Accelerating aircraft design using automated process generation

An experimental architecture for aircraft design software

M.A.Y. Ramakers

Technische Universiteit Delft

TU Delft Delft University of Technology

Challenge the future

Accelerating aircraft design using automated process generation

An experimental architecture for aircraft design software

by

M.A.Y. Ramakers

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 Tuesday Oktober 20, 2015 at 13:00.

Student number:	1507613	
Thesis number:	053#15#MT#FPP	
Supervisor:	Dr. R. Vos	TU Delft
Thesis committee:	Prof. L. Veldhuis (chair)	TU Delft
	Ir. M. F. M. Hoogreef	TU Delft

An electronic version of this thesis is available at http://repository.tudelft.nl/.

TU Delft Delft University of Technology

Acknowledgements

This master thesis is written as the final part of the Master Program in Flight Performance and Propulsion at the Faculty of Aerospace Engineering of Delft University of Technology. It marks the culmination of my time at this faculty and of my time as a student in Delft.

Since I spend over 9 months on this research I would like to thank those who supported me. First I would like to thank dr. ir. Roelof Vos and ir. Maurice Hoogreef for guiding me through this final part of my masters. I enjoyed our brainstorm sessions and talks on aircraft design. I would also like to thank prof. dr. ir. Leo Veldhuis for chairing my graduation committee.

Secondly I would like to thank my friends and colleagues for their interest in my work and their support during my thesis research. Special thanks go to my fellow students from 'Kamertje 1', for making my thesis time even more enjoyable. Thanks to my family for their continuous support throughout my education. Finally thanks to my girlfriend Pauline, for her loving support, endless patience and for keeping me motivated throughout my thesis.

M.A.Y. Ramakers
Delft, Oktober 2015

Abstract

The aircraft industry has seen many evolutionary changes in the past decades. Since the Boeing 707 however, the general shape and configuration of transport aircraft have remained similar and so has the aircraft design process. Since the aviation community is about to face a revolutionary breakthrough featuring new aircraft concepts like blended-wing body aircraft and Prandtl planes it is time to adapt the design process as well. The aircraft design process may have been extended by computational novelties and has been largely computerized, but the fundamental design process is still similar to what is used for conventional aircraft. This research investigates the effect of the design process on the design outcome and strives to find a method of automatically generating an aircraft design process given a set of computational modules, initial values and design goals.

A software architecture based on a strict separation of components and process modelling approach is proposed. A framework based on this architecture is developed in which design parameters and computational modules are modelled as nodes in a graph. A subset of the conceptual aircraft design process is simplified and implemented. Twelve Algorithms are proposed to perform the automated ordering of the computational modules. The ordering is based on the module run time, estimated impact of the module on the design or the current state of the aircraft design, among others. The design process used in the Initiator aircraft design software is used as a benchmark. Two test cases are formulated, representing Class I and Class II design problems. Two additional Class II test cases are formulated to rule out the impact of the scheduling overhead by simulating the module runtime of fully-featured modules. Each test case is then solved by the proposed algorithms.

The result is a system capable of generating a feasible (but sub-optimal) design process out of a set of stand-alone computational modules. This is achieved with no prior knowledge on the design process. From the test cases it can be concluded that the design outcome is not affected by the design process order that is employed, if the used set of modules is not changed. It is also concluded that the classical, fixed design process outperforms the design processes generated by any of the algorithms for Class I and Class II design problems by 20-40%. From the designed algorithms, the algorithm that orders modules based on the expected change of re-evaluating the module versus the expected run time of the module, performs best for the complex (Class II) design case. The impact of the scheduling overhead is shown to be negligible. Since the system is capable of generating feasible, but sub-optimal design processes, it is recommended that it be used as a tool to generate new design processes for unconventional aircraft configurations. Finally it is recommended that the process modelling based and separation of components based software structure that is presented is adopted for a next version of the Initiator.

Contents

List of Figures

Nomenclature

List of symbols

Symbol	Units	Description
n	-	number of samples/parameters/etc.
μ	–	Sample mean
σ	-	Sample standard deviation
$z_{\alpha/2}$	-	
t	s	time
p	-	priority
f	-	number of function evaluations
E	-	margin of error

Abbreviations

Abbreviation	Expansion
DEE	Design and Engineering Engine
DSM	Design Structure Matrix
xDSM	eXtended Design Structure Matrix
FDT	Functional Dependency Table
REMS	Reconfigurable Multidisciplinary Synthesis
MMG	Multi-Model Generator
XSD	XML Schema Definition
XML	eXtensible Markup Language
JSON	JavaScript Object Notation
CPACS	Common Parametric Aircraft Configuration Schema
VLM	Vortex Lattice Method
CFD	Computational Fluid Dynamics
MDO	Multidisciplinary Design Optimization
KBE	Knowledge Based Engineering
MCG	Maximum Connectivity Graph
FPG	Fundamental Problem Graph
PSG	Problem Solution Graph
DoE	Design of Experiments
KPI	Key Performance Indicators
IDE	Integrated Development Environment
MTOW	Maximum Take-off Weight
FPP	Flight Performance and Propulsion
TUDLR	Technische Universiteit Delft, Lucht- en Ruimtevaart Techniek

1

Introduction

In this chapter an introduction to the topic of this thesis is given. An overview of the aircraft design software developed and designed by TUDLR is also given. The research question and thesis goal are presented. Finally an overview of the report structure is given.

1.1. Aircraft Design

The aviation industry has come a long way since the Boeing 707 entered service in 1958. And although aviation technology has improved significantly since then the most recent commercial jet airlines are still tube-and-wing aircraft. In recent years several radically different aircraft configurations have been proposed (like Blended Wing Body aircraft or Prandtl planes). Consequentially the way aircraft are being designed is gradually changing. For a long time aircraft design has relied on empirical methods (Roskam, Raymer and Toorenbeek), based on experience with designing similar aircraft. Since the proposed aircraft configurations are new and no experience exists, empirical methods are less relevant. Therefor several initiatives exist (TUDLR Initiator [2] and DLR vampZero [3], among others) to move away from the empirical methods (Roskam [7], Raymer [8] and Torenbeek [9]) towards computational, physics-based methods.

What has not changed is the aircraft design process. Although physics-based methods have been included in conceptual aircraft design, the procedure to perform the design has remained the same. It is suggested that the process influences the design outcome and that the classical, fixed aircraft design procedure is not the most efficient design process. Therefor this thesis research focuses on the aircraft conceptual design process.

1.2. TUDLR Aircraft design software

Design & Engineering Engine The Design and Engineering Engine (DEE) concept is a hypothetical framework which was proposed by La Rocca [1]. The concept embodies a Knowledge Based Engineering (KBE) approach to support aircraft design. The DEE consists of computational design and analysis tools from different disciplines and enables data exchange through a multi-model generator (MMG). The purpose of the framework is to automate non-creative and repetitive design tasks. The DEE is focussed on improving the quality of the preliminary design phases but embodies a conceptual design tool called the Initiator. An overview fo the DEE is given by the diagram in figure 1.1.

Initiator The initiator is a conceptual aircraft sizing software tool that provides an initial set of values for the DEE. The initiator contains a vast number of design and analysis methods that are chained in a fixed iterative process to deliver a feasible design. This software is currently under heavy development at the FPP department from TUDLR. The current state boasts a semi-empirical/semi-physics-based structural weight estimation as its most distinguished feature. This feature makes it more suitable for the design of unconventional aircraft configurations.

The Initiator uses a fixed design process based on an automated, iterative implementation of Raymer/Torenbeek sizing and design modules extended by a number of physics based sizing and

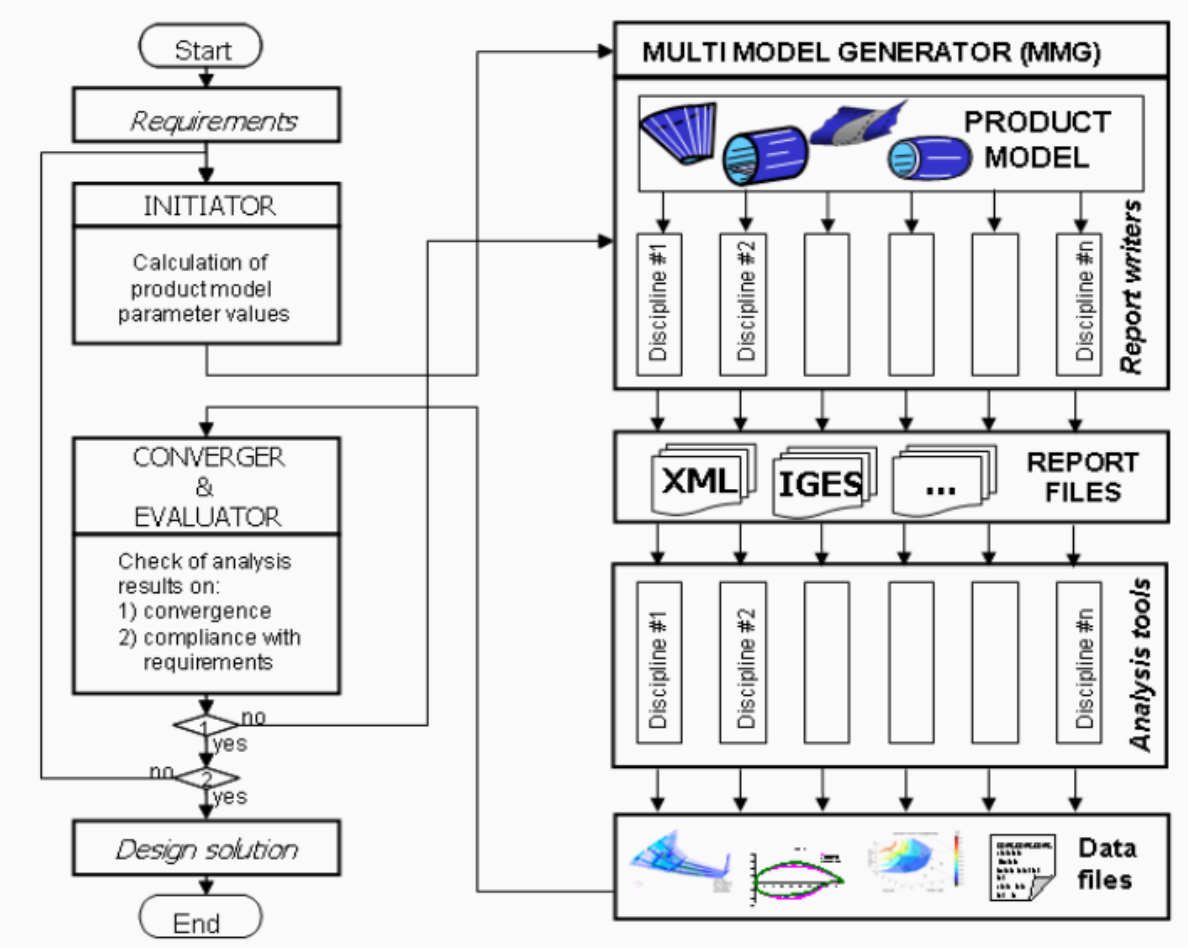

Figure 1.1: The architecture of the Design and Engineering Engine (DEE) [1]

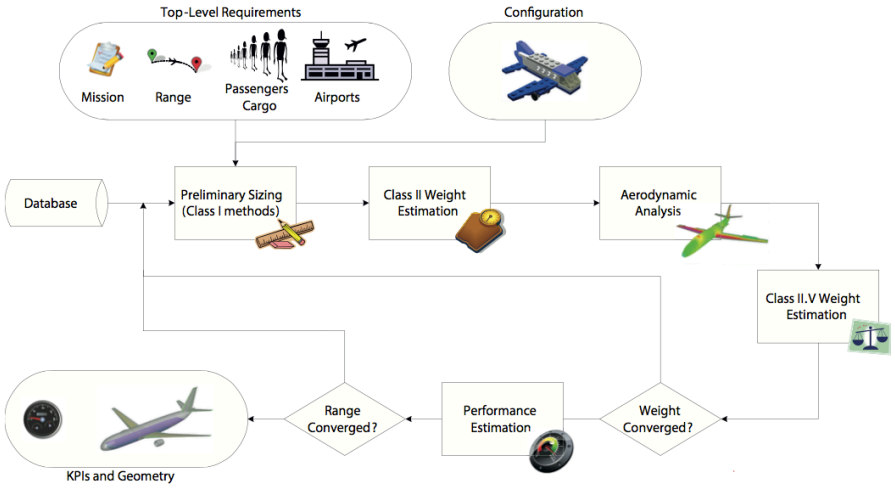

Figure 1.2: The Initiator design process [2]

analysis methods. A global overview of the design process is best illustrated by the flowchart given in figure 1.2.

The Initiator has a number of shortcoming that have been identified. These shortcomings are listed here and explained briefly.

- Dependency tracking - The Initiator treats only inter-module dependencies: before a module can be used a predefined set of preceding modules should be run. There is no registration however of the usage of specific design variables in modules. Therefor there is not possible to trace the influence of a module on the design.

- Data model - The initiator uses the initiator XSD-schema as its data-model. Although suitable for current applications, the data-model is modified ad-hoc whenever required which obscures the availability of information. Although this makes the data model extremely flexible, the data model is not fully known before run time. Therefor it is not feasible to reliably interface between modules or with third party software. Additionally the initiator XSD-schema does not comfirm to the CPACS data format that is rapidly becoming the standard format for interchanging data between aircraft design software. Basic conversion to CPACS is possible but not fully implemented.

- Fixed design process - The design process in the initiator is fixed. Modules are chained by means of a prerequisites list. Adding a module to the design process is done by modifying the prerequisites to include the new module. The fixed design process also means that the same design process is used for different types of aircraft irrespective of the design process being optimal or even suitable for the aircraft type. Therefor it is suggested that the design process, and specifically the ordering of the design steps in the design process might be of influence on the design outcome.

  Another downside of a fixed process is that computational resources are potentially wasted in the iterative convergence process. In a case where only in a single iteration only minor changes occur to a limited number of design variables, all the process steps are executed without considering if individual steps are effected by the changes that occurred.

  This is best illustrated by two examples: consider the flowchart given in figure 1.2. At some point in the design, the wing material properties are changed in the module 'Preliminary Sizing'. The wing shape however is not altered. Following the successive steps in the design process, first the 'Class II Weight Estimation' is performed (which is valid, since the weight is directly affected by the material properties) and secondly the 'Aerodynamic Analysis' is done. Since the wing shape did not change the input to the aerodynamic analysis is the same in this and the previous iteration

and the result is the same too. Therefor this evaluation of the 'Aerodynamic Analysis' module is redundant and a waste of computational effort. It is suggested that an increase in computational efficiency can be achieved by dynamically modifying the process during the execution of the design process, based on the current state of the design and the state history.

Consider the flowchart in figure 1.2 again for another example. The flowchart shows a convergence loop on the aircraft weight (Class II.V and Class II weight). The module 'Class II.V Weight Estimation' is in internally an iterative process. Thereby it is computationally rather expensive. If after the first iteration in this module it becomes clear that there exists a large difference between the class II and class II.V weight estimation, it is most probably a wasted effort to complete the expensive Class II.V iteration. Perhaps it would be more useful to return to the next iteration of the weight convergence loop after a single Class II.V evaluation. The aircraft model is then updated and both Class II and Class II.V weight estimation are then performed with updated, more feasible aircraft.

1.3. Initiator requirements

The Initiator has been used with success in quite some research. However, since the software does have clear shortcomings an improvement is desirable. Following from the working/developing experience with the current Initiator version and the shortcomings of the software explained in 1.2 a set of requirements was created. The requirements are listed and explained in the paragraphs below.

Workflow: Flexible design sequence & Stand-alone debugging To develop new aircraft design procedures and design methods for new aircraft configurations, it is desirable that the design sequence is not fixed. Multiple design workflows should be supported and developing a new workflow in order to develop or implement a new design method should be allowed. The ability to experiment with new workflows induces that a workflow can be devised encompassing a single simulation module or computation. This ability greatly simplifies debugging of such module and eases the implementation of new analysis tools.

Data Structure: Coherent, pre-defined, documented, multi-fidelity and unambiguous To anticipate the integration of new or third-party tools and to ease the integration progress it is recommended to have a clear, pre-defined aircraft data structure. A pre-defined data structure allows all tool components to communicate in a predefined way, easing tool inter-compatibility. Proper documentation of the data structure is also required to provide clarity about the data structure to users integrating new components. Additionally, it seems appropriate to have a single data-structure that is usable by both conceptual and preliminary design. This is necessary in order to reduce the effort in transferring data between the design phases, or remove the need to transferring data all together. Ultimately this might remove the distinction between conceptual and preliminary design all together. Finally it is vital that the data structure supports any configuration, as exploring new configurations is an important goal.

Dependency: Dependency tracking of variables and values Dependencies in the aircraft design can be of great importance for all users. For the design engineer knowing the dependencies during the aircraft design can reveal opportunities for improvement of the design. For research and development on new design methods, design dependencies may be very helpful in improving the design process and working towards new configurations.

Multi-fidelity: Multi- and variable-fidelity analysis A great way to improve the knowledge about the design in the conceptual design phase is to use higher-order analysis methods. Models which are generally used in later design phases. Therefore it is required that using the tool, one can vary the analysis model used to adapt the design process to the current situation.

Management of running third-party tools As mentioned the integration of third party tools is essential for both the professional designer and the academic researcher. When performing (automated) design iterations, countless calls might be made to those tools. Proper thought should be given to the

management of the third party tool processes to ensure the following occurs efficiently: Running a third party tool is often accompanied by a huge overhead caused by creating a new instance of the tool and removing it when done; On certain computer systems it might be very beneficial to run processes in parallel. Though possible, measures must be taken that allow having multiple instances of a (third party) tool avoiding conflicts between instances.

Advanced input capabilities Conceptual design is about turning the engineers' creative ideas into an initial aircraft design and about discovering whether or not the idea can be realised. Therefore it is desirable that engineers can transfer their ideas to aircraft design software, resembling their idea as closely as possible. To allow this a new way of facilitating this transfer should be devised.

Accuracy/Error quantification of analysis methods When developing scientific analysis/simulation methods, apart from software-testing, scientific validation & verification is also very important. Therefore it is suggested that a method is devised for automated/enhanced quantification of the accuracy of analysis methods. This information can then be used during selection

Technology evaluation Whenever new technologies arise the aircraft design engineer is interested whether or not the application of said technologies improves his design. Additionally new technologies may enable new configurations or concepts that were infeasible to become feasible. Therefore an aircraft design tool should allow the evaluation of a given configuration with selected technologies. Alternatively, the tool should allow the design of several instances of the same configuration, but with different technologies applied. This feature would be of great interest to all potential users.

Debugging Special attention should be paid to debugging features of the tools development environment as well as to the tool itself.

Versioning A tool under constant development should be published using a versioning system, to ensure that current development does not interfere with current usage of existing features.

Testing It is evident that modification during development will/can have effect on unforeseen aspects of the tool. Proper software-testing mechanisms should be implemented to cope with this issue.

Documentation An aircraft design tool for academic research purposes is most likely to be modified intensively after release. Therefore proper documentation on the design tool, but more importantly, on the mechanics of the framework is of vital importance to prevent duplicate efforts and to maximize usability. Next to extensive documentation of the tool functioning, proper written instructions are required to encourage and simplify the further development of the tool. Additionally guidelines for further development should be proposed. These guidelines should instruct future developers on coding practices in the framework, as well as guidance in coding style and documentation behaviour striving for consistency throughout the tool.

Maintenance The extensive modifications and development that is expected for the design tool it is important that the maintenance of the tool is taken seriously. To ensure that the tools is kept in functioning state, that developers/researches adhere to the tools structure and confirm with the guidelines that have to be defined it is recommended that maintenance responsibility is well defined. Variable approaches to software maintenance can be chosen, ranging from making the software open-source to appointing an in-house employee to the job. Without proper maintenance the software-project is likely to become a mess.

1.4. Research Question and Thesis goal

In section 1.1 it is stated that the design process is of influence on the resulting design and that the efficiency of the design process might be improved. In section 1.2 the shortcomings of the Initiator are summarized and the desire for a more flexible and adaptive design process is expressed. Therefor the research question is formulated to be:

What is the influence of the conceptual aircraft design process on the resulting design and can the performance of the design process be improved?

In support of the research question a new aircraft design framework needs to be developed that adheres to the functional requirements expressed in section 1.3. The design of the architecture of this framework as well as the use of the framework to answer the research question are the goals of this thesis. It can be formulated as follows:

1. **Develop the architecture of a next-generation aircraft design tool fundamentally based on a process modelling approach**
    - Design the software architecture of a conceptual aircraft design framework
    - Development of the framework;
    - Implementation of an aircraft design sequence in the framework.
    - Implement dynamic, automated design process generation

Due to the limited time available for the proposed thesis research, it is infeasible to implement the entire software architecture that will be designed. Therefore when time is limiting, only the minimum functionality that is required to perform the second objective, will be implemented.

1.5. Report structure

In this chapter a general introduction to the topic is given. In chapter 2 background information concerning aircraft design software and process modelling is given. Then in Chapter 3 the methodology that was used in the thesis is presented. Chapters 4 and 5 contain the experimental setup and the results. Finally the conclusions and recommendations are presented in chapters 6 and 7 respectively.

2

Background information

In this chapter some background information is presented and its relevance to the research question is indicated. The information presented considers the topics of conceptual aircraft design software (section 2.1) and process modelling (section2.2). Also the an overview of some scientific computational frameworks is given in section 2.3.

2.1. Conceptual aircraft design software

To evaluate the current state-of-the-art for conceptual aircraft design software a survey was performed on the currently avaialble software packages. In chapter 1 the software from the department of Flight Performance and Propulsion (FPP) at the faculty of aerospace engineering at the TU Delft (TUDLR) was introduced. In this section a discussion of the most prominent features of available alternatives is presented.

2.1.1. Initiator alternatives

Many alternatives to the initiator are available, ranging from out-dated, empirical design tools to preliminary design tools relying on higher-order methods. These tools have been considered and their key aspects will be discussed in the consecutive sections:

- DEE [1];
- CAESIOM [10];
- VAMPzero [3];
- Prado [11];
- Pacelab APD [12];
- ACSYNT [13];
- Piano [14];
- RAGE [15];
- openVSP [16];
- MICADO [17]

2.1.2. Design software properties

Each of the conceptual design software packages listed in section 2.1.1 has its own unique set of features. The most striking features are discussed in this section.

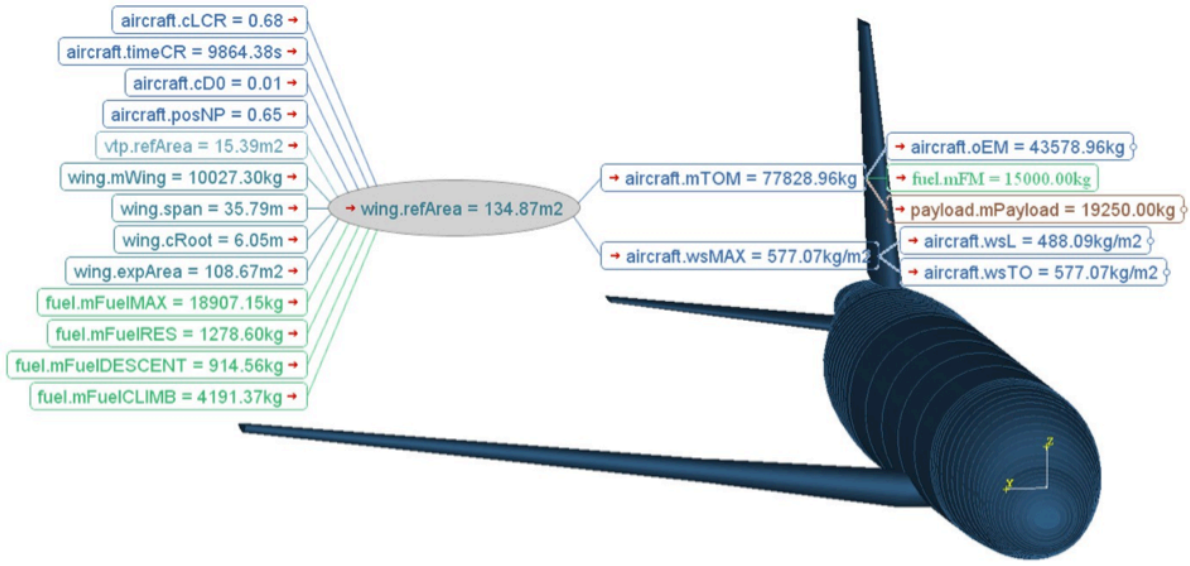

Figure 2.1: Graphical dependency tracking in VampZero [3]

Physic-based conceptual design

When designing unconventional configurations for which no empirical data is available it is essential to use physics-based analysis models to evaluate the design. Physics based models are used extensively in preliminary design tools and occasionally in conceptual design. Simple aerodynamic models like lifting-line theory or vortex-lattice methods are mostly preferred because of their short run-time. The Initiator is unique in the use of a class II.V weight estimation, which combines analytical analysis of the wing and fuselage structure and empirical data. Therefore this weight estimation is potentially very well suited for unconventional configurations, however this capability has not yet been implemented. Other tools like CAESIOM and Prado almost exclusively use physics based analysis. These are preliminary design codes where the use of such models is both essential and customary.

Modularity

For further development it is important that it is possible to add or modify analysis and design modules. For most design tools, being closed source, this is not possible. The Initiator, the DEE and VAMPzero allow extension of their toolkit. Adding modules however is not as trivial as would be desired.

Dependency tracking

An aircraft designer is naturally interested in what drives the resulting design: the driving factor provides valuable insight in the design. Room for improvement can easily be identified when the design drivers are known. A simple way to back-trace how the resulting design emerged is dependency tracking: what values are related to how they influence one another. VAMPzero lets the user track the dependencies for each design variable. This is presented visually as a mind map, as shown in Figure 2.1. This feature is a unique aspect of VAMPzero and not found in a similar way in other tools.

Variable order analysis

Variable order analysis is the use of models of different order during the same design process. For example: an initial estimate can be made based on empirical data, during the next stage a VLM model is used and finally a CFD analysis is performed. This example can be achieved using CAESIOM. Often the current state of the design and the required accuracy are driving the decision for the model to be used. In ACSYNT an interesting approach to this problem is taken. Several zones in an angle-of-attack vs. Mach number diagram are defined. Each zone is associated with an analysis model that is suited for aerodynamic analysis under the conditions of the zone. This idea is illustrated by Figure 2.

The downside of this approach though, is that discontinuities appear at the edges of the aerodynamic zones.

Another approach is used in VAMPzero. The tool interchanges modules of variable order based on the data that is available in the current aircraft model, thereby letting the tools order increase as the design progresses. VAMPzero itself is not capable of performing high order analysis like CFD. It is however possible to perform a higher-order analysis using any tool and store the results in the CPACS file-format used by VAMPzero. VAMPzero can then use the higher-order results replacing its lower-order analysis methods.

Formal, consistent data structure

A formal and consistent data structure is beneficial when integrating new or third party tools. It also helps preventing double work or redundant storage of data when it is clear what data should be stored where. VAMPzero is an excellent example of a consistent way of storing data: the CPACS format is formally defined in an XSD schema. This allows for storing aircraft geometry as well as performance parameters. An aircraft in CPACS can evolve from basic parameters to containing high-order results and detailed geometry. In VAMPzero the CPACS data model, or an extract of it, is exchanged between modules. An alternative approach, taken by the DEE and Prado, is to convert the central model into a generated set of sub models. These disciplinary specific models are then exchanged with the disciplinary analysis tools. In MICADO a strictly defined aircraft ontology is used and stored in their proprietary AiX file. The file functions as a central model for all modules in a way analogue to VAMPzeros' approach.

Data Storage In the previous paragraph data structuring of some aircraft design software is discussed. In this paragraph three technical approaches to storing data are discussed.

- In-Memory - Store data in the memory currently allocated to the software. This method is advised for short-lived data that is accessed frequently. Access to this data is lost after closing the software and not transferable between sessions/instances. In-Memory storage is implemented by having a data-model of the data and instancing the data-model. The data-model is then directly accessible from software. Since the memory allocated is limited by the available amount of Random Access Memory in a system, In-Memory can only be used for relatively small amounts of data.

- Database - A structured way of storing and accessing data for extended periods of time. The data is persistent between sessions and can be transferred between instances. The data can not be directly communicated efficiently to third parties or other systems. Database storage is implemented by setting up a database and running a database instance. Programming environments provide wrappers for accessing databases in a fast, efficient way. Traditional, SQL databases provide a fast mechanism to store and access structured data. For storage of less- or unstructured data NOSQL document databases are recommended.

- File storage - File storage is extremely slow compared to database/in-memory storage. File storage finds its application in data transmission between systems. Additionally it does not require the overhead of running a database instance and can therefor be preferred over a database for the sake of simplicity when speed is not of the essence. It is advised to adopt a standard document format such as JSON or XML to provide a structured way of storing data in a file.

Rapid geometry modelling

For quick evaluation of creative ideas for new aircraft concepts it can be a great advance to visually model the concept. After modelling the concept it can then be designed and analysed and the concepts feasibility can be evaluated. OpenVSP is an open-source parametric aircraft geometry modeller by NASA. The resulting model can be exported to a number of formats for further use. As OpenVSP is open source and integration of OpenVSP is welcomed, using OpenVSP in in-house design tools is possible. Another aircraft geometry modeller is RAGE by Desktop Aeronautics, where a solid parametric model is generated using a concise textual input file. The capabilities are comparable to the aforementioned OpenVSP. RAGE however, is commercial software and therefore not available for free.

Advanced design capabilities

The tools in Section 2.2.2 are all capable of producing a converged design. Some tools however are capable of optimizing the aircraft for a specific goal. ADS for example, allows optimization using 1 moving parameter. A different technique is available in VAMPzero. Parameters can be frozen/fixed during the design. Using this feature, parts of the aircraft can be kept constant. This allows for evaluating the influence of a certain part of the aircraft on the overall aircraft performance. ACSYNT employs a parallel design routine, compared to the default sequential design. Geometry, trajectory analysis and weight computation are evaluated in parallel. This gives the possibility of a large speed increase for the design routine to complete.

Modules

All tools mentioned in Section 2.2.2 contain at least the minimum toolset required to perform aircraft design. Some tools however incorporate additional analysis modules that provide more information about the design. An example is the IR analysis module in ACSYNT. This module is capable of analysing the IR footprint of an aircraft which can be vital in the design of stealth aircraft like fighters or bombers. Although not essential, this is a distinctive feature that gives ACSYNT the upper edge considering the design of stealth aircraft. CAESIOM contains an aero-elastics module. For certain aircraft aero-elastic effects can be driving and must be taken into account while performing the design. For all other aircraft, aero elastic behaviour must be checked and therefore this capability is critical in performing a valid design. Environmental aspects are becoming increasingly important. Lissys stresses the environmental emissions analysis as a unique feature of their aircraft design tool 'Piano'. Emissions might be driving the design during projects where reducing emissions is a goal. For all other designs emission data should not be omitted. Thus this module is a great asset to any tool. Both emission and noise capabilities are incorporated in MICADO.

Control system design

Control system design of conventional aircraft is a rather well-known process. Mostly a single solution exists and controls are allocated and sized easily. For unconventional aircraft like a three-surface aircraft, control allocation is trickier: due to the additional control surface infinitely many solutions exist for the allocation of control surfaces. When considering blended-wing body (BWB) aircraft, it becomes more complex: the concept often lacks a vertical and horizontal tail and the stability and control aspect might render the concept infeasible. AAA boasts excellent stability and control analysis for conventional aircraft. The analysis is based the use of an empirical database based on conventional aircraft and is therefore possibly not suited for the design of e.g. a BWB. The developers of CAESIOM have recognized the significance of control system design for new configurations. In CAESIOM special attention is paid to flight control system design for new configuration and it is incorporated early in the design sequence. This makes CAESIOM stand out considering the design of i.e. blended wing body aircraft.

2.2. Process modelling

In this section several approaches to modelling of the design process (or any process) are presented. The approaches are of interest when developing a new design strategy, in this case a new conceptual aircraft design process. Modelling of the design process also provides significant insight to the process itself: it allows for identification of bottlenecks or flaws in the process and may be helpful in the optimization of the process.

Most of the techniques presented here are often used to model processes and some have been used to model the aircraft design process. There are no design tools though which are fundamentally based on any process modelling approach.

2.2.1. Process modelling techniques

In this section several process modelling techniques are discussed, as is the relevance of the techniques to the goal of this thesis. The techniques that are being discussed are the (extended) Design Structure Matrix (xDSM), Graph Theory, Functional Dependency Table and Reconfigurable Multidisciplinary Synthesis (REMS).

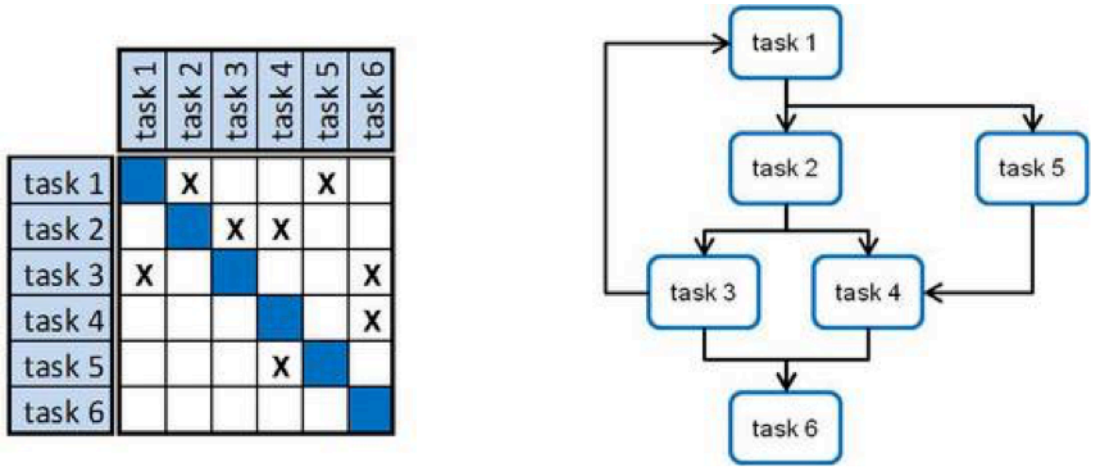

Figure 2.2: Example of a design structure matrix (DSM)[4]

Design structure matrix

A Design Structure Matrix (DSM) is a compact visualisation of a system and shows the interrelations that exist within the system. The approach was proposed by Steward in 1981 and has successfully been used since [4]. In Figure 2.2 an example of a DSM (left) is given, representing the process flowchart (right). A DSM is similar to an adjacency matrix known from graph theory. A DSM has the following useful properties:

- It is a compact visualization, also for a large number of processes/relationships;
- Because of the nature of its matrix representation, matrix-based techniques can be used to analyse and optimize the process;
- Clearly indicates feed-back and feed-forward relationships;

Extended Design Structure Matrix

The Extended Design Structure Matrix (XDSM) is an extension on the DSM discussed in the previous section. XDSM was developed by Lambe et al [5]. The XDSM is used for representing software architectures, where next to the data and processes represented by a DSM, also the process flow determined by the software architecture is shown. This is achieved using a numbering system and lines. An example is given in Figure 2.3. Additionally, optimization specific features like the objectives and constraints are included in the formulation, as well as special nodes for solvers and optimizers. This allows for a complete description of the software architecture.

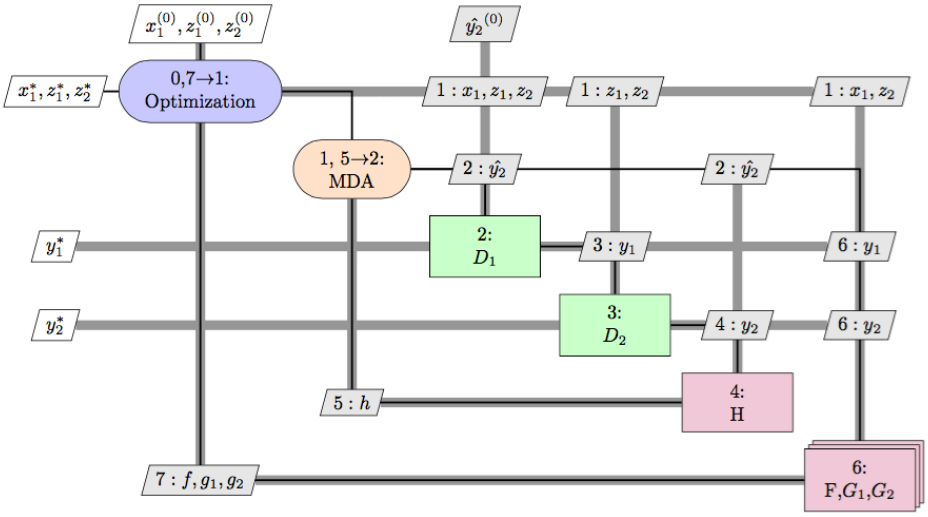

Figure 2.3: Example of an extended design structure matrix (XDSM) [5]

Functional Dependency Table

A drawback of a DSM is that a DSM contains no information on the goal or constraints that are active. This is required when performing optimization. To tackle this problem the Functional Dependency Table (FDT) was proposed by Wagner et al [18]. A FDT embodies the relationships between functions, including objectives and constraints. A drawback of the FDT is that the relations are unidirectional. Therefore the data-dependencies between modules are not captured. Therefore an FDT alone is not sufficient in describing the data-flow of a system.

REMS

Reconfigurable Multidisciplinary Synthesis (REMS) is a formal, abstract language for formatting MDO processes. REMS was proposed by Alexandrov and Lewis as a way of reasoning about an MDO problem at hand before solving the problem itself [19]. The concept is based on formulating the inputs and outputs of disciplinary modules while not considering the multi-disciplinary problem, separating the development of analysis modules and solving of the MDO problem. A directed graph based description of the data and functions of the problem, modelling both as nodes and using vertices to represent the data flow. Alexandrov and Lewis focus on the development of the abstract language that is proposed to describe the data and functions in a formal way. REMS allows for incorporating objectives and constrains as they can be modelled in a similar manner to analysis tools. It is portrayed by Pate et al as a combination of DSM and FDT. They also identify a major drawback of REMS: REMS is envisioned for the use in MDO, however it does not allow for inclusion of specific solvers or optimizers in the graph representation. Therefore specific solutions to specific problems cannot be described in full detail, forcing Pate to propose the graph based approach described in the previous section.

Graph Theory

Graph theory describes the mathematical subject of graphs. Graphs are a structured way to model relations between objects. Graphs exist of vertices (relations) and nodes (objects). Mathematical graphs are used extensively to model networks of data, communication or traffic. Many well-established methods and algorithms exist to perform operations on graphs, like finding the shortest path between two nodes or finding nodes related to a node. Additional functionality can be added to graphs by introducing the concepts of:

- Weighted graphs – assignment of weights to certain nodes/vertices
- Directed graphs – one-way relations

- Properties – additional semantics added to nodes/vertices

A simple example of a directed graph is shown in Figure 2.4.

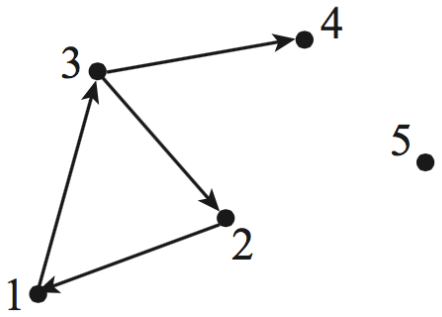

Figure 2.4: Example visualization of a directed graph [6]

Graphs can be a useful substitute of the commonly used DSM in representing a design process. Graph theory can then be used to find dependencies in a system, which can be used to select or define the appropriate design approach that has to be taken given a certain system state. To model a design system and all its modules Pate et al model both the modules and parameters in the design process as nodes in a graph. The following concepts are defined by Pate et al [6]:

- **Maximal Connectivity Graph (MCG)** – The graph containing all nodes (modules and parameters) and edges in the system. Because all nodes are contained all possible paths are present in the MCG. The MCG is the starting point for process analysis. It is also useful for finding so-called holes in the graph: required module input that is not available;
- **Fundamental Problem Graph (FPG)** – The graph that contains a minimal set of nodes and edges that are required to solve a problem. Unnecessary or redundant nodes are removed from the MCG until a minimal set of nodes remains;
- **Problem Solution Graph (PSG)** – The graph that contains both the FPG and solution strategy that is required to solve the problem. The PSG is a representation of the design process and shows the actual design steps that have to be taken.

The MCG is large by nature: Since every module, parameter and input/output relation is captured by the MCG only a single graph is possible. Depending on the structure of the MCG, one or more solutions exist for the FPG. In case of an MCG where no conflicting modules exist (modules related to the same output-parameter, i.e. modules on the same domain with different fidelity), only a single FPG is possible. In cases were modules conflict, multiple FPG scenarios are possible. The graph size of the FPG however is smallest: only the bare minimum number of modules is present to solve a problem. From the FPG a PSG is created. The PSG is the actual order of execution of the modules present in the FPG. A number of modules can be ordered in a fixed number of ways. If the FPG contains feedback however, the solution becomes of an iterative nature. The convergence speed of an iterative process depends on the process order: some ordering might prove to be more/less effective then others. Since theoretically inefficient process ordering can lead to a convergence speed that approaches zero, the number of iterations can be infinite (e.g. the iteration does not converge). Hence the number of possible PSGs is infinite (of course for every problem, one ore more optimal solutions exist). The graph size of the PSG is directly related to the number of iterations/module evaluations. Since it was stated that if the iteration does not converge the number of iterations can be infinite, the graph size can

be infinite. The minimal size of the PSG is the FPG: in case a FPG with no competing modules and feedback, the PSG is equal to the PSG. A visualization of relation of the graph size and number of possible graph between the MCG, FPG and PSG is available in figure 2.5.

Visualizing the MCG and the FPG is rather straightforward: the graph can be directly visualized as in figure 2.4. The visualization of a PSG is, in many cases, less straightforward. In cases where feedback is absent the PSG resembles the FPG and a visualization confirming to the style of figure 2.4 is valid. In a case where feedback is present and therefor an iterative approach is used a xDSM might be employed. However, this visualization approach fails if a more dynamic module ordering is applied: an xDSM only facilitates the visualization of loops of which the order does not change. If the order of modules in a loop, or the order of different loops, change, the process becomes much more dynamic and one can no longer speak of a purely 'iterative' approach. No meaning full visualization exists for a process of such a dynamic nature. Since this situation applies to most cases present in this research proper visualization of PSGs is lacking.

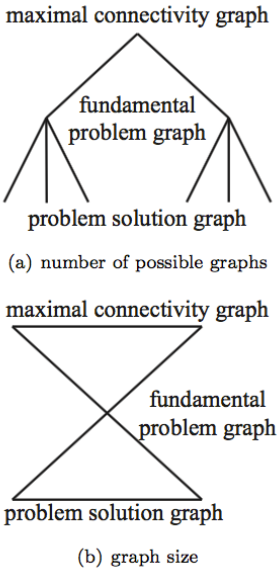

Figure 2.5: Relationship of the number of possible graphs (a) and graph size (b) between the MCG, FPG and PSG [6]

Pate et al conclude that the graph-based approach is, in comparison with other process modelling methods, very well suited for algorithmic analysis and manipulation. Additionally, graph theory contains many standard algorithms for feedback cycle detection, (minimum) spanning trees and shortest path problems. Since the goal of this thesis is to develop automatic process scheduling algorithms, graph theory is selected as the process modeling technique because of being well suited for algorithmic manipulation and analysis.

2.2.2. The significance of automated process modelling for conceptual aircraft design

As becomes clear while discussing the many modelling approaches in the previous sections, process modelling is a challenging subject and much research is spent on new approaches. Even though the design process is of great influence on conceptual aircraft design, it seems undesirable that the aircraft designer has to deal configuring the process. By automating the composition of the design process,

the aircraft designer can focus on the creative tasks and is no longer bothered with the composition of the design process. Automated process modelling may also be of great influence to the design of new aircraft configurations. Currently many aircraft design software employs a fixed design routine, based on design of the conventional aircraft configuration. This process might not be optimal, suited or even valid for new aircraft concepts. Different configurations may require a different design process or the use of different analysis modules. To get insight into the validity of the design process for unconventional aircraft configurations, automated process modelling can play an important role. It can be used to identify missing links or analysis tools. Additionally process modelling can provide guidance and direction for further development of design and analysis tools, such that unconventional configurations can be designed successfully and with confidence. Finally, having a detailed model of the design process, including all functions and data, great deals of potential advantages arise. For example, having detailed models of all input-output relations in the analysis modules makes storage of those results simple. This simplifies construction of surrogate models or estimation of derivatives, which might be useful in organizing the design process.

As mentioned before, no attempt has been made to fundamentally base an aircraft design tool on a process modelling approach, even though it is shown to have many potential advantages. Therefore a process modelling based approach might be very beneficial in the development of a new design tool.

2.3. Scientific computational frameworks

The scientific community has produced a vast amount of computational frameworks for scientific usage. These frameworks are meant to simplify linking different tools, performing optimization or facilitating distributed computation. The significance of these frameworks to this thesis is that they may provide a framework to support the development of a new design tool.

Many features are common in such frameworks (frameworks like openMDAO [20], Dassault iSight [21], Optimus [22], ModelCenter [23] and Dakota [24]). Among the most common features in such frameworks are:

- Workflow management, the functionality to set up custom workflows between interacting tools. The advantage of this feature is that data can be transferred between analysis tools;
- Optimization, Design of experiments and surrogate modelling. The frameworks provide functionality which make it trivial to set up these advanced design methods;
- Post-processing of the results. Visualization and plotting results in a convenient matter;
- Automated execution of simulation. A simulation work-flow that has been properly set up can be executed automatically, possible multiple times for varying input, without user intervention.

Some useful functionality is only implemented in specific frameworks. For example, ModelCenter comes with a rich analysis library. This can potentially reduce a lot of work when the content of this library is applicable to the project at hand. ModelCenter also provides integration with several CAD/CAE tools, useful in preliminary and detailed design. In the Dakota software uncertainty quantification is implemented out of the box. This is a valuable tool when developing new analysis tools and performing aircraft design using new tools.

Due to the closed source nature of most frameworks extending the frameworks features and implementing the planned aircraft design tool is non-trivial. Therefore they are not suited for the next-generation design tool we are considering, which leaves the open-source frameworks Dakota and openMDAO. Whether or not using one of these frameworks is wise will lead from the software architecture for the aircraft design tool, which is explained in detail in chapter 3.

3 Methodology

In this chapter the approach and methodology used in this thesis are explained. First the design philosophy for the software architecture of the Initiator is presented followed by the resulting architecture. Then the structure of the Python implementation is given. Finally the operation of the software is explained and the automatic module ordering algorithms are explained.

3.1. Initiator software architecture

First a preliminary design of the initiator software architecture is presented. The architecture is based on the requirements presented in [25].

3.1.1. Philosophy

The software architecture of the Initiator is design to comply with the functional requirements formulated in chapter 1. Two main concepts are adopted for designing the architecture: a separation of program components and a process modelling approach. The next two paragraphs are dedicated to explain this philosophy in more detail.

Separation of components The strict separation of components as presented in this section shows several advantages. The first is that it provides a great overview of the functionality and responsibilities of the different components. Secondly each component can be developed and tested as a stand-alone element. The input component for example can be developed without considering the internal functioning of analyses modules and the controller, among others. Because of clearly defined interfaces and stand-alone operation Unit-testing and regression-testing [26] can be used to guarantee and maintain quality.

Third, the current version of the Initiator is under heavy development my multiple TUDLR graduate students and staff. The separation of components eases the development of the software by multiple individuals simultaneously. By correctly using version control [27] like Git/SVN this process is further simplified.

Process modelling approach In section 2.2.2 the significance of process modelling in conceptual aircraft design is explained. Because of the advantages presented in that section a process modelling approach is adopted for this research. Consider the design process as a network of parameters and (computational) modules. Each module takes one or several parameters as its input and each module updates values for one or more parameters. In the network both parameters and modules are represented as nodes. A parameter that is input to a module is represented by a directional vertex from the parameter node to the module node. A parameter that is output of a module is represented by a directional vertex from module node to the parameter node.

3.1.2. Structure

The architecture of the Initiator should satisfy the requirements as specified in section 1.3 in a way that confirms the philosophy given in section 3.1.1. The Facade Design pattern as advocated by Zlobin

[28] is well suited for achieving a modular architecture in which there is little interdependence between different program components. According to Zlobin, the facade pattern:

- makes software easier to use and tests;
- reduces dependencies in the code;
- complicated subsystems are wrapped by a simple interface.

Thereby the pattern supports the philosophy of separation of components and is suitable for this application.

Components

To facilitate the separation of components the functionality of the framework is divided over different components. An overview of the program structure, containing the components listed below an their relations, is provided in figure 3.1. The figure clearly shows the separation of components that is so desired for reasons given in section 3.1.1. The following components can be distinguished:

- Controller component - The component that controls the operation of the initiator. The controller houses all other components and facilitates the connection and data passing between them.
- Driver component - The hearth of the Initiator. This is the component that takes input from the input component, load the analysis modules, communicates the output to the output component executes the design process.
- Input component - Component that takes input from the user and presents the input to the driver.
- Output component - Component that takes data from the driver and presents it. Presentation can be textual, visual or in any format, such as: console output, latex writer, graphs, plots, 2D/3D aircraft, CPACS XML, etc.
- Analysis/Design tool - The component that performs computations. This component is either internal or third party. It is wrapped by the module-component which is responsible for translating the data returned by the analysis tool to the format required by the driver. An example of this component is i.e. XFoil [29]. An analysis tool that has multiple modes of running it can be wrapped in different modules, each tuned for a specific run mode.
- Module component - The module component is a wrapper for an analysis or design module. This component translates input and output from the Initiators' format (required by the driver component) to the format required by the analysis tool. The wrapper also stores information about the computational result of the wrapped analysis tool such that it can be used at a later stage. A separate wrapper is intended for each different run mode of a analysis tool. This is done to ensure that the behaviour of the module is uniform, as well as the data gathered from running it.
- Instance manager component - The instance manager a component meant for managing instances of third party software. The overhead related to starting/stopping third party software is often substantial and sometimes dominant over the effective computational time. By keeping the instance alive after it has been used can reduce this overhead drastically for subsequent usage. The instance manager is also use full in managing the parallel execution of analysis modules when required. The component manages the different input and output files for each module such that no conflicts arise in concurrent existence of multiple instances of the same analysis tool.
- Meta-module component - The meta-module is meant for performing meta-modelling/surrogate modelling. These kinds of modelling techniques possibly reduce computational time by replacing a complex model by a simpler, statistical model, based on the complex models' stored input-output behaviour. These statistical method may vary from linear regression to Kriging. [30]
- Technology factor component - The technology factor component is a component that handles the impact of a technology factor on a parameter in the design of an aircraft.

- Algorithm component - The algorithm component embodies the design process algorithm. The component performs the scheduling of execution of modules for the controller to execute. For that the component receives module information for the applicable modules from the controller and present the execution schedule to the controller as a result.

3.1.3. Programming environment

Computational framework

In section 2.3 a brief overview of computational frameworks is presented. It is stated that the choice of a framework or the decision of building a proprietary framework leads from the software architecture of the design tool. The software architecture presented in section 3.1.2 does not conform to the structure used by the frameworks Dakota and openMDAO. Both frameworks are not sufficiently flexible to implement the suggested structure in the framework. Therefor it is suggested that no existing framework is used and that a new, proprietary framework implementing the given architecture is build.

Python

Several programming languages and environments were considered. Python [31] was selected for its large user-base in the scientific community. Python, being an interpreted language is easy to learn and can be executed in an interactive mode. Because it is interpreted the language is relatively slow compared to compiled languages like C++ and FORTRAN. Speed however, is not of the essence for the development of the framework itself: the computational modules are expected to be the performance bottle-neck, and the system will be set up such that these modules can be programmed in a high-performance language when required. Python is excellent for interfacing with other languages and therefor well suited for connecting modules in different languages.

Python supports modern programming paradigms and has a inherently modular setup, allowing to keep a large code-base manageable. Additionally there are plenty of packages available for performing all kinds of tasks, such as scientific an numerical programming or interfaces with third party software.

Finally Python is the programming language currently taught at TUDLR and therefor the learning curve for subsequent development is decreased significantly.

NetworkX

NetworkX [32] is a graph package for Python. The library features graph creation and manipulation. The package is open-source and therefor all algorithms and included functionality is accessible. Many default algorithms for i.e. path-finding, shortest-route and cycle detection are included in NetworkX.

The free availability, well-tested and easy integration of the package make NetworkX well suited for this study and is therefor used here.

Miscellaneous libraries

In addition to NetworkX, a number of other libraries were used. The libraries are not fundamental to the research and framework and are therefor not elaborated on any further. They are however listed below for the sake of completeness:

- pyGraphviz - Graph visualization
- Pint - Sciencitic unit framework
- Matplotlib - Plotting toolbox
- PrettyTable - Table printing to console and latex
- Unittest - Pythons internal unit testing framework

Data storage

On three occasions data storage occurs in the structure shown by figure 3.1: the aircraft data, module results/behaviour and for communication to third parties. For each type of data there is a preferred way of storage, based on the theory given in chapter 2:

- Aircraft data - The aircraft data is accessed continuously throughout the operation of the program. Considering the options presented in section 2.1.2 In-Memory storage is recommended.

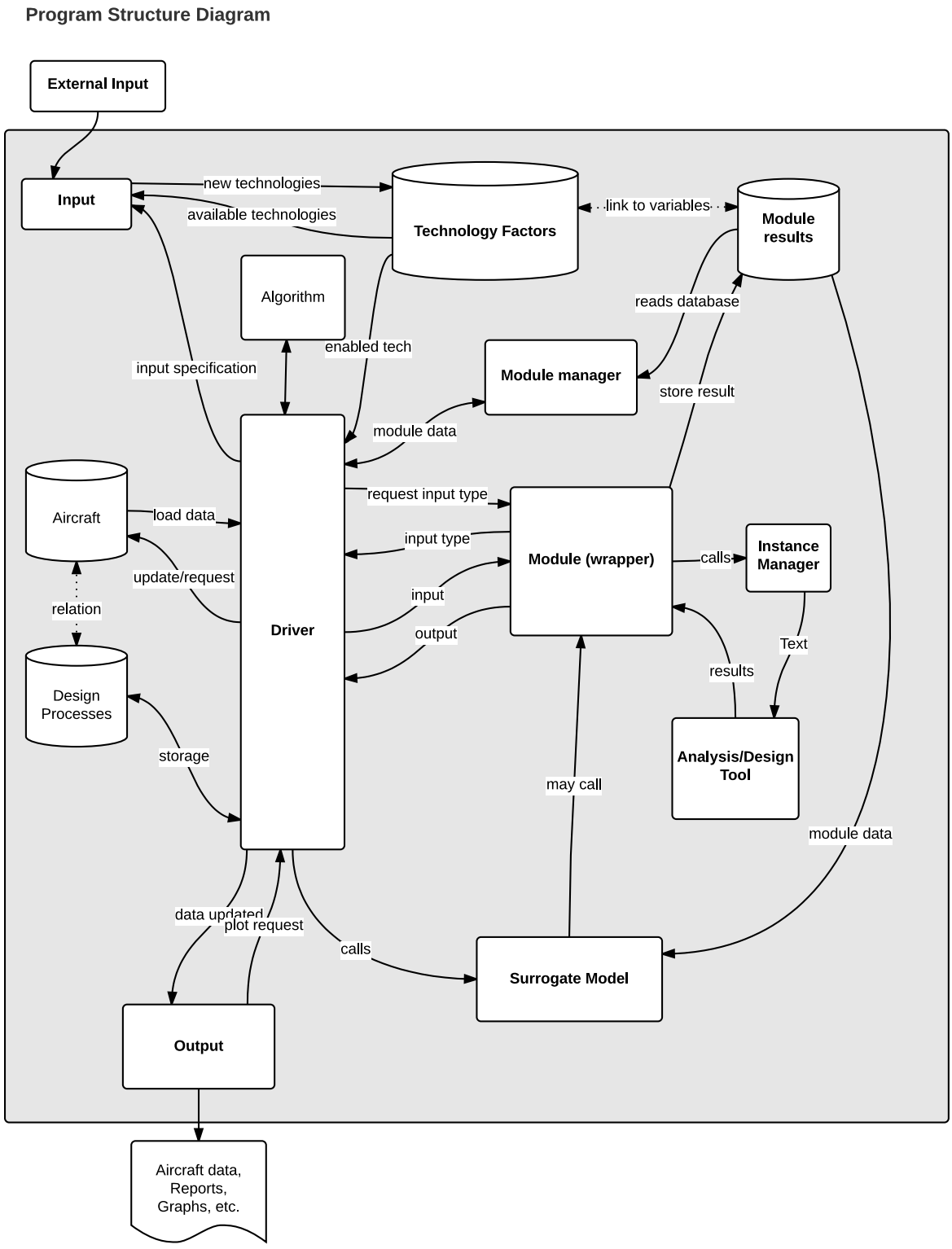

Figure 3.1: Overview of the architecture of the new Initiator

- Module results - Module results are high in volume but access occurs less frequently than accessing aircraft data. A database is advised for storing module results. Since a wide variety of module result formats is possible a documents database is advised in contrast to the structure SQL database.
- Communication to third parties - Given the portability of files compared to in-memory and databases explained in section 2.1.2 file storage is preferred to other options. The JSON document format is recommended because of its readability and small overhead compared to the XML format.

3.2. Program Structure

The software program that was developed for this research is a subset of the Initiator architecture that was presented in section 3.1. Because this research is focused on automated process modelling, the basic building blocks required to experiment with scheduling algorithms has been implemented. These essential building blocks are the driver component, algorithm component and the module component. This limited subset is represented by the diagram in figure 3.2.

This section explains basic concepts used in the implementation as well as a description of all the implemented components and design/analysis tools.

3.2.1. Concepts

To understand the terminology used in the subsequent sections some basic concepts must be introduced first.

- **Graphs** - A graph is an abstract way of modelling a network of arbitrary things (nodes) and their interconnections (vertexes).
- **Module** - A wrapper of computational design/analysis tool that is represented in the graph as a module-node.
- **Parameter** - A node representation in a graph of a variable in the design process. A parameter can be both input, output or a design goal of the design process.
- **Module error** - The difference of the sum of the normalized differences of a modules' output parameters' values of two consecutive evaluations of a module.
- **Input error** - The difference of the sum of the normalized differences of a modules' input parameters' values of two consecutive evaluations of a module.
- **System error** - The sum of the module errors' of the modules present in the FPG.
- **Module tolerance** - The maximum module error at which a module is deemed to be converged.
- **System tolerance** - The maximum system error at which the system (FPG) is deemed to be converged.
- **Convergence** - The process of approaching the system state where the system error has decreased to a value lower than the system tolerance.
- **Fixed parameter** - A parameter whose value can not be changed. Modules that have the parameter as an output are therefor not to be used: when used the constant value of the parameter is no longer guaranteed.
- **Holes (MCG)** - A hole in the MCG is a missing node. More specifically a hole is a missing parameter that is input to a module. This happens because the parameter is not defined but does occur as a input parameter of a module. The hole needs to be filled (e.g. the parameter needs to be defined) for MCG to be used.
- **Holes (FPG)** - A hole in a FPG is a parameter that is not set by any module and therefor can not be computed. An initial value needs to be set in order to solve the FPG.
- **Conflicting modules** - Conflicting modules are modules that share output parameters. Since multiple modules can set the value of the same parameter it is non-trivial to handle such a situation in generation an FPG.

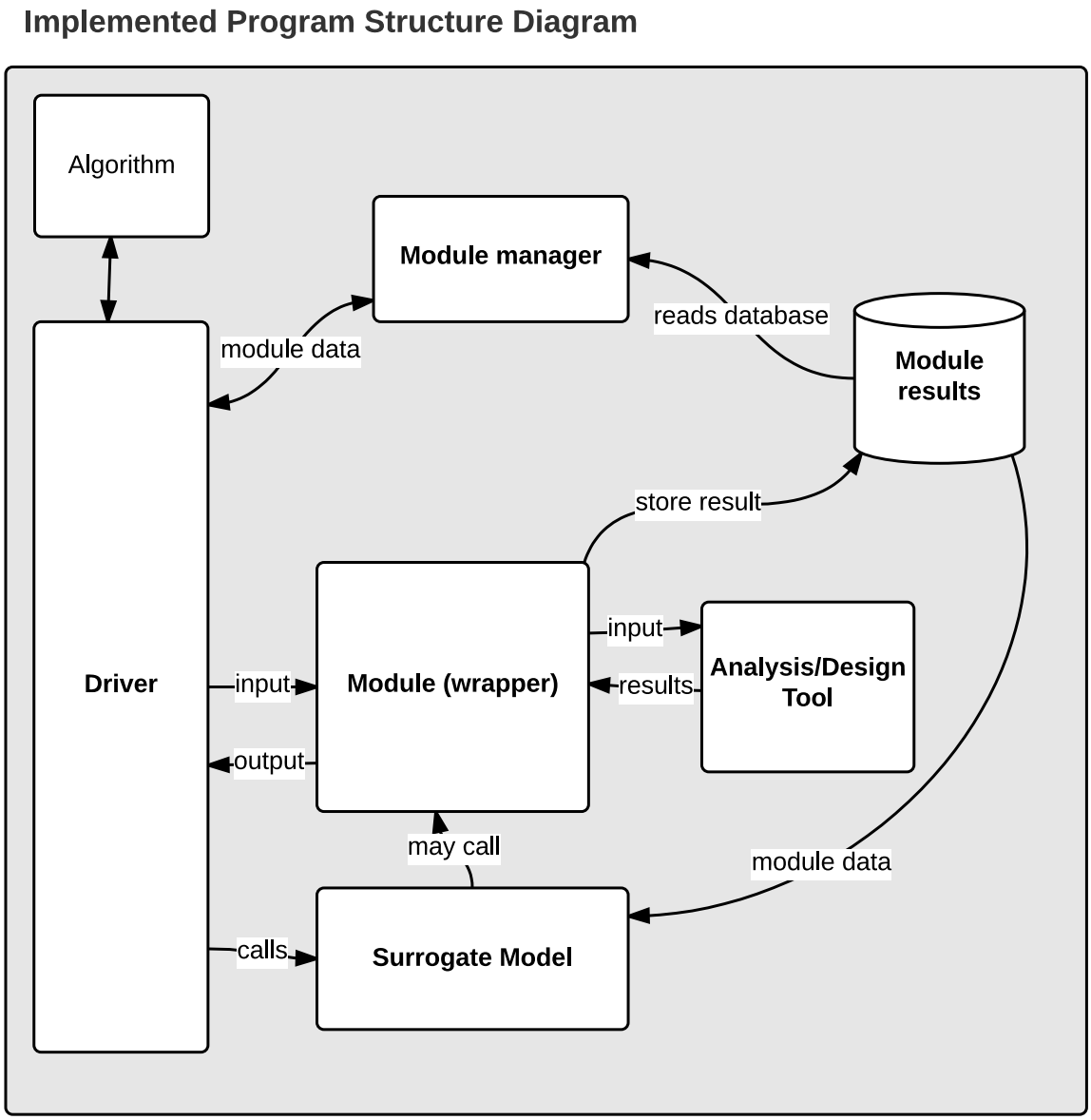

Figure 3.2: Overview of the implemented part of the new Initiator architecture

3.2.2. Components

In this section the implementation of the components in figure 3.2 is explained.

GraphController

The Graph Controller is the main component of the program. The graphController is an implementation of the 'driver' component as described in section 3.1.2. The graphController is responsible for loading parameters and modules, creating and maintaining the graph and running modules. Additionally the execution of a solving algorithm to perform the scheduling of modules is performed by the controller.

Nodes

The nodes are the basic elements of the graph. Both parameters as modules are modelled as nodes in the graph. Nodes are identified in the graph by their 'tag', a short abbreviation of their name.

Parameters A parameter is a design variable. A parameter has the following properties:

- tag
- name
- description
- unit
- value

Although the preferred way of storing parameter definitions is in a database (as advised in section 3.1.3, the overhead of setting up and running a database outweighs the speed advantage in this case, as speed is not of the essence in our test case. Hence parameters are defined in JSON format and stored in a file. As an example the definition of the gravitational acceleration parameter is given in listing 3.1.

Listing 3.1: Parameter definition example

```
{
    "tag": "g",
    "name": "Gravitational Acceleration g",
    "unit": "m/s^2",
    "description": "Gravitational pull, usually 9.81 m/s^2"
}
```

The module definition file is read by the graphController and for each defined parameter a parameter-node is created in the graph. A parameter keeps track of the history of its value. A complete list of implemented parameters is available in Appendix B.

Modules The module class defines the base class from which all module-components derive. The module class has the following data (and functionality to manipulate that data):

- Pointers to input/output parameters
- Run-time history
- Module state history
- Regression models of module behaviour
- Priority

The implementation of analysis/design tools is done by extending the module base class and implementing the mandatory init and runModule methods. In the init method the modules properties (name, description, input-output parameter tags) are defined. In runModule the execution of the analysis/design tool is performed as well as the mapping of input and output values to the analysis and design tool.

Implemented modules To experiment with automated process scheduling it is necessary to have a vast amount of different modules. Because ultimately the methods used in the current Initiator (version 2.7) are to be used in the next version, for which this framework is the base, it was chosen to replicate the methods in the current software. Due to time constraints however, it was not feasible to implement all modules present from the current Initiator. To have both a representative as a feasible set of modules with enough modules, the decision has been made to select the following design steps and split them up in smaller modules. The following design steps are implemented:

- Class I initial sizing / weight estimation
- Class I drag polar
- Design point selection
  - Thrust/weight ratio
  - Wing loading
- Class I configuration design - planform design
- Class II weight estimation

The design steps are split up in small modules. An exhaustive list of actual modules is available in Appendix B.

The underlying methods used in these modules are based on a combination of Toorenbeek and Roskam methods. Severe simplifications were made to speed up the development process. More information on the specifics of these methods can be found in the respective literature or in the documentation of the Initiator. Because these specifics are not relevant for this research no more discussion on the topic is given here.

To illustrate the code implementation of a module a brief example is given in listing 3.2. A more detailed example is the implementation code of class II wing weight estimation as by Toorenbeek. This example is given in appendix A.

Listing 3.2: Module implementation example

```
class root_tip_thickness(Module):
        # Calculate the root thickness of a wing based on
        # the thickness-to-chord ratio (considered a constant) and the
        # root chord of the wing.

        def __init__(self):
                # Module initializer. Use this init method to set basic module properties
                    like tag, name, description and in- and output-tags.
                tag = self.__class__.__name__
                name = "Estimate root and tip thickness of wing"
                description = "Estimate root and tip thickness of wing"
                self.inputTags = [ "t_over_c", "chord_root"]
                self.outputTags = ["thickness_root"]

                super(root_tip_thickness, self).__init__(tag, name, description, self.
                    inputTags, self.outputTags)

        def runModule(self):
                #Overloading of the mandatory runModule method specified in base class (
                    Module)

                # Calculation of root thickness
                root_thickness = self.inputs["chord_root"].value*self.inputs["t_over_c"].
                    value

                # Mapping of computed value to the output parameter
                self.outputs["thickness_root"].setValue(root_thickness)
```

GCTestcase

To perform experiments on using the new architecture the GraphController component needs to be set up and run in a controllable fashion. Therefor a suit of test case classes is designed that facilitates this need.

The GCTestcase, GCTestCaseCollection, GCTestCaseAlgorithmCollection and GCTestCaseRunner classes are designed to automate the initialization and running of (sets of) graphController objects. The Collection classes house a set of multiple GCTestcases in order to achieve statistically relevant sample size. Additionally the GCTestCaseReporter class is designed to perform a statistical analysis on the test results and report the results of a test-case collection. It is important to note that only a GCTestCase directly interacts with the GraphController. All other classes interact witht the GCTestCase and do not manipulate the GraphController directly.

GCTestcase The GCTestcase class was designed to generate, setup and run GraphController objects. A testcase contains a GraphController object and takes a list of initial values, goals and an algorithms for setting up the GraphController. The testcase can then run the graphController to solve for the defined goals and collect the results (goal values, run times and function evaluation).

GCTestCaseCollection The GCTestCaseCollection is a generator class that adheres to the factory design pattern as by by Zlobin [28]. Given a list of initial values for some paramters and bounds concerning these parameters, a given number of GCTestcase objects is generated. The purpose of this is to test the GraphController on varying initial values. Additionally it is meant to train the learning algorithms (detailed in the next section) by means of Design of Experiments (DOE).

GCTestCaseAlgorithmCollection The GCTestCaseAlgorithmCollection is a generator class like the regular GCTestCaseCollection class. The purpose of this class is to, given a list of algorithms and a sample size, generate and run a number of GCTestCases for each given algorithm. The results can then be collected to analyze them. A single sample per algorithm is not statistically relevant as only few algorithms produce a consistent result.

GCTestCaseRunner The GCTestCaseRunner is a helper class that manages the execution of a test case. The class has functions for running both a single testcase as well as an (algorithm) test case collection. A collection is run by treating each contained testcase as a single testcase that is run separately.

GCTestCaseReporter The GCTestCaseReporter class has the functionality to derive the relevant information from a GCTestCaseCollection and produce a readable result.

3.2.3. Program Operation

The implemented structure of the software program is presented in chapter 3.2. In this section, the chronological operation of the software is presented. Special attention is given to the communication between the software components and between the software and the user.

GraphController operation To clarify the operation chronology of the GraphController a sequence diagram of the typical operation of the GraphController (of which the structure and purpose is available in section 3.2) is given by figure 3.3. After instantiation the following steps are performed:

1. The Maximum Connectivity Graph (MCG) is composed from the defined modules (in code as illustrated in listing 3.2) and the defined parameters (in the parameters definition file, illustrated by listing 3.1).
2. After composing the MCG, the user is asked to identify the design goals. Alternatively these goals can also be specified by calling the GraphController object directly.
3. With the design goals set, the next step is to compose the Fundamental Problem Graph (FPG). The FPG is the minimal set of modules and parameters that are required to solve for the design goals.

4. The user is asked to specify the solving algorithm that is to be used to determine the module execution order. Again, alternatively this can be specified by calling the GraphController object directly.

5. The holes ('dead ends' in the graph) in the FPG are identified: these represent parameters which values can not be computed because there is no module available that has the parameter as an output. Therefor these values need to be set manually and can be considered the initial design values. The user is asked to enter values for every hole in the FPG.

6. The design problem is solved by executing the modules in the FPG until convergence is achieved. The graphController communicates with the ordering algorithm, which determines the module execution order. The exact implementation of the solving process heavily depends on the algorithm used. Some examples are presented in consecutive paragraphs.

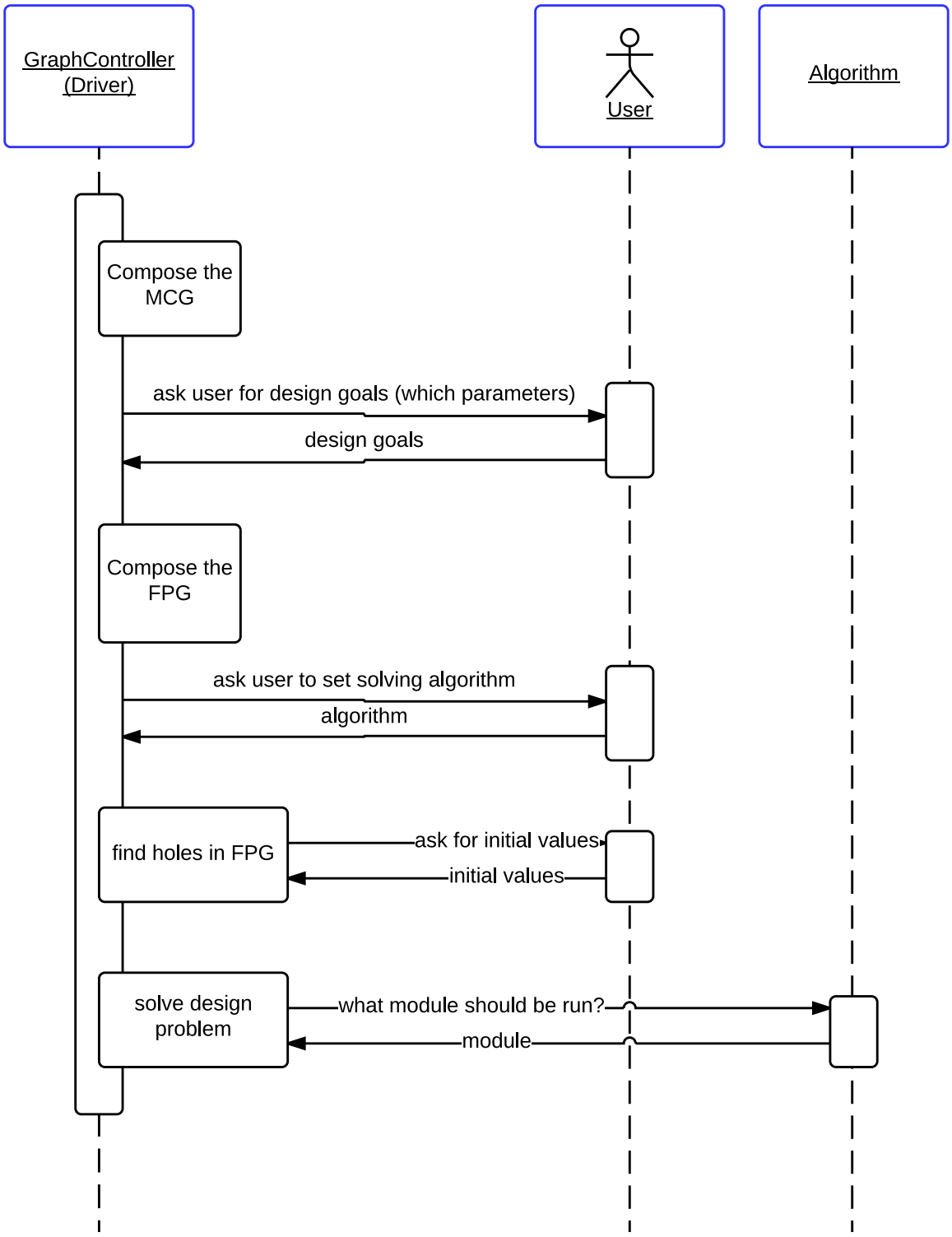

Figure 3.3: Sequence diagram of the typical operation of the graphController object

MCG composition In figure 3.3 the composition of the MCG is show as a key process in the Graph-Controller operation. The composition of the MCG is further explained by the flowchart in Figure 3.4. The following steps are taken:

1. Load parameters from parameter definition file
2. Load modules from code
3. Instantiate parameter/module objects from the loaded definitions. Add the instantiated objects as nodes to the graph
4. For every input/output parameter of every module, add a directed vertex from the parameter to the module (input) or from the module to the parameter (output) to the graph.
5. If a input/output parameter of a module is not defined in the parameter definition file, indicate this to the user and stop the program.
6. If all parameters are correctly defined then the MCG is complete.

As explained in chapter 2 the MCG is a graph network containing every possible connection between all nodes (parameters and modules). An example MCG is given in figure 3.5. Parameters are in black, modules in red and a possible design goal is highlighted in green. Note that a design goal is not strictly part of an MCG, but it is indicated to clarify the difference between a MCG and a FPG (discussed subsequently). Also note that only a arbitrary, yet limited number of modules is incorporated in this MCG to enhance the clarity of the graph. Hence not all modules listed in appendix B are present. Finally note that this network has no feedback/self-loops. The figure is generated automatically from an actual graph.

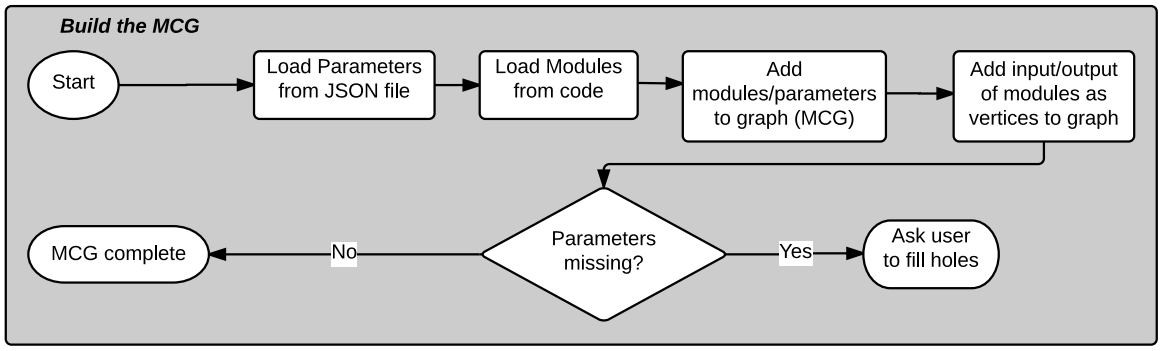

Figure 3.4: Flowchart of the composing of the Maximum Connectivity Graph (MCG).

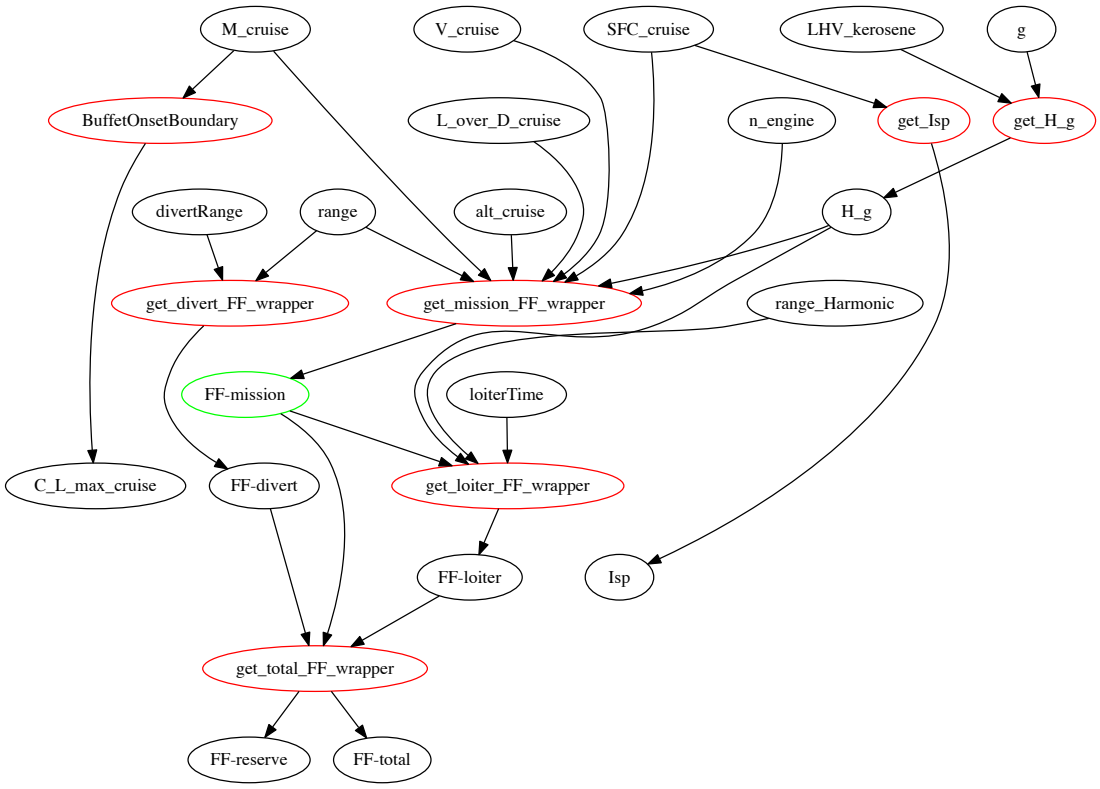

Figure 3.5: An example Maximum Connectivity Graph (MCG) given a limited set of modules (red) an parameters (black). The mission fuel flow ("FF-Mission") is highlighted as a example design goal in green.

FPG composition In a fashion comparable to the MCG, the FPG composition is an important process included in figure 3.3. The composition of the FPG is further explained by the flowchart in Figure 3.6. The following steps are taken:

1. Instantiate empty graph (FPG)
2. For every goal:
   (a) Add goal to the graph (FPG)
   (b) Recursively find all preceding nodes of the goal in the MCG and add those nodes to the FPG
3. The FPG is complete

By applying the method described by these steps and figure 3.6 given the design goal "FF-mission" and the MCG in figure 3.5, an example FPG is generated. This automatically generated FPG is presented in figure 3.7. It is immediately clear that only the nodes necessary to compute the design goal remain and that the FPG is smaller than the MCG, as predicted in section 2.2.1. Also note that the FPG is a special case in in which there are no self-loops (obviously, as there are none in the MCG).

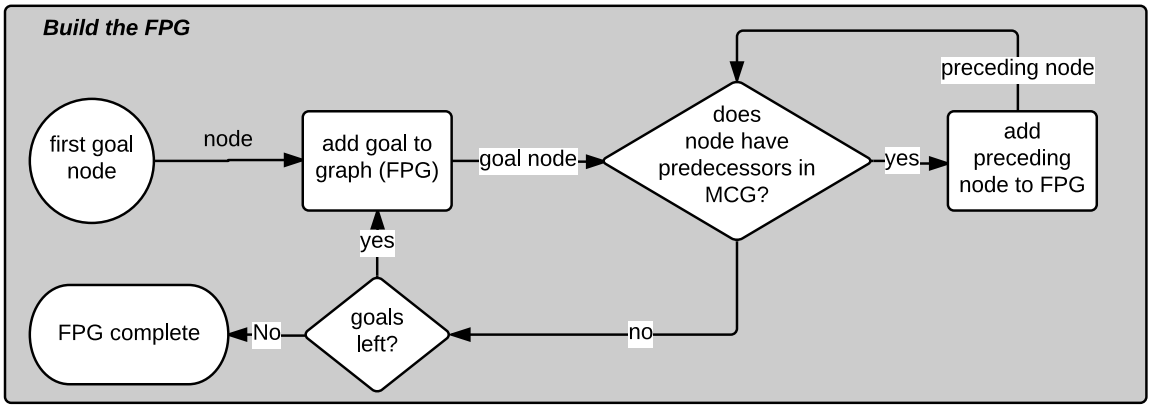

Figure 3.6: Flowchart of the compososition process of the Fundamental Problem Graph (FPG).

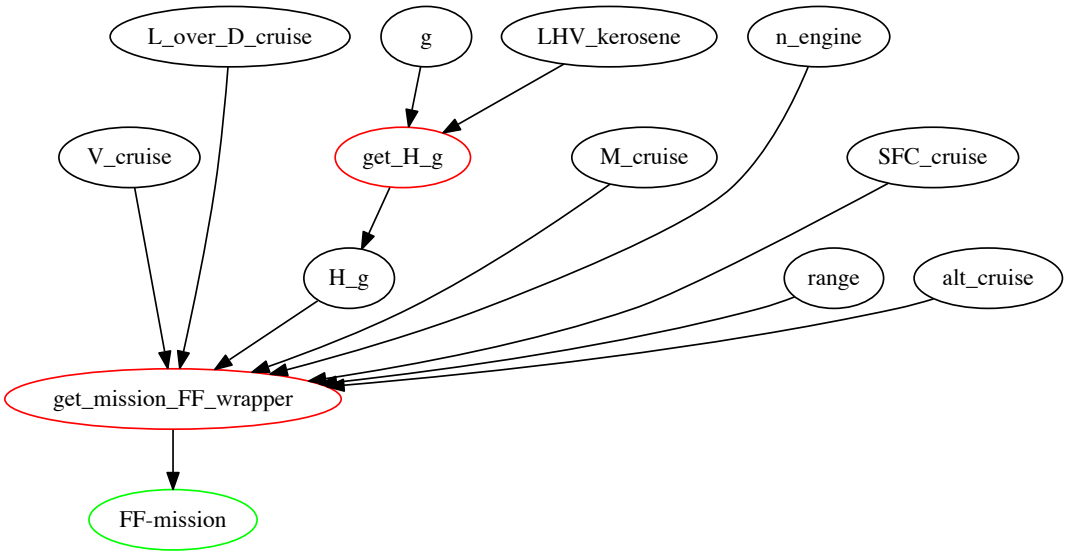

Figure 3.7: An example Fundamental Problem Graph (FPG) generated from an MCG 3.5 given a design goal (”FF-mission”), using the algorithm illustrated by figure 3.6.

GCTestcase operation In section 3.2.2 the overall structure of the GCTestcase class and related classes were given. The operation of the testcase suite is visualized in figure 3.8, indicating the chronological communication and data-flow between components of the testcase suite.

1. A GCTestcase Collection is instantiated
2. The user gives initial values, sample size and the algorithms to be used
3. The initial values and sample size(n) are presented to the testcase generator
4. n GCTestcases are instantiated given the initial values
5. For each GCTestcase, a graphController is instantiated given The intitial values and algorithm
6. The testcase collection is complete
7. Each GCTestcase in the GCTestcaseCollection is given to the GCTestcase Runner
8. The GCTestcase is executed: the graphController is asked to solve the design problem
9. The results are passed back to the GCTestcase (collection)
10. The results are passed to the GCTestcase Reporter
11. The GCTestcase Report performs a statistical analysis on the results and presents the results and the statistics to the user

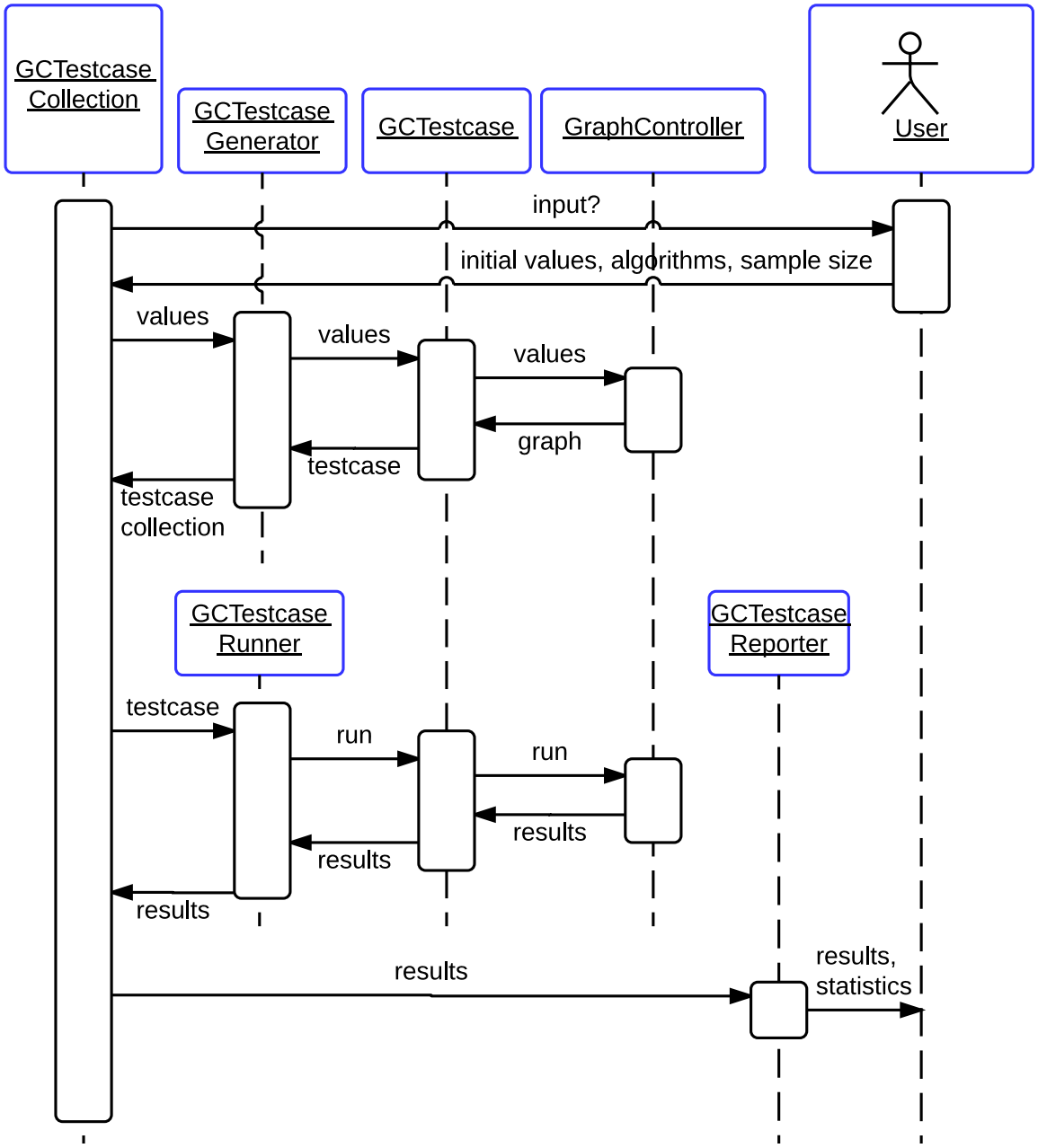

Figure 3.8: Sequence diagram of the GCTestcase suite, indicating the chronological communication and data-flow between components of the testcase suite.

3.3. Solving Algorithms

In section 3.2.3 the basic operation of the program was explained. A sequence diagram of the typical operation of a GraphController was presented in figure 3.3. Although details were presented on the composition of the MCG and the FPG no elaboration was given on solving the design problem (also: generating the PSG). This is the subject of this section.

To perform the automated scheduling of module execution several algorithms were developed with varying levels of sophistication. A base class (GraphAlgorithm) was made containing the basic logic of using an algorithm to schedule the module execution.

The algorithms listed in this section are display in an evolutionary order: the first algorithm contains the most basic functionality and was developed first. Development of the algorithms provided new insights and these have been incorporated in the development of subsequent algorithms.

3.3.1. Basic algorithms

Algorithm 1: Random order execution Modules are executed in a random order. This algorithm was developed to test the basic functionality of using an algorithm to determine the execution order. The goal is to reach a consistent, converged system. This approach uses a list of solved and a list of unsolved methods. Every module in the unsolved list is solved in a random order. If the change in value of the output parameters is significant, all modules depending on the output parameter are put back in the list of unsolved module and re-evaluated later. Because of the randomness of this algorithm it is expected that the algorithm performs terrible and scales very bad.

Algorithm 1: Randomly ordered modules

```
Data: FPG
Data: list of all modules that need to be solved
Data: list of modules that have been solved

for the first module in the list of modules that needs to be solved do
    if module can be solved then
        solve module;
        move module to list of solved modules;
        if (module error > tolerance OR first module run) AND module has predecessors then
            move all dependant modules of current module from solved list to unsolved list;
    else
        move module to the end of the list;
```

Algorithm 2: Random order execution with filtering of duplicate module execution This algorithm is an improvement of Algorithm 1 by adding filtering of the duplicate execution of modules. A major waste of computational time in Algorithm 1 occurs when a module is chosen to be executed but the values of the input parameters have not changed since the module was last executed. This behaviour is avoided by not executing the module and skipping to the next (randomly selected) method.

Algorithm 2: Randomly ordered modules with filtering of redundant module execution

```
Data: FPG
Data: list of all modules that need to be solved
Data: list of modules that have been solved

for the first module in the list of modules that needs to be solved do
    if module can be solved then
        solve module;
        move module to list of solved modules;
        if (module error > tolerance OR first module run) AND module has predecessors then
            move all dependant modules of current module from solved list to unsolved list;
    else
        move module to the end of the list;
```

Algorithm 3: Prioritizing cycles This algorithm is based on the assumption that convergence is reached faster when first all modules present in feedback loops (cycles) are considered first. Therefor after each module execution, for all cycles it is checked if they can be solved (meaning that there are no unset parameters in the cycle). If a cycle can be solved, the modules in the cycle are iterated upon until convergence is reached for the cycle. Then the next (solvable) cycle is considered.

Using the algorithm discovers that within the FPG there are many different cycles, as each cycle often contains multiple subcycles. This is because the parameters are also modelled as nodes in the graph. If there are 2 output parameters (C and D) of Module A in a cycle which are both input to Module B in the same cycle, this is counted as 2 cycles: 1 containing parameter C, 1 containing parameter D. Therefor a great many number of cycles must be converged upon, which leads to a sub-optimal result.

Algorithm 3: Prioritizing cycles

```
Data: prepare FPG
Data: list of all modules that need to be solved
Data: list of modules that have been solved
Data: list of all cycles that need to be consistent
Data: list of cycles that are consistent

if a cycle can be converged then
    for the first cycle in the list of cycles that needs to be solved do
        if cycle can be converged then
            solve cycle;
            move cycle to list of solved cycles;
            if cycle error > tolerance OR first cycle run then
                move all cycles with modules that are also in current cycle to unsolved cycle list;
        else
            move cycle to the end of the list;
else
    for the first module in the list of modules that needs to be solved do
        if module can be solved then
            solve module;
            move module to list of solved module;
            if module error > tolerance OR first module run then
                move all dependant modules of current module from solved list to unsolved list;
        else
            move module to the end of the list;
```

Algorithm 4: Prioritizing a selection of cycles This algorithm is an improvement on algorithm 3. That algorithm suffered from bad performance because of too many algorithms being present. A solution for this issue is to filter the circles that will be used in the algorithm. This filtering is performed as noted in algorithm 4.

The result of this filtering is that the number of cycles is reduced significantly, in the order of 80%-99%. As the number of duplicate cycles increases with number of nodes, the performance improvement

will increase with increasing number of nodes.

Algorithm 4: Prioritizing filtered set of cycles

```
Data: list of cycles
for cycle in cycles do
    remove parameters from cycle;
for cycle in cycles do
    if cycle is subcycle of other cycles then
        remove cycle from cycles;
do algorithm 3 with filtered cycles list;
```

3.3.2. Pre-run ordering of modules

The subsequent algorithms all apply the ordering of modules based on some criteria. The ordering occurs before the solving-process is performed. The problem solving process is then performed as in algorithm 2.

Algorithm 5: Pre-run, cycle-based ordering of modules It is assumed that the number of occurrences of a module in feedback cycles is an indication of the influence of that module on the design. By prioritizing the module with the highest number of occurrences the convergence speed is increased. Therefor this algorithm orders the modules based on the number of occurrences in the graphs cycles. The complete algorithm is given in algorithm 5.

Algorithm 5: Pre-run, cycle-based ordering of modules

```
Data: FPG
Data: list of all modules that need to be solved
Data: list of modules that have been solved
ordered list <- ordered by number of occurrences in FPG cycles
for the first module in the ordered list of modules that needs to be solved do
    if module can be solved then
        solve module;
        move module to list of solved modules;
        if (module error > tolerance OR first module run) AND module has predecessors then
            move all dependant modules of current module from solved list to unsolved list;
    else
        move module to the end of the list;
```

Algorithm 6: Pre-run, average distance-to-goals based ordering The assumption is made that the distance from a module to the goal(s) is a indicator for the impact of the module on the design. The reasoning is that a change in a modules' output propagates through consecutive modules. Therefor the impact on the total system is considered to be bigger for a module that has many succeeding modules (ergo: has a longer distance to the goal(s)). The distance from a module to a goal is computed

using Dijkstras' algorithm [33]. The calculation of the average distance is included in algorithm 6.

Algorithm 6: Pre-run, average distance-to-goals based ordering

```
Data: FPG
Data: list of all modules that need to be solved
Data: list of modules that have been solved

Data: totalPathLength = 0
Data: counter = 0

for goal in goals do
    if path from module to goal exists then
        totalPathLength += shortest path from module to goal;
        counter += 1;

average distance = totalPathLength/counter;

ordered list <- modules ordered by average distance of module to goal:

for the first module in the ordered list of modules that needs to be solved do
    if module can be solved then
        solve module;
        move module to list of solved modules;
        if (module error > tolerance OR first module run) AND module has predecessors then
            move all dependant modules of current module from solved list to unsolved list;
    else
        move module to the end of the list;
```

Algorithm 7: Pre-run, serial distance-to-goal based ordering This algorithm is a slight variation on algorithm 6 3.3.2. Instead of the average distance to multiple goals, the serial distance to the goals is use. The distance to each goal is stored separately: then sorting occurs first on the first goal, second on the second goal, etc. The full algorithm including the distance calculation is explained by algorithm 7. Note that this algorithm only varies from algorithm 3.3.2 in situations where there are multiple design goals defined.

Algorithm 7: Pre-run, serial distance-to-goal based ordering

```
Data: FPG
Data: list of all modules that need to be solved
Data: list of modules that have been solved

Data: totalPathLength = 0
Data: counter = 0

for goal in goals do
    if path from module to goal exists then
        distance[counter] += shortest path from module to goal;

order on distance[0] first, then distance[1], etc.

ordered list = ordered by sequential distance of module to goals:

for the first module in the ordered list of modules that needs to be solved do
    if module can be solved then
        solve module;
        move module to list of solved modules;
        if (module error > tolerance OR first module run) AND module has predecessors then
            move all dependant modules of current module from solved list to unsolved list;
    else
        move module to the end of the list;
```

3.3.3. Dynamic ordering of modules

The following set of algorithms uses what was called 'Dynamic' ordering. In stead of determining the module run-sequence once, the sequence is determined after every evaluation. Basically only the first item of that sequence will actually be executed, after which a new order will be determined.

Algorithm 8: Dynamic, module error based ordering This algorithm implements the dynamic ordering as explained in section 3.3.3. It is suggested that the magnitude of the module error indicates the improvement in consistency of the system that can be achieved by re-evaluating the module. Therefor in this algorithm the ordering is based on the module error. The procedure is given by algorithm 8.

Algorithm 8: Dynamic, module error based ordering

```
Data: FPG
Data: list of all modules that need to be solved
Data: list of modules that have been solved

for system error > system tolerance do
    order unsolved modules on: module error
    solve first module;
    move module to list of solved modules;
    if (module error > tolerance OR first module run) AND module has predecessors then
        move all dependant modules of current module from solved list to unsolved list;
```

Algorithm 9: Dynamic, input parameters error based ordering A similar explanation as for in section 3.3.3 holds for this algorithm: the normalized change (error) in input parameter values since the last evaluation of a module indicates the effect re-evaluation of the module will have. Therefor this algorithm orders modules based on the module input error. The procedure is given by algorithm 9.

Algorithm 9: Dynamic, input parameters error based ordering

```
Data: FPG
Data: list of all modules that need to be solved
Data: list of modules that have been solved

for system error > system tolerance do
    for module in unsolved modules do
        for input parameter of module do
            input error += input parameter[k]-input parameter[k-1]
        return input error

    order unsolved modules on: input error

    solve first module;
    move module to list of solved modules;
    if (module error > tolerance OR first module run) AND module has predecessors then
        move all dependant modules of current module from solved list to unsolved list;
```

Algorithm 10: Dynamic, run time based ordering It is suggested that the biggest gain in computational time is achieved by prioritizing fast modules with a short run time above those that take more time. Hence it is reasoned that by following this approach the computationally expensive modules are only executed when the computationally cheap modules are consistent. Thereby the evaluation of expensive modules is reduced to a minimum. Therefor in this algorithm the modules are ordered by

their average run time. The algorithm is presented in algorithm 10.

Algorithm 10: Dynamic, run time based ordering

```
Data: FPG
Data: list of all modules that need to be solved
Data: list of modules that have been solved
Data: runtime history of modules

for system error > system tolerance do
    for module in unsolved modules do
        load runtime history;
        return average runtime

    order unsolved modules on: average runtime

    solve first module;
    move module to list of solved modules;
    if (module error > tolerance OR first module run) AND module has predecessors then
        move all dependant modules of current module from solved list to unsolved list;
```

Algorithm 11: Dynamic, expected impact based ordering In this algorithm the effect of re-running a module on the module output parameters is estimated. It is suggested that the module which re-evaluation has the largest expected impact, should be prioritized because of its potential snowballing effect on consecutive modules.

The computation of the impact is based on a linear least squares estimate of the modules behaviour using the current input parameter values. The algorithm including the calculation of the impact is explained by algorithm 11.

Algorithm 11: Dynamic, expected impact based ordering

```
Data: FPG
Data: list of all modules that need to be solved
Data: list of modules that have been solved
Data: least squares model
Data: current input values

for system error > system tolerance do
    for module in unsolved modules do
        for parameter in output parameters do
            if Least squares model exists for parameter then
                least squares estimate using current input values;
                change = |current parameter value - estimated parameter value|;
                normalized change = change / current value;
            else
                skip to next output parameter;
        return ∑normalized change of output parameters

    order unsolved modules on: ∑normalized change of output parameters

    solve first module;
    move module to list of solved modules;
    if (module error > tolerance OR first module run) AND module has predecessors then
        move all dependant modules of current module from solved list to unsolved list;
```

Algorithm 12: Dynamic, expected profit based ordering This algorithm is a combination of algorithms 10 and 11 presented in section 3.3.3 and 3.3.3. The expected impact is divided by the average run time: the result is called the expected profit. This algorithm orders the modules based on this expected profit. The algorithm including the calculation of the estimated profit is given in algorithm 12.

Algorithm 12: Dynamic, expected profit based ordering

```
Data: FPG
Data: list of all modules that need to be solved
Data: list of modules that have been solved
Data: runtime history of modules
Data: least squares model
Data: current input values

for system error > system tolerance do
    for module in unsolved modules do
        for parameter in output parameters do
            if Least squares model exists for parameter then
                least squares estimate using current input values;
                change = |current parameter value - estimated parameter value|;
                normalized change = change / current value;
            else
                skip to next output parameter;
        load runtime history;
        return expected profit = ∑normalized change of output parameters / average runtime
    order unsolved modules on: expected profit
    solve first module;
    move module to list of solved modules;
    if (module error > tolerance OR first module run) AND module has predecessors then
        move all dependant modules of current module from solved list to unsolved list;
```

3.3.4. Fixed module sequence

Algorithm 13: Fixed module sequence Finally a fixed, pre-specified module sequence is implemented. This algorithm takes a pre-defined list of modules that can be run once or iterated until the set is consistent. The algorithm serves two purposes: first it provides a means to replicate the classical design process present in the current Initiator. Therefor it can serve as a benchmark for the other algorithms. The second purpose is to facilitate the storage of the resulting module sequences from other algorithms. By storing a sequence as in the format of the fixed module sequence, the stored sequence can be loaded as one and replicated exactly. The algorithm for executing a pre-defined

module sequence is given in algorithm 13.

Algorithm 13: Algorithm 13: Fixed module sequence

```
Data: runOnce, list of modules to run once
Data: iterate, list of modules to iterate

for module in runOnce do
    if module can be solved then
        solve module;
    else
        stop: the order is not feasible;
Data: i = 0
while i<2 || system error > system tolerance do
    for module in runOnce do
    if module can be solved then
        solve module;
    else
        stop: the order is not feasible;
```

4

Experiments

In chapter 3 thirteen algorithms were presented for ordering modules in the new framework. This chapter contains a set of experiments that are required to compare the different algorithms based on their performance. First the goal of these experiments will be presented in section 4.1. Then the test setup is established in section 4.2. Finally the test cases are established in section 4.3.

4.1. Goals

The research question as given in 1.4 questions the influence of the design process on the design result and whether the performance of the design process can be improved. The following questions can be formulated to support the research questions:

- Is the classical design process the best?
- Which approach is the best in which situation?
- Does the process influence the result?
- What behaviour is expected for the full initiator?

Therefor the goals of this experiment is to:

- show the effect of different design processes (by different algorithms) on the design result (goal values);
- quantify the performance of the different design processes.

4.2. Test setup

The experiment to answer to the goals set in section 4.1 is a computer simulation experiment. The different algorithms discusses in section 3 are compared. Some test cases are established first: a set of initial values and a design goal, as well as a benchmark fixed design sequence. Then each test case is given to each algorithm and the performance is recorded for comparison with other algorithms. In the oncoming section the details on the performance quantification, the initial values and the test cases is given.

4.2.1. Key Performance Indicators

In order to quantify the performance of a design process some Key Performance Indicators (KPIs) are declared that allow comparison of the different algorithms. The performance of an algorithm is quantified by the the following values:

- Runtime
  - Total runtime

  - Modules time
  - Overhead time
- Number of function evaluations
- Design goal value

Because there are multiple factors influencing the runtime of a module evaluation (and thus the runtime of solving a system), a number of samples will be taken and the values averaged. The following statistical quantities are calculated for the KPIs to assess the consistency of the different algorithms.

- Mean
- Standard deviation / Variance
- Margin of error

4.2.2. Sample size

Not all algorithms produce a consistent result each time they are used. Therefor the experiment is repeated a number of times to achieve a statistically valid sample size. The sample size for the experiments presented here is determined using equation 4.1 [34]. In the equation n is the minimum sample size for the experiment to be statistically relevant. $z_{\alpha/2}$ is the z-value belonging to the confidence interval, σ is the standard deviation and E is the margin of error. For this experiment a confidence interval of 95% is assumed to be sufficient, giving $z_{\alpha/2} = 1.96$. To determine E and σ, some knowledge of what results are to be expected is required. Since repeating the experiment is not time-consuming it is chosen to make and educated guess of the standard deviation and the margin of error. The resulting sample size is then used in the experiment. After the experiment, equation 4.1 is used to verify that the sample size was statistically valid. If not the case, a new guess can be made using the data of the experiment after which the experiment and the check of the validity of the sample size can be repeated.

From test runs during development the expected value (number of function evaluations) of the experiment is guessed to be 100. Two types of observations are to be made: how do algorithms compare Considering this value, a margin of error is accepted of 3%: if the values are withing 3% the difference in algorithm performance is considered to be negligible. On a value of 100, a standard deviation of 20 is guess to be reasonable. Using equation 4.1 and rounding the value up a sample size of 43 is calculated in equation 4.2.

$$n = \left(\frac{z_{\alpha/2} \cdot \sigma}{E}\right)^2 \tag{4.1}$$

$$n = \left\lceil \left(\frac{1.96 \cdot 10}{3}\right)^2 \right\rceil = \lceil 42.7 \rceil = 43 \tag{4.2}$$

4.2.3. Initial values

The initial values are a minimal set of design variables that are required to be able to eventually evaluate all required modules. The values that will be used are estimated values for the Boeing 737-800 aircraft as used in the current Initiator. This aircraft is chosen because it was also used as a testing configuration for the current Initiator. Note that the initial values are not particularly relevant for this study since we are mainly interested in the design process. The resulting design is less relevant and strongly affected by the many assumptions and simplifications made in the implementation of the design modules. Nonetheless the initial values are tabulated in table 4.1.

Two other settings are important for this experiment. The first is the choice of design goal. An arbitrary parameter that is within the main FPG feedback cycle is taken. The parameter that is chosen is the parameter wing area (tag: wing_area). Because the solving algorithms strive for consistence of the entire system of parameters and modules, it is rather irrelevant which specific parameter is chosen as long as they are in the same feedback cycle in the FPG. To ensure maximum usage of the modules that were defined the main feedback cycle was identified and the parameter wing area was chosen

from it. The second setting is the tolerance set on the system error. The effect of this tolerance is out of the scope of this experiment. A system tolerance of 0.01% is chosen.

Table 4.1: Initial values for design parameters used in the test cases of the experiments.

Parameter	Value	Unit
V_cruise	233.1	m/s
alt_cruise	11000	m
SFC_cruise	14.16	g/s/kN
FF-taxi	0.01	dimensionless
fuselage_nose_fineness_ratio	1.5	dimensionless
g	10	m/s^2
range_Harmonic	2500000	m
loiterTime	1800	s
LHV_kerosene	43500000	MJ/kilogram
n_engine	2	dimensionless
PAX	160	dimensionless
PAX_luggage_weight	1	kg
W_payload	18597	kg
range	2500000	m
FF-startup	0.01	dimensionless
divertRange	500000	m
rho_SL	1.23	kg/m^3
rho_cruise	0.25	kg/m^3
AR	9.45	dimensionless
maxLandingDistance	1600	m
maxTakeoffDistance	2000	m
n_max_cruise	1.3	dimensionless

4.2.4. Test system

The computer system on which the tests are run is a key to obtaining relevant timing results. Therefor the same system is used for all test cases. The test system is a Macbook Pro Late 2011 edition equipped with a 2.4Ghz Core i5 processor, 10GB of RAM and a Crucial MX100 250GB SSD drive. An attempt is made to run each test case at comparable circumstances, by taking the following measures:

- All other applications apart from the IDE are closed during the testcase evaluation.
- The computer will be run at a 100% battery level, whilst the charger is connected.
- The tests will be executed as shortly after booting the computer system as possible.
- To ensure a 'warm start' of the computer system and the runtime environment of Python, each experiment will be run twice. Only the results of the second evaluation are used for further analysis.

Nonetheless the computer system is not completely controllable and variations might be introduced by unknown artifacts. Therefor only the results obtained within the same session (test case) should be compared. Comparing the results of different sessions introduces an unknown uncertainty and is therefor not recommended.

4.3. Test cases

In this section more details on the test cases that are evaluated are given. Three test cases exist: Class I, Class II and Class II with extended run time. Each test case is explained below in detail.

4.3.1. Class I

The first test case is a class I design loop. This is the simplest feedback situation possible given the current modules. In the Class I module set there is no competition between modules: every parameter has only one module that can set its value. The benchmark situation is a fixed module sequence as shown in listing 4.1. The design goal is the maximum take-off weight. The test case is to be applied to all algorithms presented in section 3.3.

Listing 4.1: Class I fixed module sequence

```
    "modules": {
      "runOnce":[],
      "iterate": [
        # Class I
        "get_H_g",
        "get_divert_FF_wrapper",
        "get_mission_FF_wrapper",
        "get_loiter_FF_wrapper",
        "get_total_FF_wrapper",
        "get_MTOW_wrapper",
        "BuffetOnsetBoundary",
        "design_point_selection",
        "calc_wing_area",
        "wing_span_class_I",
        "wing_sweep_class_I",
        "wing_taper_class_I",
        "root_and_tip_chord",
        "fuselage_size_estimate",
        "fuselage_wetted_area_estimate",
        "wing_exposed_area_estimate",
        "wing_wetted_area_estimate",
        "aircraft_wetted_area_estimate",
        "equiv_parasite_area",
        "drag_coef_incompressible",
        "get_L_over_D_max"
      ]
    }
```

4.3.2. Class II

The second test case is a class II design loop. The class II element of this test case is the class II weight estimation. The module sequence can be read in listing 4.2. Note that this sequence does include competing modules. The class I *get_MTOW_wrapper* which determines the MTOW based on fuel fractions and an aircraft database. The competing module is the class II method *class_II_aircraft_weight_estimation* in which the MTOW is computed from the class II wing weight and the aircraft weight without the wing. The ordering algorithms handle this competition by a manually set priority indicator: for each module a priority can be set (an integer number). At each instant, the module with the highest priority that can be evaluated is used and the modules with a lower priority discarded. In this case the priority of the class I module is 1 and the priority of the class II module is 2. The design goal is the maximum take-off weight. Initially the class II experiment was to be applied to all algorithms described in section 3.3. Development of the test cases however, lead to new insights: algorithms 1-8 are not capable of coping with competing modules in a successful way and can therefor not complete the class II experiment. Therefor the only algorithms 9-13 are subjected to the class II test case.

Listing 4.2: Class II fixed module sequence

```
"modules": {
  "runOnce": [
    # Class I
```

```
    "get_H_g",
    "get_divert_FF_wrapper",
    "get_mission_FF_wrapper",
    "get_loiter_FF_wrapper",
    "get_total_FF_wrapper",
    "get_MTOW_wrapper"
  ],
  "iterate": [
    # Class I
    "BuffetOnsetBoundary",
    "design_point_selection",
    "calc_wing_area",
    "wing_span_class_I",
    "wing_sweep_class_I",
    "wing_taper_class_I",
    "root_and_tip_chord",
    "fuselage_size_estimate",
    "fuselage_wetted_area_estimate",
    "wing_exposed_area_estimate",
    "wing_wetted_area_estimate",
    "aircraft_wetted_area_estimate",
    "equiv_parasite_area",
    "drag_coef_incompressible",
    "get_L_over_D_max",

    # Class II
    "get_Empirical_Mass_Estimates",
    "w_mzf_estimation",
    "root_tip_thickness",
    "wing_weight_estimation",
    "class_II_aircraft_weight_estimation"
  ]
}
```

4.3.3. Class II - simulated module runtime, Type I

During test runs of the first two test-cases the issue arose that run time of the algorithms was dominated by the overhead time. This is mainly due to the shear simplicity of the implemented modules. Many assumptions were made to speed up development and that resulted in very simple and fast modules. Additionally the Initiator modules have been subdivided into smaller modules to arrive at a substantial number of modules. This does not correctly represent the state of the current Initiator and its modules. Where the modules in the current Initiator take anywhere from 0.1 to 20 seconds (on average) their simplified counterparts take less than 0.1 seconds.

The dominance of the overhead distorts the results: algorithms that have a significant overhead may produce a design process with a better performance, but the increase in performance does not show in the result because it is negligible compared to the algorithm overhead. Therefor a third test case was made. The third test case used the same module sequence as the Class II test case (listing 4.2). Each module evaluation however, is appended by a Python *wait* command to delay the completion of the evaluation. The delay has can be arbitrary but has to be constant for a given module such that it does not change between runs. The number of input/output parameters of a module is taken as a indicator for the numerical complexity of a module. Two different distributions of wait time will be used: one based on the number of input parameters (Type I), one based on the number of output parameters (Type II). Another measure of estimating the dummy wait time is the priority of the module. As explained the priority of a module is used facilitate the decision which competing module to execute. It is estimated that the number of input/output parameters to the power of the priority (p) gives a reasonable distribution of the wait time among modules. To reduce the total run time of the experiment, the resulting wait time is scaled linearly. A reduction by a factor of 30 is estimated to result in a run time where the overhead time is no longer dominant and the run time of the experiment is reasonable. This results in the equation 4.3 for the Type I experiment.

$$t_{wait} = \frac{\left(n_{input}\right)^{p^2}}{30.0} \tag{4.3}$$

In section 4.3.2 it was explained that progressive insights gained during development of the test

cases lead to restricting the tested algorithms for the class II test case. The same analogy holds for this test case. This test case is further restricted: additional insights showed that algorithm 10 performs an order of magnitude slower than algorithms 9, 11-13. Because of the long runtime this algorithm is not suitable for this test case, where further scaling of the run time is simulated. Therefor algorithm 10 is excluded from the test case and only algorithms 9 and 11-13 are subjected to the class II test case with simulated run time.

4.3.4. Class II - simulated module runtime, Type II

In section 4.3.3 a test case with simulated runtime was presented and motivated. It was explained that the simulated (dummy) runtime is distributed according to equation 4.3. Following that line of thought another test case is devised to asses the influence of the dummy runtime distribution. Therefor this test case is identical in almost every way to test case Class II - simulated module runtime, Type I. The difference is in the distribution of the dummy runtime among the modules. Instead of basing the dummy runtime on the number input parameters the dummy runtime is based on the number of output parameters of a module. This results in equation 4.4 for the Type II experiment.

$$t_{wait} = \frac{\left(n_{output}\right)^{p^2}}{30.0} \tag{4.4}$$

5

Results & analyses

In this chapter the results from the experiments explained in 4 are presented. Following the result a short analysis and interpretation of the results is given. The conclusions can are drawn from these results are given in chapter 6.

5.1. Results

5.1.1. Class I

The Class I experiment represents an iterative, Class I design sequence. In section 4.3.1 the details concerning the experiment are given. The results of the Class I test case are tabulated in table 5.1.

Table 5.1: Class I test case results (n = 50)

#	Algorithm	$\bar{t}_{run}$ (s)	μ_f	σ_f	$\bar{t}_{comp}$ (s)	$\bar{t}_{overhead}$ (s)	E (%)
1	Bruteforce	0.48	213	71.1	0.17	0.31	9 %
2	Bruteforce No Wasted Evals	0.42	302	60.5	0.17	0.25	6 %
3	Bruteforce Cycles	0.58	374	69.9	0.22	0.36	5 %
4	Bruteforce Selective Cycles	0.5	452	79.1	0.18	0.31	5 %
5	Ordered Cycle Based	0.44	333	70.5	0.17	0.27	6 %
6	Order Avg Distance To Goal	0.16	108	0.5	0.06	0.1	0 %
7	Ordered Serial Distance To Goals	0.16	108	0.5	0.06	0.1	0 %
8	Dynamic Error Based	0.45	201	5.6	0.06	0.39	1 %
9	Dynamic Input Error Based	0.3	134	0	0.07	0.23	0 %
10	Dynamic Time Based	0.37	113	0.9	0.04	0.33	0 %
11	Expected Change Based	0.22	108	0	0.06	0.16	0 %
12	Expected Profit Based	0.24	108	0	0.06	0.18	0 %
13	Fixed Module Sequence	0.17	126	0	0.06	0.11	0 %

The first observation that can be made is that the modal goal value is equal for all algorithms. It should also be noted that out of 12, only 5 algorithms are consistent in their goal value. This inherently means that the generated design process is not equal at each simulation. Both artifacts are highly undesirable. The number of occurrences of the modal number of function evaluations shows the same trend, confirming that the design process is varies with the same test setup.

The table shows the run time, computational time and overhead time. What is instantly clear is that for the Class I case performance is marginally increased by the expected change/profit and pre-run

ordering algorithms. Another observation is that the overhead is roughly half of the total run time. This confirms the expected distorting influence of the overhead time as was explained in chapter 4.

It is also indicated by the results that the lowest overhead is found for operations that perform the scheduling of the modules pre-run. This is obvious since this reduces the number of scheduling operations to a minimum.

The experiment also showed that the algorithm, and therefor the design process, has no influence on the design outcome: a value of 145 m^2 for the wing area was found by every algorithm. In the description of the experiment in chapter 4 a margin of error of 3% was selected. The difference in number of function evaluations between algorithms 1-5 and 9-13 though, is considerably larger than 3%. Hence it is valid to state that the number of function evaluations (50) is sufficient.

5.1.2. Class II

The Class II experiments represents a simplified version of the process used in the current Initiator. The benchmark in this experiment the fixed module sequence replicated from the Initiator. The details of this experiment are available in section 4.3.3 and the results are available in table 5.2.

From the results it shows that just the run times of algorithms 9 and 11 are within the error margin of 3%. Therefor nothing can be said about the difference in run time between the two. The other test results are outside the margin of error and are further analyzed below.

Table 5.2: Class II test case results (n = 50)

#	Algorithm	$\bar{t}_{run}$ (s)	μ_f	σ_f	$\bar{t}_{comp}$ (s)	$\bar{t}_{overhead}$ (s)	E (%)
9	Dynamic Input Error Based	0.41	156	0	0.06	0.35	0 %
10	Dynamic Time Based	2.83	936	61.3	0.07	2.75	2 %
11	Expected Change Based	0.41	195	14.7	0.07	0.33	2 %
12	Expected Profit Based	0.38	156	0.5	0.06	0.32	0 %
13	Fixed Module Sequence	0.19	126	0	0.05	0.14	0 %

First note that only 2 algorithms are consistent (standard deviation = 0) and that other algorithms do not consistently find the same process. From the results it is immediately clear that the fixed module sequence outperforms the other algorithms. Although the difference in computational time is marginal (20-40%) the number of function evaluations is significantly lower for the fixed sequence. When compared to i.e. the Dynamic Time Based algorithm it clearly shows how a preference of fast modules leads to a high number of functions evaluations (936) vs. the number of 126 evaluations for the fixed sequence.

The most striking observations regarding the results is the dominance of the overhead time in the total runtime. The share of overhead ranges from 71% to 97%. Therefor the assumption made in section 4.3.3 that the overhead dominates the results is thereby confirmed and the class II experiments with simulated runtime are required to remove the dominance of the overhead time.

The experiment again showed that the algorithm, and therefor the design process, has no influence on the design outcome: a value of 162 m^2 for the wing area was found by every algorithm. Note that the wing area is different from the Class I wing area. That is no surprise, since the module set is not the same.

5.1.3. Class II - simulated module runtime, type I

The Class II experiments with simulated module run time represents a more practical situation where the module runtime is longer and where the total runtime is not dominated by the overhead. The experiment is further explained in 4.3.3. The results can be found in table 5.3.

From the results of the type I experiment (table 5.3) it shows that the overhead times of algorithms

9 and 12 are within the error margin of 3%. Therefor no statements are made about the difference in overhead time between algorithms 9 and 12. The other test results are outside the margin of error and are further analyzed below.

Table 5.3: Class II with simulate run time test case results, type I (n = 50)

#	Algorithm	$\bar{t}_{run}$ (s)	μ_f	σ_f	$\bar{t}_{comp}$ (s)	$\bar{t}_{overhead}$ (s)	E (%)
9	Dynamic Input Error Based	26.54	156	0	26.14	0.40	0 %
11	Expected Change Based	29.06	195	19.23	28.67	0.38	3 %
12	Expected Profit Based	24.88	157	0.55	24.48	0.40	0 %
13	Fixed Module Sequence	16.96	126	0	16.10	0.86	0 %

The first thing to consider for this experiment is the ratio between the computational time and overhead: the largest percentage of overhead occurs for the fixed module sequence (5,3%). Thereby the overhead no longer dominates the timing results as found in the results in tables 5.2 and 5.1.

Again it is found that just 2 algorithms produce a consistent result: the fixed module sequence and the dynamic input-error based algorithm. From the table it is also clear that the fix module sequence is at least 34% faster than the best performing algorithm.

Finally it is remarkable that the overhead time of the fixed module sequence has increased with respect to the results in table 5.2. Since not much logic is present in the fixed module sequence algorithm, overhead is expected to be minimal as was seen in tables 5.1 and 5.2. The overhead has however increased by 514% whereas the number of operations that contribute tot the overhead (one scheduling operation for each function evaluation) is constant. One reason for this is that overhead (and run times in general) are highly dependant on the state of the computer system: available memory and other background processes. As was stated in chapter 4 measures were taken to ensure a constant computer state. Complete control over the state however, is not possible and therefor variations might be introduced between sessions (and therefor, between test cases). This does not explain however why the overhead of algorithm 13 is approximately twice the overhead of algorithm 9,11 and 12.

5.1.4. Class II - simulated module runtime, type II

As explained in section 4.3.3 two different distributions of module run time are tested. The results can be found in table 5.4.

From the results of the type II experiment (table 5.4) it shows that just the overhead times of algorithms 9, 11 and 12 are within the error margin of 3%. Therefor nothing can be said about the difference in overhead time between the three. The other test results are outside the margin of error and are further analyzed below.

Table 5.4: Class II with simulate runtime test case results, type II (n = 50)

#	Algorithm	$\bar{t}_{run}$ (s)	μ_f	σ_f	$\bar{t}_{comp}$ (s)	$\bar{t}_{overhead}$ (s)	E (%)
9	DynamicInputErrorBasedConvergence	10.362	156	0	9.72	0.63	0 %
11	ExpectedChangeBased	11.35	195	15.147	10.731	0.624	2 %
12	ExpectedProfitBased	9.40	156	0.503	8.78	0.623	0 %
13	FixedModuleSequenceClassII	7.32	126	0	6.87	0.45	0 %

The results in table 5.4 of the Type II test case show the same trend as the Type I test: algorithm 13 performs best, followed by algorithm 12 and 9. Algorithm 11 performs the worst. This shows that the distribution of dummy runtime has no noteworthy effect on the result. The overhead does not dominate the total run time.

5.1.5. Resulting design processes

In sections 5.1.1-5.1.4 the timing results were given related to the test cases defined in chapter 4. In this section the resulting design processes are considered. The resulting design processes are presented in a graphical way that resembles a punch-card. The presented results are from the class II test case and the following algorithms:

- Algorithm 13: Fixed module sequence: figures C.1-C.3, appendix C;
- Algorithm 11: Dynamic Expected change based: figures C.4-C.9, appendix C;
- Algorithm 9: Dynamic input error based: figures C.7-C.10, appendix C.

Note that the results of Algorithm 12 are not presented in appendix C. From table 5.4 it is clear that algorithms 9 and 12 produce process that has the same number of function evaluations. Inspection of the resulting process yields that both processes are identical. Therefor the process of algorithm 9 is included, 12 is omitted.

The process resulting form Algorithm 13 is obvious: an iterative pattern, repeating modules in the same fixed order until convergence is reached. Obviously the six leftmost modules in figure C.1 are run only once: it was instructed as is clear from listing 4.2. The resulting processes of algorithms 9 and 11 show a trend that somewhat resembles the trend of algorithm 13. Where they differ from algorithm 13, the difference can be expected from their nature. Algorithm 11 seems to prefer modules with many output parameters that are in the feedback cycle. This logically follows from its purpose: prefer modules with greater impact on the design. Therefor algorithm 11 shows a behaviour that can be expected.

In both algorithms a trend is visual: the diagonal 'line' that defines the process of algorithm 13 is partially present in algorithms 9 and 11. Since algorithm 13 follows a logical flow through the FPG it is obvious that algorithms 9 and 11 resemble this trend in some way.

What is key to the timed performance of algorithms 9 and 11 is that the minimum amount of function evaluations for a module not in a feedback loop is 6 (equiv_parasite_area). Apart from this module, the minimum is 7: equal to the number of iterations required for algorithm 13. Little is gained by evaluating modules not affected by feedback (fuselage_wetted_area_estimate) only twice whereas this module is included in the feedback loop of algorithm 13.

5.2. Analyses

In this section an interpretation of and a reflection on the results presented in section 5.1 is given.

In chapter 4, a set of 4 questions was formulated in support of the research question. These questions are repeated here for convenience:

- Is the classical design process the best?
- Which approach is the best in which situation?
- Does the process influence the result?
- What behaviour is expected for the full initiator?

From the results we can clearly see that the fixed module sequence, based on the classical design process used in the current initiator, outperforms the other algorithms in all test-cases. This shows that the classical design sequence is well optimised for the given design case. However, no formal proof exists to conclude that the classical design approach is the best. The results show that in each test case the same mean goal value is achieved. Therefor it is concluded that for the tested cases the design process has no influence on the outcome of the design. It does however show that the design process is of great influence on the computational time. The fixed, optimized case is much faster and requires

less function evaluations than the other algorithms. The result that the generated design processes are inferior to the fixed design routine shows that there is room for significant improvement of the algorithms.

For the full initiator, two options can be pointed out. The first is that the fixed design sequence will then implement an optimal sequence as is the case for the class I and II test case. Therefor the other algorithms will again be outperformed. However, the selection of class I and II implemented here are the most well-tested and well documented design methods and therefor more likely to be close to the optimal design sequence. For the full initiator however, many more modules are added to the system. The number of possible processes increases exponentially and the chance of the current Initiator process of being the optimal one becomes significantly smaller. Therefor second option is that the possibility exists that the new algorithms will perform much better for the full design case.

Lets consider what has been done: a system has been designed in which modules can be individually defined without any knowledge of the design process. Using the algorithms specified in 3 the system is capable of generating a (sub-optimal) design process without any prior domain knowledge. Although an optimal solution is not found, the system is expected to perform similarly given any other set of modules and test case. If it were presented with computational modules for, lets say, satellite design, it is expected to also generate a feasible design sequence. Consider the design of new aircraft configurations: Blended wing bodies and Prandtl planes. Although relatively similar, their different configuration might require a new design method/sequence to efficiently design such aircraft. It is expected that the system designed in this thesis can generate this design method (given sufficient, applicable computational modules).

Finally another remarkable advantage of this thesis' system must be highlighted. During the current development of the Initiator new computational modules are added continuously. The process of adding a module is rather crude: it is placed in the design sequence based on a best guess of the module developer. Using the result of this research a new module can be automatically integrated in the design sequence, placed feasibly by one of the algorithms. Although the results show this might not provide an optimal design process it certainly provides a guideline for designing an optimal process.

6

Conclusions

The goal of this research is to investigate the influence of the conceptual aircraft design process on the resulting design and to see if the performance of the design process can be improved. To answer to this goal, the software architecture of a next generation aircraft design tool has been formatted. From the results presented in chapter 5 is can be concluded that:

- The designed system is capable of generating a feasible (sub-optimal) design process out of stand-alone computational modules;
- The classical conceptual, fixed design process, implemented as specified in chapter 4, outperforms the design sequences generated by the algorithms given in section 3.3 by:
  - 20-40% for the Class II test case;
  - at least 34% for the Class II test case with simulated, extended module runtime.
  - if the used set of modules is equal, the order of the design process does not influence the result.
- Out of the algorithms presented in section 3.3:
  - only the dynamic input error based algorithm is consequently consistent;
  - the algorithms that perform ordering of the modules perform the best for the simple design case with no conflicting modules (class I test case);
  - the expected profit based and dynamic input error based algorithms perform best for the more complex design case with conflicting modules (class II test case);
  - the expected profit based algorithm performs best for the design case were long module run times were simulated;

It also must be concluded that the implemented set of modules does not provide an representative situation for the entire conceptual design process of the Initiator. The set that was implemented is well tested and coherent: therefor the fixed design process performs very well and outperforms the ordering algorithms.

Finally it is concluded that the performance of the fixed design processes used in the test cases can not be improved by the algorithms that were developed for this thesis.

7

Recommendations

Following from the conclusions from section 6 a number of recommendations can be made. The first recommendation is related to the current applicability of the system. The system designed in this thesis can be used as a tool to investigate feasible design processes for unconventional aircraft configurations. Although the algorithms produce sub-optimal design processes, a functional, converging design process is generated that can be a useful first start when designing a process.

Secondly it is recommended that a larger part of the current initiator is implemented(preferably all modules). As was explained in section 5.2 the implemented part of the initiator is well tested and optimized. Hence the fixed module sequence outperforms the design processes generated by the algorithms. It is expected that the algorithms that were developed yield better results for a more complex system, such as the full initiator.

Third, more test cases with a bigger variety of input values should be used to verify the behaviour of the system. It was assumed that the system tolerance, the initial values and the design goal have no influence on the results of the experiments. This however, is not verified. Therefor test cases with different initial values, system tolerances and design goals should be set up to asses the validity of this assumption.

A fourth recommendation is to implement modules for design of an unconventional aircraft, preferably physics based: a situation for which no proven, well-tested design process exists. This system was designed for such a situation and is expected to be a great help in designing such a process. Therefor the modules and test case to facilitate this situation should be implemented and tested.

A fifth recommendation is related to the syntax and practical use of the software. The syntax of defining parameters and modules is rather verbose in the current setup. This causes a lack of overview during development, caused by the fact that modules were split into smaller modules. That approach is very reasonable given that a complex network of modules was to be developed in a short time span. When implementing the full initiator however, this approach is no longer valid: the amount of boilerplate code would dominate the source code and the desired clarity and overview while developing modules would be lost. Therefor special attention should be paid to how much functionality should be contained in a 'module'. It is recommended to group code based on domain and functionality. It is expected that this creates a balance between overview and functionality.

Another recommendation is that a structured data model is designed for the software. The process modelling approach ensures that knowledge on all parameters is available at any time. The data model is flexible, but can not be modified ad-hoc: both features are desirable. In this thesis the aircraft data model is a loose collection of parameters. Considering the relatively small amount of parameters (50-100) this approach is feasible. The Initiator however has thousands of design variables. These need to be ordered in a structured manner in order to maintain an overview. It is suggested that a hierarchy is developed that is inspired by a common aircraft data format, such that transferring data

from the aircraft model to third party systems is simplified.

A sixth recommendation is to develop a simple GUI for the software. Although not discussed in this thesis, the GUI should facilitate the inspection of processes, parameters and modules. It should be aimed in providing developers/researches with information about the current state/capabilities that are implemented in the system. The information can reduce conflicts and reduce duplicate effort during development: a feat that is common when multiple entities develop a system together. The GUI should also include a simple parameter-manager to facilitate creating and editing parameters.

It is also recommended that the collection of ordering algorithms is extended. One of the aims of this research is to benefit from the suitability of graph theory for algorithmic analysis and manipulation. In the algorithms however, only use a limited amount of the capabilities of graph theory is used. Therefor it is suggested that new algorithms are defined that exploit graph theory more. An example is Dijkstras' shortest path algorithm: if proper weights (i.e. estimated impact of a module) are assigned to vertices in the graph, the shortest path algorithm could be used to estimate the snowballing effect of consecutive module impacts. This might yield an algorithm that does not take the impact of a single module into account, but the impact of chain of modules. Another example is to apply an optimization to the currently employed fixed-module sequence. This sequence could be taken as a starting point, adding or removing modules from the sequence when required (based on some analysis).

The final recommendation of this thesis is that the philosophy and software structure presented in chapter 3 be used for a next version of the Initiator. The architecture as presented in section 3.2 supports the requirements posed in section 1.3. The architecture employs clear separation of functionality and responsibilities and is based on a process modelling based approach. Hereby the approach provides a maintainable, testable architecture well suited for the simultaneous development by different researchers. The process modelling approach allows for the analysis of existing and generation of new design processes and has the potential to improve the design capability/efficiency of the Initiator.

A

Listings

A.1. Module implementation example

Example implementation of a module component. The example shown here is the (Class II) wing weight estimation module.

Listing A.1: Module implementation example

```
class wing_weight_estimation(Module):
        """Class II wing weight estimation"""

        def __init__(self):
                """Module initializer. Use this init method
                to set basic module properties like tag, name,
                 description and in- and output-tags. """

                tag = self.__class__.__name__
                name = "Estimate wing weight"
                description = "Calculate wings weight"
                self.inputTags = ["wing_sweep", "W_MZF", "wing_span",
                    "wing_area", "n_ultimate", "thickness_root"]
                self.outputTags = ["wing_weight"]

                super(wing_weight_estimation, self).__init__(tag, name,
                    description, self.inputTags, self.outputTags)

        def runModule(self):
                """Mandatory runModule method. This method is called to execute
                the analysis/design tool wrapped by the module."""

                i = self.inputs

                #mapping of input parameters to python variables
                wing_sweep = i["wing_sweep"].value
                W_MZF = i["W_MZF"].value
                b = i["wing_span"].value
                wing_area = i["wing_area"].value
                n_ultimate = i["n_ultimate"].value
                thickness_root = i["thickness_root"].value

                # Torenbeek method for commercial trainsport Airplanes as
                in Roskam part V, p.69 eq. 5.7 and 5.8
                wing_weight = self.getWingWeight(wing_sweep, W_MZF ,b,
                    wing_area, n_ultimate, thickness_root)

                # mapping analysis tool output to parameter
                self.outputs["wing_weight"].setValue(wing_weight)

@staticmethod
def getWingWeight(wing_sweep, W_MZF ,b, wing_area,
    n_ultimate, thickness_root):
```

```
    """
    Toorenbeek method for commercial trainsport Airplanes
        as in Roskam part V, p.69 eq. 5.7 and 5.8

    :param wing_sweep: leading edge wing sweep in degrees
    :param W_MZF: Maximum zero fuel weight
    :param b: Wing span in meters
    :param wing_area: Wing area in square meters
    :param n_ultimate: Ultimate load factor (usually 1.5)
    :param thickness_root: Root thickness (m)
    :return:wing weight: wing weight as by Toorenbeek
    """
    return 0.0018 * W_MZF * (b / math.cos(wing_sweep)**0.75
        * ( 1 + ( 6.3 * math.cos(wing_sweep) / b)**0.5)
            * n_ultimate**0.55 * (b * wing_area /
            ( thickness_root * W_MZF * math.cos(wing_sweep)))**0.30)
```

B

Implemented parameters & modules

B.1. Parameters

In this section the parameters that were implemented are given. The parameters are tabulated in table B.1.

Table B.1: Table of all parameters that are implemented

Tag	Name
altitude	Altitude
S_over_S_wet	Area ratio
C_L	lift coefficient
C_L_max_cruise	Maximum lift coefficient during cruise
C_L_max_TO	Maximum lift coefficient during take-off
C_L_max_L	Maximum lift coefficient during landing
C_l	Section lift coefficient
H_g	Calorific value of fuel / g
crewMember_weight	Crew member weight
W_crew	Crew weight of aircraft
L_over_D_cruise	Cruise Lift-to-drag ratio
M_cruise	Cruise Mach Number
V_cruise	Cruise Velocity
alt_cruise	Cruise altitude
SFC_cruise	Cruise specific fuel consumption
V_cruise	Cruise velocity
W_DL	Design Landing Weight
distance	Distance
FF-divert	Divert Fuel Fraction
divert_range	Divert Range
E	Endurance
FF-taxi	FF-taxi
FF-total	FF-total
FF-cruise	Fuel Fraction for cruise-segment
FF-loiter	Fuel Fraction for loiter-segment
FF-mission	Fuel Fraction for mission
FF-other	Fuel Fraction for other segments
W_fuel	Fuel weight
g	Gravitational Acceleration g
range_Harmonic	Harmonic Range
loiterTime	Loiter Time
V_loiter	Loiter velocity

LHV_kerosene	Lower Heating Value
L_over_D_max	Maximum Lift-to-drag ratio
W_ML	Maximum landing weight
W_MTO	Maximum take-off weight
W_MZF	Maximum zero fuel weight
W_fuel_mission	Mission Fuel Weight
n_engine	Number of engines
PAX	Number of passengers
W_OE	Operative Empty weight of aircraft
PAX_luggage_weight	Passenger luggage weight
PAX_weight	Passenger weight
W_payload	Payload weight of aircraft
W_ramp	Ramp Weight
range	Range
FF-reserve	Reserve Fuel Fraction
W_fuel_reserve	Reserve fuel weight
Isp	Specific Impulse
FF-startup	Startup Fuel Fraction
S	Surface area
S_wet	Wetted area of a part
W_ZF	Zero Fuel Weight
divertRange	divertRange
rho_SL	Air density at sea level
rho_cruise	Air density at cruise level
AR	Aspect ratio
ThrustLoading	Thrust Loading
WingLoading	Wing Loading
maxLandingDistance	Maximum landing distance
maxTakeoffDistance	Maximum take-off distance
n_max_cruise	Maximum load factor in cruise
Thrust_total	Total thrust force
Thrust_per_engine	Thrust force per engine
wing_area	Wing area
PAX_per_Area	Number of passengers per area
wing_span	Wing span
wing_sweep	Wing sweep angle
wing_taper	Wing taper ratio
chord_root	Wing root chord
chord_mean_geometric	Wing mean geometric chord
chord_tip	Wing tip chord
MAC	Mean Aerodynamic Chord (MAC)
t_over_c	Thickness to chord ratio of the wing
wing_exposed_area	Wing exposed area
wing_wetted_area	Wing wetted area
fuselage_nose_length	Fuselage nose length
fuselage_tail_length	Fuselage tail length
fuselage_diameter	Fuselage diameter
fuselage_center_length	Fuselage center length
fuselage_wetted_area	Fuselage wetted area
aircraft_wetted_area	Aircraft wetted area
fuselage_tail_fineness_ratio	Fuselage aft fineness ratio
fuselage_nose_fineness_ratio	Fuselage nose fineness ratio
c_f	c_f: parasite friction drag coefficent

aircraft_eq_parasite_area	Aircraft equivalent parasite area
C_D_0_incompressible	Zero lift incompressible drag coefficient
oswald_factor	Oswald factor
thickness_root	Wing root thickness
n_ultimate	Maximum load factor
wing_weight	Wing weight
W_A-W	Aircraft without wing weight

B.2. Modules

In this section the modules that were implemented are given. The parameters are tabulated in table B.1.

Table B.2: All parameters that are implemented in the system.

Module tag	**Module name**	**Input parameters**	**Output parameters**
get_W_payload_wrapper	Compute aircraft payload weight	PAX, PAX_luggage_weight, PAX_weight	W_payload
get_MTOW_wrapper	Class I AC weight estimate	W_payload, FF-total, range	W_MTO, W_OE, W_A-W
get_Empirical_Mass_Estimates	Initial mass estimates	FF-mission, FF-reserve, FF-total, FF-startup, FF-taxi, W_MTO, range	W_fuel, W_ZF, W_ML, W_ramp, W_DL, W_fuel_mission, W_fuel_reserve
get_H_g	Compute H_g	LHV_kerosene, g	H_g
get_mission_FF_wrapper	Mission Fuel fraction	range, H_g, n_engine, L_over_D_cruise, M_cruise, SFC_cruise, alt_cruise, V_cruise	FF-mission
get_divert_FF_wrapper	Divert Fuel fraction	range, divertRange	FF-divert
get_loiter_FF_wrapper	Loiter Fuel fraction	loiterTime, FF-mission, range_Harmonic, H_g	FF-loiter
get_total_FF_wrapper	Total Fuel fraction	FF-loiter, FF-mission, FF-divert	FF-total, FF-reserve
get_Range_Brequet_wrapper	Brequet Range	W_MTO, W_fuel_mission, L_over_D_cruise, V_cruise, Isp, SFC_cruise	range
get_Isp	Isp from SFC	SFC_cruise	Isp

design_point_selection	Get the design point (T/W, W/S)	maxTakeoffDistance, maxLandingDistance, W_MTO, AR, rho_cruise, rho_SL, V_cruise, n_engine, C_L_max_cruise, C_L_max_L, C_L_max_TO, g, n_max_cruise	WingLoading, ThrustLoading
BuffetOnsetBoundary	Get the buffetOnsetBoundary Cl from Mach number	M_cruise	C_L_max_cruise
engine_estimate	Estimate the thrust of the required engines	ThrustLoading, W_MTO, n_engine	Thrust_total, Thrust_per_engine
calc_wing_area	Calculate the wing area	WingLoading, W_MTO, g	wing_area
fuselage_estimate	Estimate the fuselage size and weight	PAX_per_Area, PAX, W_MTO, n_engine	Thrust_total, Thrust_per_engine
wing_span_class_I	Calculate the wing span	wing_area, AR	wing_span
wing_sweep_class_I	Estimation the wing sweep	M_cruise	wing_sweep
wing_taper_class_I	Estimation the wing taper	wing_sweep	wing_taper
root_and_tip_chord	Calculate wing chords	wing_area, wing_span, wing_taper	chord_root, chord_mean _geometric, chord_tip
mean_aerodynamic_chord	Calculate wing MAC	chord_root, wing_taper	MAC
fuselage_size_estimate	Estimate fuselage_size	PAX, fuselage_tail_fineness_ratio, fuselage_nose_ fineness_ratio	fuselage_nose_ length, fuselage_tail_length, fuselage_diameter, fuselage_center_length
wing_exposed_area_estimate	Estimate wing exposed area	wing_area, fuselage_diameter, chord_root	wing_exposed_area
wing_wetted_area_estimate	Estimate wing wetted area	t_over_c, wing_exposed_area	wing_wetted_area
fuselage_wetted_area_estimate	Estimate fuselage wetted area	fuselage_nose_ length, fuselage_tail_length, fuselage_diameter, fuselage_center_length	fuselage_wetted_ area

aircraft_wetted_area_estimate	Estimate aircraft wetted area	fuselage_wetted_area, wing_wetted_area	aircraft_wetted_area
equiv_parasite_area	Calculate equivalent parasite area (f)	c_f, aircraft_wetted_area	aircraft_eq_parasite_area
drag_coef_incompressible	Calculate incompressible drag coefficient	wing_area, aircraft_eq_parasite_area	C_D_0_incompressible
get_L_over_D_max	Calculate L over D max	AR, oswald_factor, C_D_0_incompressible	L_over_D_cruise
w_mzf_estimation	Estimate maximum zero fuel weight	W_fuel, W_MTO	W_MZF
root_tip_thickness	Estimate root and tip thickness of wing	t_over_c, chord_root	thickness_root
wing_weight_estimation	Estimate wing weight	wing_sweep, W_MZF, wing_span, wing_area, n_ultimate, thickness_root	wing_weight
class_II_aircraft_weight_estimation	Class II AC weight estimate	W_A-W, wing_weight	W_MTO

C

Resulting design processes

In this appendix the design processes (PSGs) resulting from the class II test case are presented. A graphical format was devised based on the idea of a punch-card chart. Every column represents a module (placed in the order of the fixed module sequence). Every row represents a module execution, placed in the order of execution. Therefor the chart gives a chronological overview overview of the module execution sequence. It allows for visual pattern recognition, enabling the viewer to identify recurring sequences in the process.

C.1. Fixed module sequence

The resulting fixed module sequence (which obviously matches the pre-defined sequence) is displayed in figures C.1-C.3.

	FixedModuleSequenceClassII	get_H_g	get_divert_FF_wrapper	get_mission_FF_wrapper	get_loiter_FF_wrapper	get_total_FF_wrapper	get_MTOW_wrapper	BuffetOnsetBoundary	design_point_selection	calc_wing_area	wing_span_class_I	wing_sweep_class_I	wing_taper_class_I	root_and_tip_chord	fuselage_size_estimate	fuselage_wetted_area_estimate	wing_exposed_area_estimate	wing_wetted_area_estimate	aircraft_wetted_area_estimate	equiv_parasite_area	drag_coef_incompressible	get_L_over_D_max	get_Empirical_Mass_Estimates	w_mzf_estimation	root_tip_thickness	wing_weight_estimation	class_II_aircraft_weight_estimation
1	get_H_g	■																									
2	get_divert_FF_wrapper		■																								
3	get_mission_FF_wrapper			■																							
4	get_loiter_FF_wrapper				■																						
5	get_total_FF_wrapper					■																					
6	get_MTOW_wrapper						■																				
7	BuffetOnsetBoundary							■																			
8	design_point_selection								■																		
9	calc_wing_area									■																	
10	wing_span_class_I										■																
11	wing_sweep_class_I											■															
12	wing_taper_class_I												■														
13	root_and_tip_chord													■													
14	fuselage_size_estimate														■												
15	fuselage_wetted_area_estimate															■											
16	wing_exposed_area_estimate																■										
17	wing_wetted_area_estimate																	■									
18	aircraft_wetted_area_estimate																		■								
19	equiv_parasite_area																			■							
20	drag_coef_incompressible																				■						
21	get_L_over_D_max																					■					
22	get_Empirical_Mass_Estimates																						■				
23	w_mzf_estimation																							■			
24	root_tip_thickness																								■		
25	wing_weight_estimation																									■	
26	class_II_aircraft_weight_estimation																										■
27	BuffetOnsetBoundary							■																			
28	design_point_selection								■																		
29	calc_wing_area									■																	
30	wing_span_class_I										■																
31	wing_sweep_class_I											■															
32	wing_taper_class_I												■														
33	root_and_tip_chord													■													
34	fuselage_size_estimate														■												
35	fuselage_wetted_area_estimate															■											
36	wing_exposed_area_estimate																■										
37	wing_wetted_area_estimate																	■									
38	aircraft_wetted_area_estimate																		■								
39	equiv_parasite_area																			■							
40	drag_coef_incompressible																				■						
41	get_L_over_D_max																					■					

Figure C.1: Resulting design process of the class II test case, using the fixed module sequence algorithm (part 1 of 3)

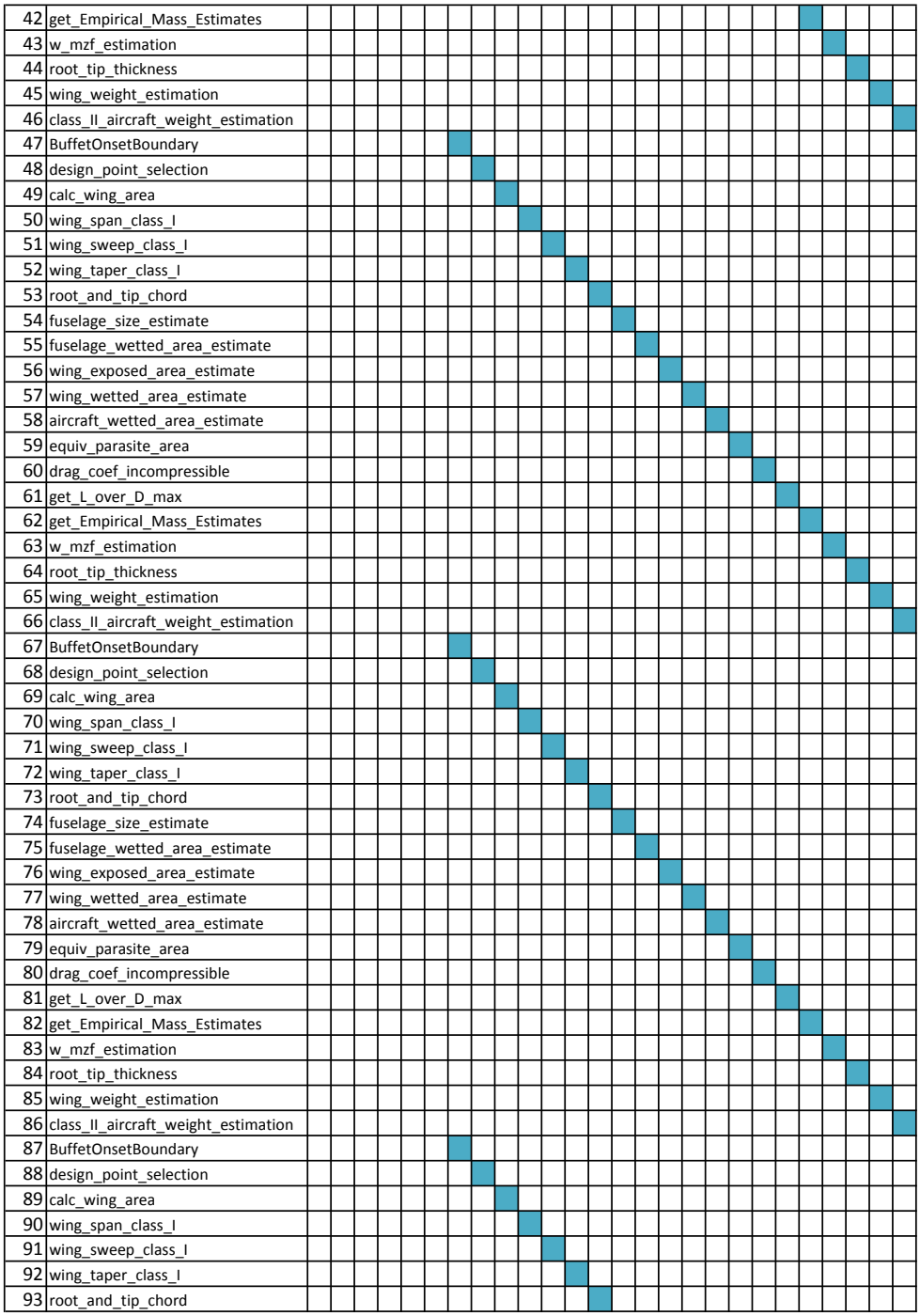

Figure C.2: Resulting design process of the class II test case, using the fixed module sequence algorithm (part 2 of 3)

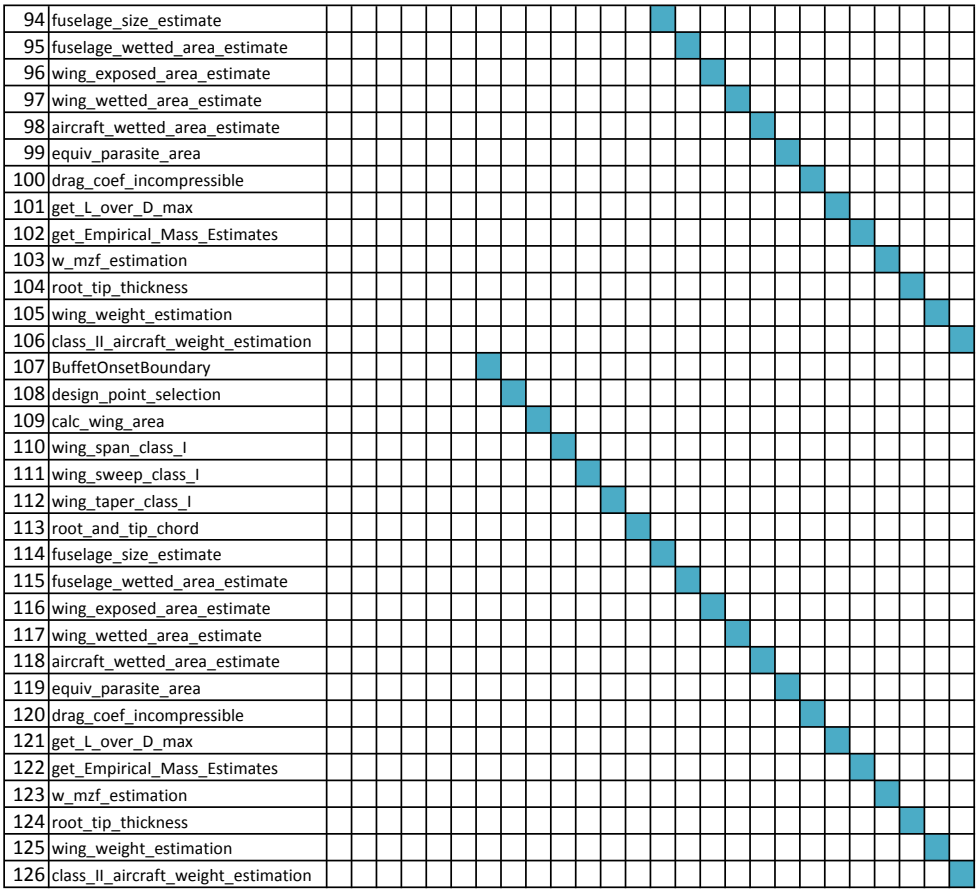

Figure C.3: Resulting design process of the class II test case, using the fixed module sequence algorithm (part 3 of 3)

C.2. Expected change based sequence

The resulting expected change based sequence is displayed in figures C.4-C.6.

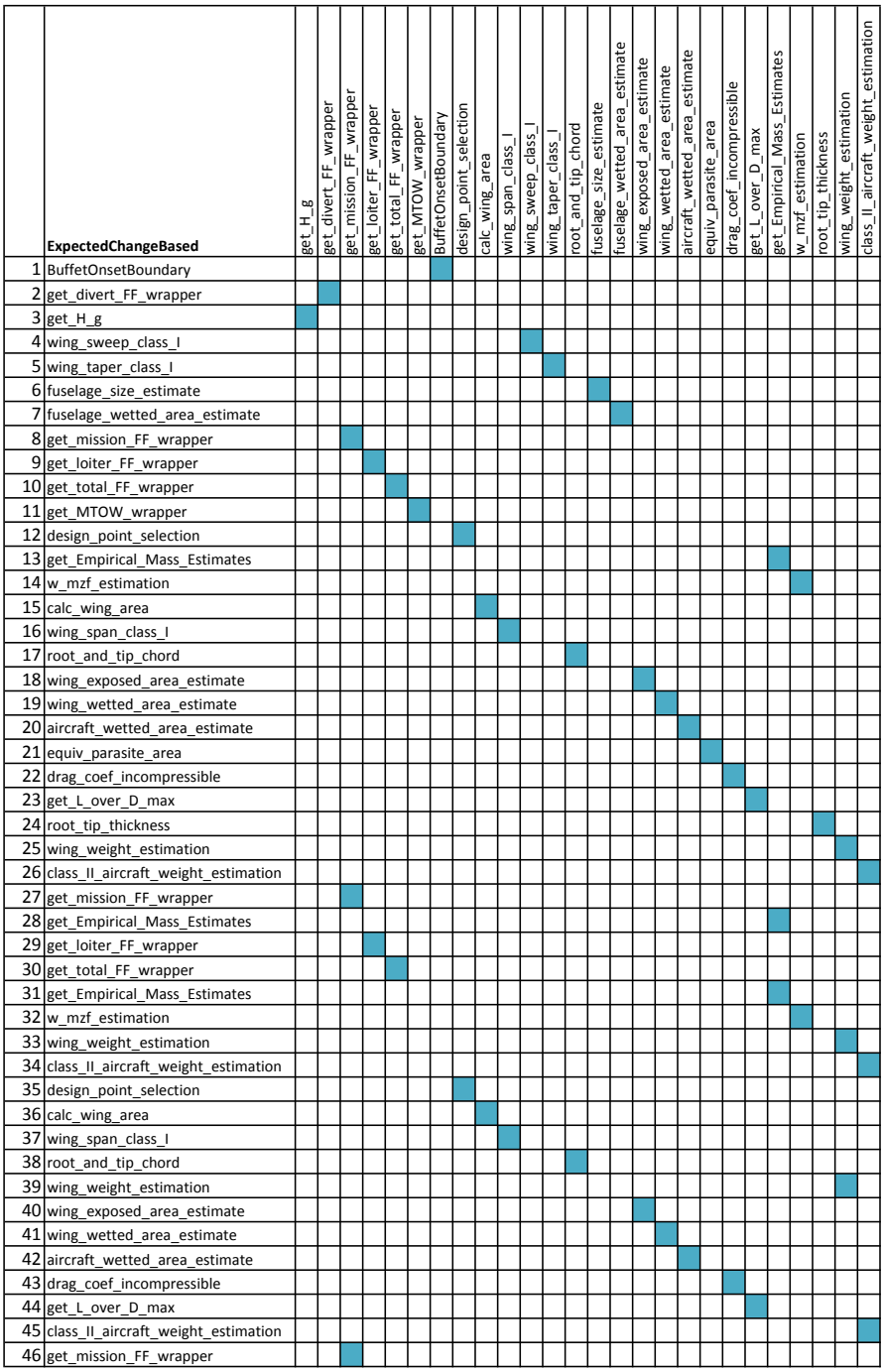

Figure C.4: Resulting design process of the class II test case, using the expected change based sequence algorithm (part 1 of 3)

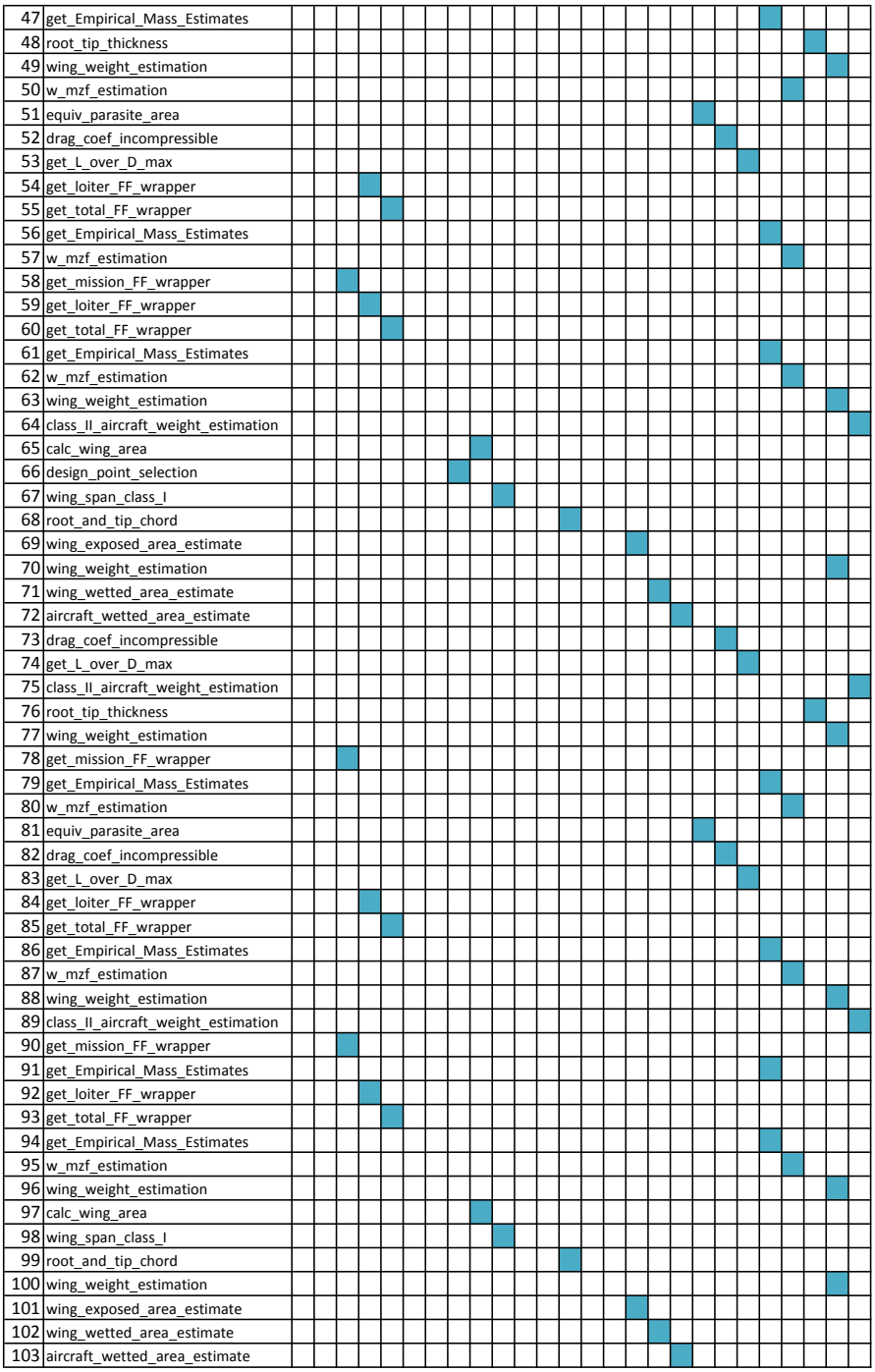

Figure C.5: Resulting design process of the class II test case, using the expected change based sequence algorithm (part 2 of 3)

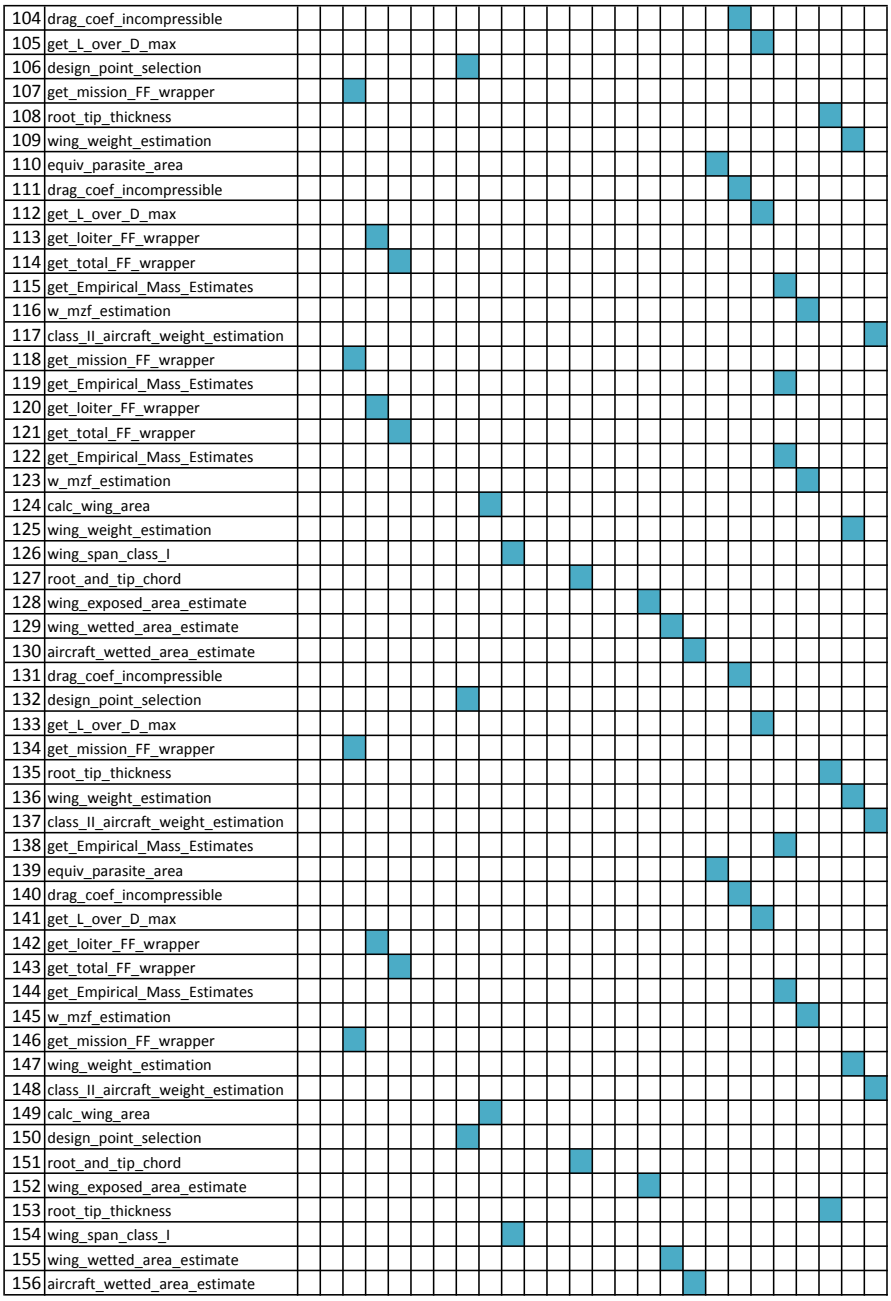

Figure C.6: Resulting design process of the class II test case, using the expected change based sequence algorithm (part 3 of 3)

C.3. Dynamic input error based sequence

The resulting dynamic input error based sequence is displayed in figures C.7-C.10.

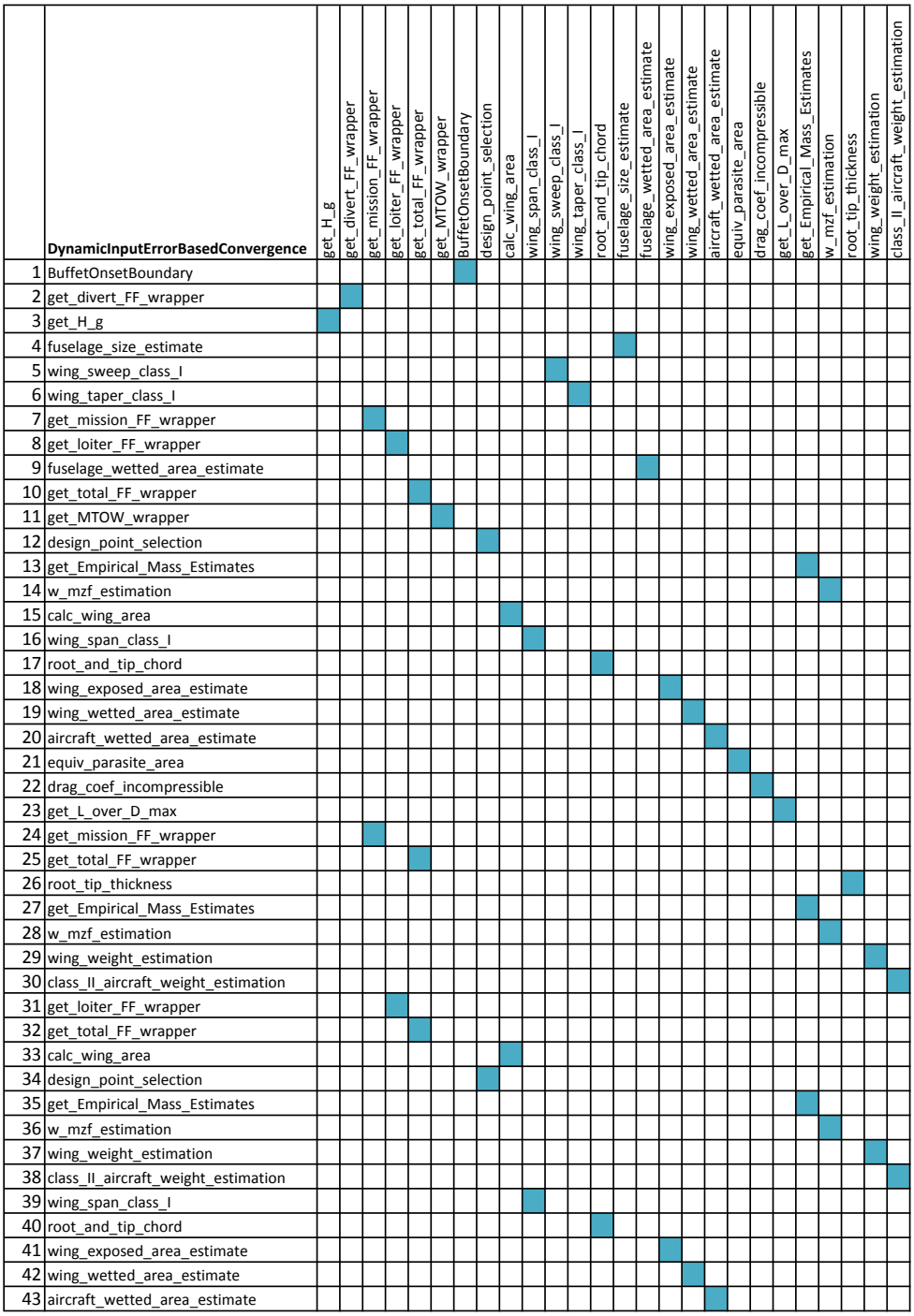

Figure C.7: Resulting design process of the class II test case, using the dynamic input error based algorithm (part 1 of 4)

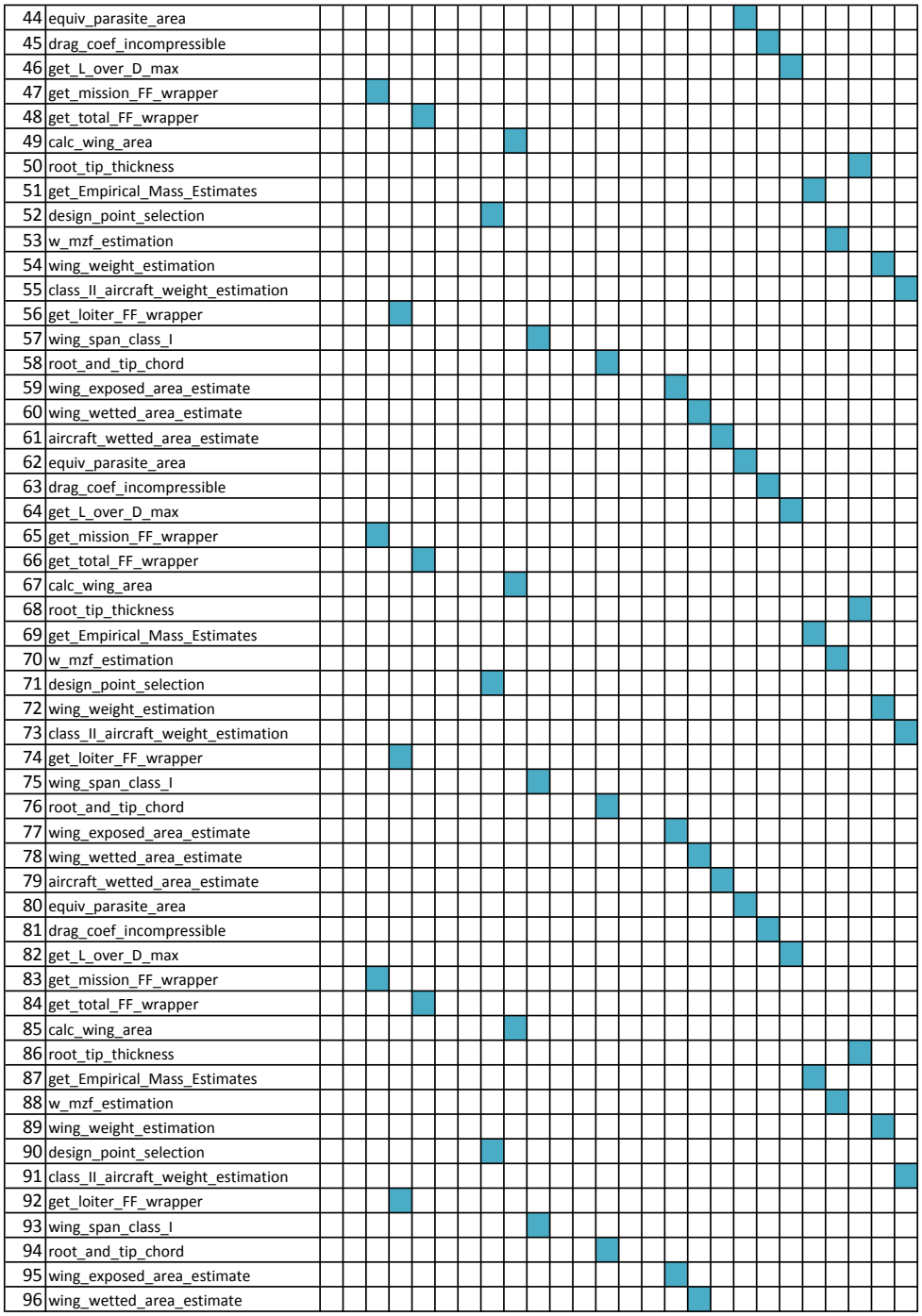

Figure C.8: Resulting design process of the class II test case, using the dynamic input error based (part 2 of 4)

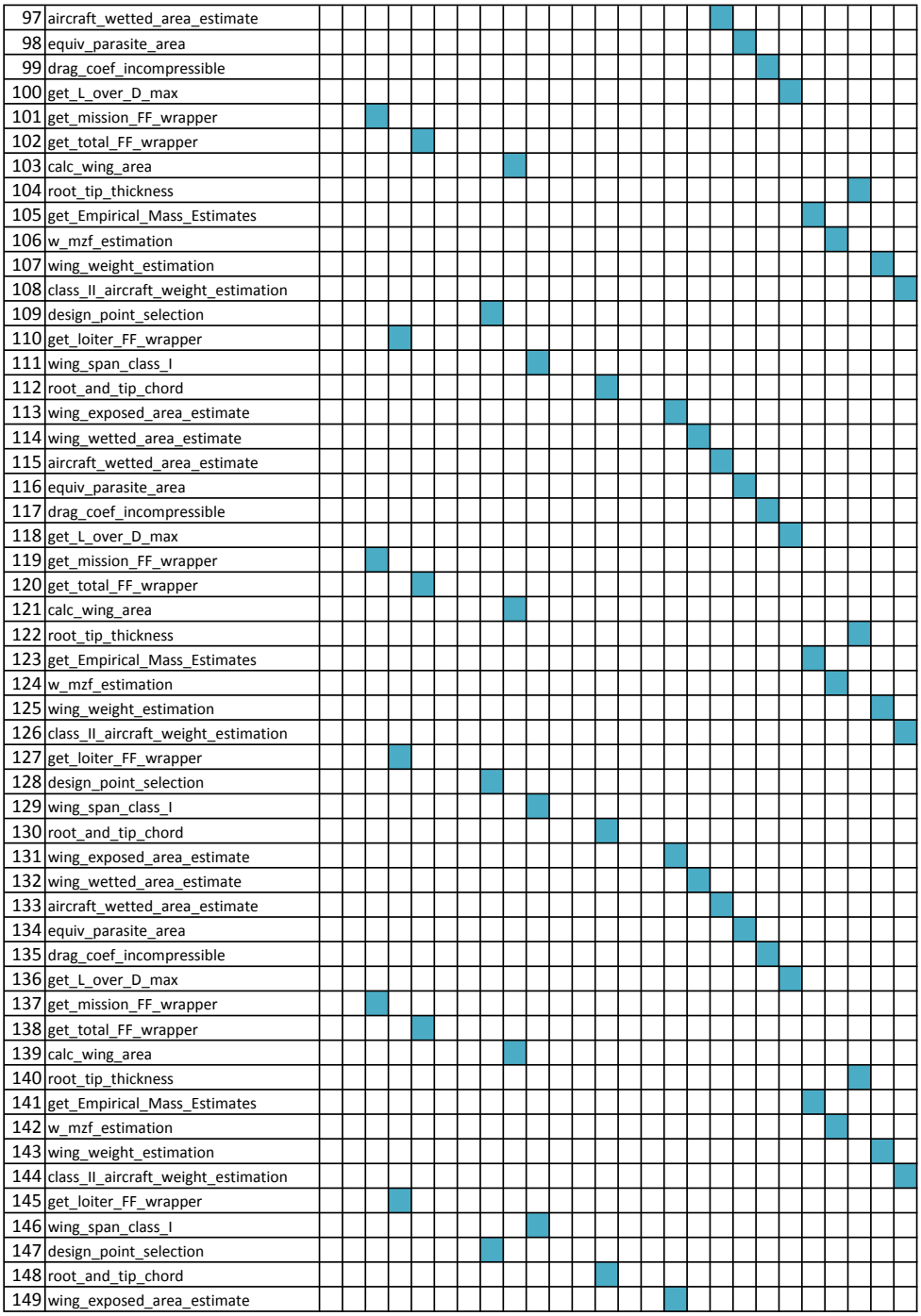

Figure C.9: Resulting design process of the class II test case, using the dynamic input error based (part 3 of 4)

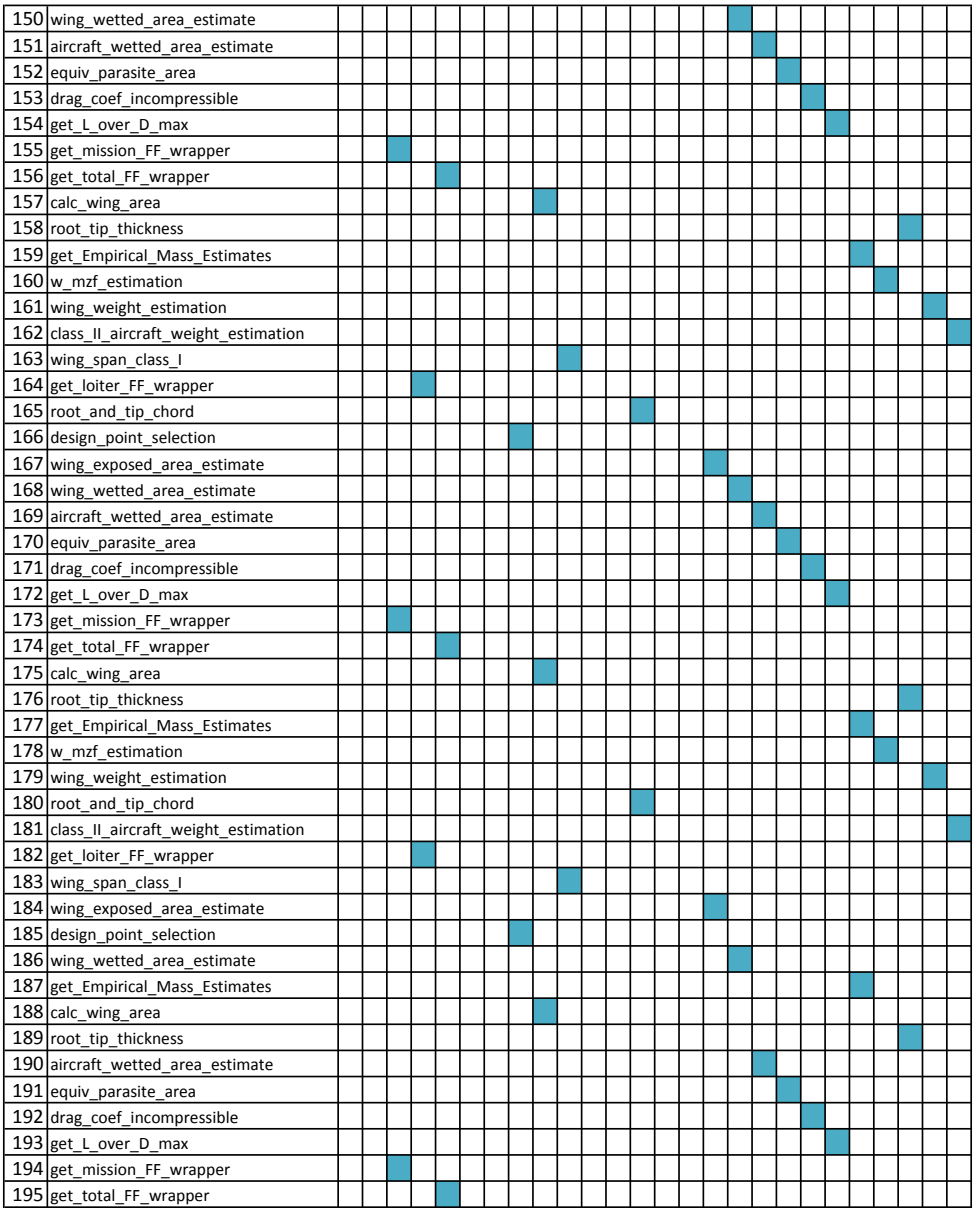

Figure C.10: Resulting design process of the class II test case, using the dynamic input error based (part 4 of 4)

Bibliography

[1] G. La Rocca and M. van Tooren, *Knowledge-based engineering approach to support aircraft multidisciplinary design and optimization,* Journal of Aircraft , 1875 (2009).

[2] R. Elmendorp, *Synthesis of novel aircraft concepts for future air travel,* (2014).

[3] N. B. Böhnke, D. and V. Gollnick, *An approach to multi-fidelity in con- ceptual aircraft design in distributed de- sign environments,* IEEE Aerospace Con- ference (2011).

[4] D. Steward, *The design structure system: A method for managing the design of complex systems,* IEEE Transactions on Engineering Management , 71 (1981).

[5] A. Lambe and J. Martins, *Extensions to the design structure matrix for the description of multidisciplinary design, analysis, and optimization processes,* Structural and Multidisciplinary Optimization 46(2) , 273 (2012).

[6] D. J. Pate, J. Gray, and B. J. German, *A graph theoretic approach to problem formulation for multidisciplinary design analysis and optimization,* Structrual and Multidisciplinary Optimization 20(2) (2013).

[7] J. Roskam, *Airplane Design Parts I through VII (v. 1-7)*, 2nd Edition (DARCorporation, 2006).

[8] D. Raymer, *Aircraft Design: A Conceptual Approach (5th Edition)*, Aiaa Education Series (Amer Inst of Aeronautics & 5 edition, 2012).

[9] E. Toorenbeek, *Synthesis of Subsonic Airplane Design* (Springer Science & Business Media, 1982).

[10] D. Engineering, *Caesiom,* http://www.ceasiom.com, accessed: 27-1-2015.

[11] L. Salavin, *Structure and function of the aircraft design program PrADO*, Tech. Rep. (Hamburg, 2008).

[12] Pacelab, *Pacelab apd,* http://www.pace.de/products/preliminary-design/pacelab-apd.html, accessed: 27-1-2015.

[13] S. Jayaram and A. Myklebust, *Acsynt - a standards-based system for parametric, computer aided conceptual design of aircraft,* (Aerospace Design Conference, Irvine, CA, 1992).

[14] L. Ltd., *Piano,* http://www.piano.aero, accessed: 27-1-2015.

[15] D. Rodriguez and P. Sturdza, *A Rapig Geometry Engine for Preliminary Aircraft Design*, Tech. Rep. (Reno, Nevada, 2006).

[16] G. G. J.D.P., *A rapid geometry modeler for conceptual aircarft*, Tech. Rep. (2015).

[17] R. Perez, H. Liu, and K. Behdinan, *Multidisciplinary Design Optimization of Aerospace Systems*, Tech. Rep. (Toronto, ON, Canada, 2005).

[18] T. C. Wagner and P. Papalambros, *A general framework work for decomposition analysis in optimal design,* Advances in Design Automation, Vol. 2 , 315 (1993).

[19] N. M. Alexandrov and R. M. Lewis, *Reconfigurability in mdo problem synthesis, part 1,* (AIAA/ISSMO, Albany, New York, 2004).

[20] K. T. Moore, *The development of an open-source framework for multidisciplinary analysis and optimization,* 10th aiaa/issmo multidisciplinary analysis and optimization conference (2008).

[21] 10th aiaa/issmo multidisciplinary analysis and optimization conference (2008).

[22] Noesis, *Optimus is a robust design optimization and multi objective optimization software that minimizes the time required to engineer new products,* http://www.noesissolutions.com/Noesis/, accessed: 27-1-2015.

[23] P. Integration, *Phx modelcenter | desktop trade studies,* http://www.phoenix-int.com/software/phx-modelcenter.php, accessed: 27-1-2015.

[24] S. N. Laboratories, *Dakota | helping analysts & decision-makers understand outcomes of predictive simulations,* http://www.phoenix-int.com/software/phx-modelcenter.php, accessed: 27-1-2015.

[25] M. Ramakers, *Next-generation conceptual aircraft design tool*, Tech. Rep. (2015).

[26] R. Patton, *Software Testing (2nd Edition)* (Sams Publishing, 2005).

[27] C. Pilato, *Version Control with Subversion* (O'Reilly Media, 2008).

[28] G. Zlobin, *Learning Python Design Patterns* (PACKT Publishing, 2013).

[29] M. Drela, *Xfoil: An analysis and design system for low reynolds number airfoils,* Volume 54 of the series Lecture Notes in Engineering , 1 (1989).

[30] D. Gorissen, *A surrogate modeling and adaptive sampling toolbox for computer based design,* Journal of Machine Learning Research , 2051 (2010).

[31] G. van Rossum, *Python tutorial, Technical Report CS-R9526*, Tech. Rep. (1995).

[32] D. A. S. Aric A. Hagberg and P. J. Swart, *Exploring network structure, dynamics, and function using networkx,* Proceedings of the 7th Python in Science Conference (SciPy2008) , 11 (1995).

[33] J. Kleinberg, *Algorithm Design* (Pearson New International Edition, 2014).

[34] statisticslectures.com, *Statistics lectures,* http://www.statisticslectures.com, accessed: 27-1-2015.