the purpose of answering typical follow-up questions (e.g. "what if" and "why is that" questions) that emerge once an initial aircraft design has been generated using a system like the *Initiator*. It is somewhat surprising that executing these studies is inherently difficult, even though the design knowledge needed to address characteristic follow-up questions is principally available within the *Initiator*. It appears that the design knowledge is not readily available in a practical or accessible form.

The reason for this appears to be related to the software architecture of the *Initiator*: The source code of the *Initiator* is not only extensive but also complex and integrated. It consists of more than 500 000 lines of code that have been written by more than 25 experts in aircraft design (with varying expertise in software design). A number of issues, which are detailed later in this report, make it difficult to comprehend and to modify the *Initiator*'s source code. Please note that the *Initiator* is not the only ADS with problematic source code. Instead, it appears that problematic source code is a fundamental issue of ADSs in general: Smith points out that ADSs are often based on "unstructured, monolithic, poorly commented code" [9]. Smith adds that engineers and researchers are primarily interested in solving a concrete design problem and do not have enough time for making their tools and methods easily accessible to others [9]. Kroo notes that "the complexity of such codes [...] often rises to the point that no one person knows what the program is doing; bugs go unnoticed for years and the results [...] become incredible" [10].

In fact, current ADS were not developed to be reconfigurable. Within the scope of this report the word reconfigureable describes the possibility **to repeatedly and reversibly configure[1] a design system** in different ways **to address a wide range of design problems**. A reconfigurable ADS **cannot only be configured to generate aircraft designs**, but **can also be reconfigured to answer characteristic follow-up questions on the generated aircraft designs**. In a reconfigurable ADS this can be achieved **without (irreversibly) changing existing source code but by (conditionally) utilizing and extending existing source code**.

Hence, current ADSs cannot be considered to be reconfigurable. This is problematic because it means that current ADSs cannot be used to investigate follow-up design problems (e.g. as shown in figure 1.2) that always emerge after the solution to an initial design problem has been obtained (especially when performing research or when investigating unconventional designs). Thus, in order to deal with these follow-up design problems, new ADSs must either be developed from scratch or assembled by hacking together components from various existing ADSs. These are time-consuming, repetitive, and error-prone undertakings. Additionally, the design knowledge already contained in the current ADSs might not be transferred to the new ADSs in its entirety and therefore might get lost over time.

[1]The word configure stems from the Latin words "con" which can be translated as "with, together, thoroughly" and "figurare" which can be translated as "to shape, form, make a likeness of" (see `https://www.merriam-webster.com/dictionary/configure`). Hence, the word configure means "to arrange something in a particular way, to make software work in the way that the user prefers" (see also `https://www.oxfordlearnersdictionaries.com/definition/english/configure`).

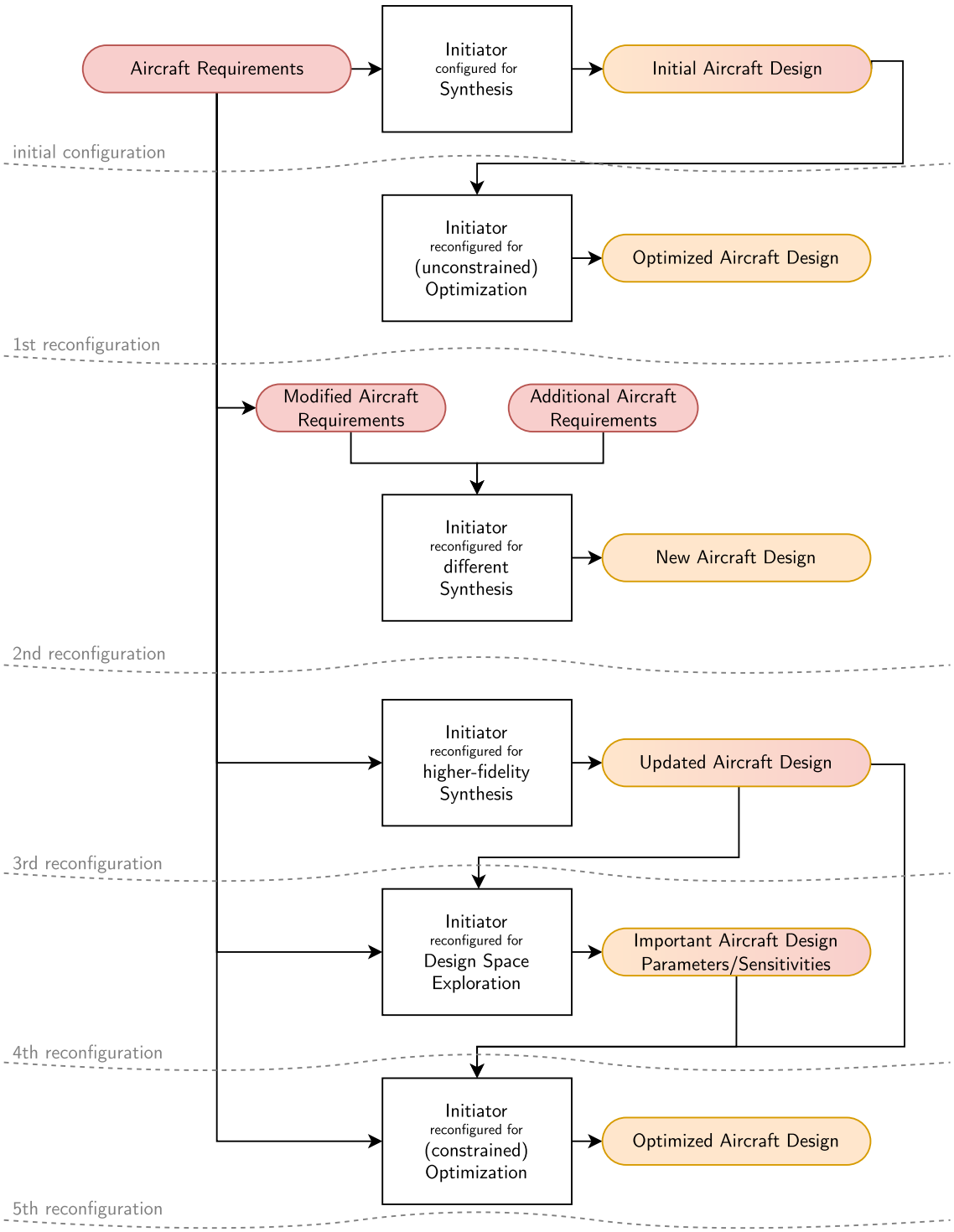


Figure 1.2: Desirable reconfigurations of the *Initiator* synthesis process

ADSs are not the only type of design systems for which reconfiguration capabilities would be desirable. Van Gent even states "the need for continuous reconfiguration lies at the core of any design practice" [11]. He recently introduced a methodology to lower the (re)configuration hurdles encountered in collaborative multi-disciplinary analysis and optimization (MDAO) projects by dividing the system design process into distinct stages (as shown in figure 1.3) and thereby separating concerns. His approach is based on:

- Utilizing a central data schema (describing the design data)
- Defining a repository of tools (which are all linked to the data schema)
- Employing a graph-based software system (termed *KADMOS*) to enable the formulation of MDAO systems (which are based on the tool repository and generated according to the problem to be solved)
- Executing these MDAO systems in (specialized) process integration and design optimization (PIDO) tools

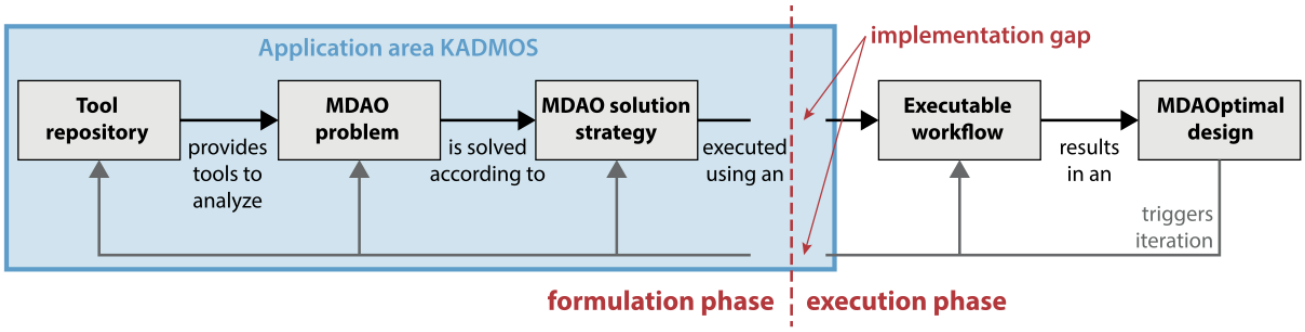


Figure 1.3: MDAO system design process proposed by Van Gent et al. [12]

Van Gent and others further realized that ADSs are akin to MDAO systems. Therefore, it should be possible to apply the methodology outlined above to make ADSs more reconfigurable. Van Gent and Bruggeman demonstrated the feasibility of this idea by investigating several different design problems (incl. design space explorations and optimizations) using a reconfigurable MDAO system based on a tool repository consisting of wrappers around a subset of the analysis & sizing modules of the *Initiator* [11, 13]. Their findings show that the analysis & sizing modules of the *Initiator* can be valuable for addressing a wide range of design problems once they are decoupled from each other and available as standalone tools. However, the reconfigurable MDAO system developed by Van Gent and Bruggeman also had some drawbacks: Most importantly, the MDAO system was unable to perform synthesis (i.e. coming up with an aircraft design based on a set of aircraft requirements) anymore. Instead, it required a (fully parameterized, but not fully converged) aircraft design as starting point. Furthermore, the wrappers around the analysis & sizing modules were rather inflexible (e.g. the MDAO system could only deal with turbo-fan aircraft but not with turbo-prop aircraft) and not entirely accurate (e.g. they sometimes demanded inputs that were not relevant for the calculations performed by the analysis & sizing modules). Moreover, the wrapped analysis & sizing modules were still based on highly complex and integrated source code. Thus, changes to the analysis & sizing modules itself still remained intricate (e.g. replacing an entire aircraft geometry estimation method with another one was easily possible, but replacing the wing geometry estimation method contained within the aircraft geometry estimation method was hardly possible). Finally, assembling and executing the MDAO system was far from computationally efficient.

The author of this report is convinced that current ADSs need to be fundamentally restructured before they can be considered reconfigurable ADSs. The MDAO system design process proposed by Van Gent (see figure 1.3) seems to be a suitable starting point for this restructuring effort.

The objective of the research project described by this report is to **explore the feasibility of developing a reconfigurable aircraft design system** by **investigating the issues that prevent current aircraft design systems from being reconfigurable** and by **proposing, implementing and testing a software architecture that addresses these issues**. In order to meet the objective some gaps in the existing body of knowledge need to be filled. To that end, the following three research questions have been established:

1. What are the **main issues** that prevent current aircraft design systems from being reconfigurable?
2. What **high-level software structure** and **low-level software artifacts** are suited to address these issues?
3. What are the **implications** of making a novel aircraft design system adhere to such a software architecture?

2 Methodology

The research questions have been addressed by the analysis of existing ADSs, followed by the development and evaluation of a novel ADS. This approach is motivated and detailed below.

2.1 Iterative software development

The flowchart presented in figure 2.1 provides an overview of the pursued methodology. The flowchart illustrates the iterative nature of the development cycle which comprises issue identification, requirement derivation, prototype development, and prototype evaluation. The development cycle concludes either when a functional and reconfigurable prototype has been obtained (see section 2.4) or when a time limit has been exceeded (i.e. when the time planned for the research project has been exceeded).

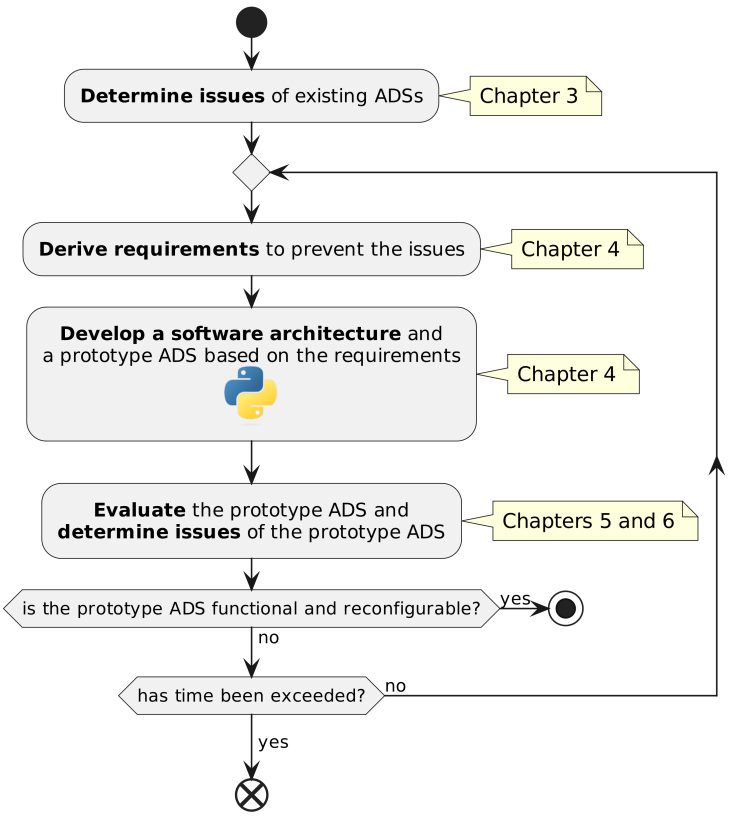


Figure 2.1: Graphical representation of the pursued methodology

Figure 2.1 might suggest that several discrete prototypes have been developed. In reality, a single prototype has been developed which underwent continuous modifications throughout multiple iterations of the development cycle. Some concepts persisted, while less promising ideas were used only during a single iteration. Current ADSs served as a baseline for the prototype development. The initial prototype was based on state-of-the-art MDAO techniques and closely resembled the demonstrator that has been developed previously by Van Gent and Bruggeman [11, 13]. The subsequent prototypes primarily revolved around the way the design data was structured or the way the analysis & sizing methods were defined.

Python was chosen as primary programming language. The main reasons for this choice were that Python is a general-purpose language, that it comes with a simple syntax (almost like pseudo-code), that it is frequently used in academic environments, and that it offers a large number of standard libraries. Python also facilitates the integration of tools written in other programming languages through wrappers. In fact, the prototype incorporated tools written in Matlab, Fortran, and C++. Furthermore, the principles of test-driven development were followed, which means that tests were created before functionality. The benefits of test-driven development are explained in chapter 6. Moreover, the object-oriented development paradigm was employed to conveniently separate concerns and to enable the final prototype to be used as both a project and a package/library (in other projects). It is worth noting that the software architecture elements proposed at a later point in this report do not necessarily have to be implemented in Python but principally could be implemented in other programming languages as well.

2.2 Utilization of existent software and established theories

The objective of the research project described in this document is the development of a reconfigurable ADS. However, it was deemed impractical to develop such an ADS from scratch. Therefore, the source code of existing ADSs was used as starting point for the development of the reconfigurable ADS. Since the prototype that has been developed during the research project is primarily based on the *Initiator*[1] (version 2.10), it is referred to as the *ReInitiator* in the following. Furthermore, several other aircraft design tools, including *SUAVE*[2] (version 2.5), *VAMPzero*[3] (version 0.8) and the *DAS*[4] (unversioned), served as sources of inspiration and design knowledge.

The direct utilization of source code from existing ADSs was not feasible (as explained in chapter 5). Furthermore, the source code of these ADSs was not sufficiently self-explanatory to be used as single source of knowledge. Consequently, the source code was checked with and compared against analysis & sizing methods outlined in literature. Specifically, analysis & sizing methods described by Torenbeek [2, 14], Raymer [15], Obert[16], Scholz[5], and ESDU[6] were taken into consideration. On occasion, the suboptimal quality and comprehensibility of the source code from existing ADSs necessitated the development of new code from the ground up, based on the analysis & sizing methods documented in literature.

[1] `https://gitlab.tudelft.nl/initiator/AircraftDesignInitiator`
[2] `https://github.com/suavecode/SUAVE`
[3] `https://github.com/DLR-boeh-da/VAMPzero`
[4] `https://svn.lr.tudelft.nl/git/DAS.git`
[5] `http://lecturenotes.aircraftdesign.org`
[6] `https://www.esdu.com/`

2.3 Selection of analysis & sizing methods

Current ADSs, and in particular the *Initiator*, contain a substantial amount of design knowledge distributed across hundreds of files and thousands of lines of source code. Thus, it was necessary to impose limitations on the amount of design knowledge utilized during the development of the *ReInitiator*. Therefore, six distinct analysis & sizing methods were selected. These analysis & sizing methods were chosen because they represent a typical aircraft synthesis loop and because they cover crucial aerodynamic, weight, and performance aspects. They were also chosen because they feature realistic interdisciplinary coupling. They are derived from the modules that make up the innermost convergence loop of the *Initiator*.

The design knowledge contained in the selected analysis & sizing methods can be summarized as follows:

- Seeder:
  Performs preliminary calculations to obtain initial values for some aircraft parameters such as the operating empty mass, zero-lift drag coefficient and maximum lift coefficient. Isolates relations for obtaining default values that were previously scattered across the *Initiator*.

- $\frac{T}{W}$ or $\frac{W}{P}$ and $\frac{W}{S}$ determination:
  Evaluates a number of standard performance requirements (e.g. take-off field length requirement, landing distance requirement, stall requirements, cruise requirements, etc.) and selects a design point (in terms of wing-loading and either thrust-loading or power-loading) satisfying these requirements. This approach is often referred to as "matching plot technique" [17].

- Geometry estimation:
  Estimates the overall aircraft geometry by sizing (and positioning) the wing, the fuselage, the empennage, and the engines (based on the wing-loading, thrust-loading/power-loading and other aircraft requirements). Includes a wrapper around the FuselageConfigurator module of the *Initiator*.

- Aerodynamic analysis:
  Estimates the aerodynamic performance of the aircraft (in terms of lift coefficient, drag coefficient, etc.) by performing a vortex-lattice analysis using the *AVL*[7] tool alongside the utilization of semi-empirical drag estimation techniques.

- Weight estimation:
  Performs a class II weight estimation and thus draws up a detailed mass breakdown for the aircraft based on relations from Torenbeek [2, 14]. In contrast to the *Initiator*, which uses an enhanced Torenbeek method to handle unconventional aircraft configurations, the actual equations as provided in literature are utilized.

- Mission analysis:
  Calculates the fuel fractions and the take-off mass for every specified mission, then derives parameters describing the harmonic mission, and finally determines the maximum take-off mass, the maximum landing mass, and the maximum zero-fuel mass (using relations from Torenbeek [2], Raymer [15] and ESDU[6]). In the *Initiator* this method is usually referred to as the class I weight estimation.

[7] `https://web.mit.edu/drela/Public/web/avl`

2.4 Reconfigurability requirements

As illustrated in figure 2.1, the development cycle can conclude once a functional and reconfigurable prototype has been acquired. This requires an evaluation of the functionality and reconfigurability of the prototype, for which three key requirements have been established:

1. The *ReInitiator* must possess the ability to automatically synthesize
   a) passenger aircraft that are comparable to existing aircraft complying to FAR part 25/EASA CS-25 standards,
   b) requiring as input only a minimal yet extensible set of requirements,
   c) using multi-disciplinary analysis & sizing methods,
   d) while supporting different aircraft configurations (e.g. tube-and-wing, blended-wing-body, or box-wing configurations),
   e) and different propulsion technologies (e.g. turbo-prop or turbo-fan engines).
2. The *ReInitiator* should support adding, removing, replacing, and reordering both
   a) entire analysis & sizing methods (e.g. a geometry estimation or an aerodynamic analysis)
   b) as well as individual elements of such analysis & sizing methods (e.g. a wing sizing method which typically is part of a geometry estimation).
3. The *ReInitiator* must enable conducting
   a) design space exploration studies,
   b) local sensitivity studies and
   c) optimization studies

   for passenger aircraft as described in the first requirement.<

The first requirement ensures that the *ReInitiator* encompasses a feature set similar to that of the *Initiator*, enabling it to synthesize realistic passenger aircraft. It also mandates the ability of the *ReInitiator* to handle not only a fixed set of requirements but also a flexible set of requirements. For instance, users may or may not choose to impose bounds on design parameters such as maximum take-off weight, wingspan, or fuselage length.

The second requirement is about confirming the *ReInitiator*'s reconfiguration capabilities with regards to the analysis & sizing methods it contains. This involves potential changes such as substituting Torenbeek's [2, 14] weight estimation method with Raymer's [15]. This also involves selectively updating individual elements of these analysis & sizing methods, for instance, replacing a low-fidelity Torenbeek wing weight equation [14] with a higher-fidelity tool such as *EMWET* [18]. Alternatively, this involves omitting individual elements of these analysis & sizing methods, for example not estimating the wings size and weight at all, and instead utilizing an existing wing design with known dimensions and a known weight.

The third requirement facilitates the versatility of the *ReInitiator*, allowing it to be used for addressing a wide range of aircraft design problems. For example, one might wish to reconfigure the *ReInitiator* for a sensitivity study, in order to find out which design parameters influence the fuel consumption the most. Subsequently, one might want to

further adapt the *ReInitiator* for an optimization study, in which key design parameters are automatically varied, to identify an aircraft design with minimal fuel consumption.

3

Issues present in existing ADSs

Current ADSs are frequently used for generating aircraft designs, but rarely employed for investigating these aircraft designs in more detail. This is due to a number of issues which make it difficult to understand and to adapt the functioning of the ADSs. The most important of these issues are described and exemplified in the following subsections. It is worth mentioning that these issues are not necessarily a threat to the validity of prior research that has been conducted with current ADSs. To be explicit, the provided examples are not intended to cast aspersions on the valuable research that has been carried out with these ADSs.

The issues are explained using examples from the *Initiator*. Nonetheless, the issues are not unique to the *Initiator*. The issues could very well be explained using similar examples from other ADSs (particularly from ADSs that were developed in academic settings). However, for the sake of presenting the issues in a concrete and coherent way, all examples are taken exclusively from the *Initiator*.

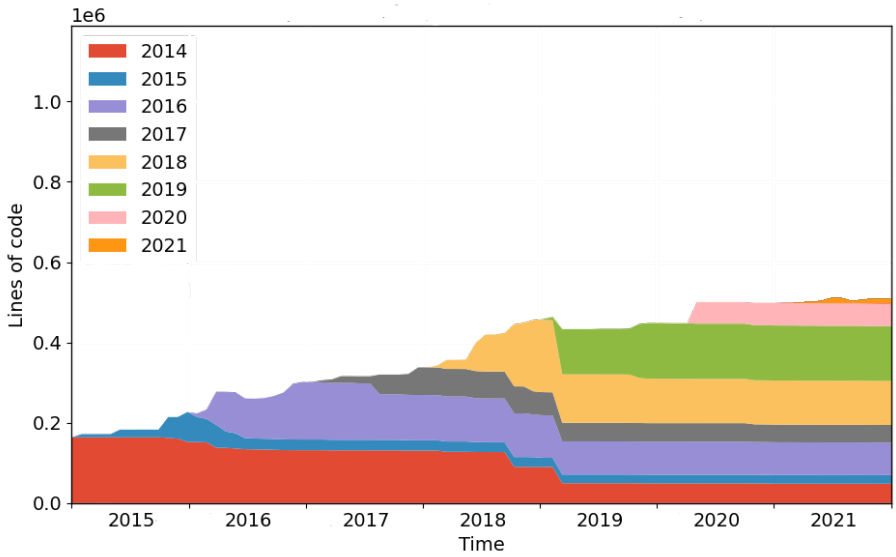


Figure 3.1: Evolution of the *Initiator* source code between the years 2014 and 2022

As indicated in the introduction of this report, the issues are linked to the source code of the ADSs. Figure 3.1 illustrates the evolution of the source code of the *Initiator*. The *Initiator* repository contained around 150 000 lines of code in the year 2014. The version of the *Initiator* repository referred to in this report contains more than 500 000 lines of code. However, the repository growth was not already well-controlled, and hence the quality

of the source code decreased over time. Notably, there are guidelines[1] that explain how the *Initiator* modules should be structured. Yet, these guidelines are hardly followed and thus the *Initiator* modules are structured in incoherent ways. The issues described in the following subsections are those that are currently present in the *Initiator*. Not all of these issues have been present since the inception of the *Initiator*. Instead, some of them have evolved gradually, sometimes as a result of unintentional misuse of the ADS.

Please note that the *Initiator* source code is written in the Matlab programming language and many structural elements inherit from the Matlab `handle` (super) class. The analysis & sizing modules of the *Initiator* (previously illustrated in figure 1.1) are implemented as subclasses of the `Module` class. Instantiated (singleton) `Module` objects are invoked trough the (singleton) `Controller` object. This `Controller` object also contains design data and keeps track of the results computed by the `Module` objects. This structure is schematically depicted in figure 3.2.

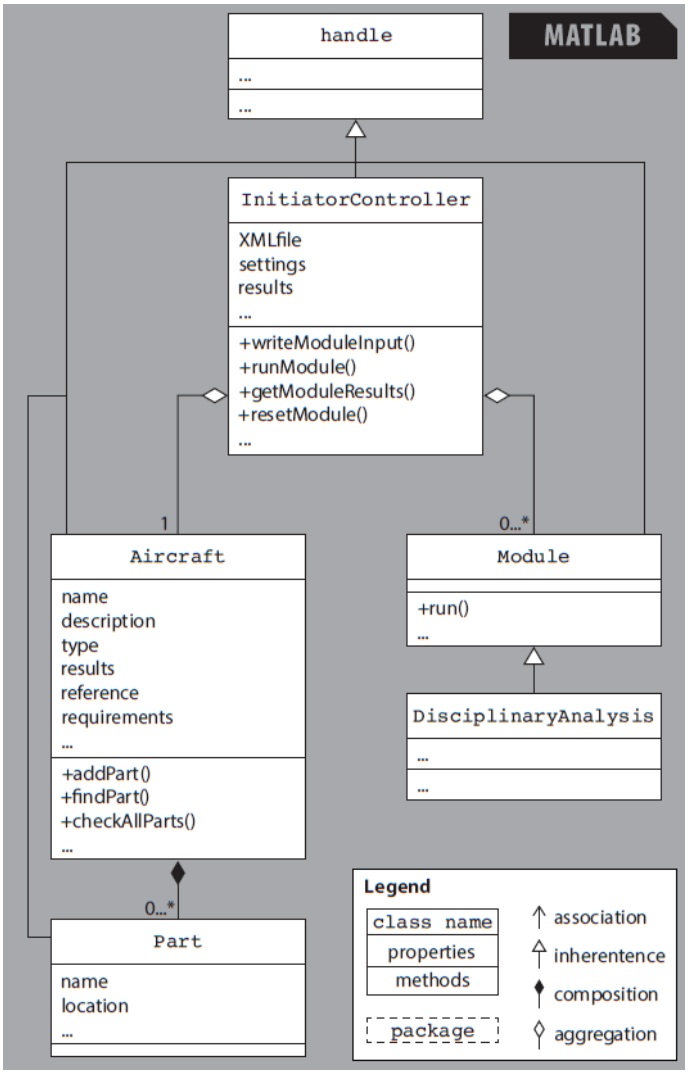


Figure 3.2: *Initiator* UML diagram [11]

[1] In the form of a wiki, see `http://fppwiki.lr.tudelft.nl/index.php/Synthesis/Initiator`.

3.1 Concealed and convoluted source code

Frequently, ADSs are seen and treated as black-boxes, with users expecting them to generate specific outputs based on provided inputs in a fully automated fashion. However, regarding ADSs as black-boxes is problematic due to the lack of emphasis on the underlying source code and the lack of incentives for writing and maintaining clear source code. Thus, the source code becomes increasingly convoluted and obscure over time, as exemplified below.

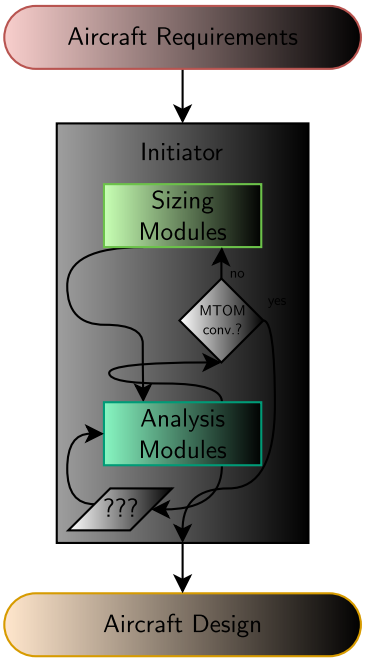


Figure 3.3: Illustration of the concealed and convoluted nature of an ADS as a whole (with intentionally reduced legibility)

Firstly, the overall design process may not be clearly defined. For example, the *Initiator* contains the `Controller` object, which, despite its name, does not control the design process but rather handles coordination tasks. The design process is not prescribed centrally in the `Controller` class (in a single class) but defined decentrally in the `Module` classes (in hundreds of classes) and determined dynamically (only) while the system is executed (cf. figure 3.2). Modules may request the execution of other modules on demand. Hence, diagrams visualizing the design process (such as the one in figure 1.1) cannot be obtained before running the ADS (by inspecting the source code) but only after executing it (by tracing the design process and observing which modules are executed). Adjustments of the design process cannot be made in straightforward ways (as every analysis & sizing module can potentially influence and reroute the design process).

Secondly, the user may not have full visibility into the overall data flow. Specifically, the input and output parameters of the ADS may not have been explicitly specified. For instance, the *Initiator* is typically invoked by providing a file containing requirements and initial values. Unnecessary parameters in the file are accepted but ignored. Missing parameters are often assigned default values (logic similar to that in listing 3.1 can be found throughout the source code). Additionally, the ADS may silently overwrite parameters. Hence, without delving into the source code, it is challenging to discern which parameters are genuinely required for the design process and which ones are optional/unnecessary. Similarly, it is difficult to ensure that parameters are not unintentionally modified. Finally, without canvassing the source code, it is demanding to differentiate whether an output is a merely defaulted or a specifically calculated parameter.

When using an ADSs for design of experiment or for optimization studies, it is crucial to have a clear understanding of and control over the data flow (e.g. it should be possible to prevent the ADS from overwriting fixed values). Furthermore, analysis & sizing methods modules are likely to be revised and rearranged before they can be utilized for optimization purposes (e.g. to ensure that coupling parameters are sufficiently converged). However, the concealed and intricate nature of the source code may discourage users from attempting these reconfigurations. It is worth noting that the issues discussed in subsequent sections can compound to the opacity and complexity of the source code.

Listing 3.1: Example code for determining a default value

```
if mach_drag_divergence is None:
    mach_drag_divergence = mach_cruise + 0.015
```

3.2 Ambiguous and cluttered data structures

ADSs are often created by combining existing analysis & sizing tools. Hence, it is not surprising that the data structures found in ADSs often mirror those present in existing analysis & sizing tools. Consequently, there usually exists a tight implicit coupling between the data structures found in ADSs and the employed analysis & sizing tools. Moreover, the data structures found in ADSs are neither homogeneous nor necessarily contiguous.

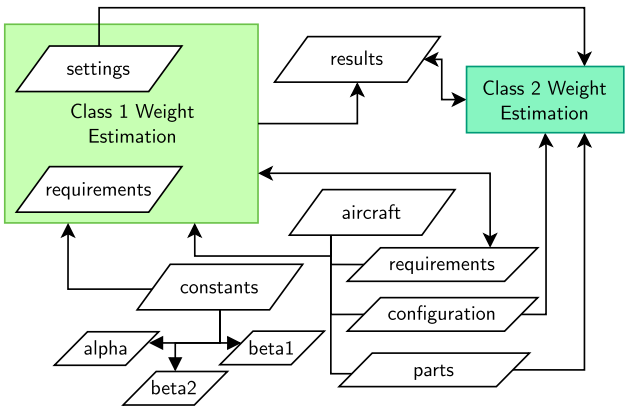


Figure 3.4: Illustration of the ambiguous data structures that are defined in multiple places

Specifically, the data structures within an ADS may exhibit fragmentation and disorder. Often, data is contained in objects such as arrays and hashes, which exhibit varying degrees of pre-established structure. Sometimes, data is stored in files, which may adhere to formats of varying readability. Incidentally, data is available in databases, which may not adhere to pre-defined schemas.

Design parameters are not always named verbosely either. While the meaning of a design parameter named `beta` may be evident when looking at an individual analysis & sizing module, it can become ambiguous when considering multiple analysis & sizing modules. The term `beta` may be used to denote different parameters within an ADS, such as the Prandtl-Glauert factor and the sideslip angle.

Similarly, for the initial developer setting up an ADSs it may be obvious that the design parameter `cruise_speed` refers to the true airspeed (and e.g. not to the equivalent airspeed) and is always specified in meters per second (and e.g. not in miles per hour). However, such details are hardly documented, and thus the next developer may interpret the `cruise_speed` differently. In the absence of adequate documentation, it becomes necessary to derive the meaning of design parameters by guessing or by inspecting the context in which the design parameters are used, which can easily lead to inconsistencies.

Since the data structures of current ADSs may have been defined in an ad-hoc manner, design parameters can be hard to locate. In case of the *Initiator* all data is principally available to all modules through the `Controller` object. Still, even similar design parameters are defined in different places, such as the `settings`, the `results`, or the `aircraft` attribute (cf. figure 3.2). Furthermore, design parameters are occasionally specified multiple times in a redundant fashion. Additionally, some design parameters are always specified, even when they are not relevant to the investigated aircraft design.

When adding a new analysis & sizing module to an ADS, it is imperative to consider the design data already available within the system. Failure to do so may prevent the seamless integration of the new module into the ADS. When replacing one analysis & sizing module with another, it is crucial to ensure the compatibility of the design data generated by both methods. The cluttered and ambiguous data structures of current ADS, as described above, prevent that such reconfigurations can be done quickly (without wasting time) and safely (without introducing inconsistencies).

3.3 Extensive and integrated source code

During the creation of analysis & sizing modules for ADSs, developers commonly face challenges related to missing input parameter values. Nevertheless, it is often feasible to establish default parameter values or estimate them using empirical relations. Hence, code fragments similar to the one presented in listing 3.1 are frequently integrated into the analysis & sizing modules of current ADSs. While this might seem beneficial initially, the excessive proliferation of such code fragments can ultimately result in bloated analysis & sizing modules that perform functions beyond their designated scope.

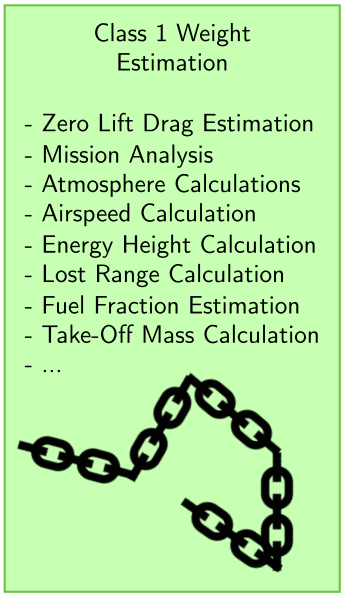


Figure 3.5: Illustration of the long chains of contiguous code used for defining modules

For instance, the weight estimation module of the *Initiator*, aside from estimating maximum take-off weight, also calculates zero-lift drag, performs a mission analysis, conducts atmospheric calculations, etc. (as illustrated in figure 3.5). Additionally, analysis & sizing modules might be overloaded with supplementary tasks, including dependency resolution, unit conversion, and plotting tasks. Frequently, these supplementary tasks are neither clearly delineated from the core tasks nor uniformly defined. In case of the *Initiator*, an analysis & sizing module is executed by invoking its `run` method (see figure 3.2). It is not uncommon that the `run` method comprises (en bloc) a few hundred lines of source code. Hence, it is demanding to understand the purpose and the operating principles of analysis & sizing modules, which is an essential prerequisite for making reconfigurations.

Noteworthy reconfigurations involve the reuse, deactivation or substitution of distinct functionalities of individual analysis & sizing modules. For example, an attempt to enhance a geometry estimation module might entail replacing a low-fidelity wing sizing process (based on empirical relations) with a higher-fidelity one (based on aerodynamic loads). However, such reconfigurations are challenging due to extensive and tightly integrated nature of the source code used to define analysis & sizing modules.

Furthermore, it is not feasible to independently reuse distinct functionalities of analysis & sizing modules. The modules can only be used as a whole or not at all. This does not only limit reconfiguration options but also limits the extent to which modules can be unit-tested (cf. figure 3.6). Moreover, the inability to conveniently reuse distinct functionalities of modules necessitates code duplication when similar tasks need to be executed in multiple modules or in disparate sections of a single module. This is problematic as subsequent updates to the initial code may not be propagated to the duplicated code (or vice versa). Furthermore, modules with overlapping functionalities can be difficult to converge.

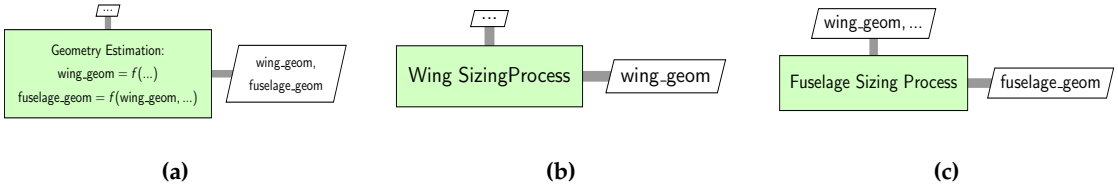


Figure 3.6: Example integrated sizing module **(a)** and alternative modular sizing modules **(b/c)**

3.4 Complex source code

When a new ADS is created, it usually can only handle a limited range of aircraft designs (e.g. conventional turbo-fan and turbo-prop aircraft). Over time, the ADS is expanded, and may be able to handle a broader range of aircraft designs (e.g. blended-wing-body or box-wing aircraft). Additionally, the analysis and sizing modules used within the ADS are often refined, in order to accurately capture subtle differences between aircraft designs. These expansions and refinements are typically achieved by incorporating nested if-else statements and nested loops into the source code, as exemplified by the pseudo-code in listings 3.2 to 3.4. While the lines of code increase almost linearly (cf. figure 3.1), the cyclomatic complexity[2] of the code grows exponentially.

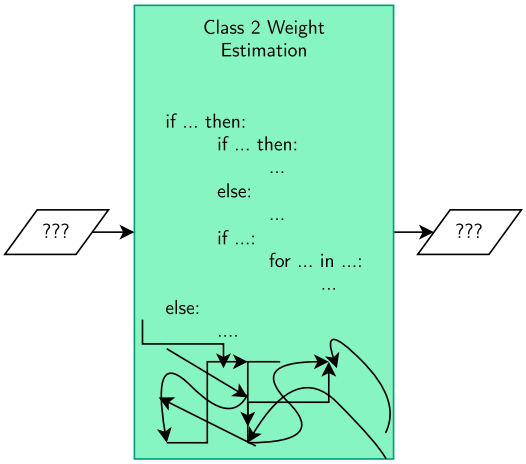


Figure 3.7: Illustration of abundant conditionals and loops used to define modules

Nowadays the modules of the *Initiator* can handle a multitude of diverse aircraft types and configurations. However, the cyclomatic complexity of the module's `run` method is often close to the number 50. As a result, some modules are so cluttered that they even contain illogical and unreachable source code.

It is troublesome to understand how a module works, which inputs are required, and which outputs are generated. The complexity of the source code becomes especially problematic when attempting to reconfigure an ADS for optimization purposes. Moreover, the complexity makes it difficult to obtain a good (i.e. continuous and differentiable) numerical representation of a module which is considered essential for solving optimization problems, especially those problems with a large number of variables [20].

Listing 3.2: Initial state

```
if engine == "TurboFan":
    func_a()
elif engine == "TurboProp":
    func_b()
```

Listing 3.3: Intermediate state

```
if engine == "TurboFan":
    if configuration == "Conventional":
        func_a()
    elif configuration == "BWB":
        func_c()
    elif configuration = "BoxWing":
        func_d()
elif engine == "TurboProp":
    func_b()
```

Listing 3.4: Current state

```
if engine== "TurboFan":
    if configuration == "Conventional":
        if engine_loc == "Wing":
            func_a()
        elif engine_loc == "Fuselage":
            func_e()
        elif:
            func_f()
    elif configuration == "BWB":
        func_c()
    elif configuration == "BoxWing":
        func_d()
elif engine == "TurboProp":
     if engine_loc == "Wing":
        func_b()
elif engine == "Hybrid":
    func_h()
```

[2]The cyclomatic complexity is a quantitative measure representing the number of paths through a code [19]. It is generally recommended to keep the cyclomatic complexity of a method below the number 11. When the cyclomatic complexity of a code is high it is often considered untestable.

3.5 Excessive coupling within source code

A key aspect of aircraft design involves the intricate network of dependencies among various aircraft subsystems. A design change in one subsystem is likely to trigger design changes in other seemingly unrelated subsystems. Thus, it is essential to also model these dependencies in an ADS.

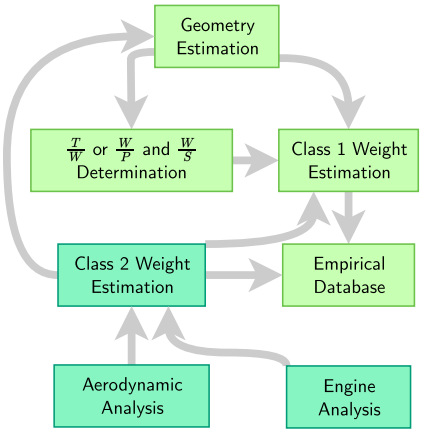


Figure 3.8: Illustration of interdependencies between modules

The straightforward approach requires hardwiring these dependencies directly into the source code. This is exemplified by the pseudo-code in listing 3.5, which shows that running the geometry estimation function invokes both a weight estimation function and a wing-thrust-loading function. This approach to handling dependencies is quick and easy to implement, which is why it can be found in various ADSs.

However, this approach carries the risk of inadvertently introducing recursive loops, which can render an ADS inoperable. To mitigate this risk, ADSs often implement some form of dependency management logic. For instance, the *Initiator* employs XML files to define dependencies between modules (as illustrated in listing 3.6) and utilizes the `Controller` object to manage the dependencies. Yet, dependency resolution processes are generally intricate. People working on ADSs, especially when facing time constraints, occasionally still revert to hardcoding dependencies, thereby bypassing the dependency resolution process. As a result, current ADSs typically utilize more than one approach to couple analysis & sizing modules to each other.

In either case, direct dependencies between analysis & sizing modules can be problematic when attempting to reconfigure an aircraft design system. For example, replacing one module with another (e.g. substituting a VLM-based aerodynamic analysis module with a CFD-based alternative) demands the replacement of all references to the initial module. Alternatively, compatibility between inputs and outputs of the initial and the replacement module must be ensured. Furthermore, altering the execution order of modules or preventing the execution of specific modules (e.g. not estimating the wing geometry and weight, but instead utilizing an existing wing geometry with known weight) is a daunting task. Finally, it is important to note that direct dependencies between analysis & sizing modules may not always accurately represent the physical dependencies between different aircraft subsystems.

Listing 3.5: Example of hardcoded design logic containing dependencies between analysis & sizing functions

```
def geometry_estimation():
    results_a = class_one_weight_estimation()
    results_b = wing_thrust_loading()
    surface = results_b.take_off_weight / results_a.wing_loading
    thrust = results_b.take_off_weight * results_b.thust_over_weight
    ...
```

Listing 3.6: Example of dependencies between analysis & sizing modules specified in XML format

```
<module>
    <name>GeometryEstimation</name>
    <dependency>Class1WeightEstimation</dependency>
    <dependency>WingThrustLoading</dependency>
</module>
```

4

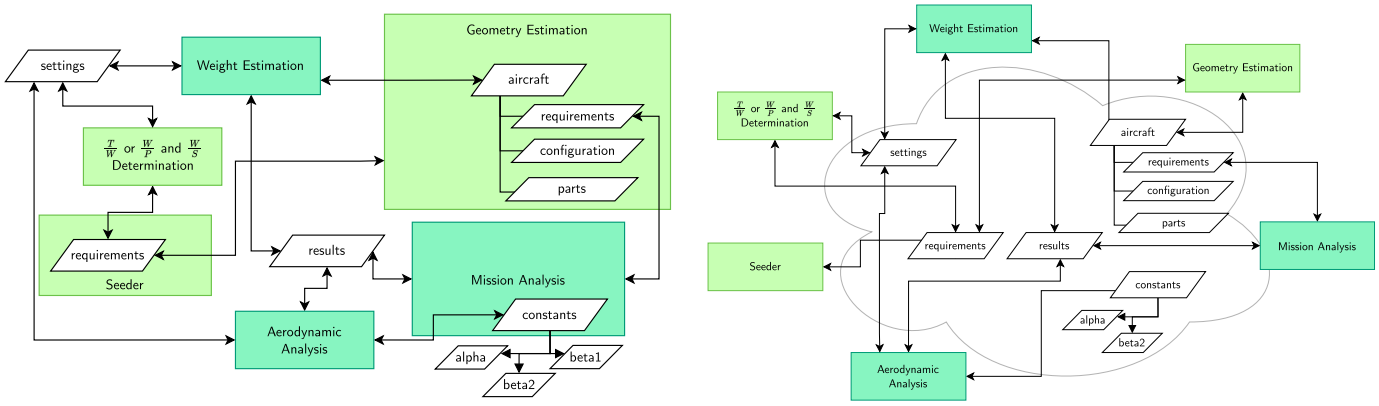
A software architecture to address the issues present in existing ADSs

The issues described in the previous section are all related to the highly integrated and highly complex software architecture of current ADSs. This implies that a less integrated and less complex software architecture is required for future ADSs. Several architectural elements for future ADSs that specifically address the previously described issues of current ADSs are presented in the following sections. The *ReInitiator* serves as a prototype for the implementation of these elements. This prototype is elaborated upon in appendix C.

4.1 Centralized data store and self-contained analysis & sizing methods

An ADS typically contains two primary components: design data and analysis & sizing methods. As pointed out before, in current ADSs, these two elements are intricately interconnected, creating a complex framework that can be difficult to understand and reconfigure. To disentangle these elements, several architectural measures are proposed.

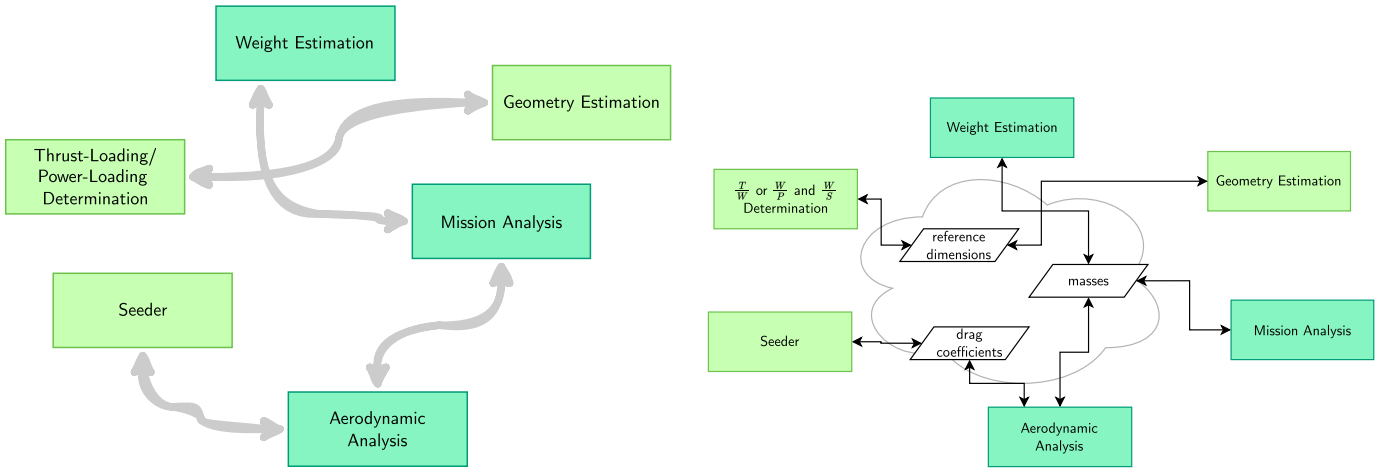
First, it is logical to separate design data from analysis & sizing methods. This separation can be achieved by establishing a centralized data store. In current ADSs, data is often distributed across various locations, sometimes combined with analysis & sizing methods or placed in dedicated data containers (see figure 4.1a). In a reconfigurable ADS, data should be accessible to all analysis & sizing methods from a central storage location (which acts as a single source of truth, see figure 4.1b).



(a) Scattered **(b)** Centralized

Figure 4.1: Distribution of design data

Second, the direct and often hard-coded interconnections between analysis & sizing methods need to be addressed. It is suggested that analysis & sizing methods should communicate exclusively through the central data store. This would involve replacing the direct links between analysis & sizing methods in current ADSs (see figure 4.2a) with indirect data connections via the central data store (see figure 4.2b). While this approach requires adding more data to the central data store, it results in self-contained analysis & sizing methods that no longer rely on direct connections to other analysis & sizing methods but instead depend solely on the presence of data within the central data store.



(a) Direct **(b)** Indirect

Figure 4.2: Connections between analysis & sizing methods

To improve data organization, it is recommended to use a structured database as the central data store. During the development of the *ReInitiator*, different database types were considered, including hierarchical file-based databases (XML, JSON), relational databases (SQLite), and graph databases (RDF). Ultimately, graph databases were chosen as the most suitable option because the design data relevant to aircraft design is often not hierarchical[1] and not homogeneous[2] but still strongly interlinked.

Finally, the implementation of standardized database endpoints is proposed. These endpoints are positioned between the database and the analysis & sizing methods (see figure 4.3). Their primary purpose is to provide a comprehensible and uniform interface for accessing the database. They are instances of a class (as illustrated in figure 4.4) and designed for temporary use. The endpoints fulfill their primary purpose by exposing a limited set of methods for query generation and dispatch, facilitating data retrieval from and writing to the database. In addition, they can seamlessly handle routine tasks, such as unit conversions (e.g. from meters to feet) and the automatic addition of metadata (e.g. specifying the analysis & sizing method responsible for a database entry, along with the modification timestamp). Moreover, endpoints can log which data is retrieved from or written to the database, an aspect that is further elaborated upon in section 4.4.

By incorporating these elements into a novel ADS, such as the *ReInitiator*, one can already achieve a software architecture that effectively separates different design concerns.

[1] An aircraft may fly at different cruise altitudes during different missions. Enforcing hierarchies, such as aircraft → mission → altitude or mission → aircraft → altitude, appears to be awkward and arbitrary.

[2] Aircraft design data cannot always be described in terms of a numerical value and an associated unit. Sometimes strings, arrays, matrices, etc. constitute more appropriate representations of aircraft design data.

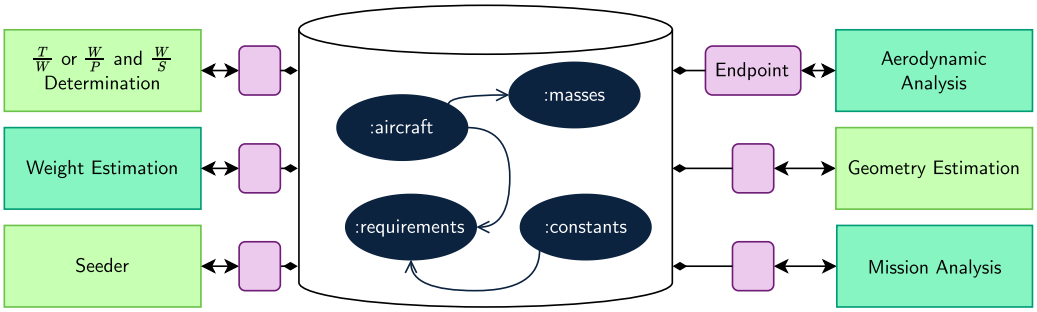


Figure 4.3: Centralized graph database with endpoints and self-contained analysis & sizing methods

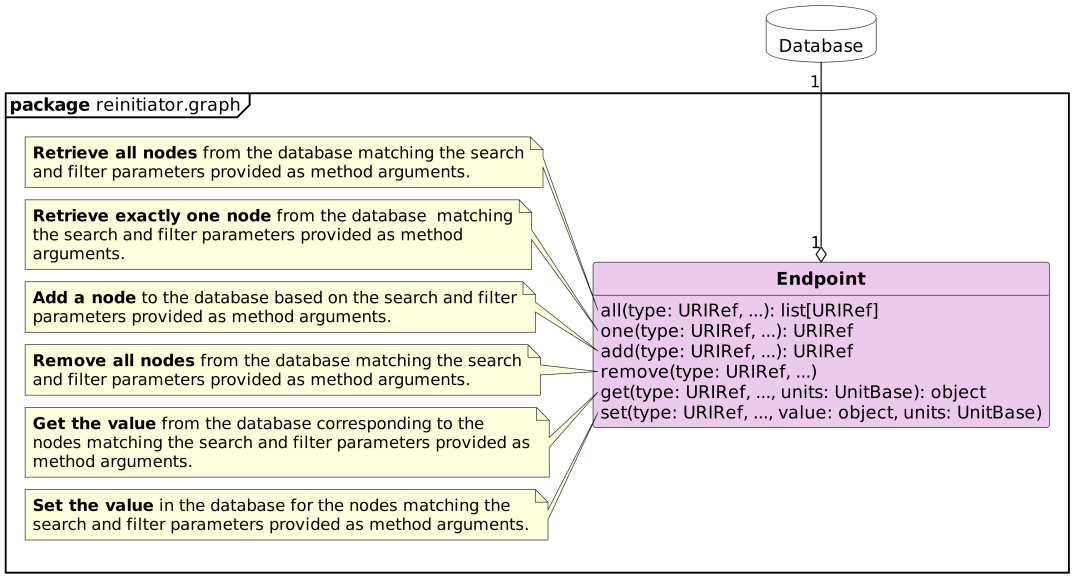


Figure 4.4: Class diagram illustrating the endpoint concept which provides a simple and uniform interface to the database

4.2 Semantic data management

While the proposed architectural elements discussed above address data clutter within contemporary ADSs, they do not address the issue of data ambiguity within these systems. In light of this, it is recommended to incorporate some architectural elements from the Resource Description Framework (RDF) when developing a reconfigurable ADS.

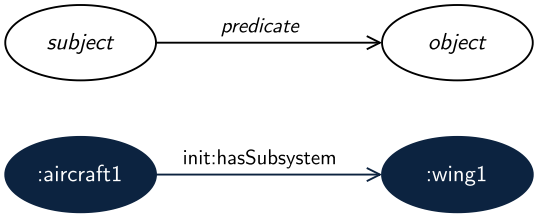


Figure 4.5: Visualization of an RDF triple as a directed arc diagram

In the RDF a graph database is often referred to as a "triple store". Triples are subject-predicate-object constructs (see figure 4.5) that can be used to express statements about

resources and their relationships with each other. The concept of triples is simple as well as powerful, and offers versatile means of representing various entities, including aircraft and aircraft subsystems (see figure 4.6).

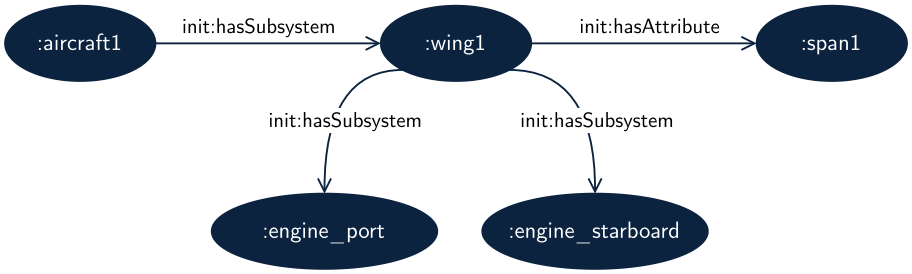


Figure 4.6: Directed arc diagram representing an example data graph

Triples can also be used to describe the meaning of data. The segment of the overall graph that defines concepts, relationships, and constraints is termed an ontology. Essentially, an ontology is a formal representation of knowledge that provides a standardized vocabulary for describing data. The adoption of an ontology enhances data comprehensibility and fosters interoperability across different systems by capturing the semantics of the data.

While numerous ontologies have been defined in the past, particularly in the medical field, there yet is no well-established ontology dedicated to aircraft design (except for an elementary prototype, cf. [21]). Although developing such an ontology for an ADS entails substantial effort, it can yield substantial benefits by enabling precise data interpretation. Best practices dictate the utilization of metadata such as alternative names, LaTeX expressions, and links (such as references to other ontologies or images) when constructing an ontology.

Figure 4.7 presents a part of an example data graph alongside an example ontology graph. The data graph is highly dynamic, taking on a different shape for each aircraft design generated with the ADS and evolving as the design process advances. Conversely, the ontology graph remains relatively static, expanding only when additional features or functionalities are integrated into the ADS. Both graphs can be contained within a single RDF database, forming an RDF dataset, which in turn is accessible through an endpoint, as depicted in figure 4.8.

The advantages of employing semantic web technologies such as the RDF have been extensively discussed in literature (cf. [22]). The example RDF dataset shown in figure 4.7 provides a few clues why these technologies are particularly useful for performing aircraft design studies: Firstly, data can be conveniently classified, here based on the system ("init:System") and attribute ("init:Attribute") base classes. Secondly, it is possible to build comprehensive system hierarchies (here both ":engine_port" and ":engine_starboard" are subsystems of ":wing1" which in turn is a subsystem of ":aircraft1"). Thirdly, attributes that are shared between/equal for different systems, here the thrust-specific fuel consumption (":tsfc1"), do not have to be defined multiple times for each system. Finally, metadata like comments ("rdfs:comment") and timestamps ("init:wasUpdatedAt") can be incorporated as needed.

There are additional semantic web technologies that can be utilized to enhance the precision and clarity of an ADS: For instance, figure 4.8 also shows a validation graph containing rules formulated in accordance with the Shapes Constraint Language (SHACL), which can be used to validate the structure and content of the data graph. Furthermore, the Web Ontology Language (intentionally abbreviated OWL and not WOL) can be employed for

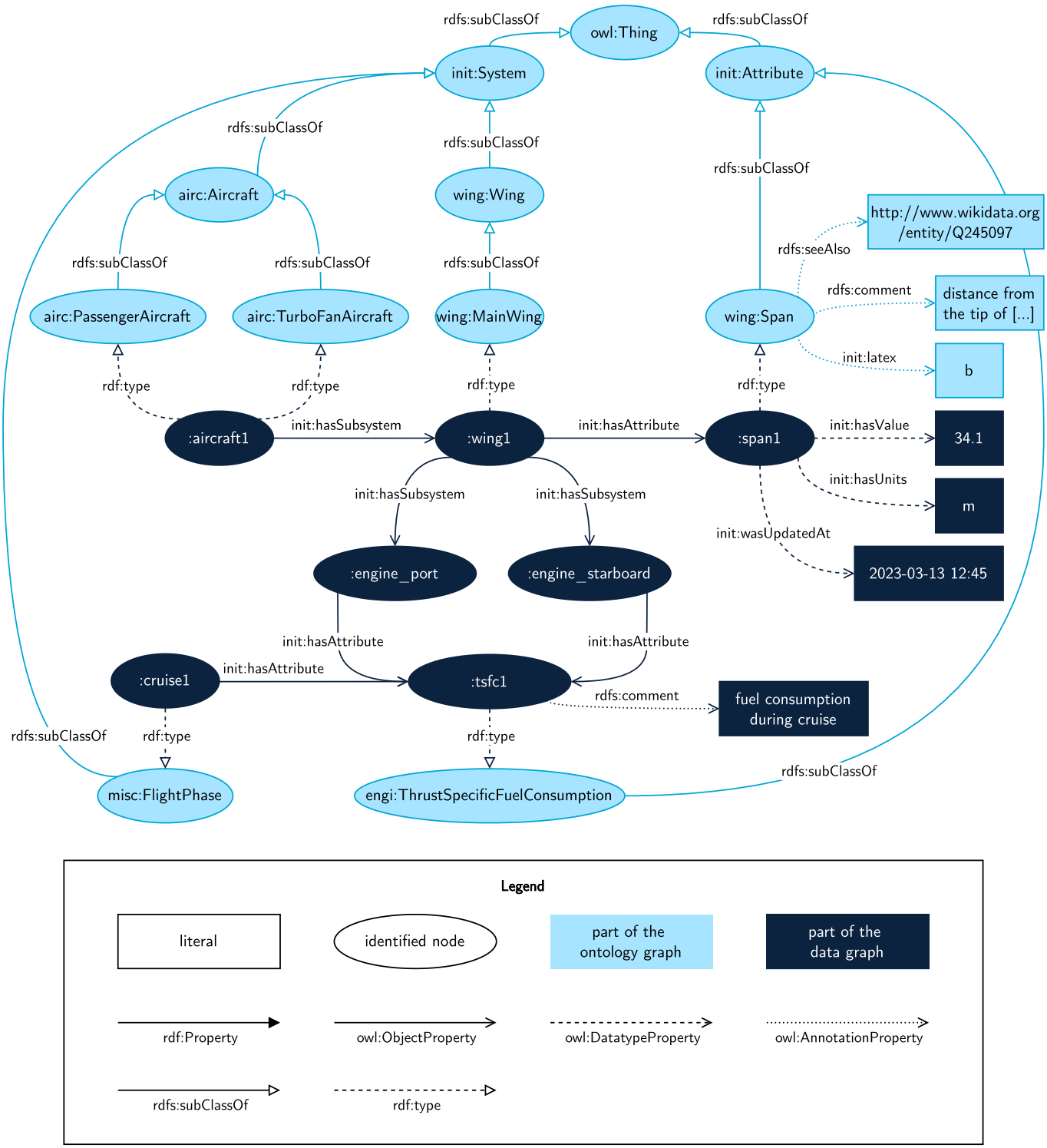


Figure 4.7: Directed arc diagram representing an example RDF dataset and the relationship between an ontology graph and a data graph

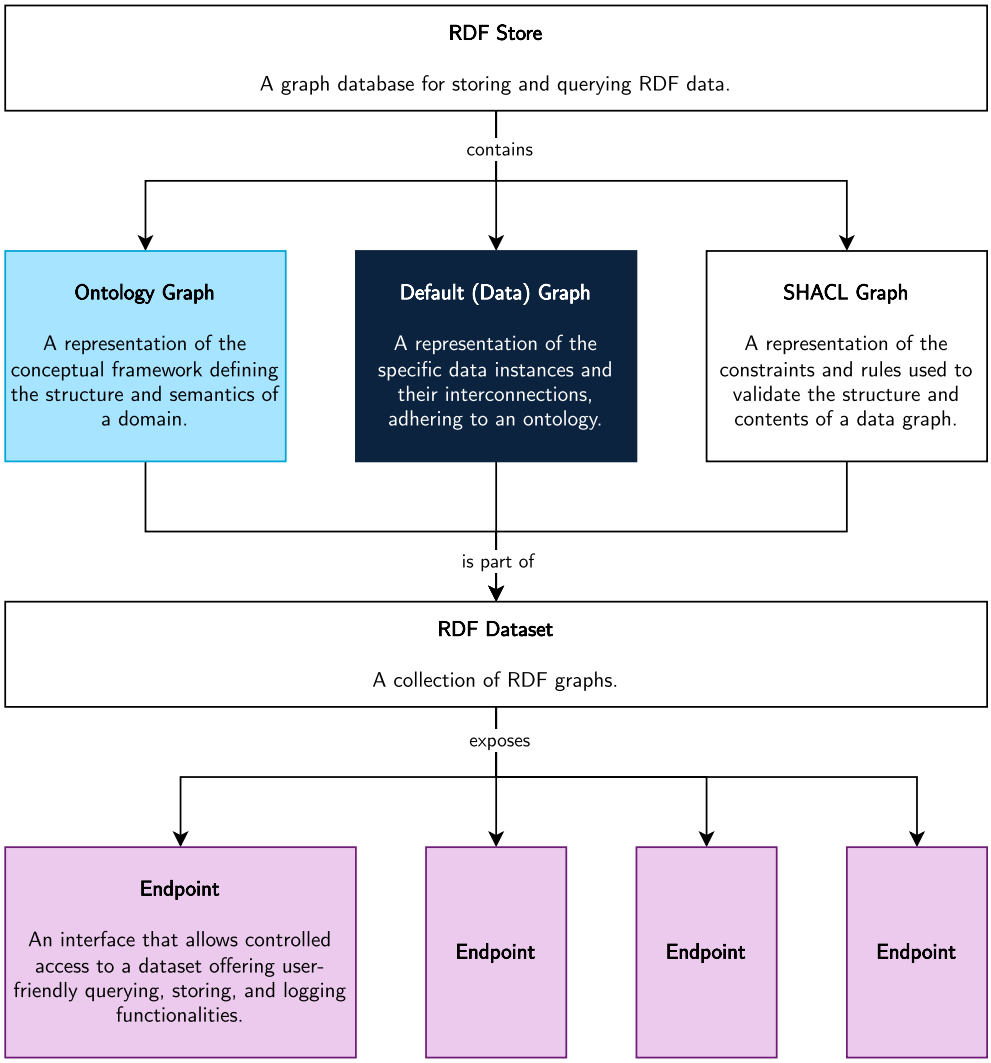


Figure 4.8: Visualization of various elements related to the RDF that prove to be useful for the development of a reconfigurable ADS

the definition of the ontology. The OWL provides formal semantics for the creation of comprehensive ontologies. Some dialects of the OWL even allow for automatic reasoning[3] to deduce additional facts about the graph (though these automatic reasoning capabilities are not a necessity for reconfigurable ADSs). Finally, the endpoints depicted in figures 4.3 and 4.8 extensively leverage the SPARQL Protocol and RDF Query Language (SPARQL) for accessing the RDF dataset. The adoption of these elements and standards ensures unified knowledge representation, fostering data consistency and enhancing the efficiency and accuracy of the ADS.

[3]Automatic reasoning, also referred to as inferencing, is a promising concept. For instance, one can use the OWL to specify that an aircraft equipped with only turbo-fan engines, but without any other engines, is to be considered a turbo-fan aircraft. Subsequently, a reasoning engine such as *HermiT* [23] or *Pellet* [24] can be used for the automatic classification of aircraft in a dataset (or for the automatic detection of aircraft that have been inconsistently defined in a dataset). However, it is important to note that automatic reasoning is computationally expensive, and formulating valid axioms is not a trivial task. Therefore, automatic reasoning was only used experimentally during this research phase. An ADS does not necessarily require automatic reasoning since the knowledge that could be automatically inferred can also be manually specified.

4.3 Standardized and modular analysis & sizing method interface

The architectural elements introduced in the previous section focused on the design data. However, the analysis & sizing methods are at least as important as the design data. Section 3.3 has shown that the extensive and integrated nature of the source code used to define analysis & sizing methods of current ADSs is problematic. To address this issue, the concept of procedures is introduced in the following.

In the first place, a procedure serves as an object for both encapsulating and explaining the computation logic contained within analysis & sizing methods. A procedure shall not be considered a plain function with a simple docstring. Instead, a procedure shall

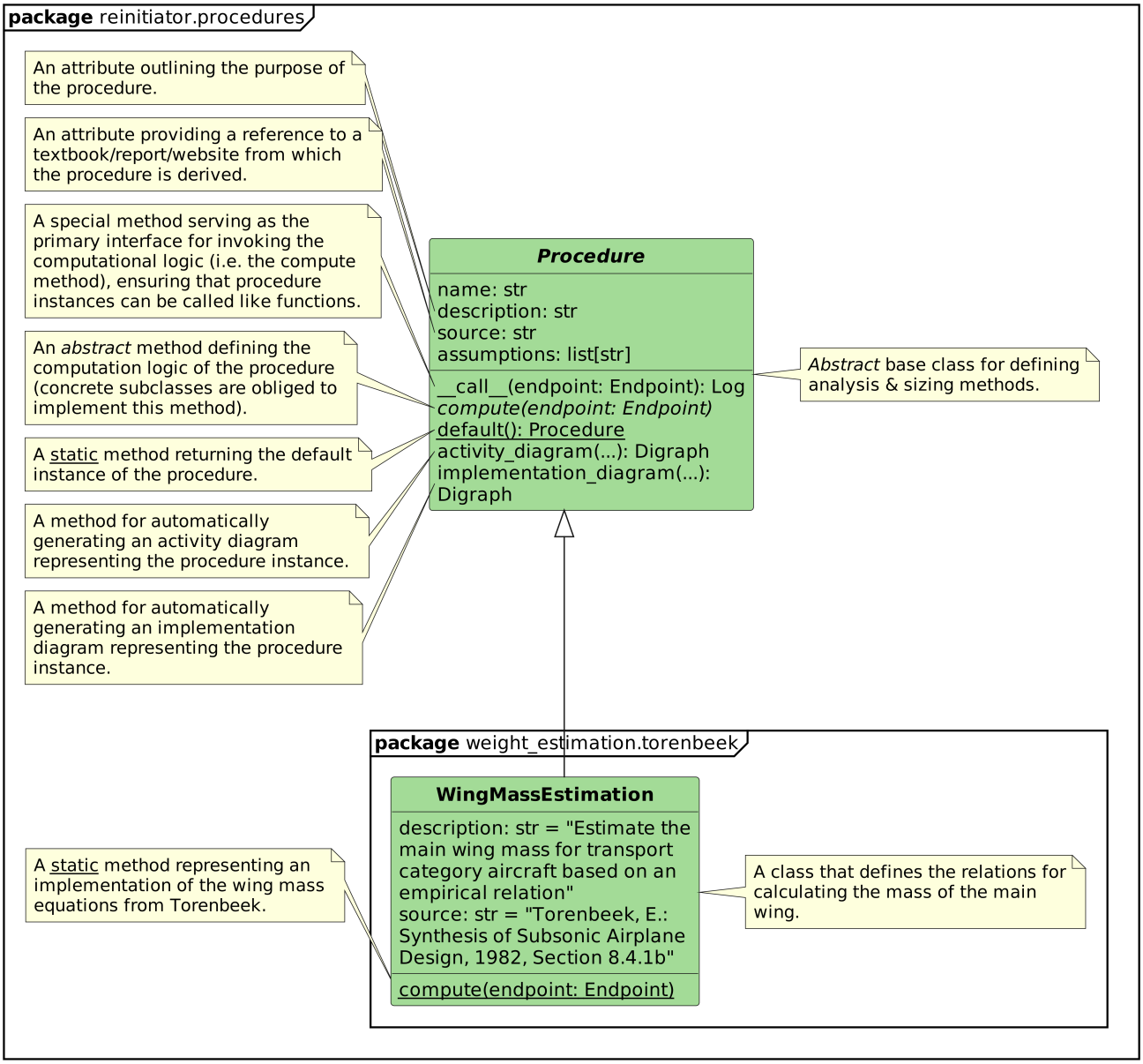


Figure 4.9: Class diagram illustrating the proposed procedure concept, showing the abstract base class (in particular its callable nature and descriptive features) alongside a concrete example class

be considered an enhanced function that comes with structured metadata and with self-documentation and self-visualization capabilities. Procedures can be defined as classes and typically only a single instance of a procedure class needs to be created (although it sometimes makes sense to create multiple instance of a procedure class, e.g. in case that the procedure is to be modified at runtime). These aspects are illustrated in figure 4.9.

Furthermore, a procedure shall be considered a callable object. This implies that once an instance of a procedure class has been created, it behaves like a function[4]. However, it does not act like a function with predefined inputs and outputs. Instead, it behaves like a function that takes an endpoint as its sole input argument and then interacts with the database (i.e. read from/write to the database) while executing the computation logic. These aspects are also illustrated in figure 4.9. Additionally, the code examples in appendix C.3 serve to clarify how procedures are intended to be created and used.

To manage the extent and complexity of analysis & sizing methods, it is proposed to establish a number of abstract procedure classes. These abstract classes, detailed in figure 4.10 and exemplified in figure 4.11 , provide an interface for organizing analysis & sizing methods in a hierarchical manner. The key idea is that only at the micro-level there are procedure that function as black-box components (referred to as "SimpleProcedures"). In contrast, at the macro-level, there are procedures that act as transparent components and that allow for introspection (the so-called "ComplexProcedures"). This facilitates the creation of nested procedures with arbitrary depth, thereby offering flexibility and scalability in structuring analysis & sizing methods[5].

Typically, at the top level of an ADS, optimization, DOE, or convergence procedures are employed. Mid-levels predominantly consist of sequential and conditional procedures, while the lowest level encompasses simple procedures. The lower-level procedures can be frequently reused (within other procedures), while higher-level procedures are less reusable and require modification to suit the specific design study or problem at hand.

Once the analysis & sizing methods have been decomposed into standardized procedures that come with methods to automate introspection, they can be effectively visualized using straightforward activity diagrams, as shown in figure 4.12. These visual representations highlight the advantages of modular procedures with high granularity, which make the design process transparent and eliminate the black-box nature of current ADSs. While this approach leads to a large number of distinct procedures, it is possible to maintain a low cyclomatic complexity within the individual procedures. By distributing the inherent complexity of analysis & sizing methods across multiple procedure levels, the design process becomes more manageable. Finally, procedures can be restructured and recombined like LEGO bricks, thereby allowing for design process reconfigurations.

[4]The concept of callable objects is a widely utilized feature in many programming languages, offering flexibility in code organization and execution. It enables instances of classes to be used with function call syntax, such as creating an object with `some_object = SomeClass()` and subsequently invoking it as `result = some_object(arguments)`. In Python, this functionality requires the definition of the `__call__` (dunder) method. In C++, this concept is commonly referred to as "functors" and requires the definition of the `operator` method.

[5]The readers of this report who are familiar with XML schemas may recognize that "SimpleProcedures" and "ComplexProcedures" bear a resemblance to "simpleTypes" and "complexTypes", which are base XSD types frequently employed when defining XML schemas.

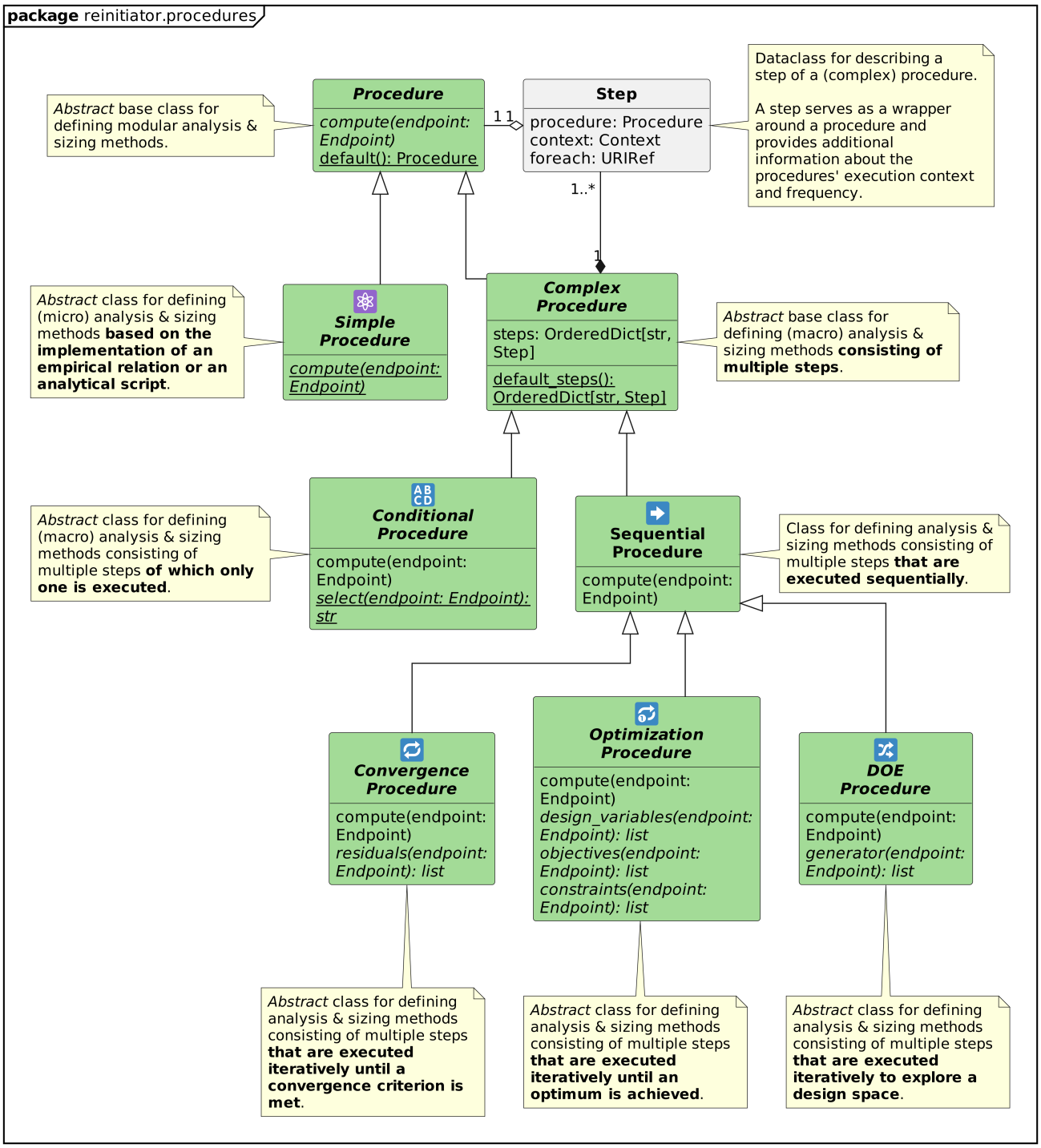


Figure 4.10: Class diagram illustrating the proposed procedure concept, depicting a hierarchy of classes that are relevant for defining typical analysis & sizing methods in a granular manner

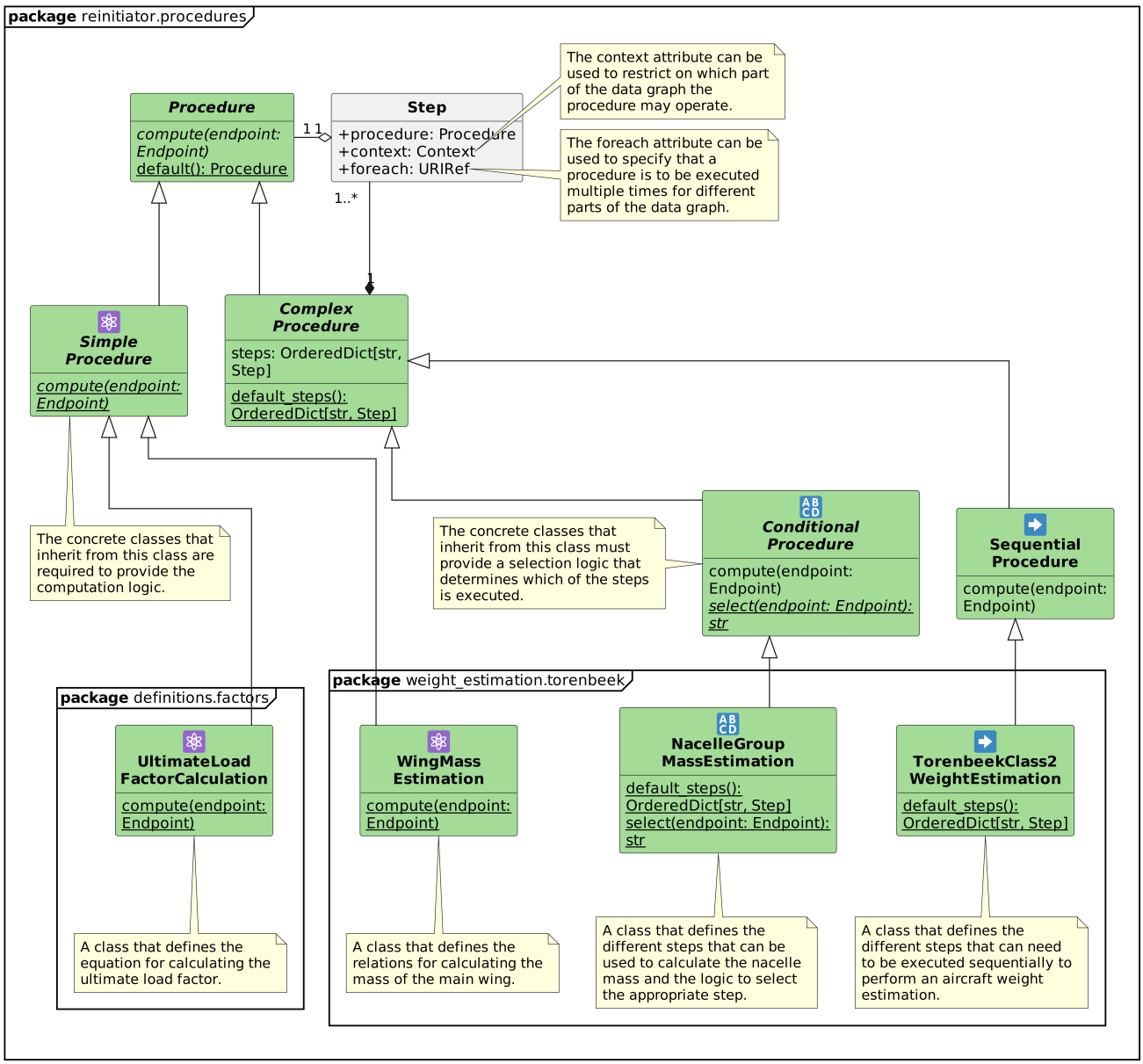
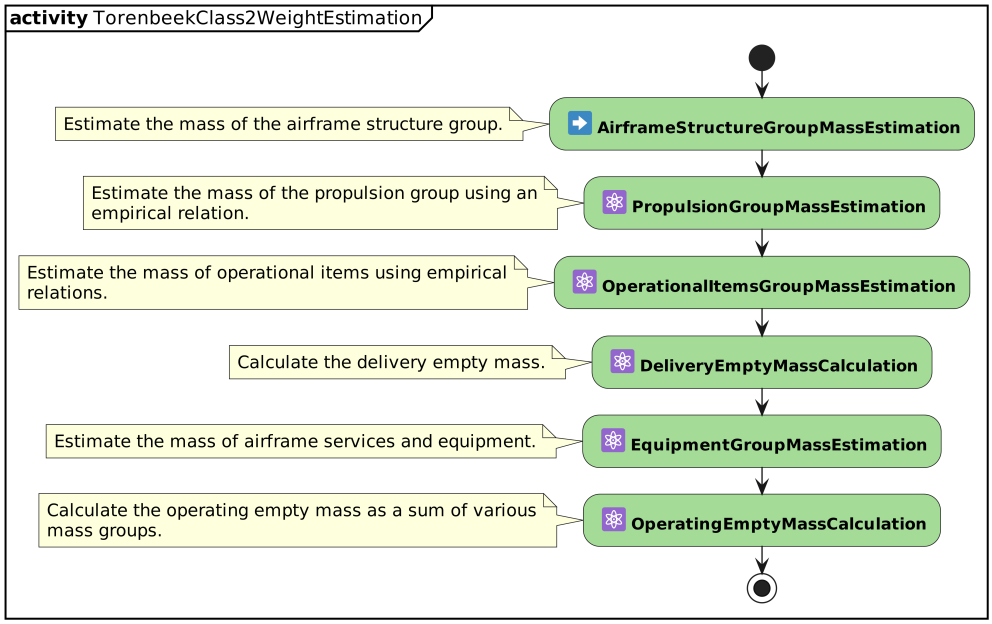
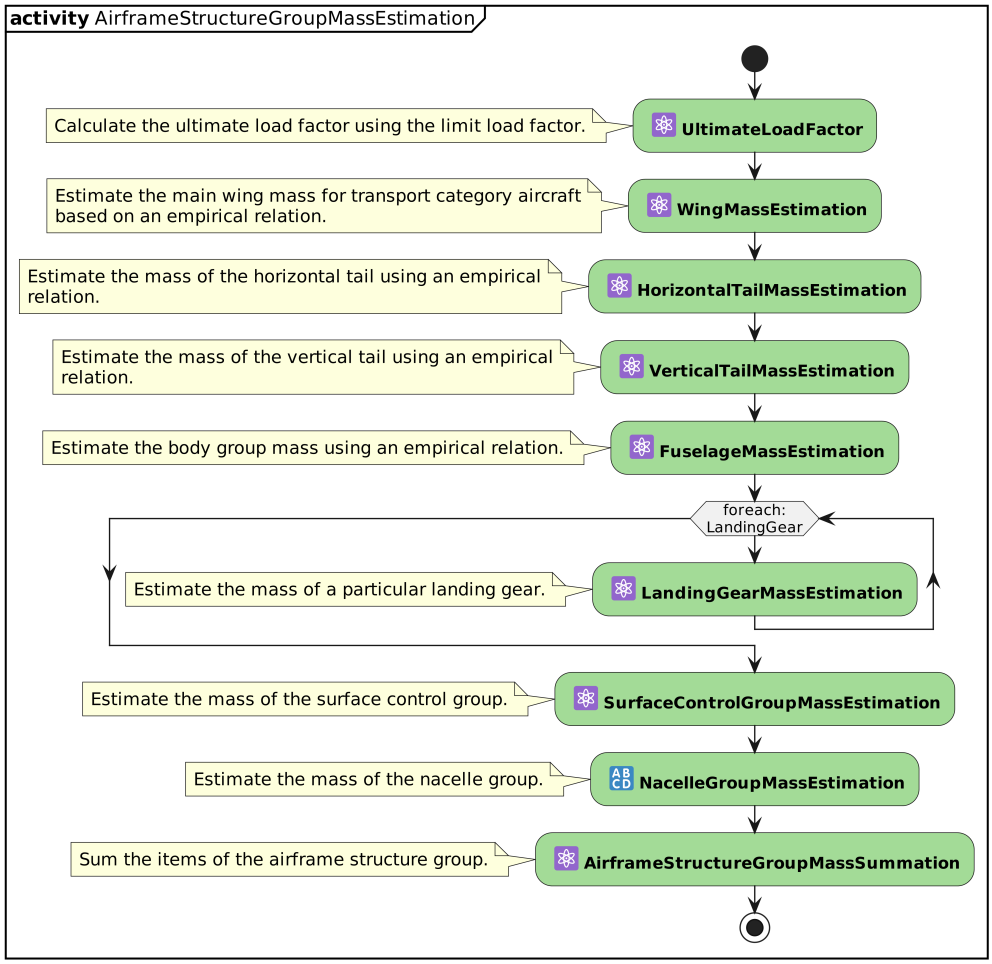


Figure 4.11: Class diagram illustrating the proposed procedure concept, providing details about essential attributes and methods along with further example classes

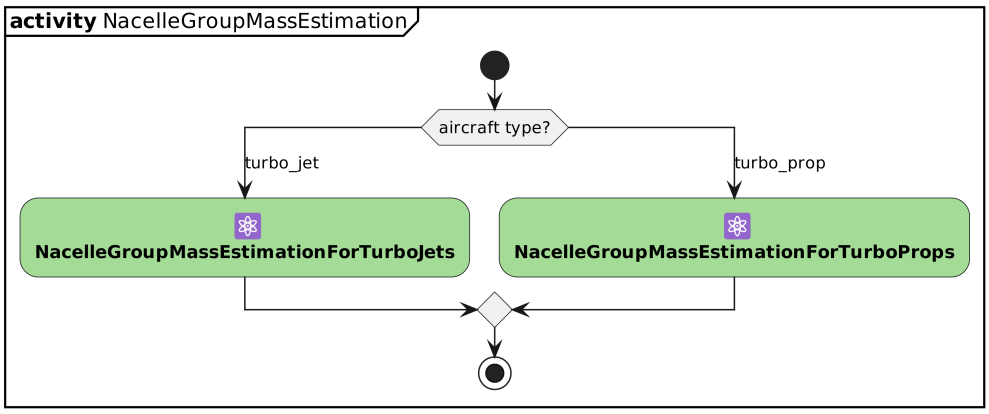


(a)



(b)

Figure 4.12: Example procedures represented as activity diagrams



(c)

Figure 4.12: Example procedures represented as activity diagrams

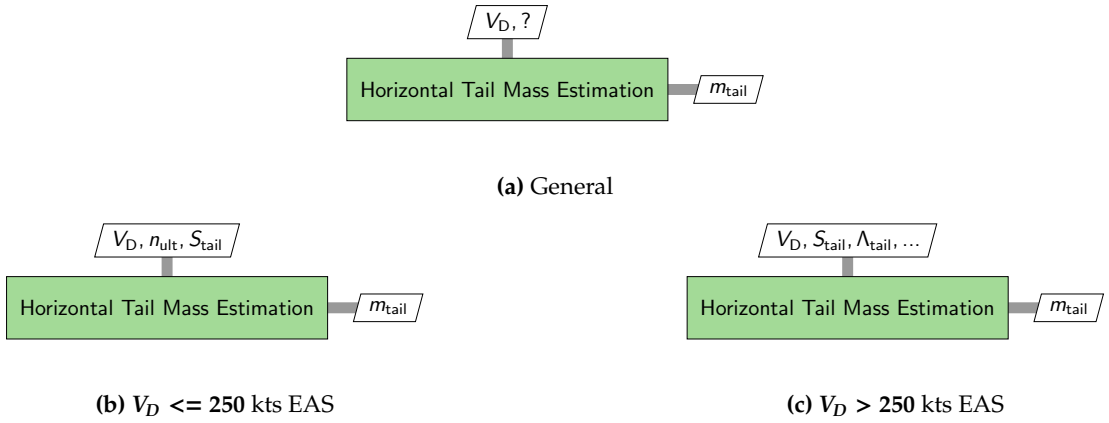
4.4 Automated logging and diagramming capabilities

The majority of the issues outlined in chapter 3 have been addressed by the architectural elements proposed above. Nevertheless, the visualization of data flow within the ADS still remains an outstanding challenge. Addressing this issue is crucial, because at their core ADSs transform a set of input data into another set of output data. When the data flow is not readily traceable, it becomes exceedingly complex to reconfigure the system with confidence, mainly because it becomes unclear whether all the interdisciplinary couplings have been accurately taken into account.

Note that the flowcharts depicted in figure 4.12 have been automatically generated from the source code defining the corresponding procedure classes. The automatic flowchart generation is a core feature of the proposed ADS that allows one to always obtain up-to-date representations of the used procedures. However, these flowcharts do not show the inputs and outputs of the specified procedures. A representation akin to an N^2 chart appears to be necessary to address this limitation.

Initially, efforts were made to specify the inputs and outputs of a procedure separately from the source code that defines the calculation/mapping logic of the procedure. This approach, however, proved overly idealistic. Retrieving the correct inputs is a non-trivial task, necessitating sophisticated logic, especially when dealing with typical synthesis methods. For instance, Torenbeek's weight estimation method stipulates the use of different equations (with different inputs) to calculate the tail weight of an aircraft depending on whether the design dive speed of the aircraft is below or above a certain threshold (see figure 4.13). The tail weight estimation can be modelled as a "ConditionalProcedure" containing two different "SimpleProcedures". In this case, the inputs and outputs of the "SimpleProcedures" can be specified explicitly. However, this is not possible for the "ConditionalProcedure". All attempts to establish a declarative input-output specification for procedures other than "SimpleProcedures" thus were futile[6]. An important observation thus is that the data flow in ADSs always depends on the available data. This is further elaborated on in section 5.3.

[6]Even for "SimpleProcedures" it is not always possible to specify all inputs in a declarative manner. This is because at a certain point it becomes unreasonable to decompose a "SimpleProcedures" into "ComplexProcedures".



(a) General

(b) V_D **<= 250** kts EAS

(c) V_D **> 250** kts EAS

Figure 4.13: Example of a procedure **(a)** that can only be described by a dynamic set of inputs, while the underlying subprocedures **(b, c)** can be represented in terms of a static set of inputs

Notwithstanding, the data flow issue can be addressed by employing the database endpoint, as introduced previously in section 4.1, to trace the inputs and outputs as the aircraft design process advances. This can be achieved by logging which nodes in the database are retrieved or updated when the endpoint is utilized. This technique is further detailed in figure 4.14. While this approach does not allow a pre-execution determination of inputs and outputs, it permits real-time determination during system execution. Then, N^2 charts, akin to the one in figure 4.15, can be readily (and automatically) generated.

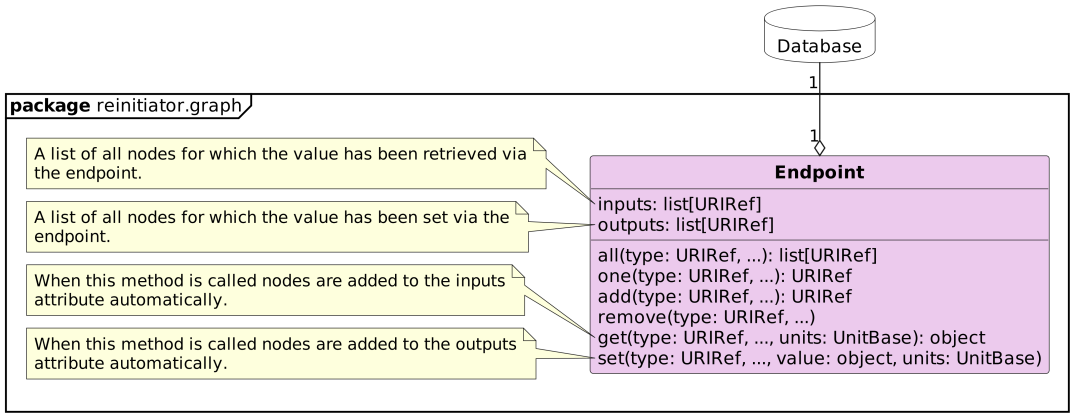


Figure 4.14: Class diagram of the endpoint concept highlighting its logging capabilities

Please observe that the activity diagrams in figure 4.12, the N^2 chart in figure 4.15, and the implementation diagram in figure 4.16 provide different views on the same procedure, each emphasizing distinct aspects of the procedure. The latter diagram serves as a custom visualization designed to offer a quick overview of the procedure while closely resembling the actual structure of the procedure instance[7]. It is important to mention that these visualizations have all been automatically generated. This automatic visualization generation is considered a key element of a reconfigurable ADS because it allows one to gain a lucid understanding of the system's functionality. It also reveals the available options for reconfiguring the ADS.

[7]Representing the nested procedure structure within a standard UML (object or structure) diagram is unfeasible, primarily because UML does not provide the means to effectively depict ordered dictionaries, which are a core element of "ComplexProcedures".

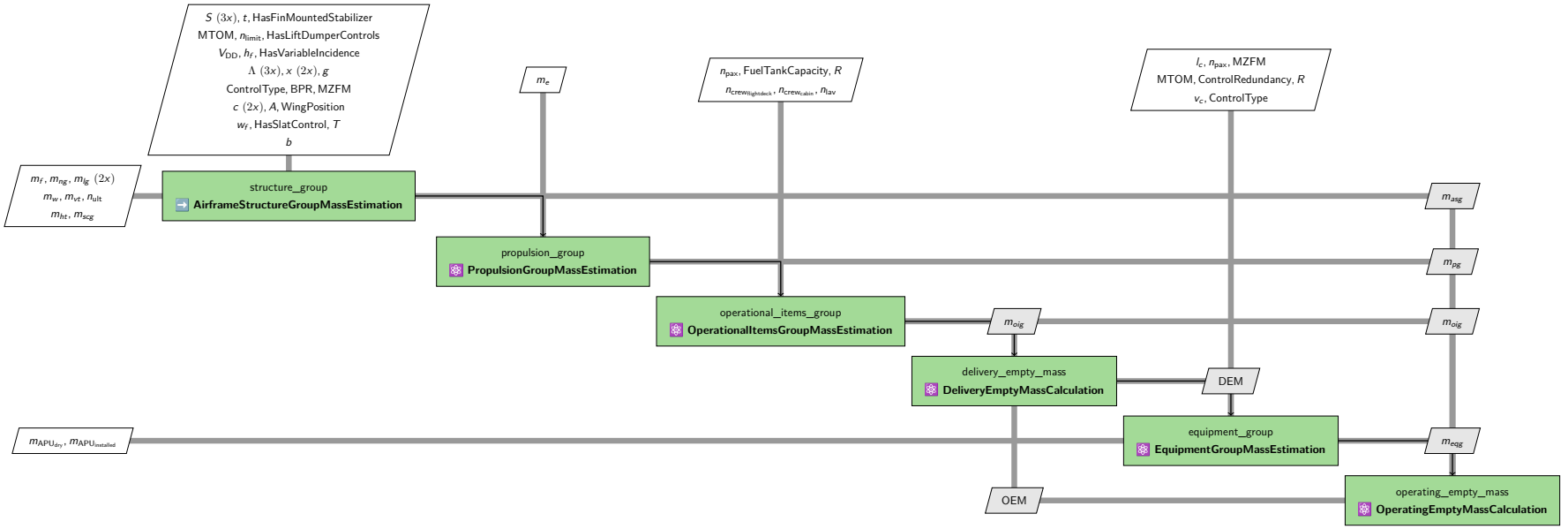


Figure 4.15: Example N^2 chart that has been automatically generated from a procedure execution log[8,9]

[9]Note that "2x", "3x", etc. signifies that there are multiple variables with the same name or symbol. For example, "S (3x)" represents the surface areas of the wing, the horizontal tail, and the vertical tail, respectively.

[9]The N^2 chart shows that the OEM needs to be converged. However, the N^2 chart does not include any convergence mechanism. This is because the convergence mechanism was not defined to be a part of the weight estimation procedure. Instead, it is usually defined to be a part of higher-level procedures (e.g. the synthesis procedure visualized in figure 5.4).

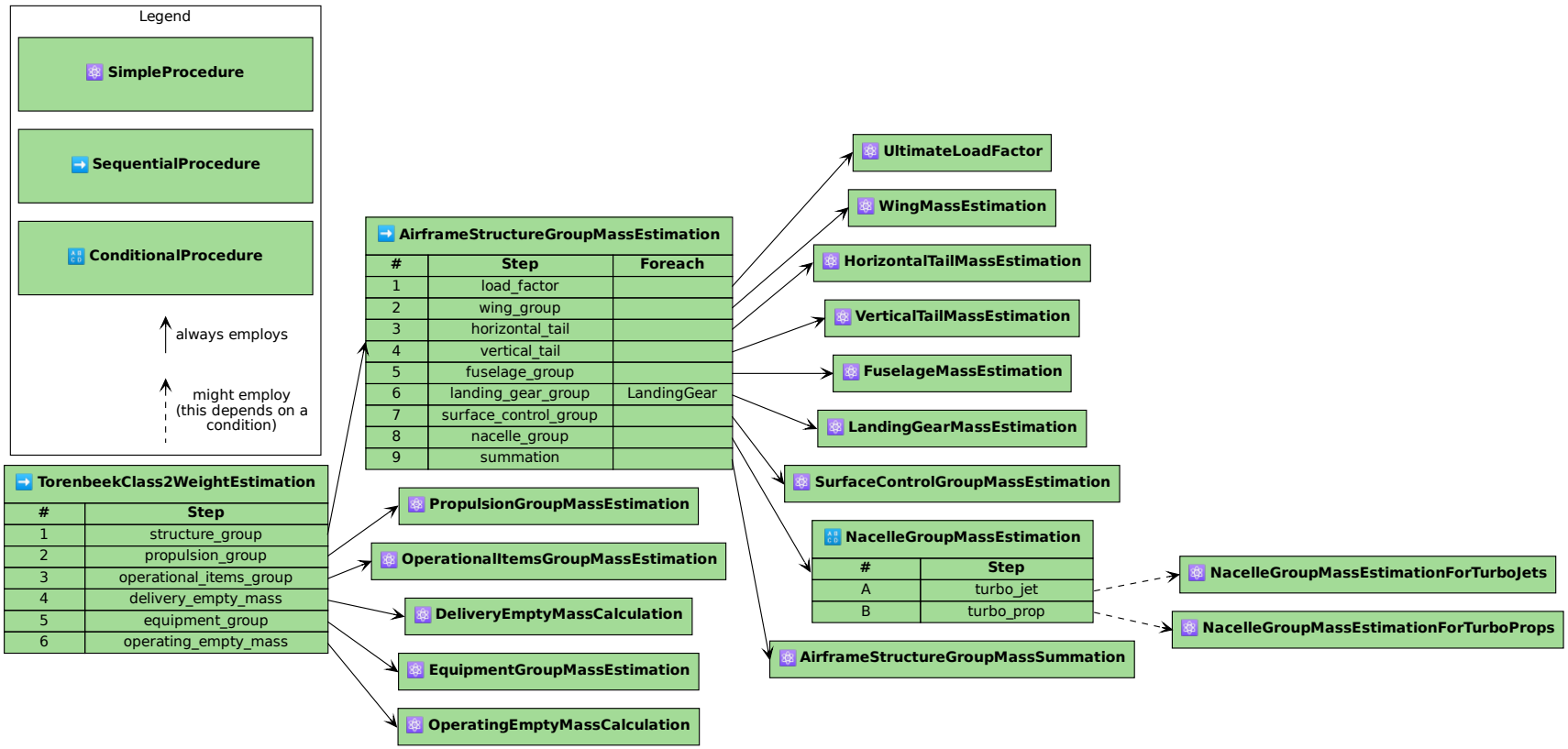


Figure 4.16: Example implementation diagram that has been automatically generated from a procedure instance

5

Implications of adopting the proposed software architecture

When developing an innovative ADS based on the proposed software architecture, notable observations can be made. First of all, the emerging ADS demonstrates distinct advantages in terms of transparency, accuracy, and modularity when compared to existing ADSs. Nevertheless, its implementation proves to be substantially more tedious than that of current ADSs. Furthermore, unforeseen obstacles arise, hindering both the completion and reconfiguration of the novel ADS. This chapter explains some of these observations in detail.

5.1 Relevance of semantic data management

The semantic data management techniques proposed in the previous chapter do not only facilitate but also necessitate precise definitions, particularly of the attributes required for/calculated by the various procedures. Formulating precise definitions is not a trivial task. For instance, the chord length is commonly defined as the "length of an imaginary straight line joining the leading and trailing edges of an airfoil" (adapted from `https://www.wikidata.org/wiki/Q1384332`, see figure 5.1a). Yet, the chord length can also be defined as the "projected length of an imaginary straight line joining [...]" (see figure 5.1b). When there is a significant amount of twist/washout, for example at the tip of a wing, the two definitions represent significantly different lengths (see figure 5.1c). The adoption of an ontology akin to the one outlined in section 4.2 (and described in detail in appendix C.1) facilitates the formulation of unambiguous attribute definitions. Textual definitions can be enriched by referencing related attributes (e.g by specifying superclass and subclass relationships) and external sources (e.g. by linking to an encyclopedia or an image) in order to precisely convey the meaning of attributes.

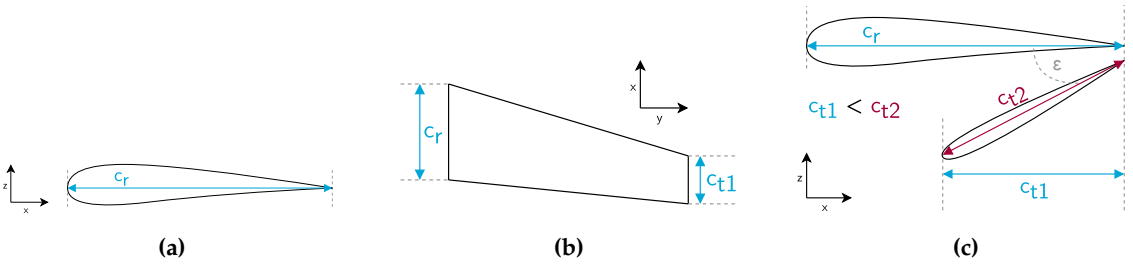
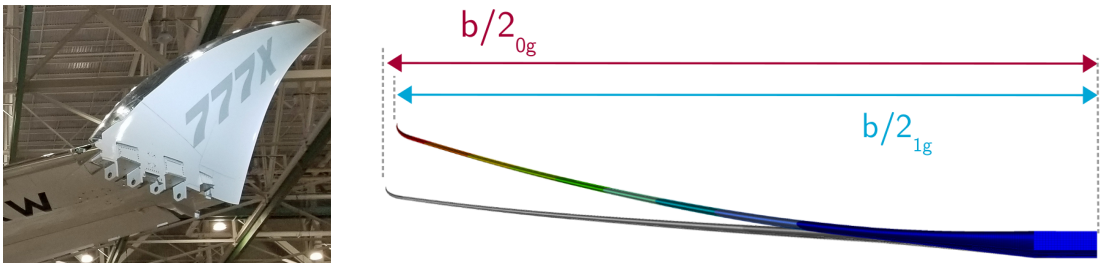


Figure 5.1: Different options for defining the "chord length" attribute

The importance of establishing not just a shared language but also common semantics becomes apparent once an ontology has been defined. It is at this point that inconsistencies

between procedures can surface. For example, a value corresponding to the first definition of the chord length might be required as input for an aerodynamic analysis. The second definition might be used to describe the value of an output of a geometry estimation. Previously, without an ontology, the same attribute might have been easily but mistakenly employed both as an output of the geometric estimation and as input for the aerodynamic analysis. Now, with an ontology, inconsistencies between the employed procedures may become evident. Note that it is challenging to estimate the impact of these inconsistencies in a reliable way. The difference between the two chord length definitions might be negligible for conventional aircraft configurations, where wing twist angles are usually small. The difference might not be negligible for unconventional aircraft configurations, where wing twist angles can be conceivably larger. Therefore, attempts were made to mitigate these inconsistencies by integrating additional calculation/conversion logic into procedures while developing the *ReInitiator*. In some cases, additional convergence cycles were required to resolve these inconsistencies.

The utilization of semantic data management techniques also offers a distinct advantage by facilitating the creation of genuinely flexible data models. This can be achieved by connecting nodes within the data graph. However, this does not require changing the ontology graph. This stands in stark contrast to the data management techniques employed in current ADSs, which typically rely on rigid and restricted data schemas, that do require adjustments to accommodate data model flexibility. For instance, the data model of the *Initiator* allows for specifying a single span per wing. The span is an attribute of a wing, and there can only be one span attribute for every wing. In contrast, the data model of the *ReInitiator* allows for specifying different spans per wing, thereby enabling one to model aircraft with variable spans (e.g. 777X-like aircraft, see figure 5.2a) or aircraft subjected to different loading conditions (see figure 5.2b). An example illustrating which nodes need to be connected to enable this kind of flexibility is shown in figure 5.2c. The semantic data management techniques applied in the *ReInitiator* facilitate the creation of both simple and detailed data models using one and the same ontology. Additionally, by applying these techniques, one can avoid the creation of data models featuring arbitrary hierarchies which often become intricate and unwieldy to modify over time.

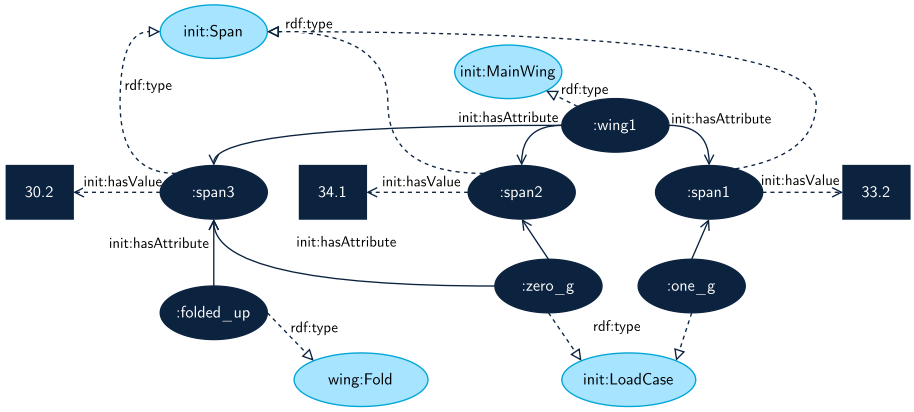


(a) Folding wingtips[1] **(b)** Different loading conditions [25]

Figure 5.2: Different ways to interpret the "span" attribute

As the data model evolves to offer increased flexibility, there arises a necessity to make procedures more universally applicable. Otherwise, one quickly reaches a point where procedures cannot handle the data models anymore. For instance, a procedure initially designed to handle a single attribute describing wing span must be adjusted to accommodate scenarios where multiple attributes of the wing span might be supplied (e.g. by specifying which specific span attribute is required, or by averaging the different span attributes). During the development of the *ReInitiator*, it was observed that these kind of

[1] `https://commons.wikimedia.org/wiki/File:777X_Roll-Out_FoldingWingtip.jpg`



(c) Example RDF dataset corresponding to the loading condition sketch

Figure 5.2: Different ways to interpret the "span" attribute

enhancements were frequently necessary to prevent procedures from raising exceptions once the possibility for a little more flexible data models was taken into account. Hence, additional conditionals (to address edge-cases represented by flexible data models) and loops (to handle scenarios where calculations need to be executed multiple times for attributes present multiple times in the data model) were integrated into the procedures of the *ReInitiator*. While these enhancements rendered the source code of the procedures more extensive and complex, it also rendered the procedures more universally applicable. It appears that the limitations intrinsic to the data model of current ADSs have, in a certain sense, been an advantage that allowed developers to formulate concise analysis & sizing methods. With the transition to more flexible data models, the formulation of appropriate analysis & sizing methods has become considerably more sophisticated and time-consuming.

5.2 Benefits and drawbacks of highly modular procedures

Utilizing modular procedures within an ADS, such as the *ReInitiator*, offers several advantages: Firstly, it provides users with the ability to gain both comprehensive as well as detailed insights into the design process. This is made possible by the transparent procedure nesting capabilities and by the automatic diagramming features of the *ReInitiator*. Secondly, it empowers developers to define unit tests for the rigorous validation of small-scale and independent (sub)procedures. This approach has revealed several bugs in existing source code, and, in one instance, an error within the underlying literature (see section 6.1). Thirdly, it enables users to reconfigure the *ReInitiator* in a LEGO-like manner, by systematically assembling (sub)procedures to tailor the ADS to specific requirements.

However, when attempting to build the *ReInitiator* using code from existing ADSs such as the *Initiator*, unexpected complications materialize. A complication similar to those discussed in the previous section occurs when modularizing procedures: It can be observed that splitting up a single procedures into two (sub)procedures typically requires that multiple additional attributes need to be defined within the ontology to ensure adequate coupling between the (sub)procedures. This means that as procedures become more granular, the ontology becomes more detailed and specific. It is essential to note that this trend is not linear (see figure 5.3, splitting up an extensive procedure requires the introduction of more coupling attributes than splitting up an already concise procedure).

This is problematic for a number of reasons: Firstly, the ontology becomes volatile and challenging to manage due to the abundance of highly specialized attributes used solely for interconnecting two (sub)procedures. Secondly, the apparent modularity of procedures becomes questionable, as these highly specialized attributes, exclusively used for interconnecting (sub)procedures, foster implicit interdependence. Predicting the future use of these attributes within other procedures is elusive. Thirdly, querying and writing data from a central database imposes computational overhead, especially when handling non-numeric data (e.g. step files, large binary data) that necessitate serialization. Furthermore, the use of a central database prevents the concurrent execution of procedures.

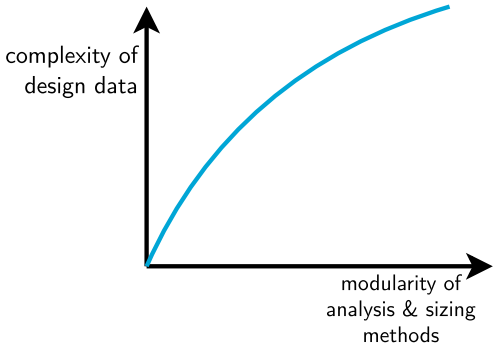


Figure 5.3: Assumed relation between modularity of procedures and complexity of the design data

Note that directly reusing analysis & sizing methods from existing ADSs is impractical. Instead, extensive redevelopment of procedures is required to establish modular procedures. Furthermore, during the development of the *ReInitiator* it became evident that overemphasizing accuracy and precision aspects of an ADS can delay or even hinder the study of aircraft designs by diverting focus towards minute details rather than the overall design. In light of this, identifying the right level of modularity remains a significant challenge. During the development of the *ReInitiator* an extremely fine-grained level of modularity was targeted for two primary reasons: Firstly, to enable desirable reconfigurations (as outlined in chapter 2), which often involve minor adjustments that need to be made deep inside the individual procedures encapsulating analysis & sizing methods. Secondly, to prevent code duplication across different procedures, thereby ensuring coherence, and averting common issues (as explained in chapter 3). As explained in the first paragraph of this section, the highly modular structure of the procedures implemented in the *ReInitiator* leads to remarkable transparency, accuracy, and reconfigurability. However, this highly modular structure comes at the expense of (unintentionally) increased complexity of the ontology. It thus remains an important recommendation to find the optimal balance between the modularity of procedures and the complexity of the ontology.

5.3 Consequences of dynamic procedure behavior

The procedures of the *ReInitiator* cannot be described by a static set of inputs and outputs. Instead, the inputs and outputs of the procedures are determined dynamically, based on the available design data. This dynamic procedure behavior is a key feature of the *ReInitiator*. At the same time this behavior is a drawback, as it makes the initial configuration and the subsequent reconfigurations of the *ReInitiator* more challenging than it would be if the procedures would exhibit a static behavior.

The dynamic procedure behavior is indispensable for an ADS, especially when the ADS

is employed for synthesis studies. This distinctive behavior enables the ADS to handle several aspects:

- Continuously evolving design data: At the start of the design process, there is hardly any data. As the design process advances, more and more data is generated. Occasionally, data might also be invalidated or deleted.
- Multiple fidelity levels: This involves seamlessly incorporating results from high-fidelity methods into lower-fidelity methods. This capability is pivotal for refining and optimizing the design iteratively.
- Various different technologies: For example, the system accommodates a range of aircraft configurations (tube-and-wing, blended-wing-body, box-wing, ...). Additionally, it can handle various propulsion technologies (turbo-fan, turbo-prop, electric, ...).

Figure 5.4 illustrates the synthesis process employed within the *ReInitiator*. Table 5.1 shows the evolution of the database and the procedures throughout this synthesis process. As the synthesis process advanced, the database is populated with design data. Similarly, the feedback/feedforward connections between the procedures vary as the design process advances. This is because different (sub)procedures become relevant based on the design data present in the database which might have been generated by preceding procedures.

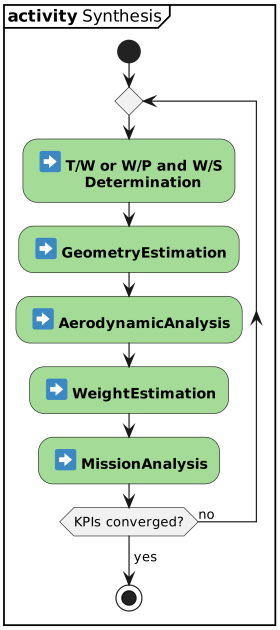
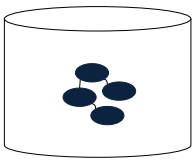
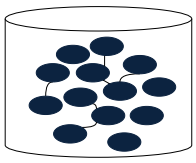
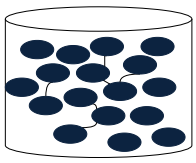
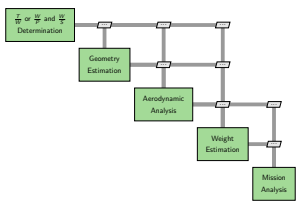
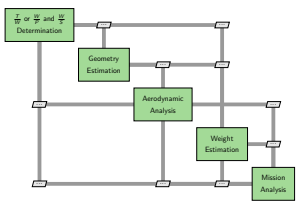
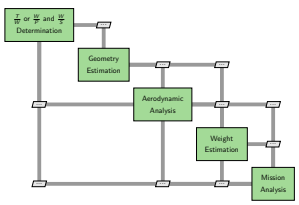


Figure 5.4: A typical design/synthesis process

It is important to realize that N^2 charts are only representative for a single iteration of a synthesis process. In subsequent iterations, the procedures within the N^2 charts typically feature different input/output/coupling variables. Moreover, N^2 charts do only show what attributes from the database are needed to execute a synthesis process. However, the structure of the database plays an equally important role to determine how a synthesis process is to be executed. Hence, while N^2 charts are valuable tools for reviewing a synthesis process, their utility for structuring a synthesis process is limited (and the same holds for charts derived from N^2 charts, such as XDSM charts). Therefore, activity dia-

Table 5.1: Illustrations of design data and analysis & sizing methods during subsequent iterations of a typical design/synthesis process

iteration	**1**	**2**	**...**	**n**
design data			...	
analysis & sizing methods			...	

grams (as the one shown in figure 5.4) and implementation diagrams (as the one shown in figure 4.16) are intended to structure the design process of the *ReInitiator*. N^2 charts can be used to review the design process after it has been executed with the *ReInitiator*. It shall be noted that there have been numerous attempts to integrate logic into N^2 charts in the past (see [26, 27] for an overview of such possibilities to extend N^2 charts). However, it shall also to be noted that such N^2 charts containing logic can no longer offer a straightforward overview of the actual data flow within a synthesis process.

The *ReInitiator* allows its users to define dynamic procedures. It also allows its users to perform reconfigurations by restructuring these procedures. While doing so the users need to pay close attention that no procedure along the way requires an input for which a value has not yet been calculated. However, anticipating the inputs required to execute a procedure/the outputs generated by a procedure is challenging. The ability of the *ReInitiator* to create N^2 charts eases such reconfigurations. Workflow management tools like *KADMOS* [11] and *InFoRMA* [22] could further ease such reconfigurations but are currently unsuitable for assisting users in these scenarios because they depend on analysis & sizing methods with fixed inputs and outputs. Nevertheless, using these tools would be helpful to prevent one from making seemingly straightforward reconfigurations which can render the *ReInitiator* inoperable: For example, it might not be possible to execute the system anymore (when some procedures lack elementary inputs). Alternatively, the system may fail to converge (when there is excessive coupling between the procedures).

It has also been attempted to modify procedures with the aim of achieving static inputs and outputs. However, these attempts were mostly unsuccessful due to inherent interdependencies within procedures. For example, the geometry estimation contains two different relations for calculating the so-called "tail arm". One of these relations is an estimation that only needs to be executed during the first iterations when the geometry of the tail is not known yet. The other relation is a more accurate estimation but it can only be utilized once the geometry of the tail is known. It has been attempted to exclude the first relation from the geometry estimation procedure and instead include it in a separate procedure that is only executed once at the beginning of the design process (outside the main convergence loop). However, this turned out to be infeasible because the first relation itself had dependencies on other design parameters calculated earlier within the geometry estimation. This example shows that the order in which the procedures are normally executed is not random, but it is the result of careful considerations and experiential insights. Hence, there is a lot of implicit knowledge embedded in the order in which procedures are normally executed. When using the *ReInitiator* in a reconfigurable way, e.g. by unduly rearranging procedures, one may inadvertently lose this implicit but invaluable knowledge.

5.4 Impact of assumptions ingrained within analysis & sizing methods

When attempting to reconfigure the *ReInitiator*, it further becomes apparent that certain assumptions ingrained within the analysis & sizing methods can lead to critical problems. This is because these assumptions may cease to be valid once the *ReInitiator* is used in a reconfigurable way. In particular, this applies when using the *ReInitiator* to perform design space exploration or optimization studies (instead of synthesis studies). When the assumptions conflict with explicit constraints imposed by the respective studies, this can lead to convergence issues or unexpected (and possibly invalid) outcomes. Three examples of assumptions that are frequently encountered in the utilized analysis & sizing methods are discussed below.

Some of the utilized analysis & sizing methods incorporate sub-level optimizations. For example, the $\frac{W}{S}$ and $\frac{T}{W}$ determination procedure selects a design point from the matching plot which maximizes the take-off wing loading (but does not necessarily minimize take-off thrust-to-weight ratio), as illustrated in figure 5.5. Here, the assumption is that users of the ADS invariably wants to maximize the take-off wing loading. The feasible design space is much larger and may conflict with the high-level optimization objectives pursued by the users.

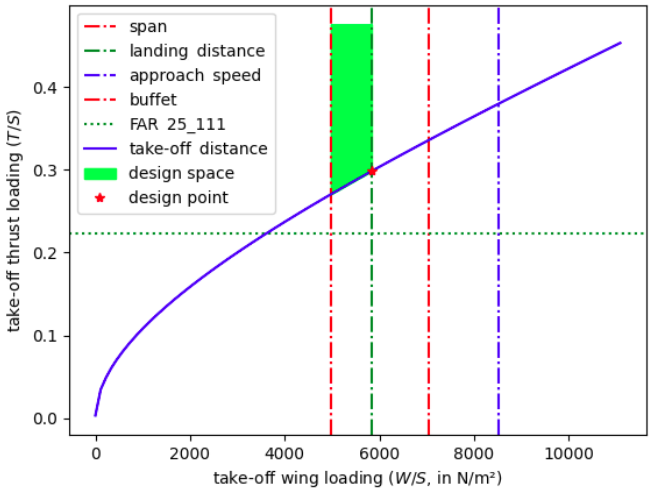


Figure 5.5: Matching plot showing selected requirements, the feasible design space constrained by these requirements, and the selected design point

Similarly, the fuselage sizing procedure always attempts to generate a fuselage with minimal length (or surface area). In this case, the assumption is that users of the ADS invariably seek to minimize the fuselage length (or surface area). However, this assumption likely limits the design space and may conflict with the high-level optimization objectives pursued by the users. This is illustrated in figure 5.6.

Many of the utilized analysis & sizing methods are based on reversed requirements. For example, in the detailed wing sizing procedure, instead of verifying if the spar can support a specific load, the spar is designed precisely to withstand the given load. While the obtained spar dimensions are rather lower limits, they are treated as definite values. In this case, the assumption is that the spar should never have larger dimensions than necessary to withstand the given load. Similarly to the examples above, this assumption also limits the design space significantly.

The assumptions exemplified represent fundamental characteristics of synthesis methods. Their presence in synthesis methods is crucial. Their absence would result in an overwhelming degree of design freedom, rendering synthesis unfeasible. At the same time, these assumptions prevent realistic design space exploration and optimization studies.

Please note that these assumptions have come to light primarily due to the remarkable transparency of the *ReInitiator*. On the contrary, conventional ADSs tend to obscure these assumptions. In theory, the high degree of modularity within the *ReInitiator* opens up the possibility of modifying the assumptions ingrained within the analysis & sizing methods. In practice, this would necessitate numerous intricate adjustments and result in entirely different analysis & sizing methods.

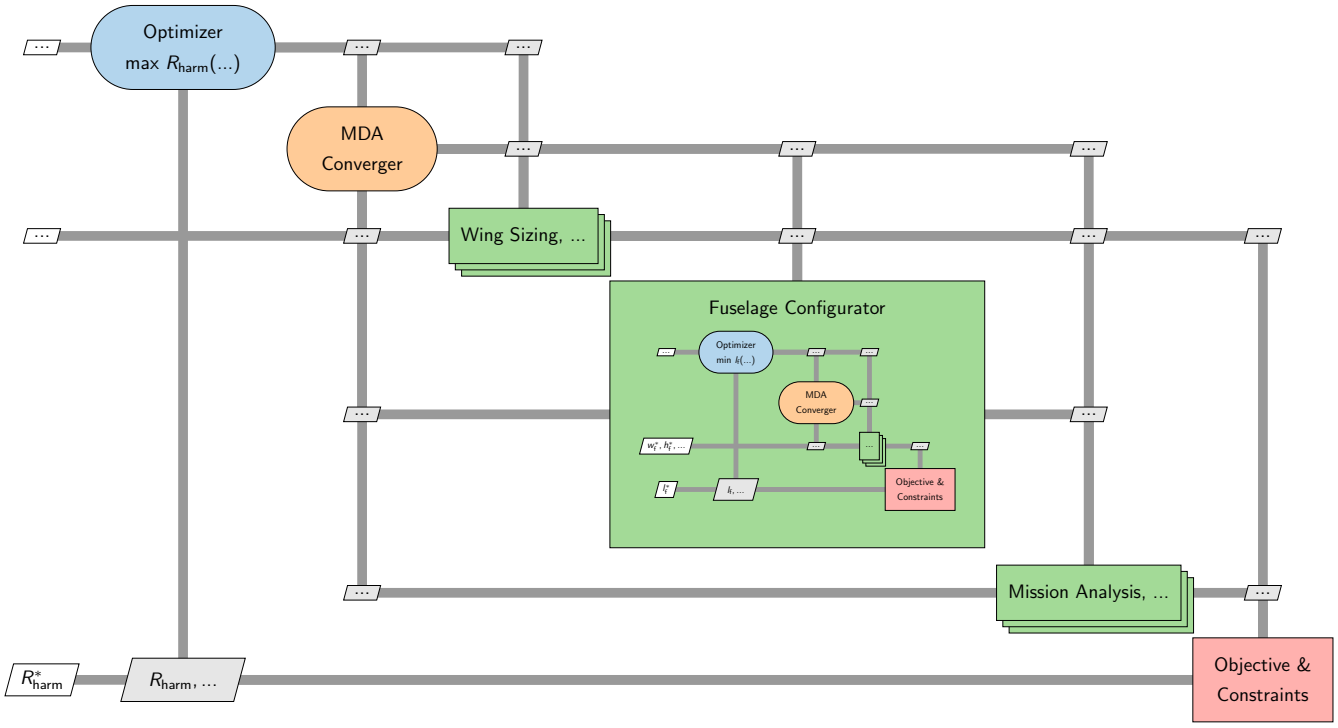


Figure 5.6: XDSM of a multi-level optimization with (potentially) conflicting objectives

Initially, it was envisioned that the modularity of the *ReInitiator* would allow for simply removing all procedures that perform sizing tasks from the system, so that only the procedures that perform analysis tasks remain within the system. However, this turned out to be infeasible because even after modularizing analysis & sizing methods by developing procedures it was not always possible to clearly determine if a procedure is of analysis or rather of sizing type. Furthermore, it turned out to be infeasible since it was not always possible to determine the requirements that lead to the sizing tasks (which would need to be implemented as constraints in optimization studies).

6
Verification & Validation

In the following section, the effectiveness of the proposed software architecture is evaluated by describing the software tests and quality checks carried out on the *ReInitiator* (i.e. the prototype implementation of the software architecture). Additionally, the most important characteristics of the *ReInitiator* are compared with those of other ADSs and related MDAO systems.

The *ReInitiator* cannot be considered a fully functional ADS. Establishing the overall architecture, setting up the repository, formulating a self-explanatory ontology, constructing the endpoint interface, and developing the self-documenting base procedures classes demanded a significant amount of time. Hence, there was only insufficient time left for implementing a comprehensive set of procedure classes essential for materializing a thorough aircraft design synthesis loop. Furthermore, it became evident that the existing analysis & sizing methods, on which the *ReInitiator* is based, come with inconsistencies and limitations, as detailed in the previous chapter. Deliberately incorporating these inconsistencies and limitations into the *ReInitiator* was deemed inappropriate, since their impact on the overall design process could not be reliably estimated. As a consequence of the time constraints imposed on this research project, achieving a fully operational ADS proved unattainable. Consequently, it was not possible to demonstrate this ADS can be used in a reconfigurable way, despite indications that this could very well be the case.

6.1 Automated tests & quality checks

In the realm of software development, assuring the correct functioning of the software is of utmost importance. As briefly mentioned in section 2.1, the principles of test-driven development have been adhered to during the development of the *ReInitiator*. The key idea is to create tests in parallel with the code and to execute them frequently. Three categories of fully automated tests have been employed during the *ReInitiator*'s development: unit tests, code quality checks, and integration tests.

Unit tests, that is tests of small-scale units of code, played a pivotal role. Over 100 of these unit tests have been set up to verify that the analysis & sizing methods work correctly and deterministicly (i.e. that they always produce the same output for the same input). This level of testing was made possible by the modular structure of the procedures. An example unit test is delineated in appendix C.4. Unit testing proved exceptionally effective in the identification of bugs, including unit conversion errors and inadvertent omissions in equations. Remarkably, unit testing even revealed a minor discrepancy in the underlying literature/a standard aircraft design textbook[1].

[1] Equation 3.11 from [2] that can be used to calculate the overall engine efficiency of a turbo-fan engine should not read $\eta_0 = \frac{a_{sl}}{H/g} \frac{M\sqrt{\theta}}{C_T}$ but $\eta_0 = \frac{a_{sl}}{H} \frac{M\sqrt{\theta}}{C_T}$ (note the superfluous gravitational acceleration factor in the

Furthermore, code quality checks were carried out through pre-commit hooks, automating various checks prior to committing code to the code repository. This encompassed a code formatting tool[2] ensuring uniform coding style by automatically reformatting the code according to certain standards. Among others, a linting tool[3] was employed, checking that there is no "dead" code and that there are no unnecessary, duplicated, or overly complex code artifacts. Finally, an ontology analysis tool[4] was used, examining the ontology, detecting potential problems and identifying inconsistent classes. Although commonly adopted in professional software projects, these tools remain underutilized in research projects, despite their potential to enhance code quality and comprehension. This becomes especially relevant when multiple people and/or people without experience in software design collaborate on the same source code. As explained in chapter 3, although there are guidelines that indicate how code should be structured, they are often not adhered to. The use of automated quality checks can help to enforce such guidelines.

Last but not least, integration tests were devised to confirm compliance with the requirements specified in section 2.4. These tests were intended to validate that the entire *ReInitiator* is indeed reconfigurable. Regrettably, due to the aforementioned time constraints, it was not possible to demonstrate this on system level, by, for example, executing a synthesis or optimization study. Nevertheless, successful integration tests were conducted at subsystem level. It could be shown that it is possible to seamlessly substitute a procedure with another one by modifying the steps attribute of complex procedure objects (see appendix C.4 for implementation details). In principle, such modifications can be performed even at runtime. Furthermore, it was shown that it is a straightforward task to remove entire procedures and specify design parameters, which would normally have been computed by said procedures, manually in the database. In fact, this was done frequently while setting up unit tests (see also appendix C.4 for an example).

6.2 Comparison of the *ReInitiator* with other ADSs and related MDAO systems

In the following, a brief comparison of the *ReInitiator* with other ADSs is presented. The objective of this section is to underscore the commonalities and distinctions between the *ReInitiator* and other ADSs, shedding light on emerging trends aimed at improving ADSs in general.

An emerging trend in ADSs and MDAO systems involves establishing a central data base with a corresponding data schema that is independent of the used analysis & sizing methods. For instance, the developers of *SUAVE* refer to this concept as "attribute-method orthogonality" [28]. Furthermore, it is one of the main motives that led to the creation of *CPACS* [29] and *KADMOS* [12]. The *ReInitiator* builds upon this trend but adopts a more extensive approach, employing an ontology (that comes with well-defined semantics) instead of traditional XML or database schemas (that might not adequately capture the nuanced meanings of design data and that often lack the flexibility to represent complex relationships present within design data). The shift from ad-hoc data structures, through static schemas, to expressive ontologies appears promising and has the potential to reduce data interpretation inconsistencies between analysis & sizing methods and entire systems (without the need for additional interpretation layers as proposed in [30] to address the

first equation). The first (incorrect) equation has been used several times in an existing ADS. Interestingly, the error did not significantly affect the overall design process and hence remained undetected.

[2] `https://github.com/psf/black`

[3] `https://github.com/PyCQA/flake8`

[4] `https://gitlab.com/lukasmu/ontocop`

deficiencies of schemas). Utilizing ontologies also facilitates the construction of flexible and multi-fidelity data models (as explained earlier in section 5.1).

During the development of the software architecture for the *ReInitiator* two alternative approaches for managing design data were considered: One involved the utilization of the well-known *CPACS* schema (e.g. as done in MDAO systems [29, 31, 32]), while the other one involved the development of an object-oriented data model (e.g. similar to the models employed in *SUAVE* [28] or *ADEBO* [3]). The first approach was swiftly discarded since *CPACS* turned out to be too inflexible[5], unpractical[6], and ambiguous[7] to be used in the early design phases which were targeted by this research project. Furthermore, handling XML files based on a schema like *CPACS* has proven to be inefficient in the past [13]. The second approach was investigated in more detail. In particular, setting up an object-oriented data model based on class hierarchies that is linked to a relational database by means of standard object-relational mapping techniques was investigated. However, this approach also turned out to be too inflexible, as these techniques were predominantly tailored to more heterogeneous and more structured data sources (e.g. corporate databases). Moreover, it was challenging to adequately represent the multitude of interconnections between attributes and systems (as illustrated previously in figure 5.2). Eventually this approach evolved into a kind of entity-attribute-value data structure, which was already similar to the adopted graph database, but much more cumbersome to work with.

A distinctive facet of the *ReInitiator* is its innovative approach to analysis & sizing method definition. In contrast to prevailing ADSs that predominantly treat analysis & sizing methods as integrated, black-box tools with extensive functional breadth, the *ReInitiator* treats them as transparent, modular, and nested procedures. This approach enhances user understanding and potential customization of analysis & sizing methods, thereby fostering trust and confidence in the ADS. Furthermore, it enables the integration of new (sub)methods into existing (super)methods. The development of procedures for the *ReInitiator* demands a comprehensive understanding of analysis & sizing methods, requiring a substantial upfront time investment. In situations where time constraints prevail, engineers or researchers focusing on specific aircraft concepts may find it more practical to employ wrappers around existing tools. By employing such wrappers it is possible to quickly craft proof-of-concept research projects. For this reason wrappers are frequently used in current ADSs (see e.g. [3, 29, 31]). However, such wrappers hinder in-depth reconfigurations and impede proper unit testing. Hence, for long-term research projects, an approach akin to the one implemented in the *ReInitiator*, featuring self-documentation and self-visualization capabilities, likely constitutes a more sustainable choice. Such an approach facilitates introspection and reconfigurations, allowing for greater longevity in the context of evolving research requirements.

Existing ADSs typically rely on a coordinating object to manage the orchestration of analysis & sizing methods and facilitate data exchange among them. The "Controller" object of the *Initiator* (shown in figure 3.2) is a prime example of such an object. Similarly, in *ADEBO* there is an "Artificial Engineer" [3] and in *MICADO* there is a "Study Manager" [34]. In contrast, the *ReInitiator* operates differently, with the procedures themselves serving as coordinating entities. This approach facilitates a flexible configuration of the design system, manifesting a "system-of-systems" approach at method level. Furthermore, the central database ensures that methods can autonomously access the required data. The endpoint fulfils a crucial role by logging which data is retrieved and/or updated. A

[5] *CPACS* is specifically designed to excel at a particular level of fidelity [33].

[6] *CPACS* is intentionally designed not to support the storage of certain parameters relevant for conceptual aircraft design, such as wing span, to mitigate the risk of data inconsistencies [33].

[7] *CPACS* can easily be interpreted in inconsistent ways, as pointed out in [30].

comprehensive overview of the data flow can only be obtained post-execution of analysis & sizing methods, as elaborated upon in section 5.3. It should be noted that the data flow, once available, can be supplied to a workflow management tool like *KADMOS* [12]. During this research project, an attempt was made to check if reordering procedures using an automatically determined function order based on different algorithms provided by *KADMOS* could lead to faster convergence and, hence, a more efficient design process. Whether or not this is the case could not be conclusively determined yet.

The concept of representing analysis & sizing methods as workflows is not entirely new and shares similarities with constructs like "Jobs" in *ADEBO* [3] and "Workflows" in *RCE* [35]. However, this representation does not align well with classical MDAO studies, where advanced optimizing algorithms often rely on fixed input-output relationships, differentiable representations of analysis & sizing methods, and predetermined initial values. Classical MDAO studies primarily use strictly mathematical relationships, whereas synthesis studies (although they can be considered specific subset of MDAO studies) require more logical relationships. It becomes evident that analysis & sizing methods used for different types of studies are not necessarily compatible and should be distinguished.

7 Conclusion

Current ADSs are frequently used to perform synthesis studies. However, it is inherently difficult to reconfigure these systems. As a result, current ADSs are rarely used to investigate the synthesized aircraft in greater detail or to perform a broader range of aircraft design studies.

To address this problem, a research project has been initiated. The objective of this research project was to explore the feasibility of developing a software architecture that would facilitate the reconfiguration of ADSs. The findings of this research project have been documented in the report at hand.

7.1 Review

First, the main issues that prevent current ADSs from being used in a reconfigurable manner were identified. Afterwards, an iterative development methodology was employed to come up with a software architecture aimed at addressing these issues. Simultaneously, the *ReInitiator*, a reference implementation of the proposed software architecture, was created. An overview linking the identified issues to the corresponding architectural elements is presented in table 7.1.

Table 7.1: Mapping between the proposed architectural elements for a reconfigurable ADS and the corresponding issues of current ADSs

architectural element	**addressed issues**
Centralized data store and self-contained analysis & sizing methods	Ambiguous and cluttered data structures Excessive coupling within source code
Semantic data management	Ambiguous and cluttered data structures Concealed and convoluted source code
Standardized and modular analysis & sizing method interface	Complex source code Extensive and integrated source code
Automated logging and diagramming capabilities	Concealed and convoluted source code Extensive and integrated source code

The creation of the *ReInitiator* required significantly more time than initially estimated. Ultimately, time constraints prevented that the *ReInitiator* could be utilized to demonstrate that adopting the proposed software architecture leads to a reconfigurable ADS. Nevertheless, the creation of the *ReInitiator* yielded valuable insights that are relevant for the enhancement of current ADSs and the development of future ADSs. Core observations, which stand as a significant outcome of this research project, are listed in the following.

- By applying semantic design data management techniques, such as making use of a standard graph database and establishing a formal ontology, it becomes possible to construct genuinely unambiguous and flexible data models. The adoption of semantic design data management techniques mandates accuracy and precision. When applied diligently, these techniques may (automatically) reveal inconsistencies and limitations in the employed analysis & sizing methods (and thus may induce a need for further examination and potential improvement of the employed analysis & sizing methods).
- Formulating analysis & sizing methods in a modular way using standardized procedure classes appears to be the key to achieving reconfigurability. Self-visualization features incorporated into these classes can significantly enhance the comprehension of analysis & sizing methods. Furthermore, the modular nature of the procedure classes facilitates the creation of unit tests, thereby fostering trust and confidence into the analysis & sizing methods. Most importantly, instances of these classes can be treated like LEGO bricks, which empowers users to configure and reconfigure a design system as desired.
- The instances of procedure classes exhibit a dynamic behavior (i.e. they cannot be described by a static set of inputs and outputs). This behavior is essential for synthesis-oriented ADSs, but appears to be inadequate for optimization-oriented ADSs. Although activity and implementation diagrams outlining the general functioning of procedure instances can be obtained pre-execution, N^2 charts that accurately represent the data flow within an ADS can only be obtained post-execution. This complicates reconfiguration efforts but appears to be unavoidable when analysis & sizing methods designed for synthesis are employed.
- The transparency and modularity of the *ReInitiator* revealed that many of the employed analysis & sizing methods are not merely based on inverted requirements but directly target implicitly specified optima. While this was already anticipated when starting the development of the *ReInitiator*, the extent of these optimality assumptions and their concealment deep inside the analysis & sizing methods was surprising. These optimality assumptions may no longer hold when using the ADS in a reconfigurable manner. However, it became evident that these assumptions cannot be removed from the analysis & sizing methods without rendering the initial objective of the ADS (i.e. performing synthesis) infeasible. This significantly restricts the extent to which the analysis & sizing methods can be reused in a reconfigurable ADS and necessitates further investigation.

7.2 Recommendations

During the development of the *ReInitiator* some interesting ideas emerged that could not be investigated due to time constraints. Specifically, it remains a recommendation to:

- Examine if workflow management tools like *KADMOS* or *InFoRMA* can be adapted to accommodate complex procedures that are not accompanied by precisely predetermined inputs and outputs. It might be possible to rely on other indicators than inputs and outputs for estimating the coupling between complex procedures. If this is the case, then these workflow management tools could be valuable for reconfiguring ADSs. In this respect, it might also be beneficial to define principal inputs and outputs when creating complex procedures. This approach could pave the way for applying demand-driven or dependency-resolution strategies to automatically determine the procedure execution order (instead of relying on a predefined procedure

execution order that might not be an optimal one).

- Investigate which level of modularity would be most desirable for procedures. It has been demonstrated that more modular procedures require more complex ontologies. Creating procedures with an extremely fine-grained level of modularity while developing the *ReInitiator* demanded a significant amount of time, while the advantages of this level of modularity remain ambiguous. Furthermore, it would be interesting to assess whether exposing procedures with different levels of modularity to an optimizer has an impact on the optimization runtime and/or outcomes.
- Continue the development of the ontology created during this research project and investigate its applicability to other research projects within the field of aircraft design. This ontology offers the potential for precise and unambiguous communication of aircraft design data and could be a replacement for *CPACS* as it allows for a more consistent interpretation of aircraft design data. Additionally, the ontology could be used to compile a reference database containing aircraft design data for verifying and validating various ADSs.
- Investigate the establishment of a well-documented library of aircraft design equations, methods, and tools that are not yet integrated into complex ADSs. This approach would empower users to create custom ADSs, enabling a more in-depth understanding of the underlying assumptions and the selection of appropriate functions and relations for the specific design problem at hand. An example of such a custom ADS could be a synthesis system (e.g. containing additional procedures that specify synthesis-specific logic), while another one could be an optimization system (e.g. containing additional procedures that specify constraint calculations).

7.3 Closing

This thesis project revolved around ADSs. However, it deviated from the typical projects dealing with the development of new analysis or sizing functionalities for such systems. Instead, the project focused on the overall structure and functioning of these systems. This project has also been an exercise in reflecting on current ADSs. Investigating the challenges stemming from their source code and proposing software architecture elements to address these challenges formed a core part of this project. Subsequently, the *ReInitiator* was developed as a prototype implementation of the proposed architecture.

Regrettably, due to time constraints, validation of whether the *ReInitiator* constitutes a reconfigurable aircraft design system could not be completed. Nevertheless, it could be shown that adopting the proposed architectural elements, tabulated above, enhance comprehensibility and credibility of the ADS and can even lead to the detection of inconsistencies and limitations in the utilized analysis & sizing methods. Furthermore, it is possible to selectively employ the proposed architectural elements to incrementally enhance existing ADSs. Hence, it can be concluded that this thesis project yielded insights that are not only relevant for the development of future ADSs but also for the enhancement of current ADSs.

Bibliography

[1] Anemaat, W. A. J., "Conceptual Airplane Design Systems," *Encyclopedia of Aerospace Engineering*, Wiley, Hoboken, New Jersey, 2010. URL https://doi.org/10.1002/9780470686652.eae394.

[2] Torenbeek, E., *Advanced aircraft design: conceptual design, analysis, and optimization of subsonic civil airplanes*, Wiley, Chichester, United Kingdom, 2013. URL https://onlinelibrary.wiley.com/doi/book/10.1002/9781118568101.

[3] Herbst, S., "Development of an Aircraft Design Environment Using an Object-Oriented Data Model in MATLAB," Ph.D. thesis, TU München, München, Germany, 2018. URL http://mediatum.ub.tum.de/doc/1431402/1431402.pdf.

[4] Elmendorp, R., Vos, R., and La Rocca, G., "A conceptual design and analysis method for conventional and unconventional airplanes," *Proceedings of the 29th Congress of the International Council of the Aeronautical Sciences*, ICAS, St. Petersburg, Russia, 2014. URL http://resolver.tudelft.nl/uuid:1dc55ce5-18c3-4986-b668-f70d9b24aac0.

[5] Brown, M., and Vos, R., "Conceptual Design and Evaluation of Blended-Wing Body Aircraft," *Proceedings of the AIAA Aerospace Sciences Meeting*, AIAA, Kissimmee, Florida, 2018. URL https://doi.org/10.2514/6.2018-0522.

[6] Zohlandt, C. N., "Conceptual Design of High Subsonic Prandtl Planes," Master's thesis, TU Delft, Delft, The Netherlands, 2016. URL http://resolver.tudelft.nl/uuid%3Ae1f01743-e2eb-4d8b-8b2c-131f50f41a2c.

[7] Hoogreef, M., Vos, R., de Vries, R., and Veldhuis, L. L., "Conceptual Assessment of Hybrid Electric Aircraft with Distributed Propulsion and Boosted Turbofans," *Proceedings of the AIAA Scitech 2019 Forum*, AIAA, San Diego, California, 2019. URL https://doi.org/10.2514/6.2019-1807.

[8] Vos, R., Wortmann, A., and Elmendorp, R., "The optimal cruise altitude of LNG-fuelled turbofan aircraft," *Journal of Aerospace Operations*, Vol. 4, No. 4, 2017, pp. 207–222. URL https://doi.org/10.3233/AOP-160063.

[9] Smith, H., Sziroczák, D., Abbe, G., and Okonkwo, P., "The GENUS aircraft conceptual design environment," *Proceedings of the Institution of Mechanical Engineers, Part G: Journal of Aerospace Engineering*, Vol. 233, No. 8, 2019, pp. 2932–2947. URL https://doi.org/10.1177/0954410018788922.

[10] Kroo, I., "A quasi-procedural, knowledge-based system for aircraft design," *Proceedings of the Aircraft Design, Systems and Operations Conference*, AIAA, Atlanta, Georgia, 1988. URL https://doi.org/10.2514/6.1988-4428.

[11] van Gent, I., "Agile MDAO Systems: A Graph-based Methodology to Enhance Collaborative Multidisciplinary Design," Ph.D. thesis, TU Delft, Delft, The Netherlands, 2019. URL http://resolver.tudelft.nl/uuid%3Ac42b30ba-2ba7-4fff-bf1c-f81f85e890af.

[12] van Gent, I., and La Rocca, G., "Formulation and integration of MDAO systems for collaborative design: A graph-based methodological approach," *Aerospace Science*

and Technology, Vol. 90, 2019, pp. 410–433. URL https://doi.org/10.1016/j.ast.2019.04.039.

[13] Bruggeman, A.-L., "Automated Execution Process Formulation using Sequencing and Decomposition Algorithms for Collaborative MDAO," Master's thesis, TU Delft, Delft, The Netherlands, 2019. URL http://resolver.tudelft.nl/uuid%3A7d402ca8-2f9e-41e4-abaa-ff0e600fbc14.

[14] Torenbeek, E., *Synthesis of subsonic airplane design: an introduction to the preliminary design of subsonic general aviation and transport aircraft, with emphasis on layout, aerodynamic design, propulsion and performance.*, Kluwer, Dordrecht, The Netherlands, 1982. URL http://resolver.tudelft.nl/uuid:229f2817-9be9-49b6-959a-d653b5bac054.

[15] Raymer, D. P., *Aircraft Design: a conceptual approach*, 2nd ed., AIAA, Washington, District of Columbia, 1992. URL https://arc.aiaa.org/doi/book/10.2514/4.104909.

[16] Obert, E., Slingerland, R., Leusink, D. J. W., van den Berg, T., Koning, J. H., and van Tooren, M. J. L., *Aerodynamic design of transport aircraft*, IOS Press, Amsterdam, The Netherlands, 2009. URL https://doi.org/10.3233/978-1-58603-970-7-i.

[17] Glizde, N., "Wing and Engine Sizing by Using the Matching Plot Technique," *Transport and Aerospace Engineering*, Vol. 5, 2017, pp. 48–59. URL https://doi.org/10.1515/tae-2017-0018.

[18] Elham, A., La Rocca, G., and van Tooren, M. J. L., "Development and implementation of an advanced, design-sensitive method for wing weight estimation," *Aerospace Science and Technology*, Vol. 29, No. 1, 2013, pp. 100–113. URL https://doi.org/10.1016/j.ast.2013.01.012.

[19] McCabe, T., "A Complexity Measure," *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 4, 1976, pp. 308–320. URL https://doi.org/10.1109/TSE.1976.233837.

[20] Gray, J. S., Hwang, J. T., Martins, J. R. R. A., Moore, K. T., and Naylor, B. A., "OpenMDAO: an open-source framework for multidisciplinary design, analysis, and optimization," *Structural and Multidisciplinary Optimization*, Vol. 59, No. 4, 2019, pp. 1075–1104. URL https://doi.org/10.1007/s00158-019-02211-z.

[21] Ast, M., Glas, M., and Roehm, T., "Creating an Ontology for Aircraft Design," *DLRK 2013*, DGLR, Stuttgart, Germany, 2013. URL https://www.dglr.de/publikationen/2014/301356.pdf.

[22] Hoogreef, M. F. M., "Advise, Formalize and Integrate MDO Architectures: A Methodology and Implementation," Ph.D. thesis, TU Delft, Delft, The Netherlands, 2017. URL https://doi.org/10.4233/uuid:cc2af611-6d78-4439-9b10-7e62ae579029.

[23] Glimm, B., Horrocks, I., Motik, B., Stoilos, G., and Wang, Z., "HermiT: An OWL 2 Reasoner," *Journal of Automated Reasoning*, Vol. 53, No. 3, 2014, pp. 245–269. URL https://doi.org/10.1007/s10817-014-9305-1.

[24] Sirin, E., Parsia, B., Grau, B. C., Kalyanpur, A., and Katz, Y., "Pellet: A practical OWL-DL reasoner," *Journal of Web Semantics*, Vol. 5, No. 2, 2007, pp. 51–53. URL https://doi.org/10.1016/j.websem.2007.03.004.

[25] Hermanutz, A., and Hornung, M., "Influence on the Flutter Behavior of Pre-Stressed Wing Structures Under Aerodynamic Loading," *Proceedings of the International Forum on Aeroelasticity and Structural Dynamics*, AIAA, Savannah, Georgia, 2019. URL https://mediatum.ub.tum.de/1507249.

[26] Browning, T. R., "The many views of a process: Toward a process architecture framework for product development processes," *Systems Engineering*, Vol. 12, No. 1, 2009, pp. 69–90. URL https://doi.org/10.1002/sys.20109.

[27] Browning, T. R., "Design Structure Matrix Extensions and Innovations: A Survey and New Opportunities," *IEEE Transactions on Engineering Management*, Vol. 63, No. 1, 2016, pp. 27–52. URL https://doi.org/10.1109/TEM.2015.2491283.

[28] Lukaczyk, T. W., Wendorff, A. D., Colonno, M., Economon, T. D., Alonso, J. J., Orra, T. H., and Ilario, C., "SUAVE: An Open-Source Environment for Multi-Fidelity Conceptual Vehicle Design," *Proceedings of the 16th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, AIAA, Dallas, Texas, 2015. URL https://doi.org/10.2514/6.2015-3087.

[29] Böhnke, D., Rizzi, A., Zhang, M., and Nagel, B., "Towards a Collaborative and Integrated Set of Open Tools for Aircraft Design," *Proceedings of the AIAA Aerospace Sciences Meeting*, AIAA, Grapevine, Texas, 2013. URL https://doi.org/10.2514/6.2013-222.

[30] Jepsen, J., Ciampa, P. D., and Nagel, B., "Avoiding Inconsistencies between Data Models in Collaborative Aircraft Design Processes," *DLRK 2016*, DGLR, Braunschweig, Germany, 2016. URL https://elib.dlr.de/111232/1/Jepsen_DLRK_2016.pdf.

[31] Moerland, E., Langen, T., Nagel, B., Spangenberg, H., Schumann, H., and Zamov, P., "Application of a Distributed MDAO Framework to the Design of a Short- to Medium-Range Aircraft," *DLRK 2012*, DGLR, Berlin, Germany, 2012. URL https://elib.dlr.de/79623/1/281412.pdf.

[32] Pfeiffer, T., Moerland, E., Böhnke, D., Nagel, B., and Gollnick, V., "Aircraft configuration analysis using a low-fidelity, physics based aerospace framework under uncertainty considerations," *Proceedings of the 29th Congress of the International Council of the Aeronautical Sciences*, ICAS, St. Petersburg, Russia, 2014. URL https://www.icas.org/ICAS_ARCHIVE/ICAS2014/data/papers/2014_0750_paper.pdf.

[33] Gent, I. v., Aigner, B., Beijer, B., and Rocca, G. L., "A Critical Look at Design Automation Solutions for Collaborative MDO in the AGILE Paradigm," *Proceedings of the Multidisciplinary Analysis and Optimization Conference*, AIAA, Atlanta, Georgia, 2018. URL https://doi.org/10.2514/6.2018-3251.

[34] Risse, K., Anton, E., Lammering, T., Franz, K., and Hoernschemeyer, R., "An Integrated Environment for Preliminary Aircraft Design and Optimization," *Proceedings of the 53rd AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics and Materials Conference*, AIAA, Honolulu, Hawaii, 2012. URL https://doi.org/10.2514/6.2012-1675.

[35] Boden, B., Flink, J., Först, N., Mischke, R., Schaffert, K., Weinert, A., Wohlan, A., and Schreiber, A., "RCE: An Integration Environment for Engineering and Science," *SoftwareX*, Vol. 15, 2021. URL https://doi.org/10.1016/j.softx.2021.100759.

[36] Ziemer, S., Glas, M., and Stenz, G., "A Conceptual Design Tool for Multi-Disciplinary Aircraft Design," *2011 Aerospace Conference*, IEEE, Big Sky, Montana, 2011. URL https://doi.org/10.1109/AERO.2011.5747531.

[37] Danis, R. A., Green, M. W., Freeman, J. L., and Hall, D. W., "Examining the Conceptual Design Process for Future Hybrid-Electric Rotorcraft," Contractor Report NASA/CR-2018-219897, NASA Ames Research Center, Moffett Field, California, 2018. URL https://ntrs.nasa.gov/citations/20180003214.

[38] Mattingly, J. D., Heiser, W. H., and Pratt, D. T., *Aircraft Engine Design*, Vol. 2, AIAA, Reston, Virginia, 2002. URL https://arc.aiaa.org/doi/book/10.2514/4.861444.

[39] Lefebvre, T., Schmollgruber, P., Blondeau, C., and Carrier, G., "Aircraft Conceptual Design In A Multi-Level, Multi-Fidelity, Multi-Disciplinary Optimization Process," *Proceedings of the 28th International Congress of the Aeronautical Sciences*, ICAS, Brisbane, Australia, 2012. URL http://www.icas.org/ICAS_ARCHIVE/ICAS2012/PAPERS/042.PDF.

[40] Raymer, D., "A computer-aided aircraft Configuration Development System," *Proceedings of the AIAA Aerospace Sciences Meeting*, AIAA, New Orleans, Louisiana, 1979. URL https://doi.org/10.2514/6.1979-64.

[41] Sliwa, S. M. A., "OPDOT: A computer program for the optimum preliminary design of a transport airplane," Technical Memorandum NASA/TM-81857, NASA Langley Research Center, Hampton. Virginia, 1980. URL https://ntrs.nasa.gov/citations/19830002844.

[42] Bil, C., "ADAS - A design system for aircraft configuration development," *Proceedings of the Aircraft Design and Operations Meeting*, AIAA, Seattle, Washington, 1989. URL https://doi.org/10.2514/6.1989-2131.

[43] McCullers, L. A., "Aircraft configuration optimization including optimized flight profiles," *Recent Experiences in Multidisciplinary Analysis and Optimization, Part 1*, NASA Langley Research Center, Hampton, Virginia, 1984. URL https://ntrs.nasa.gov/citations/19870002310.

[44] Li, Y., "A Parametric Approach to Preliminary Design for Aircraft and Spacecraft Configuration," *Proceedings of the 18th Congress of the International Council of the Aeronautical Sciences*, ICAS, Beijing, China, 1992. URL https://www.icas.org/ICAS_ARCHIVE/ICAS1992/ICAS-92-7.2.1.pdf.

[45] Jayaram, S. M., "ACSYNT - A standards-based system for parametric, computer aided conceptual design of aircraft," *Proceedings of the Aerospace Design Conference*, AIAA, Irvine, California, 1992. URL https://ntrs.nasa.gov/citations/19920050721.

[46] Raymer, D., "RDS - A PC-based aircraft design, sizing, and performance system," *Proceedings of the Guidance, Navigation and Control Conference*, AIAA, Hilton Head Island, South Carolina, 1992. URL https://doi.org/10.2514/6.1992-4226.

[47] Rentema, D. W. E., "AIDA: Artificial Intelligence supported conceptual Design of Aircraft," Ph.D. thesis, TU Delft, Delft, The Netherlands, 2004. URL http://resolver.tudelft.nl/uuid%3Aef473d71-e384-4f2f-b9c2-881eb2fb9918.

[48] Salavin, L., "Structure and function of the aircraft design program PrADO," Tech. rep., Hamburg University of Applied Science, Hamburg, Germany, 2008. URL https://www.fzt.haw-hamburg.de/pers/Scholz/arbeiten/TextSalavin.pdf.

[49] Zhang, M., Rizzi, A. W., Nicolosi, F., and De Marco, A., "Collaborative Aircraft Design Methodology using ADAS Linked to CEASIOM," *Proceedings of the 32nd AIAA Applied Aerodynamics Conference*, AIAA, Atlanta, Georgia, 2014. URL https://doi.org/10.2514/6.2014-2012.

[50] Morino, L., Bernardini, G., and Mastroddi, F., "Multi-Disciplinary Optimization for the Conceptual Design of Innovative Aircraft Configurations," *Computer Modeling in Engineering & Sciences*, Vol. 13, No. 1, 2006, pp. 1–18. URL https://doi.org/10.3970/cmes.2006.013.001.

[51] Kirby, D. M. R., and Mavris, D. D. N., "The Environmental Design Space," *Proceedings of the 26th Congress of International Council of the Aeronautical Sciences*, ICAS, Anchorage, Alaska, 2008. URL https://www.icas.org/ICAS_ARCHIVE/ICAS2008/PAPERS/586.PDF.

[52] Liu, H., Wu, Z., Wang, G.-l., and Wang, X.-l., "Implementation of a Sketch Based Approach to Conceptual Aircraft Design Synthesis and Modeling," *Chinese Journal of Aeronautics*, Vol. 17, No. 4, 2004, pp. 207–214. URL https://doi.org/10.1016/S1000-9361(11)60238-0.

[53] Cassidy, P., Gatzke, T., and Vaporean, C., "Integrating Synthesis and Simulation for Conceptual Design," *Proceedings of the AIAA Aerospace Sciences Meeting*, AIAA, Reno, Nevada, 2008. URL https://doi.org/10.2514/6.2008-1443.

[54] Afsar, R., and Salam, A., "CEASIOM: An Open Source Multi Module Conceptual Aircraft Design Tool," *International Journal of Engineering Research*, Vol. 2, No. 7, 2013. URL https://www.ijert.org/ceasiom-an-open-source-multi-module-conceptual-aircraft-design-tool.

[55] Feng, H., Luo, M., Liu, H., and Wu, Z., "A Knowledge-based and Extensible Aircraft Conceptual Design Environment," *Chinese Journal of Aeronautics*, Vol. 24, No. 6, 2011, pp. 709–719. URL https://doi.org/10.1016/S1000-9361(11)60083-6.

[56] Seeckt, K., and Scholz, D., "Application of the Aircraft Preliminary Sizing Tool PreSTo to kerosene and liquid hydrogen fueled regional freighter aircraft," *DLRK 2010*, DGLR, Hamburg, Germany, 2010. URL https://www.fzt.haw-hamburg.de/pers/Scholz/PreSTo/PreSTo_PUB_DLRK_10-08-31.pdf.

[57] Greitzer, E. M., Bonnefoy, P. A., Hall, D. K., Hansman, R. J., Hileman, J. I., Liebeck, R. H., Lovegren, J., Mody, P., Pertuze, J. A., Sato, S., Spakovszky, Z. S., Tan, C. S., Hollman, J. S., Duda, J. E., Fitzgerald, N., Houghton, J., Kerrebrock, J. L., Kiwada, G. F., Kordonowy, D., Parrish, J. C., Tylko, J., and Wen, E. A., "N+3 Aircraft Concept Designs and Trade Studies, Final Report," Contractor Report NASA/CR-2010-216794/VOL2, NASA Glenn Research Center, Cleveland, Ohio, 2010. URL https://ntrs.nasa.gov/citations/20100042398.

[58] Böhnke, D., Nagel, B., and Gollnick, V., "An approach to multi-fidelity in conceptual aircraft design in distributed design environments," *2011 Aerospace Conference*, IEEE, Big Sky, Montana, 2011. URL https://doi.org/10.1109/AERO.2011.5747542.

[59] Elmendorp, R. J. M., "Synthesis of Novel Aircraft Concepts for Future Air Travel," Master's thesis, TU Delft, Delft, The Netherlands, 2014. URL http://resolver.tudelft.nl/uuid%3A1e1941d0-2171-4bc4-95b8-7e01b9c8425d.

[60] Marco, A. D., Nicolosi, F., Vecchia, P., and Cusati, V., "A Java Toolchain of Programs for Aircraft Design," *Proceedings of the 6th CEAS Air and Space Conference*, CEAS, Bucharest, Romania, 2017. URL https://core.ac.uk/download/pdf/148706906.pdf.

[61] Welstead, J. R., Caldwell, D., Condotta, R., and Monroe, N., "An Overview of the Layered and Extensible Aircraft Performance System (LEAPS) Development," *Proceedings of the AIAA Aerospace Sciences Meeting*, AIAA, Kissimmee, Florida, 2018. URL https://doi.org/10.2514/6.2018-1754.

[62] Munjulury, R. C., Staack, I., Berry, P., and Krus, P., "A knowledge-based integrated aircraft conceptual design framework," *CEAS Aeronautical Journal*, Vol. 7, No. 1, 2016, pp. 95–105. URL https://doi.org/10.1007/s13272-015-0174-z.

[63] Kirschen, P. G., York, M. A., Ozturk, B., and Hoburg, W. W., "Application of Signomial Programming to Aircraft Design," *Journal of Aircraft*, Vol. 55, No. 3, 2018, pp. 965–987. URL https://doi.org/10.2514/1.C034378.

[64] Vegh, J. M., Botero, E., Clarke, M., Smart, J., and Alonso, J., "Current Capabilities and Challenges of NDARC and SUAVE for eVTOL Aircraft Design and Analysis," *AIAA Propulsion and Energy 2019 Forum*, AIAA, Indianapolis, Indiana, 2019. URL https://doi.org/10.2514/6.2019-4505.

[65] Schouten, T., Hoogreef, M., and Vos, R., "Effect of Propeller Installation on Performance Indicators of Regional Turboprop Aircraft," *Proceedings of the AIAA Scitech 2019 Forum*, AIAA, San Diego, California, 2019. URL https://doi.org/10.2514/6.2019-1306.

[66] Wells, D. P., Horvath, B. L., and McCullers, L. A., "The Flight Optimization System Weights Estimation Method," Technical Memorandum NASA/TM–2017–219627/VOL1, NASA Langley Research Center, Hampton, Virginia, 2017. URL https://ntrs.nasa.gov/citations/20170005851.

[67] Raymer, D. P., "RDSwin: Seamlessly-Integrated Aircraft Conceptual Design for Students & Professionals," *Proceedings of the AIAA Aerospace Sciences Meeting*, AIAA, San Diego, California, 2016. URL https://doi.org/10.2514/6.2016-1277.

[68] Moran, P. J., "Developing An Open Source Option for NASA Software," Technical Report NAS-03-009, NASA Ames Research Center, Moffett Field, California, 2003. URL https://ntrs.nasa.gov/citations/20030054432.

[69] La Rocca, G., "Knowledge based engineering techniques to support aircraft design and optimization," Ph.D. thesis, TU Delft, Delft, The Netherlands, 2011. URL http://resolver.tudelft.nl/uuid:45ed17b3-4743-4adc-bd65-65dd203e4a09.

[70] Drela, M., "Development of the D8 Transport Configuration," *Proceedings of the 29th AIAA Applied Aerodynamics Conference*, AIAA, Honolulu, Hawaii, 2011. URL https://doi.org/10.2514/6.2011-3970.

[71] Choi, S., Alonso, J. J., and Kroo, I. M., "Two-Level Multifidelity Design Optimization Studies for Supersonic Jets," *Journal of Aircraft*, Vol. 46, No. 3, 2009, pp. 776–790. URL https://doi.org/10.2514/1.34362.

[72] Decyk, V. K., Norton, C. D., and Gardner, H. J., "Why Fortran?" *Computing in Science & Engineering*, Vol. 9, No. 4, 2007, pp. 68–71. URL https://doi.org/10.1109/MCSE.2007.89.

[73] Cai, X., Langtangen, H. P., and Moe, H., "On the Performance of the Python Programming Language for Serial and Parallel Scientific Computations," *Scientific Programming*, Vol. 13, No. 1, 2005, pp. 31–56. URL https://doi.org/10.1155/2005/619804.

[74] La Rocca, G., Langen, T. H. M., and Brouwers, Y. H. A., "The design and engineering engine. Towards a modular system for collaborative aircraft design," *Proceedings of the 28th International Congress of the Aeronautical Sciences*, ICAS, Brisbane, Australia, 2012. URL https://www.icas.org/ICAS_ARCHIVE/ICAS2012/PAPERS/603.PDF.

[75] van Tooren, M., Nawijn, M., Berends, J., and Schut, J., "Aircraft Design Support using Knowledge Engineering and Optimisation Techniques," *Proceedings of the 46th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics and Materials Conference*, AIAA, Austin, Texas, 2005. URL https://doi.org/10.2514/6.2005-2205.

[76] Schut, E. J., and van Tooren, M. J. L., "Design "Feasilization" Using Knowledge-Based Engineering and Optimization Techniques," *Journal of Aircraft*, Vol. 44, No. 6, 2007, pp. 1776–1786. URL https://doi.org/10.2514/1.24688.

[77] Schut, E. J., "Conceptual Design Automation: Abstraction complexity reduction by feasilisation and knowledge engineering," Ph.D. thesis, TU Delft, Delft, The Netherlands, 2010. URL http://resolver.tudelft.nl/uuid%3A94fd1664-9fdc-4868-b2c6-d977c0fbc2e2.

[78] Zijp, S. O. L., "Development of a Life Cycle Cost Model for Conventional and Unconventional Aircraft," Master's thesis, TU Delft, Delft, The Netherlands, 2014. URL http://resolver.tudelft.nl/uuid%3Ade09f6fc-3bc0-4aa8-8d33-dbe3a89ef4b3.

[79] Hettema, A. P., "Vertical Tail Design: Development of a rapid aerodynamic analysis method," Master's thesis, TU Delft, Delft, The Netherlands, 2015. URL http://resolver.tudelft.nl/uuid%3A011a25b6-5b0b-40a3-873c-ab0026e9a45b.

[80] Mutluay, T., "The Development of an Inertia Estimation Method to Support Handling Quality Assessment," Master's thesis, TU Delft, Delft, The Netherlands, 2015. URL http://resolver.tudelft.nl/uuid%3A10ee1352-2f45-46b5-babc-6ea7d3579995.

[81] Van den Dungen, N. H. M., "Synthesis of an Aircraft Featuring a Ducted-Fan Propulsive Empennage," Master's thesis, TU Delft, Delft, The Netherlands, 2017. URL http://resolver.tudelft.nl/uuid%3A82986b0c-2b29-462f-8c6f-746236259ea3.

[82] Mulder, H. A., "Modular Initiator Modelling of Engines," Master's thesis, TU Delft, Delft, The Netherlands, 2018. URL http://resolver.tudelft.nl/uuid:a1129eb8-87bb-47b1-9df0-9704ef5f5284.

[83] La Rocca, G., and Li, M., "Conceptual design of a passenger aircraft for aerial refueling operations," *Proceedings of the 29th Congress of the International Council of the Aeronautical Sciences*, St. Petersburg, Russia, 2014. URL https://www.icas.org/ICAS_ARCHIVE/ICAS2014/data/papers/2014_0425_paper.pdf.

[84] De Smedt, S. D., "Knowledge-Based Engineering Approach to the Finite Element Analysis of Fuselage Structures," Master's thesis, TU Delft, Delft, The Netherlands, 2014. URL http://resolver.tudelft.nl/uuid%3A1e0cf654-70db-43bd-aad0-13cb2aa901da.

[85] Van Haver, S., and Vos, R., "A Practical Method for Uncertainty Analysis in the Aircraft Conceptual Design Phase," *Proceedings of the AIAA Aerospace Sciences Meeting*, AIAA, Kissimmee, Florida, 2015. URL https://doi.org/10.2514/6.2015-1680.

[86] Sol, M. B., "Conceptual Design of Swept Wing Root Aerofoils," Master's thesis, TU Delft, Delft, The Netherlands, 2015. URL http://resolver.tudelft.nl/uuid:5bf482f6-5910-4346-abd5-785748980bc9.

[87] Ramakers, M. a. Y., "Accelerating aircraft design using automated process generation: An experimental architecture for aircraft design software," Master's thesis, TU Delft, Delft, The Netherlands, 2015. URL http://resolver.tudelft.nl/uuid%3A1cdca7a1-a440-4dce-9fce-859fd9701839.

[88] Van Keymeulen, Q. P. D., "Design of a modular fuselage for commercial aircraft: To cope with seasonal variation in passenger demand," Master's thesis, TU Delft, Delft, The Netherlands, 2015. URL http://resolver.tudelft.nl/uuid%3A3f319e5c-437f-4b17-88fc-bdf8bc21e838.

[89] Vargas Jimenez, J. A., "Development of a Wave Drag Prediction Tool for the Conceptual Design Phase," Master's thesis, TU Delft, Delft, The Netherlands, 2015. URL http://resolver.tudelft.nl/uuid%3Afadcbb6a-67f8-419b-9e4f-a546866c738a.

[90] Higgs, T. a. C., "Investigation into the effects of advanced technologies on overall aircraft performance in a collaborative design environment," Master's thesis, TU Delft, Delft, The Netherlands, 2015. URL http://resolver.tudelft.nl/uuid%3A155a86db-60c3-45ac-99fd-0f6e9cf7d65f.

[91] De Smedt, S., and Vos, R., "Knowledge-Based Engineering Approach to the Finite Element Analysis of the Oval Fuselage Concept," *53rd AIAA Aerospace Sciences Meeting*, AIAA, Kissimmee, Florida, 2015. URL https://doi.org/10.2514/6.2015-1899.

[92] Jansen, Q. J. M., "Relaxed Static Stability Performance Assessment on Conventional and Unconventional Aircraft Configurations," Master's thesis, TU Delft, Delft, The Netherlands, 2015. URL http://resolver.tudelft.nl/uuid%3Ad0450f45-f383-411e-b455-1eda6ace08e8.

[93] Jansen, Q., and Vos, R., "Assessing the Effect of Decreased Longitudinal Stability on Aircraft Size and Performance," *Proceedings of the AIAA Aerospace Sciences Meeting*, AIAA, San Diego, California, 2016. URL https://doi.org/10.2514/6.2016-1281.

[94] Boersma, J. Y., "Business Jet Design Using Laminar Flow," Master's thesis, TU Delft, Delft, The Netherlands, 2016. URL http://resolver.tudelft.nl/uuid:4d1818ec-5842-47b0-b144-173aa77a3803.

[95] Bertels, F. G. A., Dijk, R. v., Elmendorp, R., and Vos, R., "Impact of pulsed jet actuators on aircraft mass and fuel consumption," *CEAS Aeronautical Journal*, Vol. 7, No. 4, 2016, pp. 535–549. URL https://doi.org/10.1007/s13272-016-0201-8.

[96] Decloedt, D., "Investigation into the effect of relaxed static stability on a business jet's preliminary design," Master's thesis, TU Delft, Delft, The Netherlands, 2016. URL http://resolver.tudelft.nl/uuid%3A1e3a106b-236d-4e9c-8153-dc707a0fdc59.

[97] Brown, M. T. H., "Conceptual Design of Blended Wing Body Airliners," Master's thesis, TU Delft, Delft, The Netherlands, 2017. URL http://resolver.tudelft.nl/uuid:6f66cd83-673c-4a20-ae5f-c3ea1b7ce3c3.

[98] Li, M., "Conceptual Design Study for In-flight Refueling of Passenger Aircraft," Ph.D. thesis, TU Delft, Delft, The Netherlands, 2017. URL http://resolver.tudelft.nl/uuid%3A5657a63d-1549-4080-8805-a122679cb707.

[99] Jansen, R. A. J., "Conceptual Design Study of a Hydrogen Powered Ultra Large Cargo Aircraft," Master's thesis, TU Delft, Delft, The Netherlands, 2017. URL http://resolver.tudelft.nl/uuid:e38233ed-91a3-46cc-8376-37fa7c2d8d21.

[100] Cosenza, D., and Vos, R., "Handling Qualities Optimization in Aircraft Conceptual Design," *Proceedings of the 17th AIAA Aviation Technology, Integration, and Operations Conference*, AIAA, Denver, Colorado, 2017. URL https://doi.org/10.2514/6.2017-3763.

[101] Bouquet, T., and Vos, R., "Modeling the Propeller Slipstream Effect on Lift and Pitching Moment," *Proceedings of the AIAA Aerospace Sciences Meeting*, AIAA, Grapevine, Texas, 2017. URL https://doi.org/10.2514/6.2017-0236.

[102] Rousseau, R. N. J., "Semi-Analytical Closed-Wing Weight Estimation during Conceptual Design," Master's thesis, TU Delft, Delft, The Netherlands, 2017. URL http://resolver.tudelft.nl/uuid:948ab573-cea8-43a7-af94-40a38994016b.

[103] Voskuijl, M., van Bogaert, J., and Rao, A. G., "Analysis and design of hybrid electric regional turboprop aircraft," *CEAS Aeronautical Journal*, Vol. 9, No. 1, 2018, pp. 15–25. URL https://doi.org/10.1007/s13272-017-0272-1.

[104] Schouten, T., "Assessment of Conceptual High-Capacity Regional Turbopropeller Aircraft," Master's thesis, TU Delft, Delft, The Netherlands, 2018. URL http://resolver.tudelft.nl/uuid%3A236ad212-eb0b-443d-a40d-602ec6fe64f9.

[105] Vos, R., and Hoogreef, M. F. M., "System-level assessment of tail-mounted propellers for regional aircraft," *Proceedings of the 31st Congress of the International Council of the Aeronautical Sciences*, ICAS, Belo Horizonte, Brazil, 2018. URL http://resolver.tudelft.nl/uuid:a1674c88-2df1-4365-8c3a-efde47de7f8c.

[106] Mancini, A., "The Effect of Maneuver Load Alleviation Strategies on Aircraft Performance Indicators," Master's thesis, TU Delft, Delft, The Netherlands, 2018. URL http://resolver.tudelft.nl/uuid%3A4ae25b51-34aa-4028-b3d5-98839c3599be.

[107] Peerlings, B., "Holistically improving screening decisions under uncertainty in aircraft conceptual design and technology assessment: Insights on bottom-up uncertainty quantification and propagation and integrated socio-technical group decision making," Master's thesis, TU Delft, Delft, The Netherlands, 2019. URL http://resolver.tudelft.nl/uuid%3Ad14b5f60-7000-439b-8785-62e513902bf8.

[108] van Oene, N., "Landing Gear Design Integration for the TU Delft Initiator," Master's thesis, TU Delft, Delft, The Netherlands, 2019. URL http://resolver.tudelft.nl/uuid%3Ae08c31c2-1371-465d-a5bc-666433945249.

[109] Mancini, A., and Vos, R., "The Effect of Maneuver Load Alleviation Strategies on Aircraft Performance Indicators," *AIAA Aviation 2019 Forum*, AIAA, Dallas, Texas, 2019. URL https://doi.org/10.2514/6.2019-3272.

[110] Ciampa, P. D., and Nagel, B., "Preliminary Design for Flexible Aircraft in a Collaborative Environment," *Proceedings of the 4th CEAS Air & Space Conference*, CEAS, Linkoping, Sweden, 2013. URL https://elib.dlr.de/95187/.

[111] Brandt, S. A., Post, M., Hall, D. W., Gilliam, F., Jung, T., and Yechout, T. R., "The Value of Semi-Empirical Analysis Models in Aircraft Design," *Proceedings of the 16th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, AIAA, Dallas, Texas, 2015. URL https://doi.org/10.2514/6.2015-2486.

[112] Gu, X., Ciampa, P. D., and Nagel, B., "An automated CFD analysis workflow in overall aircraft design applications," *CEAS Aeronautical Journal*, Vol. 9, No. 1, 2018, pp. 3–13. URL https://doi.org/10.1007/s13272-017-0264-1.

[113] Ciampa, P. D., Zill, T., and Nagel, B., "A Hierarchical Aeroelastic Engine for the Preliminary Design and Optimization of the Flexible Aircraft," *Proceedings 54th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*, AIAA, Boston, Massachusetts, 2013. URL https://doi.org/10.2514/6.2013-1820.

A

Background on ADSs

A multitude of ADSs have been developed since the 1980s at public research institutes, universities, and private companies. Some of the ADSs are already considered as legacy while others are recognized as state-of-the-art. This appendix serves inventorial and inspirational purposes.

A.1 Overview

More than 35 ADSs have been discovered during this research project. Some of them (e.g. *Piano* and *ADP*) are well known, while others (e.g. *ADAS (D)* and *ACODE*) are barely recognized in literature [9]. In figure A.1 all discovered ADSs and their initiation years[1] are shown. Since all ADSs are rather similar, individual descriptions are spared out in this report. The interested reader is referred to [1, 3, 9] for particular descriptions of the ADSs.

It should be noted that figure A.1 is not exhaustive because some ADSs are not available to the public domain and descriptions of those systems can hardly be found in literature. More specifically, the author of this report is aware of commercially used systems from aircraft original equipment manufacturers that do not appear in literature at all. Additionally, there are ADSs that are solely developed and used by individuals and thus are not featured in literature either.

The discovered ADSs are all tools that are used specifically for conceptual and preliminary aircraft design. They are sometimes used within more general frameworks such as *RCE* [35] or *CDT* [36]. They are all truly multidisciplinary as well. Related systems that are centered around single disciplines (e.g. *Propulsion Airframe iNTegration for Hybrid Electric Research (PANTHER)* [37] and *Aircraft Engine Design System Analysis Software (AEDsysDP)* [38]) are purposely disregarded here.

The different ADSs have similar working principles: They are all based on some sort of analysis & sizing methods that are wrapped into an (mostly iterative) process flow (sometimes in the form of an optimization loop). For some ADSs the process flow diagrams presented in the accompanying literature are remarkably similar to the one of the Initiator presented in figure B.5 (e.g. see [3] or [39]).

[1] The initiation year of an ADS is defined as the year when its development commenced. If the initiation year is not provided in literature, the year of the first publication mentioning the ADS is used instead.

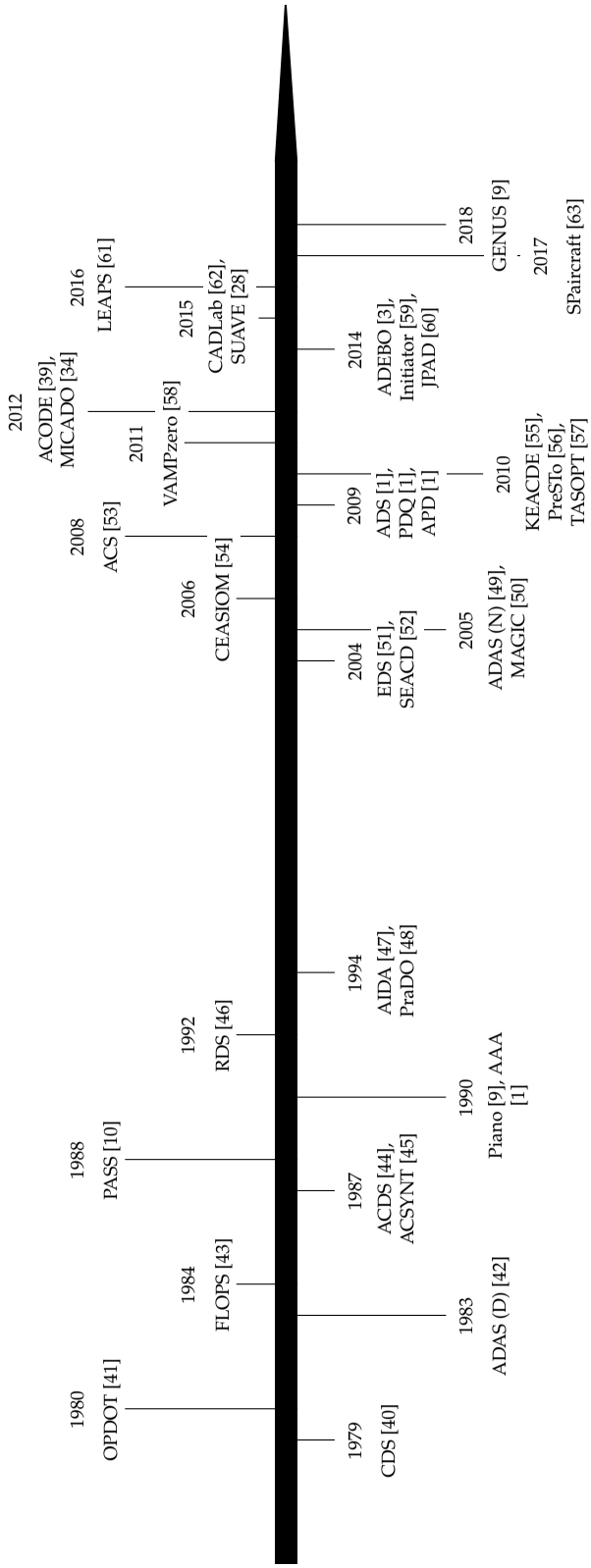


Figure A.1: ADSs initiation timeline

A.2 Comparison

Six substantial aspects of the ADSs are analyzed in the following. Some aspects are treated more elaborately than others, while this does not necessarily mean that these aspects are more important. An overview is presented in table A.1. The aim of this section is to highlight the differences between the various ADSs.

A.2.1 Initiation & lifespan

From figure A.1 it can be seen that development of ADSs started in the 1970s. Especially in the 1980s the development of many ADSs was initiated, but in the 1990s the number of new developments stagnated a little. Since 2004 a new ADS is emerging almost every year. This clearly shows that there is indeed a lot of interest in such systems.

Some ADSs (e.g. the *Initiator* and *SUAVE*) are continuously used in research projects and frequently mentioned in literature (see e.g. [64, 65]). However, most ADSs are only cited infrequently and can therefore be assumed to be deprecated by now (e.g. *CDS*, *ACSYNT*, *AIDA*, *MAGIC*). Some more recent ADSs are designated as successors of older ones (e.g. *ACS* is the successor of *ACSYNT*, and *LEAPS* is intended to become the successor of *FLOPS* [61]).

It can only be speculated on the reasons behind the fact that some ADSs are deprecated earlier than others. It appears that strong ownership helps keeping an ADS at life (e.g. one of the first ADSs, *FLOPS*, received an updated weight documentation in 2017 which indicated that this tool is still being used and valued [66]). Also having a dedicated lead developer helps keeping an ADS up to date (e.g. one ADS that exists since almost 30 years, *RDS*, is still receiving regular updates and is running on modern operating systems [67]). Nonetheless, it seems that all systems have a limited lifespan and new tools are emerging with the advent of programming languages and frameworks [1].

A.2.2 Ownership

Slightly more than fifty percent of the analyzed ADSs originate from an academic environment (i.e. universities). The other half originates from either institutional or commercial entities (i.e. institutes or companies respectively). As outlined above, it is expected that there are more ADSs in institutional and commercial use that are not known to the public. However, the relatively large amount of discovered ADSs is surprising under the assumption that the amount of people dealing with conceptual and early preliminary aircraft design is rather limited [61].

Interestingly, only about one sixth of the systems is open source[2]. This unfortunately hinders collaboration and prevents stimulation of innovation [61, 68]. Sometimes systems are described as open source in the accompanying literature, while the source is not publicly available (e.g. *ADEBO* [3]). Others are not labeled as open source, whereas the source code is publicly available (e.g. the entire code for *OPDOT* is listed in the appendix of the accompanying report [41]). As one might expect, the majority of open source ADSs originates from academic environments (4 from academic entities, 2 from institutional entities, 1 from a commercial entity).

[2]Here open source implies that the code of an ADS was publicly available at the time of writing this report.

Table A.1: Overview of ADSs

Acronym	Name	Design phase	Configurations	Open-source	Programming language	Type	Owner	Initiated	Reference
CDS	Configuration Development System	conceptual	transport aircraft, fighters	no	Fortran	commercial	Rockwell International, Los Angeles	1979	[40]
OPDOT	Optimum Preliminary Design of a Transport Airplane	preliminary	transport aircraft	yes	n/a	institutional	NASA, Hampton	1980	[41]
ADAS (Delft)	Aircraft Design and Analysis System	conceptual	transport aircraft	no	Fortran	academic	TU Delft, Delft	1983	[42]
FLOPS	Flight Optimization System	conceptual	transport aircraft, fighter aircraft, general aviation aircraft, and HWB aircraft	no	Fortran	institutional	NASA, Hampton	1984	[43]
ACDS	Aircraft and spacecraft Configuration Design System	preliminary	transport aircraft, spacecraft	no	n/a	academic	Northwestern Polytechnical University, Xi'an	1987	[44]
ACSYNT	AirCraftSYNThesis	conceptual	transport aircraft	no	Fortran and PHIGS and C	institutional	ACSYNT Institute, Blacksburg	1987	[45]
PASS	Program for Aircraft Synthesis Studies	conceptual, early preliminary	transport aircraft	no	Fortran	academic	Stanford University, Stanford	1988	[10]
AAA	Advanced Aircraft Analysis	conceptual	transport aircraft, unconventional configurations, including VTOL	no	n/a	commercial	DARcorporation, Lawrence	1990	[1]
Piano	Project Interactive Analysis and Optimisation	conceptual, preliminary	transport aircraft	no	Lisp	commercial	Lissys, Woodhouse Eaves	1990	[9]
RDS	Raymer's Design System	conceptual	transport aircraft	no	n/a	commercial	Conceptual Research Corporation, Playa del Rey	1992	[46]
AIDA	Artificial Intelligence supported conceptual Design of Aircraft	conceptual	transport aircraft	no	C and C# and others	academic	TU Delft, Delft	1994	[47]
PraDO	Preliminary aircraft Design and Optimization	preliminary	transport aircraft	no	Fortran and Java	academic	TU Brunswick, Brunswick	1994	[48]
EDS	Environmental Design Space	conceptual	transport aircraft, future conventional aircraft configurations	no	n/a	institutional	FAA, Washington	2004	[51]
SEACD	Synthetic Environment for Aircraft Conceptual Design	conceptual	transport aircraft	no	n/a	academic	Beijing University of Aeronautics and Astronautics, Beijing	2004	[52]
ADAS (Neaples)	Aircraft Design and Analysis Software	conceptual, preliminary	transport aircraft, light aircraft	no	Visual Basic	academic	University of Naples Federico II, Naples	2005	[49]
MAGIC	Multidisciplinary Aircraft desiGn of Innovative Configurations	conceptual	innovative aircraft configurations	no	n/a	academic	Roma Tre University, Rome	2005	[50]
CEASIOM	Computerized Environment for Aircraft Synthesis and Integrated Optimization Methods	conceptual, preliminary	general aviation aircraft, transport aircraft	yes	Matlab or Python	commercial	CFS Engineering, Lausanne	2006	[54]
ACS	AirCraft Synthesis	conceptual, preliminary	transport aircraft, fighter, unmanned aircraft	no	n/a	commercial	AVID, Yorktown	2008	[53]
ADS	Aircraft Design Software	conceptual	aircraft, commuter category aircraft	no	n/a	commercial	Optimal Aircraft Design, Namur	2009	[1]
AirplanePDQ	AirplanePDQ	conceptual	light-sport aircraft, ultralight aircraft, experimental aircraft, general aviation aircraft	no	n/a	commercial	DaVinci Technologies, Aubum	2009	[1]
APD	Aircraft Preliminary Design	conceptual, preliminary	transport aircraft	no	n/a	commercial	Pacelab, Berlin	2009	[1]
KEACDE	Knowledge-based and Extensible Aircraft Conceptual Design Environment	conceptual	transport aircraft	no	n/a	academic	Beihang University, Beijing	2010	[55]
PreSTo	Aircraft Preliminary Sizing Tool	preliminary	transport aircraft	yes	n/a	academic	HAW, Hamburg	2010	[56]
TASOPT	Transport Aircraft System OPTimization		transport aircraft	yes	Fortran	academic	Massachusetts Institute of Technology, Cambridge	2010	[57]
VAMPzero	VAMPzero	conceptual	transport aircraft	yes	Python	institutional	DLR, Hamburg	2011	[58]
ACODE	Airliner COnceptual DEsign	conceptual	transport aircraft	no	n/a	institutional	Onera, Toulouse	2012	[39]
MICADO	Multidisciplinary Integrated Conceptual Aircraft Design and Optimization	conceptual	transport aircraft	no	C++	academic	RWTH Aachen University, Aachen	2012	[34]
ADEBO	Aircraft Design Box	conceptual, early preliminary	fixed-wing aircraft	no	Matlab	academic	TU Munich, Munich	2014	[3]
Initiator	Initiator	conceptual, early preliminary	transport aircaft, business jets, novel configuartions (boxwing, bwb)	no	Matlab	academic	TU Delft, Delft	2014	[59]
JPAD	Java toolchain of Programs for Aircraft Design	preliminary	transport aircraft	partly	Java	academic	University of Naples Federico II, Naples	2014	[60]
CADLab	Conceptual Aircraft Design Laboratory	conceptual, preliminary	transport aircraft, fighters, very-light jets	no	n/a	academic	Linköping University, Linköping	2015	[62]
SUAVE	Stanford University Aerospace Vehicle Environment	conceptual	transport aircraft, unconventional configs, UAVs, eVTOLs, esp. non-TAW configurations	yes	Python	academic	Stanford University, Stanford	2015	[28]
LEAPS	Layered and Extensible Aircraft Performance System	conceptual	advanced aircraft concepts	no	Python	institutional	NASA, Hampton	2016	[61]
GENUS	GENUS Aircraft Conceptual Design environment	conceptual	various, potentially radically different aircraft configuration	no	Java	academic	Cranfield University, Cranfield	2018	[9]
SPaircraft	Signomial Programming Aircraft	conceptual	transport aircraft	yes	Python	academic	Massachusetts Institute of Technology, Cambridge	2017	[63]

A.2.3 Objectives

Primarily, ADSs have been developed to automate (and consequently accelerate) sophisticated aircraft design methods [3]. For example *MICADO* is meant "for automated aircraft design [...] with a minimum of user input, i.e. a set of top-level requirements and specifications" [34] and *GENUS* is intended "to enable the conceptual level design of various [...] aircraft configurations" [9]. But ADSs also have many secondary objectives of which the most distinguished ones are analyzed in the following.

The automation of aircraft design methods naturally frees the user of manual tasks [40]. Thus, the user can perform more creative and innovative tasks which play an important role in the early design phases [69]. Alternatively, the user can perform tasks that were traditionally neglected during the conceptual and preliminary design phases such as the assessment of ecological and economic impacts [34, 51].

All ADSs can principally be used in optimization workflows, but the newer ones (e.g. *TASOPT* and *SUAVE*) are geared to support optimization processes out of the box [70]. In particular, the more recent ADSs (e.g. *MICADO*) allow to optimize a design towards a freely selectable design parameter or a freely selectable combination of design parameter [34] (hence not only towards a previously selected design parameter such as the minimum mass or fuel). Some ADSs can therefore be utilized to perform optimization studies of entire aircraft or of individual aircraft parts rapidly and affordably [40, 70].

Sometimes an ADS also serves as a geometry generator or a geometry database (especially in the case of *ACSYNT* and *CADLab*). This allows visualizing aircraft concepts easily. More importantly, this allows introducing higher fidelity and physics-based tools in the early design phases because these tools often require geometry data as input [40, 45].

Additionally, existing ADSs can be used to access the capability of computer tools in the conceptual and preliminary design phases [42]. Some ADSs have been specifically developed for this purpose (e.g. *ADAS (D)* and *EDS*).

In conclusion, ADS have different objectives and therefore can be used to solve problems of different types. Not all ADSs are equally suited for achieving a particular objective. Therefore, the developers of any ADS need to take into account that there is often more than one objective even though additional objectives might not be specified in the first place.

A.2.4 Design phases

The investigated ADSs are described as conceptual, early preliminary, or preliminary design tools in the accompanying literature. Logically, the conceptual design takes place before the preliminary design. However, this can lead to confusion since the terms are used interchangeably in the accompanying literature. For example, *OPDOT* is described as a tool for preliminary design, while its outputs are less detailed than those of *RDS*, which is described as a tool for conceptual design [41, 46].

One shall also note that it is common practice that the conceptual design of a subsystem (e.g. an aileron) may happen during the detailed design of a system (e.g. an aircraft). The adoption of ADSs accelerates the individual design phases and one tends to include more details earlier on in the design process [69]. This eventually will render the previous classifications of design phases insignificant.

In one design study (on a supersonic jet which was conducted using *PASS*) it was proposed to merge the early design phases and to consider different Fidelity Levels (FLs) instead [71]. Indeed, FLs or Technology Readiness Levels (TRLs) might be future classification means

that have the potential to replace the current unclear definition of design phases. However, yet no universally accepted definition has emerged.

A.2.5 Programming languages

A variety of programming languages are utilized within the different ADSs as can be seen from table A.1. In fact, the languages also provide an indication of the comprehensiveness of the ADSs, which is further explained in the following.

It comes as no surprise that the first ADSs (e.g. *CDS*, *FLOPS*, *PASS*) were almost exclusively written in Fortran, which was one of the most popular high-level languages for scientific computing back in the times [72]. The main advantages of Fortran are its high computational performance and its straightforward syntax for representing and solving mathematical (systems of) equations. These advantages were especially valuable when computational resources were scarce and when mostly simple aircraft configurations were investigated.

Nowadays, both the models and modules used in ADSs tend to be more sophisticated. Therefore, higher-level languages such as Java, Matlab and Python are used more frequently (e.g. in *SUAVE*, *ADEBO*, *GENUS*). The main advantages of those languages are high programming productivity and extensive object-oriented programming features. Their performance deficiencies (when compared to the classical compiled languages such as Fortran and C) do not weigh too much anymore [73].

For about one-third of the investigated ADSs the language is not available. In this case the ADS is only available as precompiled executables (e.g. in case of *AAA*, *RDS*, *ADS*) and/or no accompanying literature specifying a distinct programming language has been published (e.g. in case of *MAGIC*, *ACDS*, *ADP*). Oftentimes, these ADSs offer a graphical user interface and are (thus) rather limited in scope (since modifications of the modules employed in the ADSs is not envisaged).

Sometimes the authors of an ADS explain their choice for the selected programming language in accompanying literature. For example, the decision to use Python was driven by its "combination of object-oriented programming, duck typing, concise language, portability in open source, and community standard as glueware" for *SUAVE* [28]. The decision for utilizing Matlab was made based on its status as "state-of the art in engineering" and the "multiple communication interfaces to other frameworks" that are offered by this language [3]. The *GENUS* architecture relies on the "object-orientation and the use of inheritance and polymorphism" of Java [9].

A.2.6 Programming paradigms

A variety of programming paradigms are utilized within the different ADSs. These patterns always influence the functioning of an ADS. Some relevant programming paradigms are analyzed in the following.

All ADSs rely on disciplines which typically are of sizing type (e.g. engine sizing, wing sizing, fuselage sizing) or analysis type (e.g. mission analysis, aerodynamic analysis, stability analysis, performance analysis). The number of disciplines within an ADS differs substantially, as some ADSs have less than ten generic disciplines (e.g. in *JPAP*, *SEACD*, *ACODE*, *VAMPzero*), while others have more detailed disciplines (e.g. in *SUAVE*, *AAA*). Occasionally, there are disciplines of different fidelity levels (e.g. in *ADAS (D)*) which are sometimes split in class I and II disciplines as presented in classical design textbooks (e.g. in *AAA*) [1].

The ADSs handle design data in different ways. The first systems (e.g. *OPDOT*) only stored data in standard data structures behind plain variables. Later (e.g. in *AAA*, *ADAS (D)*) data was stored in dedicated databases. Now, data is often stored in files, most frequently in the XML format (e.g. in *JDAP*, *VAMPzero*, *MICADO*, *CAESIOM*, *JPAD*), and in objects (e.g. in *SEACD*, *SUAVE*, *ADEBO*). It was recently realized that handling data is a bottleneck for performance [11]. Data is frequently organized hierarchically in a tree structure (see [29, 54] for illustrations) although a tree structure is not always unambiguous.

The disciplines and the design data are always interlinked. Abstractly, this has been described as attribute-method orthogonality [28] or as component-discipline relationship [29]. Effectively, the disciplines amend the design data in an iterative process. One ADS (i.e. *AIDA*) makes use of artificial intelligence [47] during this process. While most ADSs aim for optimality, in case of non-conventional configurations feasibility is oftentimes critical [9].

ADSs offer different user interfaces. For example in literature on ADSs it is stated that programming interfaces accommodate flexibility (see e.g. [9]) whereas graphical interfaces accommodate usability (see e.g. [60]). Some ADSs have a GUI (e.g. *RDS*, *CAESIOM*, *ACSYNT*), some are accessed via simple command line prompts (e.g. *CDS*, *OPDOT*), but most are initialized from a programming environment (e.g. *MICADO*, *SUAVE*). This is not surprising as ADSs are systems that are made by/for expert users and thus they are not supposed to be used by laymen without any programming experience.

Last but not least, it can be observed that most ADSs follow a seemingly object-oriented paradigm. However, only a few modern ones (e.g. *VAMPzero*, *SUAVE*, *MICADO*) respect object-oriented principles such as inheritance and encapsulation. The earlier ones follow a rather procedural paradigm although they contain objects and methods.

A.3 Trends

The first ADSs were conceived in the 1970s. Since then these systems improved gradually but substantially. Some notable trends can be observed when investigating the development of ADSs over time. The objective of this section is to distill these trends. But this section is not just a chronological listing of aspects from the previous two sections. It rather is an evolutionary illustration of conceptual and preliminary aircraft design methods.

One main tendency is that the objectives of ADSs previously have been confined but now are more comprehensive. The early ADSs have been mainly developed "for the initial configuration development of aircraft concepts" [40] and for "aircraft configuration optimization [...] for use in conceptual design of new aircraft and in the assessment of the impact of advanced technology" [43]. But newer ADSs can additionally be used for "parameter variations and optimizations with individually selected free variables and objectives" [34] or for "parametric interactive design" [62] and sometimes are "capable of estimating source noise, exhaust emissions, and performance for potential future aircraft designs under different technological, operational, policy, and market scenarios" [51]. Thus, ADSs appear to serve an increasing number of purposes nowadays (see section A.2.3).

It can also be observed that the fidelity level of ADSs is increasing continuously. Furthermore, ADSs with support for multiple fidelity levels are emerging recently. In particular, it has been shown how to "close a multi-fidelity design loop" [58] and how to incorporate "the right level of fidelity at the right time" [28]. This generally yields more trustworthy designs and prevents misjudgments in the early design phases. Therefore, less redesign efforts are required in subsequent design phases and eventually the overall aircraft development risk

is cut down [2, 74]. However, the fidelity level increase of ADSs also contributes to the confusion that is caused by the inconsistent use of the terms conceptual and preliminary when referring to the early design phases (see section A.2.4).

Another trend (closely related to the previous one) is that the disciplines of ADSs are becoming less based on statistics and instead increasingly based on physics. One of the investigated ADSs has main disciplines (i.e. structures, aerodynamics and aeroelasticity) that are "all first-principles based" [50]. This trend is substantiated because historical correlations, expert opinions, and statistical models are mostly unreliable for assessing innovative aircraft designs as they are all biased towards existing aircraft designs [9, 51, 61]. Somewhat contrary to this trend, ADSs themselves are nowadays used to generate empirical relations and charts (e.g. showing how varying one aircraft design parameters influences other aircraft design parameters).

There is also a tendency that ADSs are increasingly supporting unconventional aircraft configurations. While the early ADSs were mainly used to synthesize and analyze one distinct aircraft configuration, the newer ones are often used to compare several advanced configurations (e.g. hybrid electric aircraft, non-tube-and-wing aircraft, vertical take-off and landing aircraft). It shall be noted that such a comparison can be precarious though when different (i.e. non-generic and non-physics-based) methods are used to evaluate different configurations [9]. Notwithstanding, a strong demand for such comparisons exists, since the aircraft requirements become increasingly stringent or too stringent for conventional aircraft configurations [61].

The last tendency to be discussed is that ADSs are presently written in higher-level programming languages with richer feature sets. This allows for more flexible program structures but can lead to more complicated program code [61] as well. For example one ADS uses "an object-oriented architecture that enables arbitrary aircraft and propulsion system topologies" [28] and another ADSs has a repository that is "capable of containing all kinds of standalone software providing an accessible application interface" [3] Whereas disciplines were previously loosely integrated in procedural code, they are now clearly divided in more object-oriented code. Whereas the disciplines were previously simpler, they are now more abstract and they are accompanied by more code overhead (see sections A.2.5 and A.2.6).

In summary, five major trends have been identified by analyzing the development of ADSs over time. This analysis also illustrates the status quo of the conceptual and preliminary aircraft design methods, but it does not predict the future of these methods.

B

Background on the *Initiator*

The *Initiator* is an ADS which was developed and is used at the section of Flight Performance and Propulsion (FPP) at the faculty of Aerospace Engineering (AE) of the Delft University of Technology (TUD). It has been successfully used in various studies to investigate a wide range of problems related to the conceptual and early preliminary design of passenger transport aircraft. This appendix provides background information on the *Initiator*.

B.1 Development

The development of the *Initiator* commenced roughly 16 years ago. The development phases are illustrated in figure B.1 and are also described in a few words in this section. This shall demonstrate that the development of an ADS is less straightforward than one might imagine.

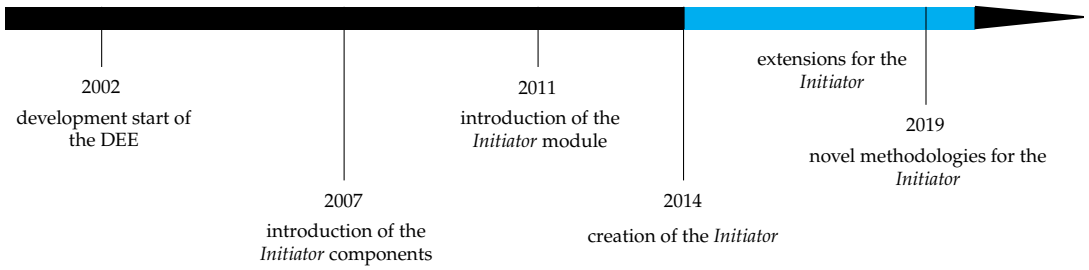


Figure B.1: *Initiator* development phases

Since 2002 the so-called Design and Engineering Engine (DEE) has been actively developed at the TUD [69]. The DEE was originally "defined as an advanced design environment, where the design process of complex products can be supported and accelerated through the automation of non-creative and repetitive design activities" [75].

In 2007 the so-called *Initiator* components were first introduced "for providing feasible starting parameter values for the instantiation of the (parametric) product model" within the DEE [76, 77]. The *Initiator* components were merely intended to provide reasonable estimates of aircraft design parameters satisfying a set of aircraft requirements that could subsequently be used in a multidisciplinary optimization or in an advanced analysis.

In 2011 the so-called *Initiator* module was introduced as a coupled "collection of sizing tools" [69]. Thus, the *Initiator* module had to consider interaction effects between the different *Initiator* components. This eventually made sure that the aircraft design parameter estimates were not only reasonable but also coherent.

The *Initiator* as we know it today was finally created in 2014 as "a tool designed to synthesize a preliminary aircraft design from a set of top-level requirements" [4]. The system was primarily developed to compare different conventional as well as unconventional aircraft configurations. However, it also was a proper implementation of the *Initiator* module as defined previously.

Since 2014 the *Initiator* was regularly updated and extended. In the following some examples are listed:

- a cost estimation method was added in 2014 [78],
- a rapid vertical tail sizing method was added in 2015 [79],
- an inertia estimation method was added in 2015 [80],
- a ducted-fan propulsive empennage sizing method was added in 2017, [81],
- a modular engine design method was added in 2018 [82],
- and a distributed propulsion design method was added in 2019 [7].

More updates and extensions of the *Initiator* are listed in table B.1.

In 2019 an attempt was made to restructure the *Initiator* by reimplementing it using a novel graph-based methodology [11, 13]. The result was a more structured and agile MDAO system with improved modularity. But not all parts of the *Initiator* were considered during the restructuring and the computational efficiency of the novel implementation was far from optimal.

The development phases of the *Initiator* can be summarized by drawing an analogy to a generic aircraft design process. First, the design problem is broken down and reduced in size (the *Initiator* components). Then one realizes that too much simplification is not suitable for solving the aircraft design problem at hand and certain interaction effects need to be considered (the *Initiator* module). Afterwards, the design problem is worked out in detail (the *Initiator*) and the level of complexity increases (updates and extensions for the *Initiator*). Finally, the design problem is updated which oftentimes requires a new methodical approach (novel methodologies for the *Initiator*).

B.2 Objectives

As pointed out earlier, the *Initiator* is an ADS for the conceptual and early preliminary design phases. This definition is rather ambiguous and therefore a more elaborate description of the scope of the *Initiator* is presented in this section.

The most important feature of the *Initiator* probably is its ability to synthesize an aircraft in an almost fully automated manner [11, 59]. The synthesis process often takes only a few minutes on a modern personal computer. Therefore, the *Initiator* can be used to compare many different aircraft configurations rapidly. This has been demonstrated repeatedly (see appendix B.4) and sometimes has been illustrated nicely (see figures B.2 to B.4). In light of this, the *Initiator* can also be described as an aircraft configuration generator.

Another important feature of the *Initiator* is its ability to handle conventional (tube-and-wing) as well as unconventional (canard, three-surface, box-wing, blended-wing-body) aircraft configurations [4, 110]. This distinguishes the *Initiator* from most of the alternative ADSs (see appendix A) which are mainly geared towards conventional aircraft configurations.

It should also be noted that *Initiator* studies are dealing almost exclusively with civil passenger transport aircraft. There are almost no *Initiator* studies dealing with other aircraft categories (e.g. general aviation aircraft, fighter aircraft, vertical-take-off-and-landing aircraft, cargo aircraft, supersonic aircraft). Principally, the *Initiator* can deal

Table B.1: Publications resulting from studies with the *Initiator*

Type	Publication year	Title	Reference
master thesis	2014	Synthesis of Novel Aircraft Concepts for Future Air Travel	[59]
conference paper	2014	A conceptual design and analysis method for conventional and unconventional airplanes	[4]
journal article	2014	Conceptual design of a passenger aircraft for aerial refueling operations	[83]
master thesis	2014	Development of a Life Cycle Cost Model for Conventional and Unconventional Aircraft	[78]
master thesis	2014	Knowledge-Based Engineering Approach to the Finite Element Analysis of Fuselage Structures	[84]
conference paper	2015	A Practical Method for Uncertainty Analysis in the Aircraft Conceptual Design Phase	[85]
master thesis	2015	Conceptual Design of Swept Wing Root Aerofoils	[86]
master thesis	2015	Accelerating aircraft design using automated process generation: An experimental architecture for aircraft design software	[87]
master thesis	2015	Design of a modular fuselage for commercial aircraft: To cope with seasonal variation in passenger demand	[88]
master thesis	2015	Development of a Wave Drag Prediction Tool for the Conceptual Design Phase	[89]
master thesis	2015	Investigation into the effects of advanced technologies on overall aircraft performance in a collaborative design environment	[90]
conference paper	2015	Knowledge-Based Engineering Approach to the Finite Element Analysis of the Oval Fuselage Concept	[91]
master thesis	2015	Relaxed Static Stability Performance Assessment on Conventional and Unconventional Aircraft Configurations	[92]
master thesis	2015	The Development of an Inertia Estimation Method to Support Handling Quality Assessment	[80]
master thesis	2015	Vertical Tail Design: Development of a rapid aerodynamic analysis method	[79]
conference paper	2016	Assessing the Effect of Decreased Longitudinal Stability on Aircraft Size and Performance	[93]
master thesis	2016	Business Jet Design Using Laminar Flow	[94]
master thesis	2016	Conceptual Design of High Subsonic Prandtl Planes	[6]
journal article	2016	Impact of pulsed jet actuators on aircraft mass and fuel consumption	[95]
master thesis	2016	Investigation into the effect of relaxed static stability on a business jet's preliminary design	[96]
master thesis	2017	Conceptual Design of Blended Wing Body Airliners	[97]
doctoral thesis[a]	2017	Conceptual Design Study for In-flight Refueling of Passenger Aircraft	[98]
master thesis[a]	2017	Conceptual Design Study of a Hydrogen Powered Ultra Large Cargo Aircraft	[99]
conference paper	2017	Handling Qualities Optimization in Aircraft Conceptual Design	[100]
conference paper	2017	Modeling the Propeller Slipstream Effect on Lift and Pitching Moment	[101]
master thesis	2017	Semi-Analytical Closed-Wing Weight Estimation during Conceptual Design	[102]
master thesis	2017	Synthesis of an Aircraft Featuring a Ducted-Fan Propulsive Empennage	[81]
journal article	2017	The optimal cruise altitude of LNG-fuelled turbofan aircraft	[8]
journal article	2018	Analysis and design of hybrid electric regional turboprop aircraft	[103]
master thesis	2018	Assessment of Conceptual High-Capacity Regional Turbopropeller Aircraft	[104]
conference paper	2018	Conceptual Design and Evaluation of Blended-Wing Body Aircraft	[5]
master thesis	2018	Modular Initiator Modeling of Engines	[82]
conference paper	2018	System-level assessment of tail-mounted propellers for regional aircraft	[105]
master thesis	2018	The Effect of Maneuver Load Alleviation Strategies on Aircraft Performance Indicators	[106]
conference paper	2019	Conceptual Assessment of Hybrid Electric Aircraft with Distributed Propulsion and Boosted Turbofans	[7]
conference paper	2019	Effect of Propeller Installation on Performance Indicators of Regional Turboprop Aircraft	[65]
master thesis	2019	Holistically improving screening decisions under uncertainty in aircraft conceptual design and technology assessment: Insights on bottom-up uncertainty quantification and propagation and integrated socio-technical group decision making	[107]
master thesis	2019	Landing Gear Design Integration for the TU Delft Initiator	[108]
conference paper	2019	The Effect of Maneuver Load Alleviation Strategies on Aircraft Performance Indicators	[109]
doctoral thesis	2019	Agile MDAO Systems: A Graph-based Methodology to Enhance Collaborative Multidisciplinary Design	[11]
master thesis	2019	Automated Execution Process Formulation using Sequencing and Decomposition Algorithms for Collaborative MDAO	[13]

[a] In this publication the use of the *Initiator* is initially considered but eventually discarded.

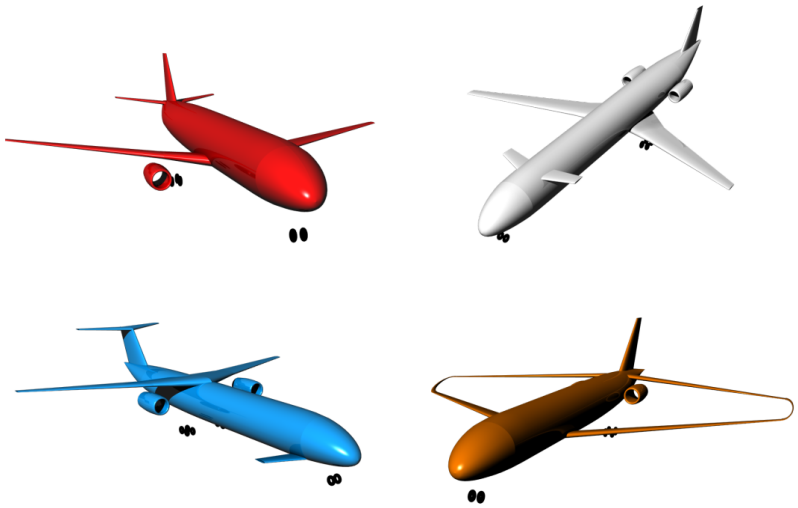


Figure B.2: Example illustration of selected aircraft concepts generated with the *Initiator* [4]

Figure B.3: Example illustration of selected hybrid electric aircraft concepts generated with the *Initiator* [7]

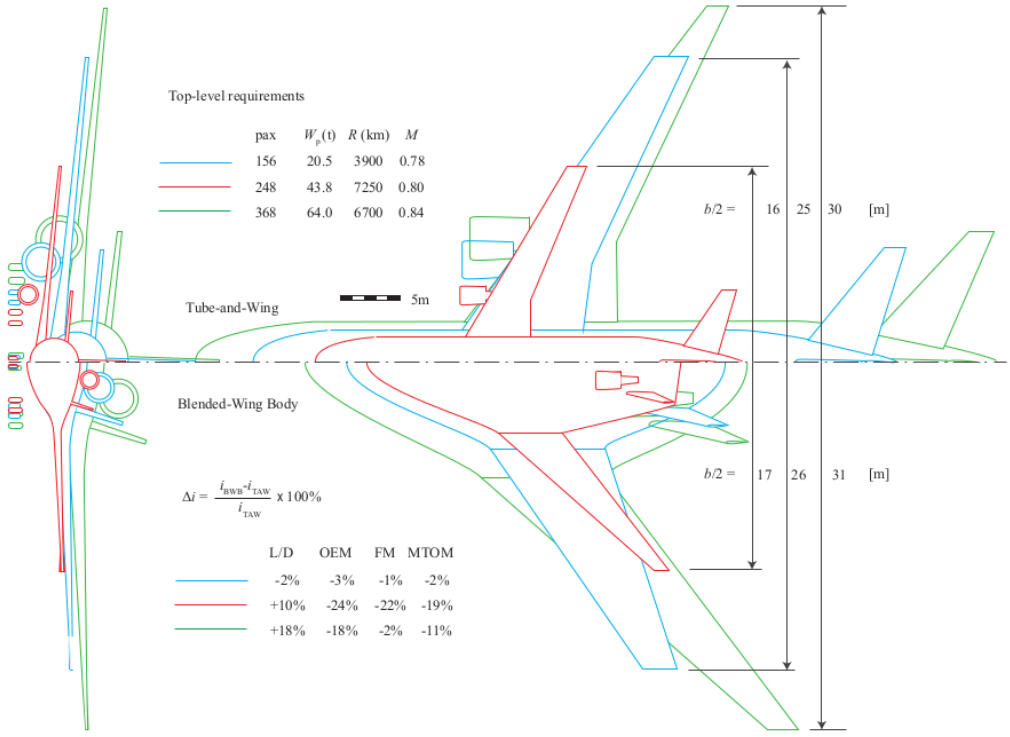


Figure B.4: Example illustration of selected tube-and-wing and blended-wing-body aircraft concepts generated with the *Initiator* [5]

with other aircraft categories too, although this often requires extensive adjustments. For example, the *Initiator* has been used twice to synthesize business jets in the past [94, 96]. But sometimes it is considered more efficient to develop a completely new ADS for other aircraft categories instead of reusing existing systems [98, 99].

From a high-level perspective the *Initiator* inputs are aircraft requirements (e.g. number of passengers, cruise speed, runway length), an aircraft configuration (e.g. tube-and-wing, box-wing, blended-wing-body) and several empirical databases (e.g. of airfoil or engine properties). The *Initiator* outputs are aircraft design parameters such as the aircraft geometry (e.g. span, length, height) and its key performance indicators (e.g. weights, operating costs, emissions) [4]. Therefore, the *Initiator* also allows investigation of the interaction effects caused by small and large changes to the aircraft requirements [7].

The modules used within the *Initiator* are either physics-based or empirical. Both types of modules have their own advantages and disadvantages during the early design phases (see [111–113]). Physics-based modules are generally more accurate than empirical ones while at the same time being more expensive – both with regard to implementation efforts and computational resources. On the other hand, empirical modules are inexpensive but they are also unsuitable for unconventional or novel designs – because empirical data is simply not available for those. Thus, the modules used within the *Initiator* can and need to be aligned with the problem at hand continuously [4].

The aircraft designs obtained with the *Initiator* can be compared with existing aircraft having similar design requirements (but generally different/unknown objectives). The difference in MTOW is typically between 1-10% which is considered meticulous for an ADS [4, 5, 95, 103, 105]. The accuracy of other aircraft design parameters is within a similar order of magnitude (e.g. the span usually is more accurate while the sweep is often overestimated and less accurate). Since the MTOW is (in one way or another) influenced by almost all other aircraft design parameters, it is commonly considered a good indication for the overall accuracy when validating the *Initiator* [4]. Although the aircraft design parameters from the *Initiator* are considered accurate, it is not advised to state definitive numbers when jointly evaluating different aircraft configurations. Instead, relative numbers are commonly used for this purpose [107].

Generally speaking, the *Initiator* is considered a part of the DEE. An interesting aspect is that the *Initiator* is sometimes also described as a DEE on its own [69, 76]. Thus, one can put forward the hypothesis that the *Initiator* and the DEE can principally be used interchangeably. This implies that an *Initiator* with more sophisticated modules can theoretically replace the DEE. Alternatively, a DEE with simplified modules can theoretically replace the *Initiator*. Demonstrating this conjecture is out of scope for now.

Certainly, the *Initiator* is not an all-in-one solution suitable for every aircraft design purpose. It is a powerful tool for aircraft design synthesis that supports its users in translating aircraft requirements to aircraft design parameters. It can also be used to find interaction effects that are hidden or too complex to understand at first sight. However, it is still up to the users to consciously select and balance appropriate aircraft requirements.

B.3 Implementation

In the previous section, the *Initiator* was explained in rather general terms. In this section, a more technical description is given.

The most common process flow diagram of the *Initiator* implementation is depicted in figure B.5. Since the *Initiator* is frequently updated the process flow is adjusted incidentally

as well (see e.g. [4, 7, 74, 104]). In the given diagram the inputs and outputs are visualized by rounded boxes. The modules are characterized by square boxes instead. There are both sizing and pure analysis modules. The modules have different fidelity levels as can be deduced directly from the naming of the three weight estimation modules.

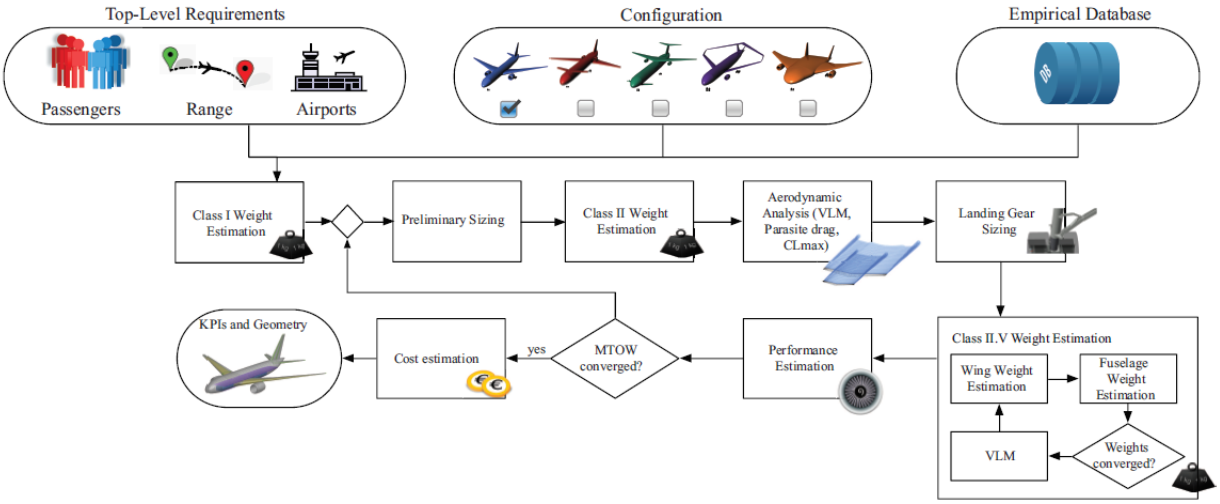


Figure B.5: *Initiator* process flow diagram [4]

Figure B.5 shows that the *Initiator* modules are wrapped into several nested convergence cycles. Indeed, the synthesis process of the *Initiator* can be described as a convergence or feasilization process [7, 105]. The *Initiator* updates the aircraft design parameters in an iterative manner until a predefined subset of those parameters (usually only the MTOW as indicated in the figure above) converge below a certain threshold while still satisfying all aircraft requirements [7, 105].

Although the *Initiator* is written in Matlab some of its modules make use of executables (e.g. *XFOIL*, *AVL*, *DATCOM*) that were developed in different programming languages. This is unproblematic as long as the executables are treated as black-boxes. The *Initiator* modules themselves are often treated as black-boxes as well.

B.4 Use cases

A multitude of problems related to aircraft design have been investigated with the *Initiator*. The results were documented in journal articles, conference proceedings, and (master/-doctoral) theses. This section presents a selection of the problems studied with the *Initiator* (in chronological order). The intention is to convey a better impression of the problem types that can be solved with the *Initiator*. When provided in literature, specific *Initiator* implementation details are also presented here.

For this study a survey was conducted which revealed around 40 of the above-mentioned documents were published between 2014 and 2019. A little more than half of them are theses while the other half are academic papers, which sometimes originate from the former. For an overview, the interested reader is referred to table B.1.

B.4.1 Impact of pulsed jet actuators on aircraft mass and fuel consumption (2016)

In 2016 the impact of active flow control on aircraft mass and fuel consumption was studied [95]. In fact, a system was conceived where the pulsed jet actuators were integrated into the flaps of an aircraft.

It was shown that the installation of pulsed jet actuators can lead to a decrease in OEW and MTOW when the actuators are capable of increasing the maximum lift coefficient by a certain amount. The weight decrease is mainly due to a smaller wing. It was also demonstrated that the observed overall weight decrease does not necessarily lead to a reduced fuel consumption because powering the actuators also requires some energy.

For this study three different aircraft were synthesized with the *Initiator*: one conventional aircraft and two modified aircraft with slightly different pulsed jet actuator systems. There was a distinct difference between the conventional and modified aircraft from which the conclusions presented above could be derived. There were only insignificant differences between the aircraft with slightly different pulsed jet actuator systems though. This was attributed to the relatively crude convergence criteria used within the *Initiator* (i.e. 1 % for the MTOW and 0.2% for mission range).

For this study the *Initiator* was only slightly modified: two additional design modules were integrated into the existing weight estimation modules. The additional modules could model the pulsed jet actuator system and their effects on aircraft design parameters. Some of the code in the additional modules was relatively inefficient (e.g. an exhaustive search) but it was still considered appropriate (i.e. the analysis ran approximately three days on a personal computer in total). It was also recommended to include additional cost modules to account for the increased complexity when adding an active flow control system.

B.4.2 Conceptual design and evaluation of blended-wing-body aircraft (2017/ 2018)

In 2017 the modelling and analysis capabilities of the *Initiator* for blended-wing-body aircraft were significantly improved within the scope of a master thesis [97]. A follow-up conference paper about the comparison between conventional and blended-wing-body aircraft was published by the same author in 2018 [5].

The objective was to allow a more trustworthy comparison of conventional aircraft and blended-wing-body aircraft based on the same aircraft requirements and fidelity levels. It was shown that the blended-wing-body aircraft generally feature favorable characteristics such as lower weights (e.g. MTOW), higher aerodynamic efficiencies (e.g. lift over drag values) and thus also lower fuel consumption (e.g. fuel burn per passenger kilometer) when compared to their conventional counterparts.

For this study, three different aircraft classes were synthesized with the *Initiator* both as conventional and blended-wing-body aircraft: a 150-passenger, a 250-passenger, and a 400-passenger aircraft. It was observed that the above-mentioned favorable characteristics become more distinct with increasing aircraft size. It was noted though that the quantitative results obtained were more of a qualitative or provisional nature. This was attributed to simplifications in the analysis modules which then lead to obvious misestimates (e.g. of the drag).

For this study the *Initiator* was heavily modified: a novel parametrization for modelling blended-wing-bodies was introduced and the existing modules were adjusted accordingly.

This enhanced the *Initiator*'s robustness and broadened the geometric design space. However, the overall design space for blended-wing-aircraft now appeared smaller than the one for conventional aircraft. Eventually, it was recommended to include even more details and setscrews in the novel parametrization.

B.4.3 Conceptual assessment of hybrid electric aircraft with distributed propulsion and boosted turbofans (2019)

In 2019 different innovative aircraft concepts with hybrid electric propulsion systems were investigated with the *Initiator* [7]. The synergistic effects of adopting such novel propulsive technologies emerged clearly from this investigation.

It was demonstrated that especially the distributed propulsion systems influence the design points (e.g. in wing and power loading diagrams) significantly. This led to excessive weight increases (w.r.t. both MTOW and OEW) that would annihilate nearly all other benefits these systems offer. However, nondistributed systems with boosted propulsion systems appeared to be promising (w.r.t. both weights and energy consumption).

For this study four different aircraft were synthesized with the *Initiator*: an aircraft of conventional type (as reference), an aircraft with boosted turbofan engines, an aircraft with a distributed leading-edge propulsion system and an aircraft with an over-the-wing distributed propulsion system and with a propulsive empennage. The obtained aircraft design parameters differed distinctively.

For this study the *Initiator* was significantly modified: additional modules (e.g. for power train modelling and analysis) were added, existing modules were modified (e.g. the constraint analysis, the mission analysis) and the process flow was adapted accordingly. This way, the interaction effects between the propulsive system and the overall aerodynamic performance could be captured successfully as well. Nevertheless, it was also emphasized that the *Initiator* would still benefit from more detailed (but at that point not available) models of the propulsive systems.

C

Implementation of the *ReInitiator*

The *ReInitiator* has been developed as a reference implementation of the software architecture proposed in chapter 4. This appendix sheds light on some implementation aspects. Figure C.1 offers an overview of the repository structure of the *ReInitiator*, showing the most significant files and directories. The purpose of this appendix is to explain the design decisions that have shaped this repository structure. Moreover, this appendix aims to clarify how the proposed software architecture elements can be implemented.

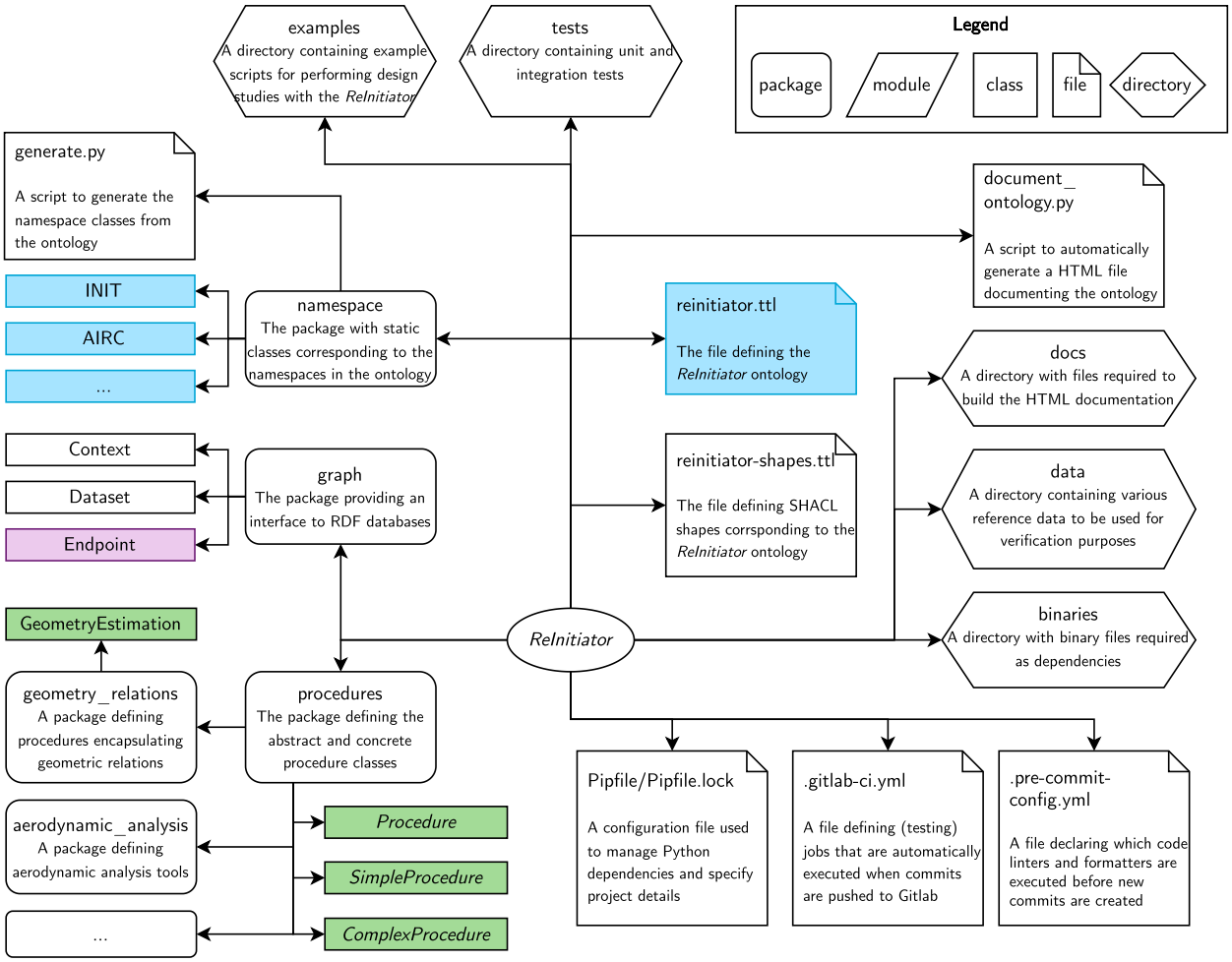


Figure C.1: Overview showcasing important files, directories, packages, modules and classes contained in the *ReInitiator* repository

C.1 Ontology

One key element of the proposed software architecture is the use of an ontology. The *ReInitiator* ontology is based on well-known and openly available schemas/vocabularies, including RDF, RDFS, OWL, DCTerms, and others. A selection of important high-level entities of the ontology are shown in figure C.2.

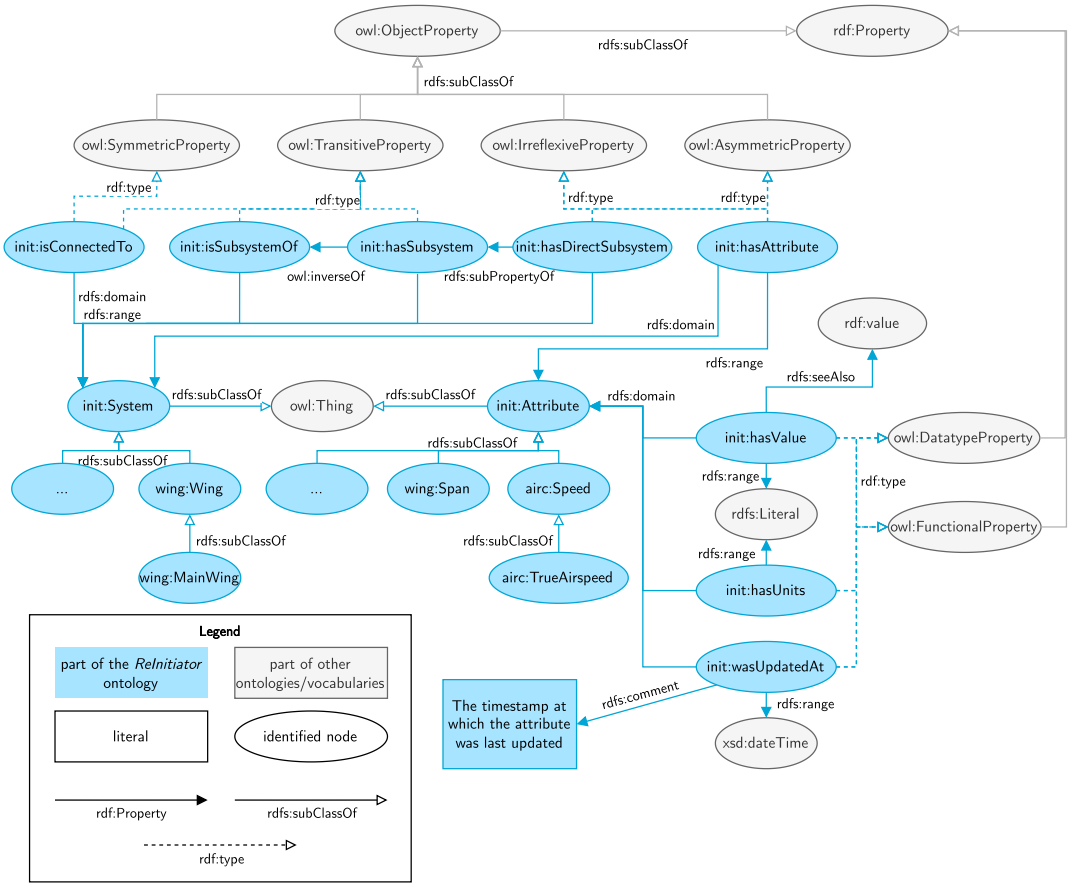


Figure C.2: Selection of important high-level entities of the *ReInitiator* ontology

Please note that the *ReInitiator* ontology encompasses approximately 350 nodes and 1100 edges. Sketches such as the ones in figure 4.7 and figure C.2 only show a small subset of these nodes and edges. The true number of elements and actual size of the ontology is more accurately represented by figure C.3. In this figure, two distinct clusters of nodes are visible, which are the system and attribute classes, respectively.

The meticulously defined system and attribute classes constitute the core entities of the *ReInitiator* ontology. Each of these classes is defined with at least a description and/or a link to a corresponding entry in the Wikidata[1] knowledge base, as illustrated in figure C.4a. Furthermore, these classes are organized hierarchically, as exemplified in figure C.4b. It is important to note that the concept of a class in the context of RDF differs from that in OOP. An RDF class defines a set of resources with shared characteristics, while an OOP class defines a blueprint for creating objects with specific attributes and behaviors in software development. RDF classes represent sets of data, while OOP classes encapsulate sets of objects. Therefore, it is not surprising to encounter classes repeatedly in the hierarchy.

[1] `https://www.wikidata.org`
[2] `http://vowl.visualdataweb.org/webvowl.html`

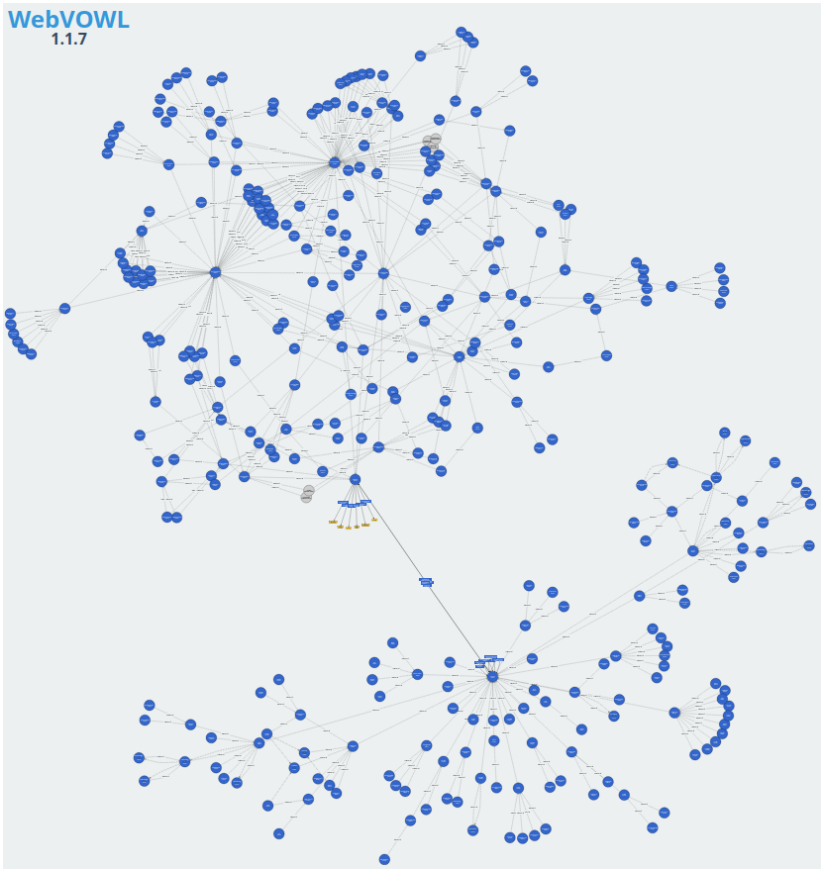
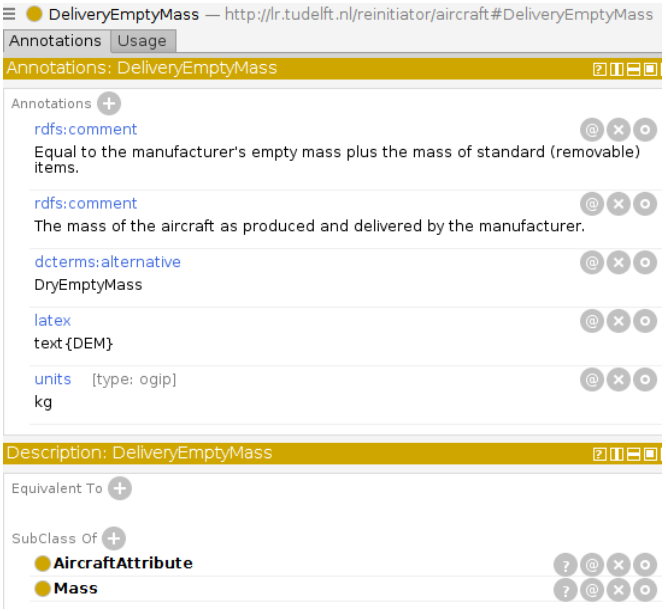
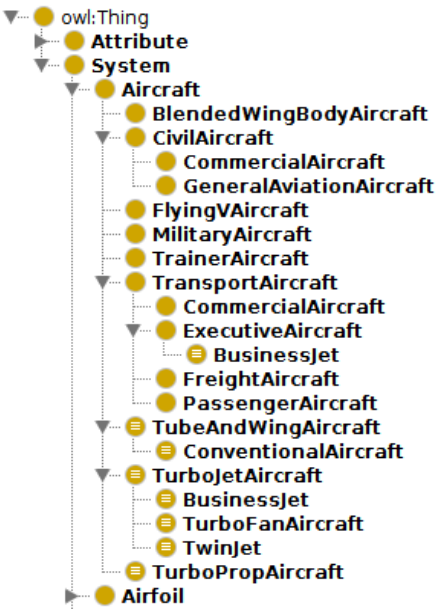


Figure C.3: *WebVOWL*[2] visualization of the *ReInitiator* ontology



(a) Example of the definition of an attribute



(b) Extract of the class hierarchy

Figure C.4: Screenshot from *Protégé*[3] showing distinct aspects of the *ReInitiator* ontology

In order to avoid that the *ReInitiator* ontology becomes too confusing and too cumbersome to work with (given the numerous classes it defines), a number of namespaces (with corresponding prefixes) have been established. For instance, core `init:System` and `init:Attribute` classes along with core properties (e.g. `init:hasSubsystem` and `init:hasAttribute`) have been defined in the main namespace `http://lr.tudelft.nl/reinitiator#` with prefix `init`. More specific classes have been defined in subordinate namespaces. For example, the `wing:Wing` and `wing:Span` classes have been defined in the namespace `http://lr.tudelft.nl/reinitiator/wings#` with prefix `wing`, while the `engi:Fan` and `engi:DryEngineMass` classes have been defined in the namespace `http://lr.tudelft.nl/reinitiator/engines#` with prefix `engi`.

It is important to note that the different systems and attributes have been intentionally modeled as classes inheriting from `owl:Thing` (cf. listing C.1). An alternative (and maybe more conventional) approach would have been to make the different systems inherit from `owl:Thing`, while the different attributes are treated as instances of `owl:ObjectProperty` (cf. listing C.2). However, the adopted approach offers practical benefits: Firstly, it makes it possible to define attributes even without specifying a system (e.g. one can define axioms such as `attribute2 rdf:type misc:AirportCategory` and `attribute2 rdf:type "ClassIV"`). Secondly, it ensures that the type of an attribute is always defined together with its value, units, etc. (e.g. one and the same attribute cannot accidentally be referred to with multiple unrelated properties such as **`airc:hasTrueAirspeed`** and **`airc:hasAirportCategory`**). Thirdly, it simplifies querying, particularly when property path constructs are involved in the queries.

Listing C.1: Adopted approach for modelling systems and attributes in an ontology

```
init:System rdfs:subClassOf owl:Thing
init:Attribute rdfs:subClassOf owl:Thing
init:hasAttribute rdf:type owl:ObjectProperty

airc:Aircraft rdfs:subClassOf init:System
airc:TrueAirspeed rdfs:subClassOf init:Attribute

attribute1 rdf:type airc:TrueAirspeed
attribute1 hasValue "850"
attribute1 hasUnits "km/h"

aircraft1 rdf:type airc:Aircraft
aircraft1 init:hasAttribute attribute1
```

Listing C.2: Alternative approach for modelling systems and attributes in an ontology

```
init:System rdfs:subClassOf owl:Thing
init:Attribute rdfs:subClassOf owl:Thing
init:hasAttribute rdf:type owl:ObjectProperty

airc:Aircraft rdfs:subClassOf init:System
airc:hasTrueAirspeed rdfs:subPropertyOf init:hasAttribute

attribute1 rdf:type airc:Attribute
attribute1 hasValue "850"
attribute1 hasUnits "km/h"

aircraft1 rdf:type airc:Aircraft
aircraft1 airc:hasTrueAirspeed attribute1
```

The *Protégé*[3] tool was used for the creation of the ontology due to its user-friendly GUI. Figure C.5 shows a screenshot from *Protégé*, providing some statistics about the *ReInitiator* ontology. It was decided to store the ontology in the TTL format because this format provides a human-readable and compact representation of RDF data. Unlike more verbose formats such as XML or JSON, the use of TTL eases the creation and maintenance of an

[3] `https://protege.stanford.edu/`

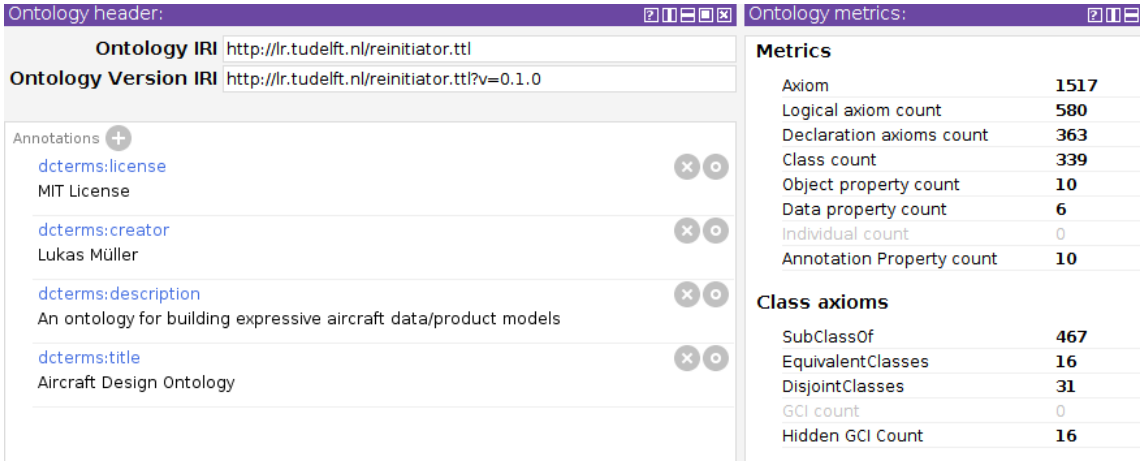


Figure C.5: Metadata of the *ReInitiator* ontology as displayed in *Protégé*[3]

ontology, even in the absence of a tool like *Protégé* (easy conversion between formats is still possible). An example of an attribute definition in the TTL syntax is given in listing C.3 (note that this figure corresponds to the attribute definition shown in figure C.4). Additionally, the quality of the ontology and potential pitfalls were assessed using the OOPS![4] tool.

Listing C.3: Example definition of an attribute contained in the *ReInitiator* ontology in TTL syntax

```
###  http://lr.tudelft.nl/reinitiator/aircraft#DeliveryEmptyMass
airc:DeliveryEmptyMass rdf:type owl:Class ;
                       rdfs:subClassOf airc:AircraftAttribute ,
                                       misc:Mass ;
                       dcterms:alternative "DryEmptyMass" ;
                       rdfs:comment "Equal to the manufacturer's empty mass plus the mass of
                            standard (removable) items." ,
                                    "The mass of the aircraft as produced and delivered by the
                                       manufacturer." ;
                       init:latex "\\text{DEM}" ;
                       init:units "kg"^^ogip:ogip .
```

In parallel to creating the *ReInitiator* ontology, corresponding SHACL shapes have been defined in a dedicated file. These shapes impose constraints on RDF data with the intention of enhancing consistency. For example, they can be used to specify that nodes of the class `init:System` can be linked to nodes of the class `init:Attribute` via the `init:hasAttribute` object property but not via the `init:hasSubsystem` object property. Similarly, they can be used to dictate that a node of the class `misc:Efficiency` can only be assigned literals between 0 and 1 via the `init:hasValue` data property.

C.2 Graph package

In order to work with RDF data and in order to make use of the *ReInitiator* ontology, the graph package was developed. This package is structured around three main classes: the dataset class, the endpoint class, and the context class. These classes are visualized in figure C.6 and explained in the following. Note that the graph package is based on the *RDFlib*[5] package, a well-established library serving as the de facto standard for RDF data manipulation with Python. In the course of developing the *ReInitiator*, some contributions

[4] `https://oops.linkeddata.es/`
[5] `https://github.com/RDFLib/rdflib`

were made to enhance the *RDFlib* package as well[6,7].

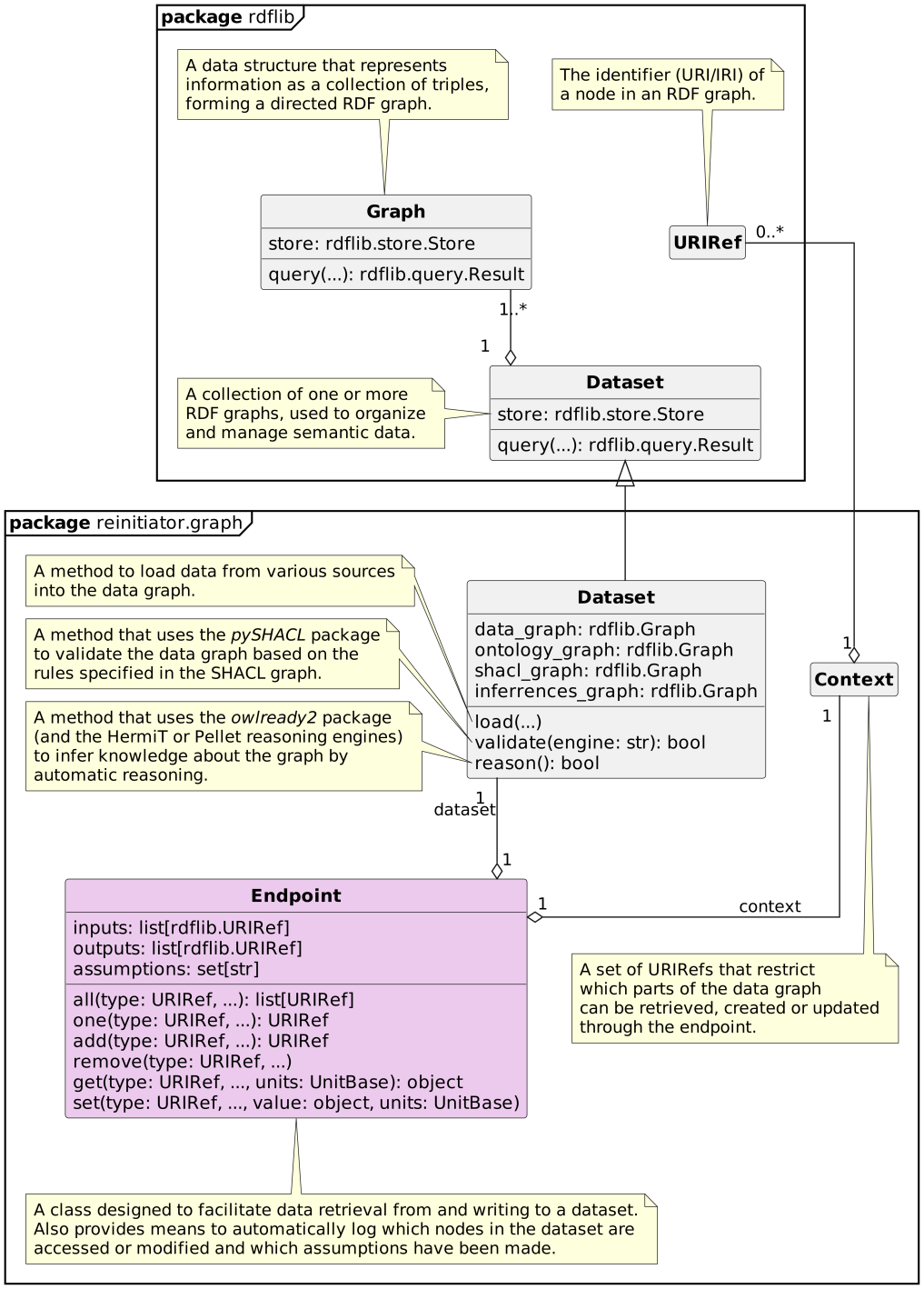


Figure C.6: Class diagram illustrating core classes of the graph package

The dataset class of the graph package extends the dataset class of the RDFlib package by providing four default (named) graphs, as illustrated in figure C.6. Upon instantiation of the dataset class, two of these graphs, namely the ontology and shapes graph, are pre-populated. The data graph is to be initially populated by the user of the *ReInitiator*,

[6] `https://github.com/RDFLib/rdflib/pull/2504`
[7] `https://github.com/RDFLib/rdflib/pull/2520`

while the inferences graph is automatically populated after running a reasoning engine. Additionally, the dataset class provides utility methods for populating the data graph with initial data, for validating the data graph, and for automatic inferencing of design knowledge.

A dataset instance needs to be backed by a store instance, which provides an interface to an actual graph database implementation. The *RDFlib* package provides various stores out of the box. For the *ReInitiator* two different stores were considered: the non-persistent default in-memory store and a modern file-based store leveraging *Oxigraph*[8] database written in the highly efficient Rust programming language. Initially, the default in-memory store seemed to be a lot less performant than the modern file-based store. After memoizing/caching the process of SPARQL query preparation/interpretation, the in-memory store appeared to be more performant than the file-based store. Both stores proved to be suitable and were employed interchangeably during the development of the *ReInitiator*.

Another key element of the proposed software architecture is the endpoint class, which offers a standardized interface for accessing the dataset and additional logging features. For this, the endpoint class provides various methods, as illustrated in figures 4.4 and 4.14, each of which accepts a number of search and filter parameters as input arguments. Subsequently, these parameters are translated into a SPARQL query, which is then automatically prepared/interpreted[9], and executed on the dataset. The translation logic is exemplified in table C.1. Please note that the translation logic is non-trivial and extensively leverages property path constructs to obviate the necessity for inferred knowledge about transitive relationships between nodes. Finally, the methods check if the data returned by the dataset is as expected (and otherwise raise an exception). The get and set methods also automatically update the input and output attributes, respectively, when being called. Note that the input and output attributes (i.e. the attributes used for logging purposes) are semi-private attributes. They can be read globally but modified only from within the endpoint class[10]. In the future the endpoints might potentially be extended to incorporate access control mechanisms that can be used to, for example, prevent that some parts of the data graph can only be modified by any other than a pre-determined procedure. Such a mechanism could be useful to avoid unintended behaviors resulting from multiple procedures concurrently updating a single design parameter.

Another noteworthy class contained in the graph package is the context class. As its name indicates, the purpose of this class is to provide context to SPARQL queries. This class can be used to limit which parts of a dataset can be accessed through an endpoint. A context can be applied at multiple levels: either directly when calling the endpoint methods, or indirectly when creating an endpoint, or even when defining procedures. The context is intended to always be propagated down to the query level. The functioning of the context is best illustrated by an example: When the context `init:MainWing` is required, this means that the targeted attribute/system must either be an attribute/system contained in a system of class `init:MainWing` or, alternatively, it must be an attribute/system not contained in any system of the class `init:Wing` (which is the base system class of `init:MainWing`). The latter condition becomes imperative to prevent undue query restrictions when a context is specified during endpoint creation or procedure definition. For instance, a scenario may arise where the context `init:MainWing` is applied to a generic procedure designed for calculating wing weight. This procedure might very well require an attribute that is defined for the aircraft system (but not the wing system) as input. Without the latter

[8] `https://github.com/oxigraph/oxrdflib`

[9] This process should be memoized/cached as explained above.

[10] While in Python one cannot define truly private attributes, the use of the property decorator facilitates the emulation of semi-private attributes.

Table C.1: Examples of the translation of search and filter parameters into SPARQL queries

arguments	query

`type = 'fuse:Fuselage'`
`types_transitive = True`
	`SELECT DISTINCT ?system`
	`WHERE`
	`{`
	`  ?system rdf:type/rdfs:subClassOf* fuse:Fuselage.`
	`}`

`type = 'misc:Length'`
`types_transitive = True`
`system = 'fuse:Fuselage'`
	`SELECT DISTINCT ?attribute`
	`WHERE`
	`{`
	`  ?attribute rdf:type/rdfs:subClassOf* misc:Length.`
	`  fuse:Fuselage ^rdfs:subClassOf*/^rdf:type?/init:hasAttribute ?attribute.`
	`}`

`type = 'wing:OswaldEfficiency'`
`types_transitive = False`
`context = {'wing:MainWing'}`
	`SELECT DISTINCT ?attribute`
	`WHERE`
	`{`
	`  ?attribute rdf:type wing:OswaldEfficiency.`
	`  FILTER (`
	`  EXISTS { wing:MainWing ^rdfs:subClassOf*/^rdf:type?/(init:hasSubsystem|init:hasDirectSubsystem)*/init:hasAttribute ?attribute. } ||`
	`  NOT EXISTS { wing:Wing ^rdfs:subClassOf*/^rdf:type?/(init:hasSubsystem|init:hasDirectSubsystem)*/init:hasAttribute ?attribute. }`
	`).`
	`}`

`type = 'engi:OverallEfficiency'`
`types_transitive = False`
`context = {'misc:HarmonicMission',`
`'misc:Cruise'}`
	`SELECT DISTINCT ?attribute`
	`WHERE`
	`{`
	`  ?attribute rdf:type engi:OverallEfficiency.`
	`  FILTER (`
	`  EXISTS { misc:HarmonicMission ^rdfs:subClassOf*/^rdf:type?/(init:hasSubsystem|init:hasDirectSubsystem)*/init:hasAttribute ?attribute. } ||`
	`  NOT EXISTS { misc:Mission ^rdfs:subClassOf*/^rdf:type?/(init:hasSubsystem|init:hasDirectSubsystem)*/init:hasAttribute ?attribute. }`
	`  ).`
	`  FILTER (`
	`  EXISTS { misc:Cruise ^rdfs:subClassOf*/^rdf:type?/(init:hasSubsystem|init:hasDirectSubsystem)*/init:hasAttribute ?attribute. } ||`
	`  NOT EXISTS { misc:FlightPhase ^rdfs:subClassOf*/^rdf:type?/(init:hasSubsystem|init:hasDirectSubsystem)*/init:hasAttribute ?attribute. }`
	`  ).`
	`}`

condition, it would not be possible to query such an attribute anymore.

The graph package further encompasses utility functions. For example, it offers robust serializing and parsing features, which allow for not only storing numerical values and strings but also complex objects in the graph database. This becomes particularly relevant when working with highly modularized procedures, where there is often a need to pass around matrices, dictionaries, etc. To achieve this, a method has been devised that first leverages the *dill* package[11,12] to convert arbitrary object structures into a byte stream and then encodes this byte stream into Base64 format, which can be stored seamlessly as literal in a graph database (and which can be retrieved, decoded and parsed as needed).

Lastly, the namespace package shall be mentioned. Though the namespace package is not included in the graph package, it is very much related to it. The namespace package comprises multiple files with static classes that can be automatically generated from the *ReInitiator* ontology (see figure C.7). The classes correspond to the various namespaces defined in the *ReInitiator* ontology (as previously explained in appendix C.1) and among others contain all attribute and system classes defined in the *ReInitiator* ontology as class attributes. This makes it possible to refer to these attributes and systems from the source code without relying on plain strings. Instead, one can refer to the class attributes, which allows for auto-completion, type-checking, etc. After modifications to the *ReInitiator* ontology, a straightforward script execution suffices for regenerating the files containing the namespace classes (see also appendix C.4).

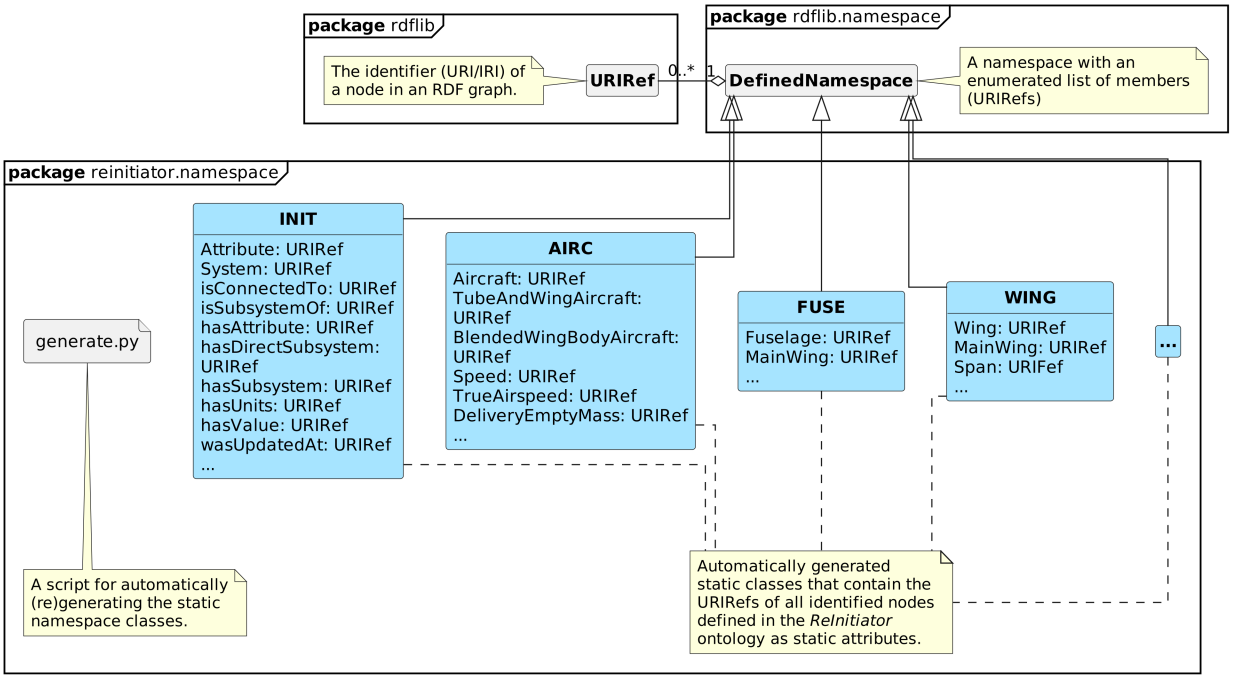


Figure C.7: Class diagram illustrating a selection of classes from the namespace package

[11] `https://github.com/uqfoundation/dill`

[12] The *dill* package is a drop-in replacement for the well-known *pickle* module but can handle a larger range of object types.

C.3 Procedures package

Another core element of the *ReInitiator* is the procedures package. This package encompasses abstract procedure base classes, along with concrete procedure classes. The following section provides a more detailed explanation of the classes contained in the procedures package and the overall package structure.

The procedure base class is shown in figure 4.9. It is an abstract class, which means that it cannot be instantiated. Instead, it specifies various abstract attributes/methods. These abstract attributes/methods must be defined in subclasses that inherit from the base class. Only when a subclass defines all abstract attributes/methods can it be considered a concrete class, and only then can it be instantiated. The main abstract method of the procedure base class is the compute method, which is supposed to contain the actual calculation logic of a procedure. However, the procedure base class delineates additional abstract attributes/methods that must be defined in subclasses, for example for self-documentation and self-visualization purposes, as illustrated in figure C.8.

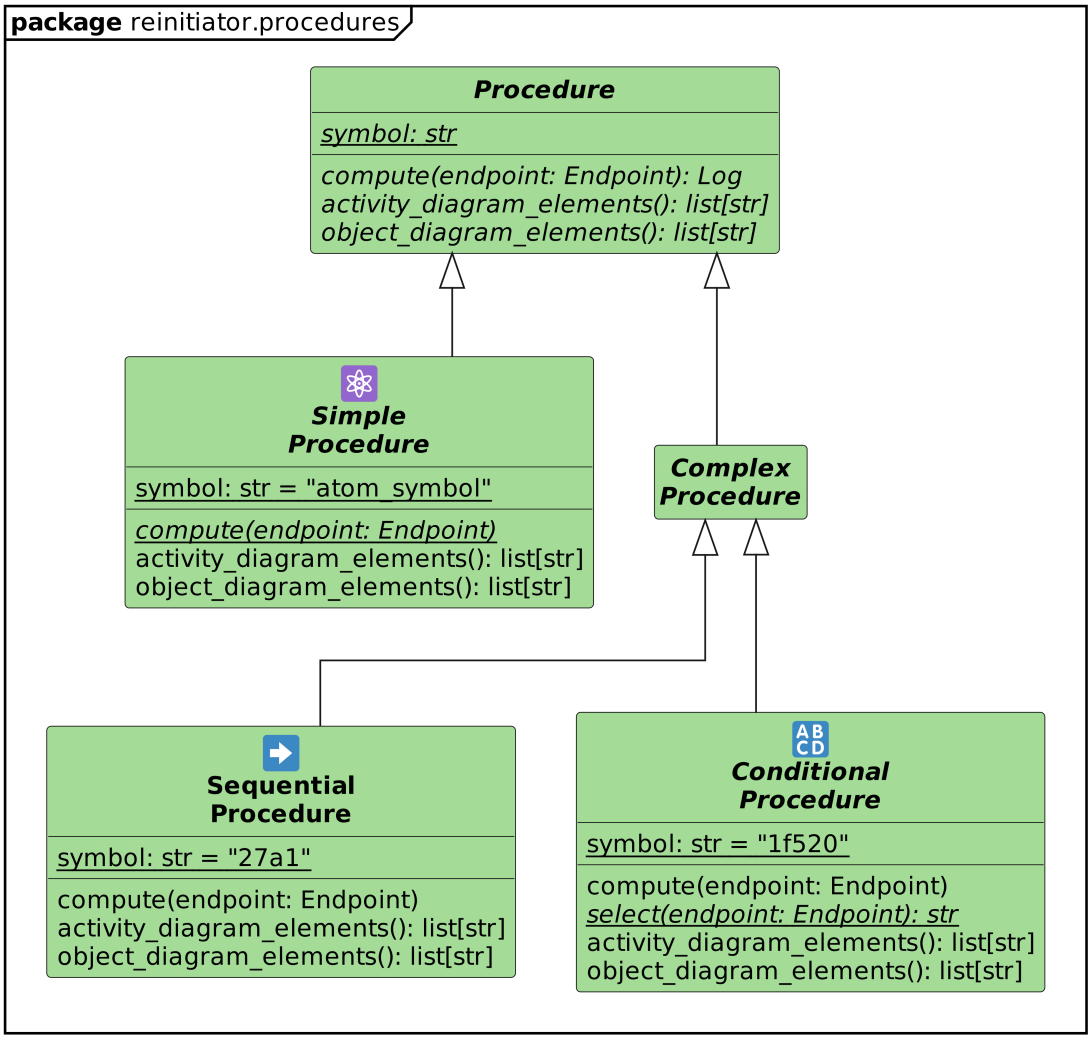


Figure C.8: Class diagram showcasing procedure classes with a specific emphasis on the strategic utilization of abstract attributes and methods

The procedures that exclusively specify calculation logic and do not invoke other proce-

dures are classified as simple procedures. Listing C.4 provides an example class definition of such a procedure. The listing illustrates that the procedure defines a few attributes with metadata. Additionally, the procedure implements the compute method, which accepts an endpoint as input argument and uses the methods provided by this endpoint to retrieve data from and to write data to a dataset. It is important to emphasize that the compute method normally does not directly return any values.

Listing C.4: Example class definition of a simple procedure (cf. figure 4.9)

```
from math import cos
from astropy import units

from reinitiator.graph import Endpoint
from reinitiator.namespace import AIRC, MISC, WING
from reinitiator.procedures import ProcedureException, SimpleProcedure

class WingMassEstimation(SimpleProcedure):

    description = "Estimate␣the␣main␣wing␣mass␣for␣transport␣category␣aircraft␣based␣on␣an␣
        empirical␣relation."
    source = "Torenbeek,␣E.:␣Synthesis␣of␣Subsonic␣Airplane␣Design,␣1982,␣Section␣8.4.1b"
    assumptions = {
        "Assumed␣civil␣airplane␣with␣aluminium␣alloy␣cantilever␣wings"
    }

    def compute(self, endpoint: Endpoint):
        # Check basic assumption
        if endpoint.get(AIRC.MaximumTakeOffMass, units=units.kg) < 5670:
            raise ProcedureException
        # Get empirical factors & physical parameters
        k_w = 6.67e-3
        b_ref = 1.905
        b = endpoint.get(WING.Span, system=WING.MainWing, units=units.m)
        Lambda = endpoint.get(WING.HalfChordSweep, system=WING.MainWing, units=units.rad)
        b_s = b / cos(Lambda)
        ...
        # Calculate base mass
        base_mass = (
            k_w * b_s**0.75 * (1 + math.sqrt(b_ref / b_s)) * n_ult**0.55 *
            ((b_s / t_r) / (m_g / S)) ** 0.3 * m_g
        )
        # Determing correction factors
        multiplier = 1.0
        if endpoint.all(WING.Spoiler, system=WING.MainWing):
            multiplier += 0.02
        ...
        # Calculate and set final mass
        mass = base_mass * multiplier
        endpoint.set(MISC.Mass, system=WING.MainWing, value=mass, units=units.kg)
```

The procedures that consist of and invoke other subordinate procedures are classified as complex procedures. Multiple types of complex procedures have been established, as illustrated in figures 4.10 and 4.11, each employing a distinct and pre-defined compute method. The subordinate procedures contained in a complex procedure are stored in an ordered dictionary[13] which can be modified even during runtime. Listing C.5 presents an example of how to define a complex procedure. Please note that a number of attributes with metadata are specified (as with simple procedures). Furthermore, a method[14] specifying the default steps is provided. However, the logic for calling these steps (sequentially in this particular case) does not need to be defined in the exemplified procedure class, as it has already been defined in a superclass.

[13]The `OrderedDict` type from the Python standard library has been utilized. It is similar to the `dict` type but comes with additional methods for rearranging dictionary entries. Since Python version 3.7, the default `dict` type also incorporates ordered entries (ordered by insertion order) but lacks methods for rearranging dictionary entries.

[14]The default steps cannot be specified directly in an attribute; they need to be defined within a method. This is necessary to accommodate for referencing/copying peculiarities within Python.

Listing C.5: Example class definition of a complex procedure (cf. figure 4.11)

```
from reinitiator.namespace import GEAR
from reinitiator.procedures import SequentialProcedure
from reinitiator.procedures.weight_estimation.torenbeek import WingMassEstimation, ...

class AirframeStructureGroupMassEstimation(SequentialProcedure):

    description = "Estimate the mass of the airframe structure group."
    source = "Torenbeek, E.: Synthesis of Subsonic Airplane Design, 1982, Section 8.4.1"

    def default_steps(self):
        return {
            "wing_group": Step(WingMassEstimation.default()),
            "horizontal_tail": Step(HorizontalTailMassEstimation.default()),
            "vertical_tail": Step(VerticalTailMassEstimation.default()),
            "fuselage_group": Step(FuselageMassEstimation.default()),
            "landing_gear_group": Step(
                LandingGearMassEstimation.default(),
                foreach=GEAR.LandingGear
            ),
            "surface_control_group": Step(SurfaceControlGroupMassEstimation.default()),
            "nacelle_group": Step(NacelleGroupMassEstimation.default()),
            "summation": Step(AirframeStructureGroupMassSummation.default()),
        }
```

As already elucidated in section 4.3, procedures are callable objects and can be used akin to standard functions, as illustrated in listing C.6. The dunder method used to make the procedure callable (i.e. the `__call__` method) serves as a wrapper around the compute method. Furthermore, it performs some supplementary tasks, such as measuring execution time and returning a log containing aggregated execution details.

Listing C.6: Example code snippet illustrating how a procedure can be instantiated and invoked

```
from reinitiator.procedures.weight_estimation.torenbeek import WingMassEstimation
...
procedure = WingMassEstimation()
log = procedure(endpoint)
...
```

The procedure package is organized hierarchically. At top level, it predominantly contains abstract procedure classes, described above. At subordinate levels, it mainly encompasses concrete classes, as illustrated in figure C.9. The subordinate levels are divided by discipline (e.g. aerodynamics, weights, performance) but not by function (e.g. sizing, analysis) because one procedure typically performs multiple functions concurrently. A procedure implementation chart is shown in figure C.10. In contrast to the procedure implementation chart shown earlier in figure 4.16, this chart demonstrates that the same procedure can be used several times within different contexts. A related activity diagram is shown in figure C.11 to further clarify this idea. Moreover, the procedure package incorporates additional abstract procedures at subordinate levels to facilitate reusability. For instance, the conditional procedures `LiftOverDragEstimation` and `LostRangeEstimation` depicted in figure C.10 both inherit from the abstract `TurboJetOrTurboProp` class that defines the selection logic applicable to all its subclasses, as visualized in figure C.12.

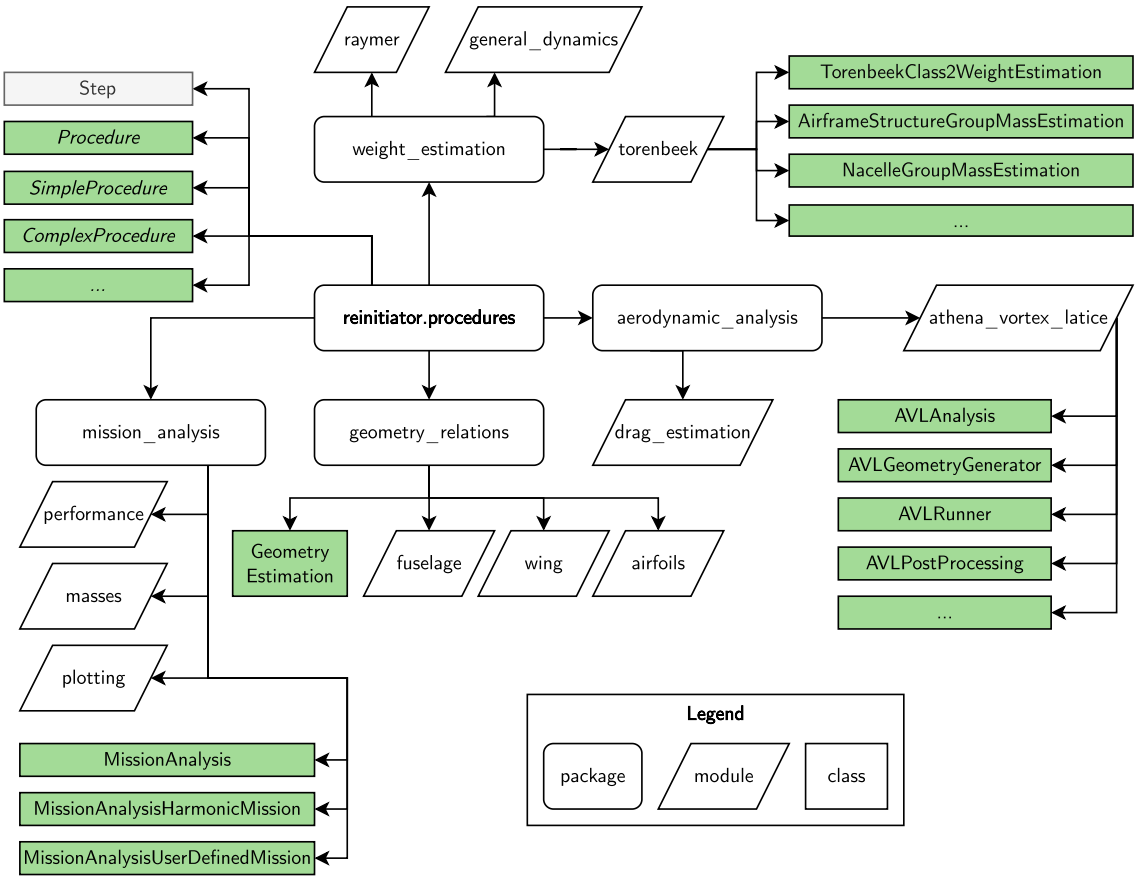


Figure C.9: Partial view of the internal structure of the procedures package

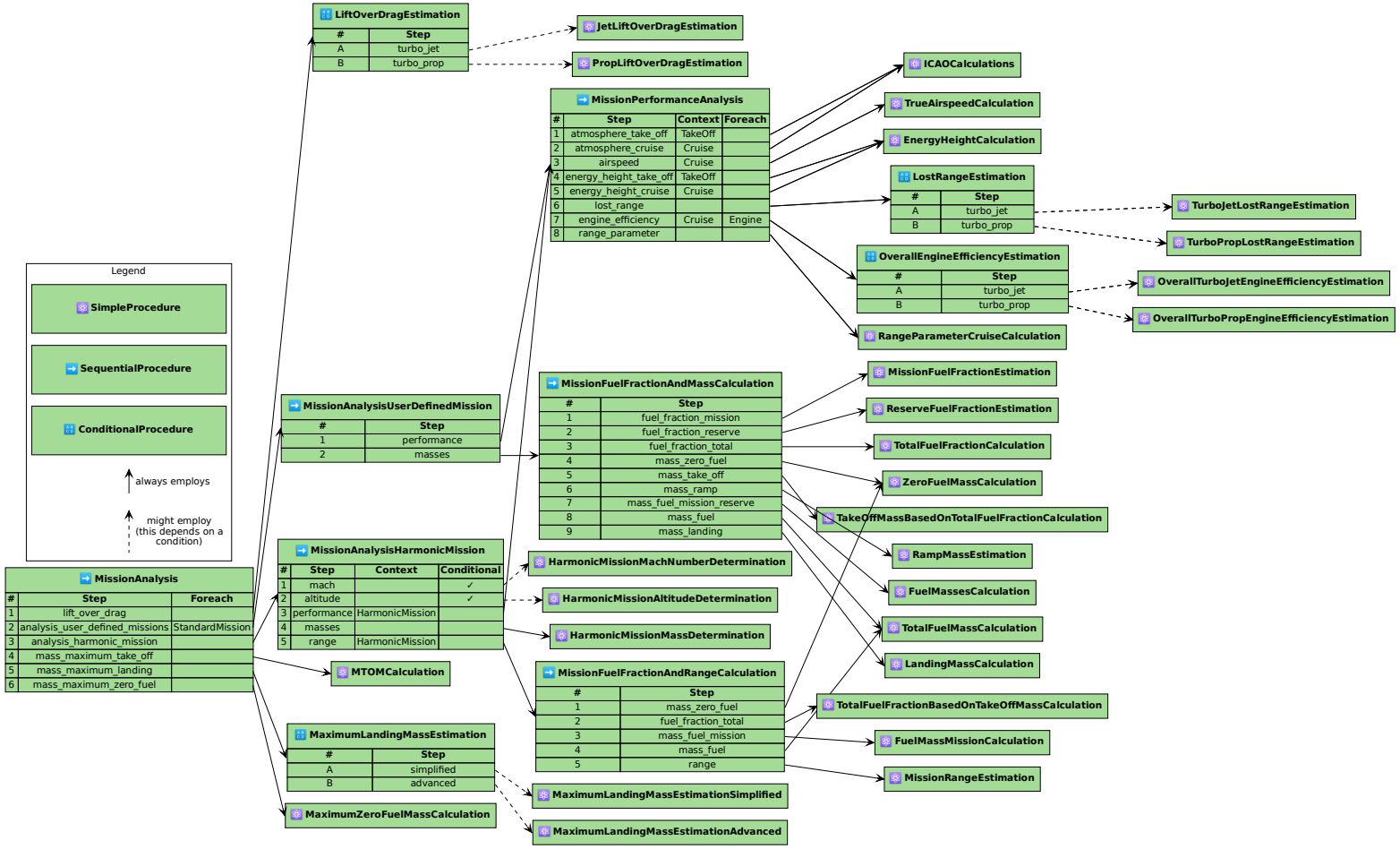


Figure C.10: Example implementation diagram that has been automatically generated from a procedure instance illustrating the reuse of simple procedures (in different contexts)

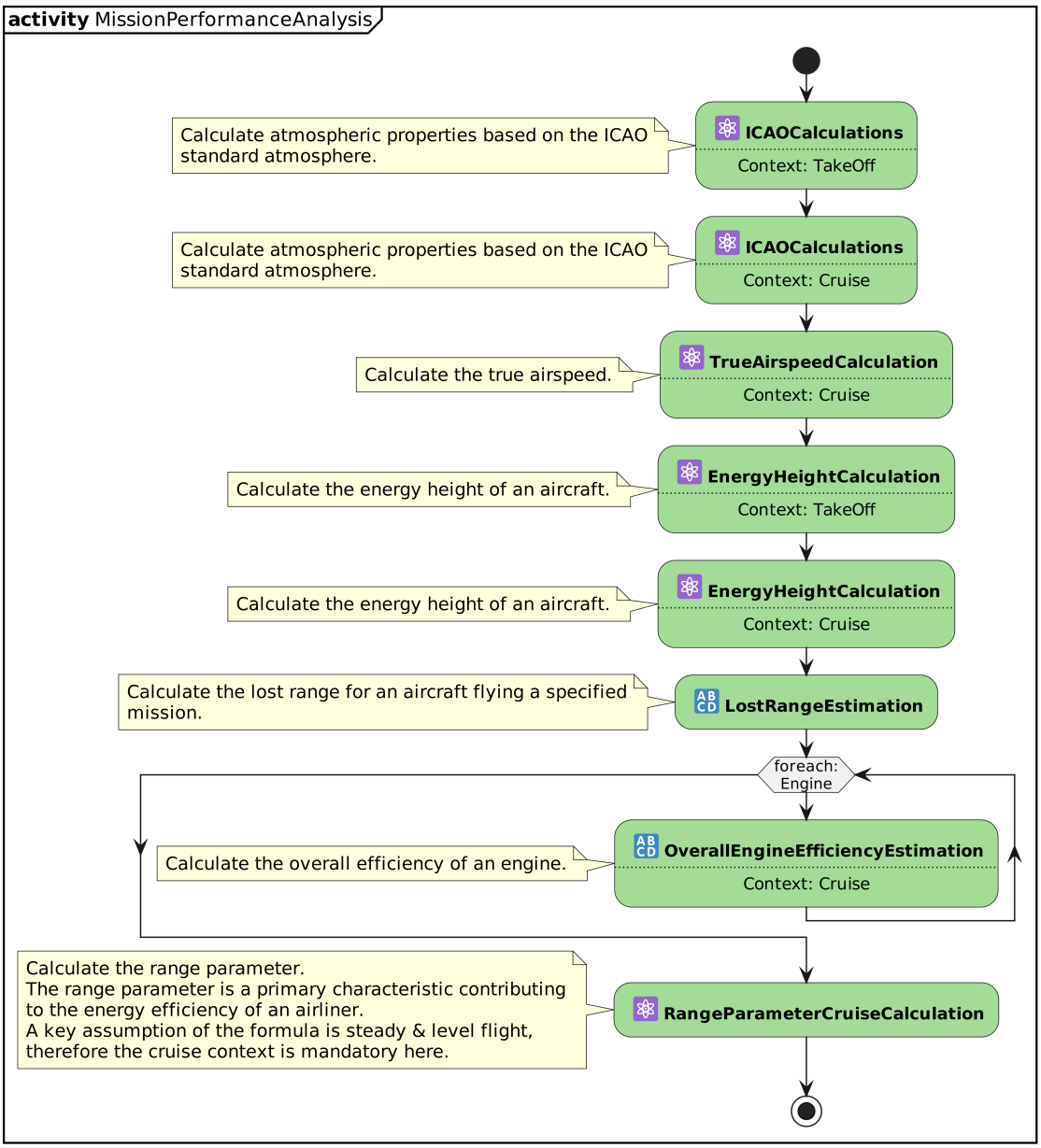


Figure C.11: Example activity diagram representing one of the sequential procedures contained in figure C.10

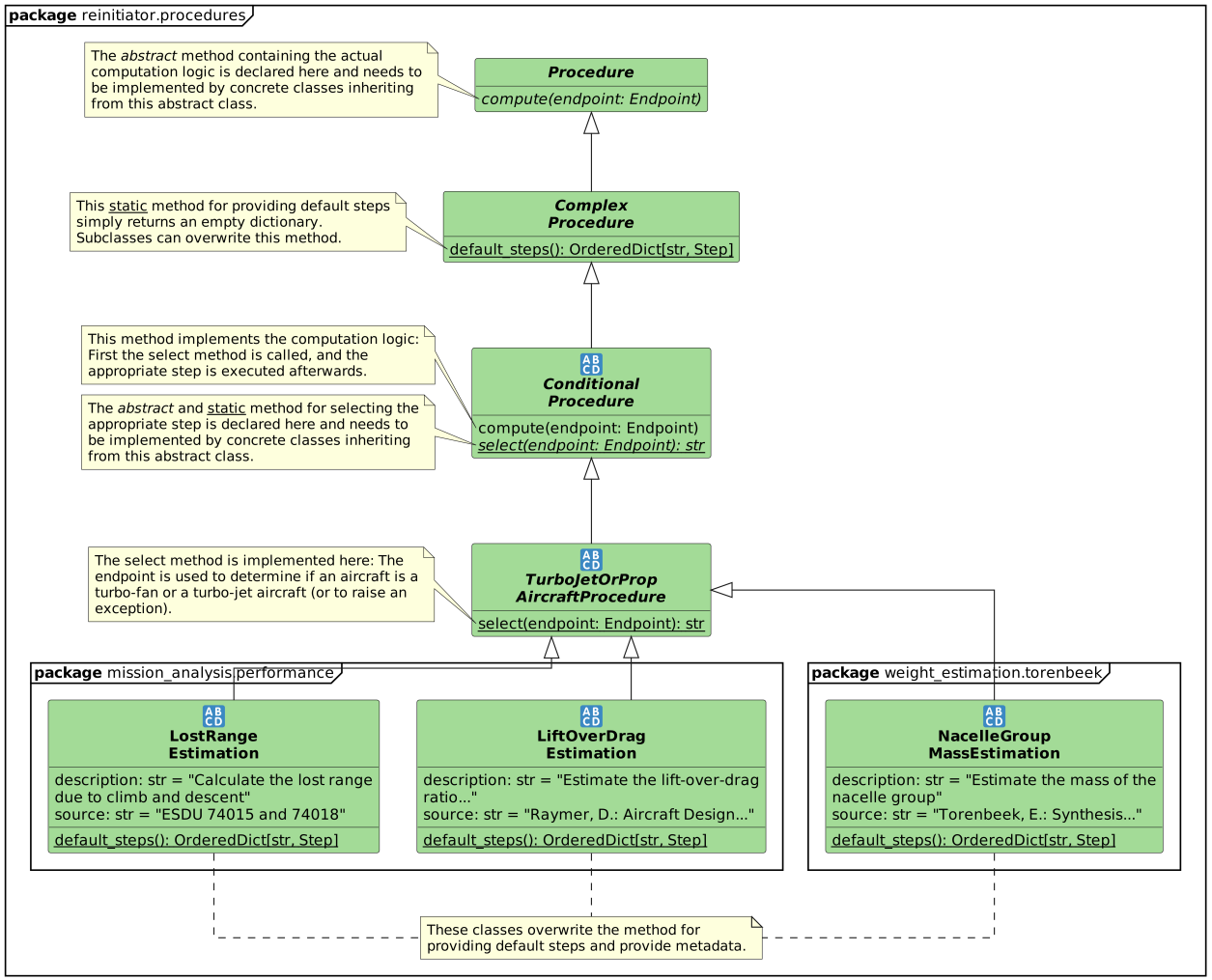


Figure C.12: Example inheritance diagram for conditional procedures contained in figures 4.16 and C.10

C.4 Scripts

Scripting played a crucial role in the development of the *ReInitiator*. The core idea behind scripting revolves around automating routine tasks through the use of simple scripts. This not only frees users from time-consuming manual work, but, more importantly, also ensures reproducibility and consistency. The following section provides examples of how scripting can be used during the development of an ADS such as the *ReInitiator*.

The most important scripts are most likely contained in the examples directory of the *ReInitiator* repository (cf. figure C.1). This directory contains several scripts that demonstrate how the *ReInitiator* can be utilized for conducting aircraft design studies. The scripts in this directory typically feature a similar structure: Firstly, a new dataset is created and populated with initial data such as aircraft requirements or an initial aircraft topology. Additionally, an endpoint is established. This is followed by the instantiation and subsequent execution of one or more procedures. Afterwards, aircraft KPIs and other aircraft design parameters can be retrieved from the database. Moreover, it is also possible to run supplementary procedures, for instance procedures that generate plots of the aircraft geometry. All these steps can be included in a single and straightforward script, as the one shown in listing C.7.

Listing C.7: Example script demonstrating how the *ReInitiator* can be used for aircraft design synthesis purposes

```
from astropy import units

from reinitiator.graph import Dataset, Endpoint
from reinitiator.procedures.utilities import Synthesis
from reinitiator.procedures.geometry_relations import AircraftGeometryPlotter

# Create a new dataset and populate it with initial data
dataset = Dataset()
dataset.load('example-tube-and-wing-configuration.xml')
dataset.load('example-requirements.xml')

# Create a new endpoint
endpoint = Endpoint(dataset)

# Initialize and execute the synthesis procedure (might take a few minutes)
synthesis = Synthesis()
synthesis(endpoint)

# Retrieve aircraft design parameters
mtom = endpoint.get(type='airc:MaximumTakeOffMass', units=units.t)
fm = endpoint.get(type='airc:FuelMass', context={'misc:HarmonicMission'}, units=units.t)
print(f"MTOM: {mtom} t")
print(f"Fuel mass (harmonic mission): {fm} t")

#  Initialize and execute a procedure for plotting the aircraft geometry
plotter = AircraftGeometryPlotter()
plotter(endpoint)
```

It is also possible to reconfigure the *ReInitiator* by means of scripting. This is exemplified in listings C.8 and C.9. The first example shows how the aircraft design process can be enhanced by substituting two procedures with higher-fidelity alternatives. Here one key advantage of representing the steps of a complex procedure as an ordered dictionary, rather than an ordered list, becomes apparent; namely the ease of referencing and replacing individual steps[15]. The second example shows how the aircraft design process can be reconfigured to prevent all engine sizing and weight estimation activities, and to make use of a known engine with fixed geometry and performance characteristics instead. The

[15]Note that the `__getitem__` and `__settitem__` dunder methods have been defined for the complex procedure class so that complex procedure objects can be treated like a dictionary object. Calling these methods essentially modifies the `steps` attribute of a complex procedure object.

second example also shows how activity diagrams can be generated prior to execution, and how N^2 charts can be generated post execution.

Listing C.8: Example script demonstrating how the *ReInitiator* can be used to enhance a synthesis process by substituting procedures

```
from reinitiator.graph import Dataset, Endpoint
from reinitiator.procedures.utilities import Synthesis
from reinitiator.procedures.weight_estimation.others import EMWET
from reinitiator.procedures.aerodynamic_analysis.tornado import TornadoAnalysis

# Create a new dataset and populate it with initial data
dataset = Dataset()
dataset.load('example-tube-and-wing-configuration.xml')
dataset.load('example-requirements.xml')

# Create a new endpoint
endpoint = Endpoint(dataset)

# Initialize, modify and again execute the synthesis procedure
synthesis = Synthesis()
synthesis['aerodynamic_analysis] = TornadoAnalysis()
synthesis['weight_estimation]['structure_group']['wing_group] = EMWET()
synthesis(endpoint)
```

Listing C.9: Example script demonstrating how the *ReInitiator* can be used to enhance a synthesis process by removing procedures and adding additional aircraft design data

```
from reinitiator.graph import Dataset, Endpoint
from reinitiator.procedures.utilities import Synthesis
from reinitiator.procedures.geometry_relations import AircraftGeometryPlotter

# Create a new dataset and populate it with initial data
dataset = Dataset()
dataset.load('example-tube-and-wing-configuration.xml')
dataset.load('example-requirements.xml')
dataset.load('example-engine-data.xml')

# Create a new endpoint
endpoint = Endpoint(dataset)

# Initialize and modify the synthesis procedure
synthesis = Synthesis()
del synthesis['geometry_estimation']['engine_sizing']
del synthesis['weight_estimation']['propulsion_group']

# Plot several activity diagrams
synthesis.activity_diagram()
synthesis['geometry_estimation'].activity_diagram()
synthesis['weight_estimation'].activity_diagram()

# Execute the synthesis procedure
log = synthesis(endpoint)

# Plot the N2 charts corresponding to the first and
# last iteration of the synthesis procedure respectively
log[0].n2()
log[-1].n2()
```

Similar to scripting entire design workflows, it is also possible to script unit and integration tests, in order to verify that the individual elements of design workflows behave as expected. An example unit test is shown in listing C.10. Note the similarity to the previous listings: Firstly, initial data is generated. Then, a procedure is initialized and executed. Subsequently, calculated data is retrieved and validated. Both unit and integration tests can be formulated as functions, and over 100 of these functions have been developed for the *ReInitiator* so far. Next to design workflows, tests for core functionalities, such as generating SPARQL queries and modifying the dataset, have been set up. The *pytest*[16]

[16] `https://pytest.org`

framework has been leveraged since it simplifies and standardizes test creation, organization, and execution. The framework's concise syntax and rich features, such as fixtures[17], enhance test readability and maintainability, while providing test coverage reports[18]. Furthermore, it shall be noted that the *ReInitiator* repository is hosted on a *Gitlab* server. A so-called pipeline[19] has been configured to automatically execute all tests as soon as new commits are pushed to the *Gitlab* server. The pipeline itself is scripted and defined in the .gitlab-ci.yml file (cf. figure C.1).

Listing C.10: Example unit test formulated for the *ReInitiator* as used within the *pytest* framework

```
import pytest

from initiator.workflows.weight_estimation.torenbeek import VerticalTailMassEstimation
from initiator.graph import Endpoint
from initiator.namespace import AIRC, MISC, WING

def test_vertical_tail_mass_estimation(endpoint: Endpoint):
    endpoint.add(WING.VerticalTail)
    endpoint[WING.GrossArea] = 21.5 * units.m**2
    endpoint[WING.HalfChordSweep] = 34 * units.deg
    endpoint[AIRC.DesignDiveSpeed] = 381 * units.imperial.kt
    endpoint[WING.HasFinMountedStabilizer] = False
    procedure = VerticalTailMassEstimation()
    procedure(endpoint)
    mass = endpoint.get(MISC.Mass, system=WING.VerticalTail, units=units.kg)
    assert mass == pytest.approx(467.05)
```

Moreover, supplementary scripts are employed to ensure that the *ReInitiator* repository stays clean and consistent. These scripts are defined in the .pre-commit-config.yaml file and are automatically executed whenever new code is committed to the repository (as already elaborated upon in section 6.1). It is noteworthy that, unlike tests, these scripts only take fractions of a second to run and thus can be executed frequently without impeding development activities. Nonetheless, these scripts are also included in the *Gitlab* pipeline.

Last but not least, the creation of accessible documentation files can be scripted as well. However, due to time constraints, this has only been explored experimentally during the development of *ReInitiator* until now. Firstly, a script has been set up that utilizes the *pyLODE*[20] tool to document the ontology through an HTML page, which might be more accessible than the TTL file. Secondly, a script based on the *Sphinx*[21] framework has been employed to automatically generate multiple HTML pages documenting distinct facets of the source code. Additionally, activity diagrams like the ones shown in figures 4.12 and C.11 can be automatically included within these HTML pages.

[17] A fixture is a reusable component, allowing the execution of common code before and/or after test functions. The endpoint passed as argument to the unit test in listing C.10 serves as an example of such a fixture. This fixture prevents that a dataset and endpoint have to be manually instantiated within every test function.

[18] A test coverage report reveals the extent to which a software's source code has been tested by evaluating the portions of code that are executed when test functions run.

[19] Within *Gitlab* a pipeline is an automated CI/CD system that allows for the configuration, execution, and monitoring of a series of jobs. Other code hosting platforms may use different names for their CI/CD systems. For example, within *Github* the term actions is used instead of pipelines.

[20] `https://github.com/RDFLib/pyLODE`

[21] `https://www.sphinx-doc.org/`