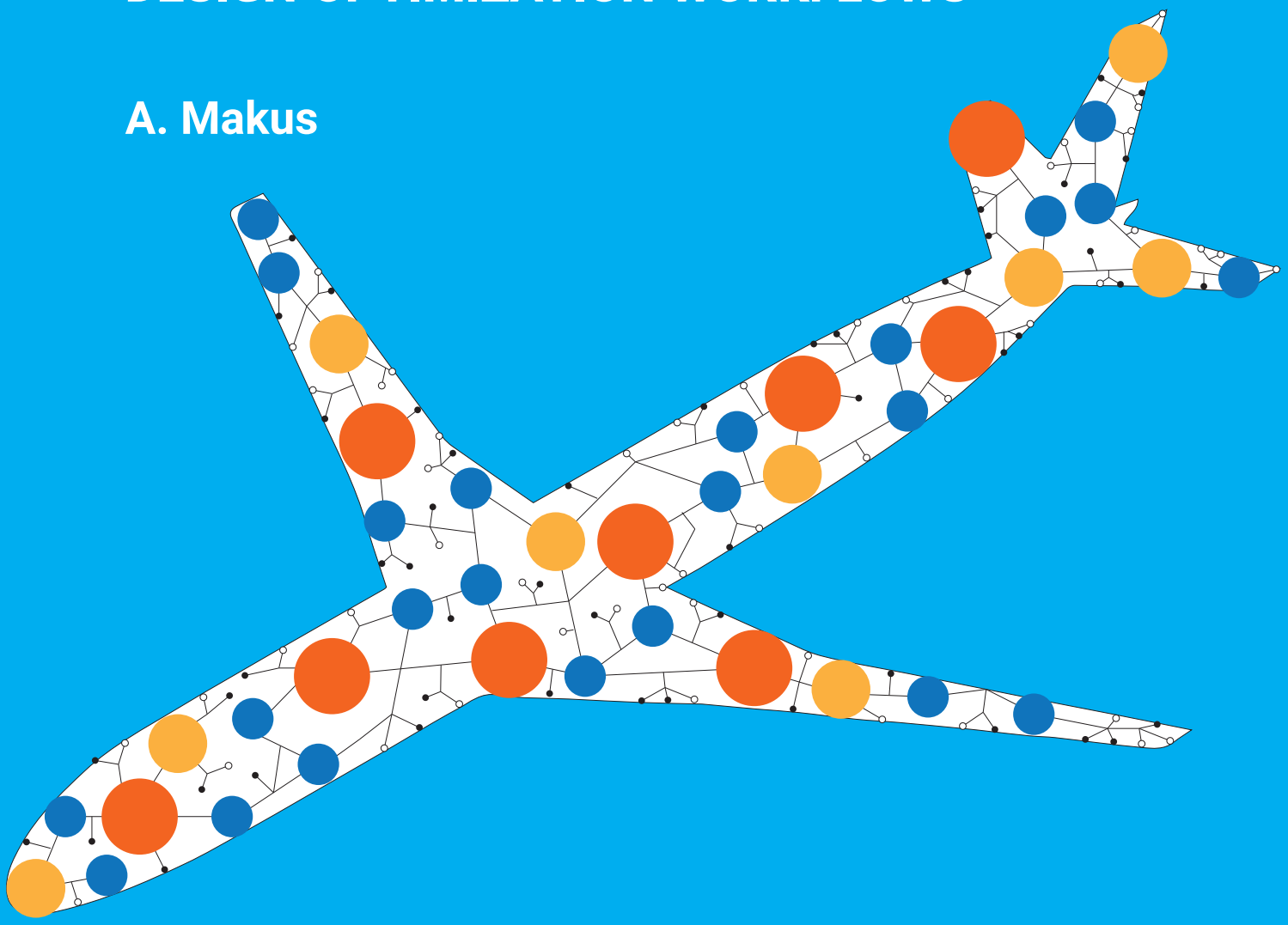
DEVELOPMENT OF A KNOWLEDGE-ENABLED TOOL REPOSITORY TO SUPPORT AUTOMATED GENERATION OF MULTIDISCIPLINARY DESIGN OPTIMIZATION WORKFLOWS

A. Makus



Technische Universiteit Delft

TU Delft

DEVELOPMENT OF A KNOWLEDGE-ENABLED TOOL REPOSITORY TO SUPPORT AUTOMATED GENERATION OF MULTIDISCIPLINARY DESIGN OPTIMIZATION WORKFLOWS

by

A. Makus

to obtain the degree of Master of Science

at the Delft University of Technology,

to be defended publicly on Monday April 3, 2017 at 9:30 AM.

Student number:	4105958	
Report number:	152#17#MT#FPP	
Thesis committee:	Prof. dr. ir. L. Veldhuis,	TU Delft, Chairman of the exam committee
	Ir. I. van Gent,	TU Delft, Supervisor
	Dr. ir. G. La Rocca,	TU Delft , Flight Performance and Propulsion
	Dr. ir. R. de Breuker,	TU Delft, Aerospace Structures and Materials
	Ir. P. D. Ciampa,	DLR, Integrated Aircraft Design

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

TU Delft

Acknowledgments

With the completion of this thesis, my time in Delft as a student at the Faculty of Aerospace Engineering comes to an end. Without the support from many people, I could have not come this far, and I would like to take the opportunity to express my thankfulness.

I would like to thank my supervisor Imco van Gent for his support in all matters of my research, for his motivational attitude and for his intelligent advice. His guidance throughout my project helped me to keep an eye on the big picture and the important things about my work. I would also like to thank Gianfranco La Rocca for taking his time to check my work, for his useful input and helpful feedback. Finally, I want to thank my girlfriend Julia for her help and patience throughout the course of my project, and most importantly, for being there for me in the times of need.

A. Makus
Delft, April 2017

Summary

The EU-funded program *Aircraft 3rd Generation MDO for Innovative Collaboration of Heterogeneous Teams of Experts*, short AGILE, is concerned with the development of knowledge-enabled information technologies to support interdisciplinary design campaigns, making the use of multidisciplinary design optimization (MDO) more agile by removing major barriers to this engineering design methodology. As part of this project, the TU Delft is tasked with the development of a system to support an automated generation of MDO workflows with the primary goal of shortening their configuration time, especially in large optimization problems. The *Knowledge-Based Agile Design for Multidisciplinary Optimization System* (KADMOS) is being developed to provide a support framework for design engineers in order to generate MDO workflows following an MDO problem formulation and according to available analysis tools and MDO architectures.

The work in this research lays the foundation for KADMOS through the development of a tool repository that enables the creation, modification and debugging of MDO workflows through the use of graph networks. The tool repository uses an XML-based system to establish a human and machine readable interface and converts it to graph structures that enable an effective analysis and representation of the information stored in the repository. In order to enable a more efficient interdisciplinary collaboration, the *Common Parametric Aircraft Configuration Schema* (CPACS) is used. CPACS serves as a straightforward data exchange schema that simplifies the coupling of design tools and forms a standardized data structure for the development aircraft designs. Knowledge rules about the relationship between nodes defined by the data schema are stored in the repository and allow the automation of complex and time-consuming tasks in the generation of MDO workflows.

The developed system was integrated into KADMOS and demonstrated its capabilities in a realistic case study involving the multidisciplinary optimization of an aircraft wing design. The use case was set up both manually and automatically in order to illustrate the agility of the created repository, where most of the repetitive configuration tasks are automated. It was shown that the automated generation of MDO formulations significantly decreases their configuration time, and that engineers are supported in the decision-making process by helping to identify desirable workflow configurations.

The most important benefit of the system is its capability to rapidly redefine and reorganize MDO formulations when adding or removing disciplinary tools, changing tool settings, or adjusting the initial aircraft design or top level requirements. This results in a system that can quickly switch between aircraft designs, making the optimization process more agile and more accessible to interdisciplinary collaborators. The developed tool repository contributes to the elimination of barriers to the widespread use of MDO methodologies, which ultimately entails numerous economic benefits in terms of cost and time that come with their use as compared to traditional design methodologies.

Contents

List of Figures

List of Tables

Nomenclature

AGILE	Aircraft 3rd Generation MDO for Innovative Collaboration of Heterogeneous Teams of Experts
CPACS	Common Parametric Aircraft Configuration Schema
FPG	Fundamental Problem Graph
KADMOS	Knowledge-Based Agile Design for Multidisciplinary Optimization System
MCG	Maximal Connectivity Graph
MDO	Multidisciplinary Design Optimization
PSG	Problem Solution Graph
RCG	Repository Connectivity Graph
XDSM	eXtended Design Structure Matrix

1

Introduction

The onset of the digital age in the aerospace industry several decades ago has caused a significant shift away from traditional design methodologies towards the computerization of information. Aircraft design methodologies have made considerable progress, especially due to the increased availability and affordability of computational resources, leading to the development of sophisticated software applications that can make precise predictions about the performance of a product with respect to various fields of study.

Engineers today can access a vast amount of numerical analysis tools of varying fidelities to support their design activities. However, the impact of the advance in computational performance is limited by the widespread application of traditional design practices, where each engineering domain remains a segregate entity during the design process, limiting the amount of information and data exchange between them (Belie et al., 2002).

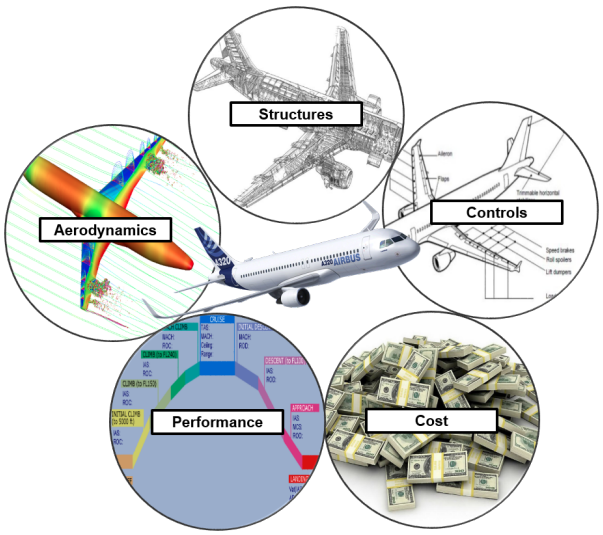


Figure 1.1: Many separate disciplines go into the design of a complete aircraft.

The segregation of domains was originally established as a means to deal with the growing complexity of non-linear interdependencies amongst engineering systems and subsystems (Belie et al., 2002), leading to the issue of design engineers primarily focusing on optimizing their own discipline, and less so on the state of the entire system. This becomes especially critical in areas that are

highly integrated (Bowcutt, 2003), as in the case of aerodynamic and structural analysis which often requires close cooperation to determine aero-elastic effects.

With the introduction of *Multidisciplinary Design Optimization* (MDO), engineers attempt to eliminate the segregation of disciplines by aggregating them into one system. Although MDO has first found its purpose in structural synthesis (Schmit and Farshi, 1974), it has soon been adopted as a design methodology for the conceptual and preliminary design phases in aerospace and other domains (Agte et al., 2010).

As opposed to legacy design methodologies, MDO attempts to connect separate domains through the coupling of disciplinary analysis tools and optimizers, enabling an interdisciplinary exchange of information throughout the design process. This not only has the effect of permitting design engineers to generate more concepts at a faster rate, but also reduces the competition between separate engineering domains by having an increased focus on the overall optimization of the system (Belie et al., 2002).

The impact of disciplinary coupling and optimization as introduced through MDO has been investigated by Flager and Haymaker (2007), and compared to legacy methodologies. Although the engineers spent more time with the problem specification using the MDO methodology, they spent significantly less time on execution and management tasks, and had an almost five-fold increase in the amount of time allocated to analyze results and reason over design options, as presented in Figure 1.2.

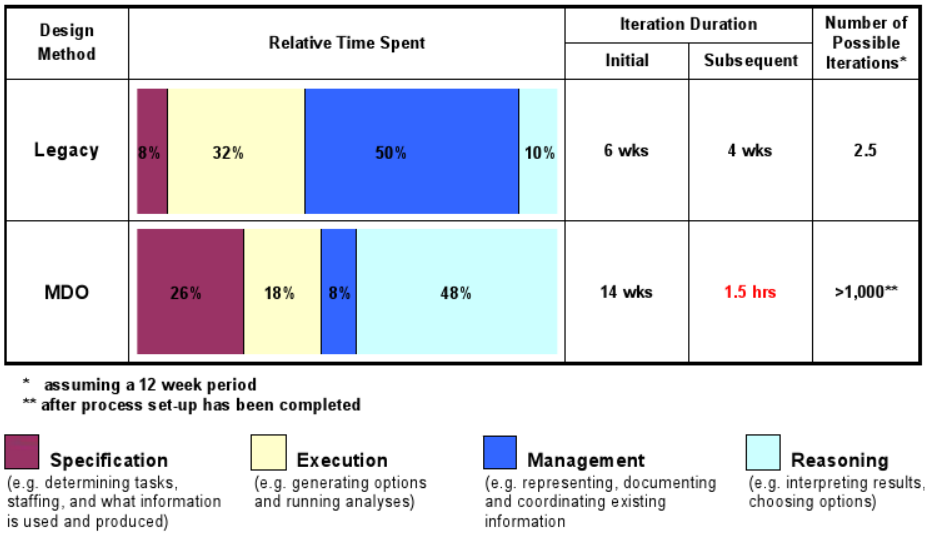


Figure 1.2: Comparison of time allocation between MDO and Legacy methods (Flager and Haymaker, 2007).

This study suggests an important trend that is facilitated through the use of MDO: it not only enables the design engineer to reduce the time of the traditional design cycle while analyzing more configurations, but also heavily reduces the time spent on non-creative work, meaning that the engineer can spend more time interpreting results and evaluating design decisions (Flager and Haymaker, 2007).

1.1. Barriers and Challenges in MDO

There are, however, numerous technical and non-technical barriers to the widespread use of MDO methodologies. As Sobieszczanski-Sobieski and Haftka (1996) observed, one of the main issues with

MDO is its organizational complexity. The more disciplines become interconnected in an analysis, the more data is exchanged between them to find a feasible optimum, and the more difficult it becomes for an engineering team to keep an overview of the MDO process that is used to solve the optimization problem (Venkataraman and Haftka, 2002), such as in the example of Fig 1.3. Although this observation has been made more than 20 years ago, optimization systems have steadily become even more complex, which is why this issue still persists today.

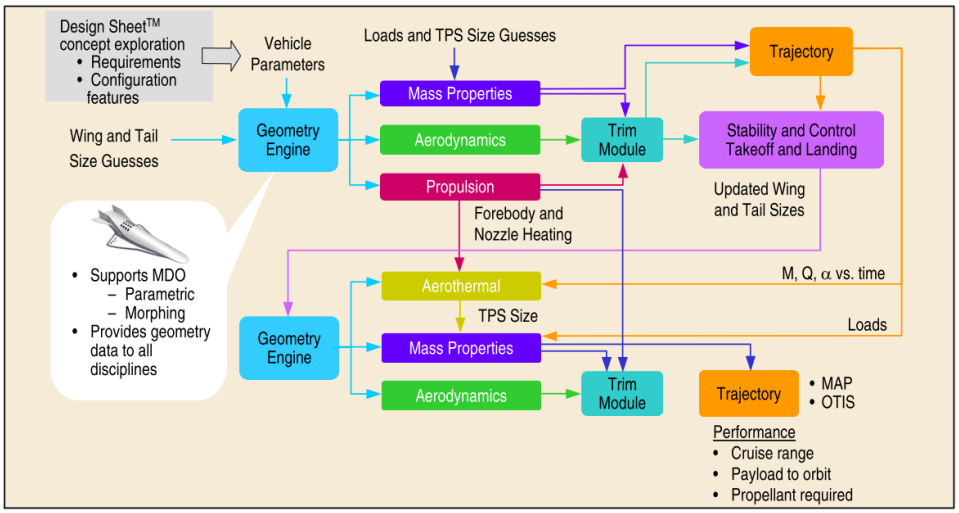


Figure 1.3: Example of a highly coupled analysis and optimization model for a hypersonic vehicle (Bowcutt, 2003).

An increased complexity entails a high configuration time and effort for large systems, which can be obeserved in the high "Specification" time and "Initial Iteration Duration" in Figure 1.2, constituing a significant deterrent to the use of MDO due to the implied cost in the initial stage of the design phase. This aspect, as Belie et al. (2002) argue, greatly impairs the widespread implementation of MDO in industry.

1.2. Current Developments

Despite significant advances in the application of MDO methodologies since their first conception in the 1970's, their full potential is not yet exploited. To counter the critical issues that prevent a more extensive use of MDO, the *AGILE* program was initiated, providing the context for the work in this research.

1.2.1. AGILE

The *Aircraft 3rd Generation MDO for Innovative Collaboration of Heterogeneous Teams of Experts* project (Nagel and Ciampa, 2014), short *AGILE*, is funded by the European Commission and tries to set new benchmarks in the reduction of aircraft development costs by improving current MDO design methodologies and enhance collaboration among heterogeneous teams in large projects, as shown in Fig. 1.4.

Its four technical objectives include **(1)** the development of advanced multidisciplinary optimization techniques and their integration, **(2)** the development of processes and techniques for efficient multi-site collaboration in the overall design teams, **(3)** development of knowledge enabled information technologies to support interdisciplinary design campaigns and **(4)** the development of an Open MDO Test Suite (Nagel and Ciampa, 2014). In short, the program aims at establishing a system

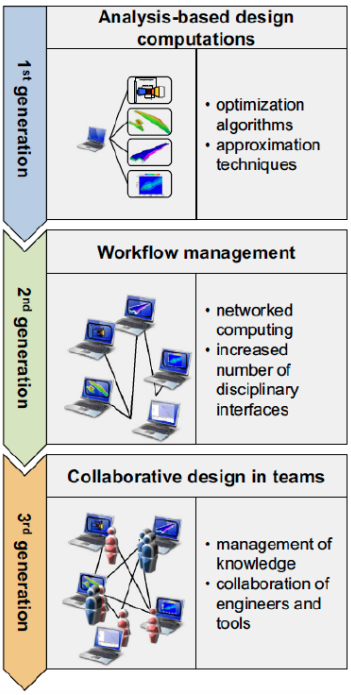


Figure 1.4: Improving the collaboration between disciplines and disciplinary actors (Moerland et al., 2015).

that enables teams of engineers from different disciplines to cooperate closely on complex MDO problems by removing existing barriers that impede the widespread use of MDO methodologies.

1.2.2. KADMOS

As an integral part of AGILE, the *Knowledge-Based Agile Design for Multidisciplinary Optimization System* (KADMOS) acts as a platform to assist heterogeneous teams of engineers in the creation and configuration of MDO workflows (Van Gent and La Rocca, 2017). KADMOS intends to support engineers with complex and time-consuming tasks that are required in the set-up of MDO systems, ultimately enabling a quick and agile generation of these intricate systems. Agility in this context can be understood as having the flexibility to perform a rapid definition and redefinition of a system by automating repetitive, complex, and time-consuming processes in order to maximize the engineer's time for analysis and interpretation of the results.

Time consuming tasks, for instance, are often found in the creation of interfaces between design tools to facilite information exchange between them. These tasks, however, are highly repetitive as they need to be established for each tool in each use case. By providing a framework that supports the engineer in establishing the tool connections, time, effort, and the need for expert knowledge in the set up of MDO workflows can be reduced significantly.

The research presented in this report aims to contribute to the development of KADMOS and is specifically related to the first two steps in the MDO development process seen in Fig. 1.5. The main goal of this research is to establish a knowledge enabled tool repository to facilitate an automated generation of MDO workflows, which are ultimately transformed into executable simulation workflows. Knowledge rules play an important part in this project as they provide grounds for a more dynamic construction of use case specific tool inputs and outputs, and allow for the re-use of the information of previosuly generated workflows. By providing KADMOS with the ability to create MDO workflows in an *agile* manner, the specification time of these workflows, as defined by Flager

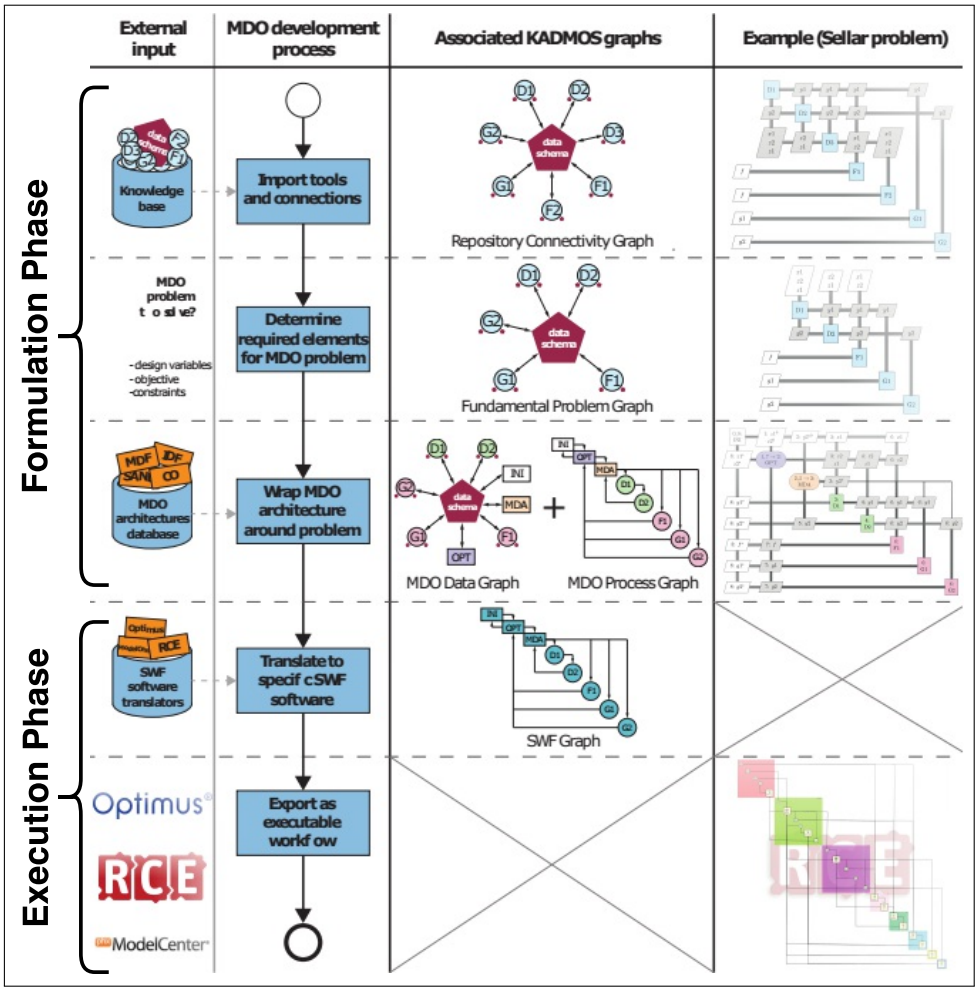


Figure 1.5: MDO developement process in *KADMOS* (Van Gent and La Rocca, 2017).

and Haymaker (2007), will be significantly reduced and a major component of complexity in the set-up of MDO systems removed, eliminating some of the major barriers to the use of MDO as an integral part of aircraft design methodologies.

1.3. Research Objectives

The previous discussion introduced the issues that arise in the generation of MDO workflows. The more design tools are present in an MDO system, the more interactions exist between these tools, resulting in complex and often repetitive tasks in the set up of such workflows. These tasks not only include the creation of appropriate couplings among the tools, but also their placement in a workflow that allows for an efficient solution strategy.

To support the engineers with these tasks, the research in this report attempts to improve the current state of MDO applications by providing a methological approach in the set up of MDO formulations using a knowledge-enabled tool repository. The research is placed in the "Formulation Phase" of KADMOS, specifically with the initial two stages in the development of MDO workflows in the form of graphs. This, in essence, lays the ground work for subsequent developments in KADMOS, as its operation entirely depends on the structure and composition of the tool repository, as shown in Fig. 1.5.

The ultimate ambition of this research is to develop a "smart" tool repository that provides an interface between KADMOS and design engineers, and that uses knowledge-rules to reduce the complexity and workload in the set up of MDO workflows. The created repository must enable KADMOS to not only estimate which tools can be coupled in order to define an optimal MDO workflow, but also how these tools can be coupled to limit its computational expense. The goal of KADMOS is to develop a system that cuts the set-up time and effort in order to make MDO methodologies more accessible to engineers in the aerospace industry. This is reflected in the research objectives listed below:

- Establish a tool repository structure that enables the automatic generation of MDO formulations according to an arbitrary MDO problem definition through the application of knowledge rules and knowledge reuse
- Develop a methodology that facilitates the swift and uncomplicated addition of design tools and knowledge rules to the knowledge-enabled tool repository
- Extend KADMOS with the knowledge-enabled tool repository and integrate it with other KADMOS modules to allow for a rapid generation of MDO formulations

These objectives directly translate to the following research question:

- How can a knowledge-enabled tool repository facilitate the automatic generation of efficient MDO formulations for arbitrary MDO problems using a given set of analysis tools?

1.4. Report Outline

The structure of this document is as follows. Chapter 2 provides an overview of the technologies used in this research and described the used approach for the creation of graph-based MDO workflows. In Chapter 3, the structure of the tool repository and all its required elements are presented, and Chapter 4 discusses the creation of graph structures using a such tool repository. Chapter 5 introduces the main issues of the presented approach for the setup of the tool repository and presents an extended repository structure that enables a more agile set-up of MDO workflows. A case study is presented in Chapter 6 giving practical insights into the developed tool repository, and Chapter 7 finalizes this thesis report with a conclusion and recommendations.

2

Methodology and Enabling Technologies

In this chapter, the most important aspects in the development of the knowledge-enabled tool repository are discussed. Following the approach of Pate et al. (2014), Section 2.2 presents the graph-based methodology to represent MDO workflows in the form of graph structures. Section 2.3 introduces the reader to the concepts of the XML Schema, which presents the basis for the common description of aircraft, and Section 2.4 gives insight into XML and its use in this research. The chapter ends with an overview of the terminology used in this report.

2.1. MDO Workflow Representation

Multidisciplinary Design Optimization (MDO) is a field of engineering that focuses on the use of numerical optimization for the design of systems that involve a number of disciplines or subsystems. It can be used to improve existent designs, to develop new, innovative systems, and to simplify the search for feasible design solutions (Martins and Lambe, 2013). As opposed to legacy design methodologies, MDO tries to connect separate domains through the coupling of disciplinary analysis tools and optimizers, leading to an interdisciplinary exchange of information throughout the design process. An abstracted MDO process is shown in Figure 2.1.

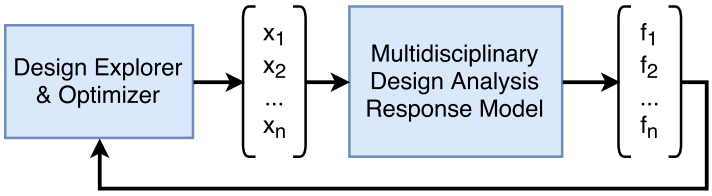


Figure 2.1: Abstracted example of an MDO process, adapted from (Vandenbrande et al., 2006).

A useful representation of the *Multidisciplinary Design Analysis Response Model* in Fig. 2.1 can be obtained through the *eXtended Design Structure Matrix* (XDSM), as proposed by Lambe and Martins (2012). This illustration method, shown on the left in Figure 2.2, enables the engineer to recognize data flows (grey lines), process flows (black lines), generic processes (rectangles), data in- or output (parallelograms) and iterative driver processes (rounded boxes) in a straightforward and efficient manner.

Although the XDSM facilitates an excellent representation of MDO workflows in a human-readable manner, it does this in a format that is not machine-readable (Pate et al., 2014). A different repre-

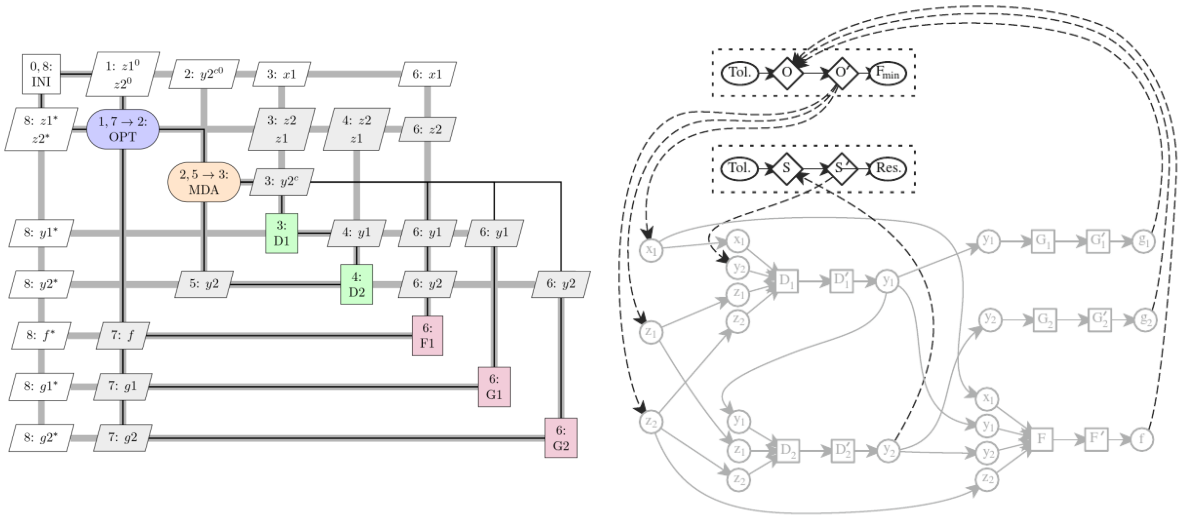


Figure 2.2: Example of an XDSM (left)(Van Gent and La Rocca, 2017) and Graph-based (right) (Pate et al., 2014) diagram of the Sellar Problem.

sentation format is introduced by Pate et al. (2014) that tries to circumvent this issue through the application of standard expressions of graph theory to create a new graph syntax. Even though this syntax is more difficult for humans to read, especially in large problems with a lot of data exchange, it provides new constructs tailored to algorithmic analysis and manipulation (Pate et al., 2014). This is particularly interesting for the construction of MDO workflows of given problem formulations, since it administers rules and guidelines for the generation of such workflows. A comparison of the both representations for the same MDO workflow can be viewed in Fig. 2.2.

2.2. Graph-Based Approach

Graphs are mathematically defined to consist of **nodes** and **edges**, and are used to represent relationships between objects. Fig. 2.3 shows a directed and an undirected graph, of which only the former is used throughout this research. All edges in a directed graph have a certain direction, indicating the path of the exchange of information. Graphs can be used to represent the input and output variables of tools in the form of nodes, and allow to build up networks based on the nodes that are provided to the system.

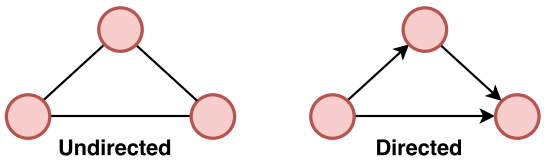


Figure 2.3: Example of an undirected and directed graph.

As shown above, a possible approach towards the formulation of MDO problems and their integration into workflows is proposed by Pate et al. (2014), who build upon a graph-theoretical approach to derive a graph syntax to not only illustrate workflows, but also generate these for a given MDO problem according to provided disciplinary tools and MDO architectures.

The application of the graph-based syntax starts with the analysis of all inputs and outputs of the

available analysis tools, constraints, and objectives of the defined problem, and is followed by the generation of a so-called *Maximal Connectivity Graph* (MCG). Here, all relationships among all tools and variables are illustrated. As an example, the MCG of a subsonic transport aircraft problem is provided on the top of Figure 2.4. In this MCG, each tool is represented by a pair of squared nodes, with each either having incoming edges or outgoing edges, which are always connected to variable nodes. The MCG represents all available tools in the database which can be used to determine the variable nodes *"Total weight"* and *"Performance"*.

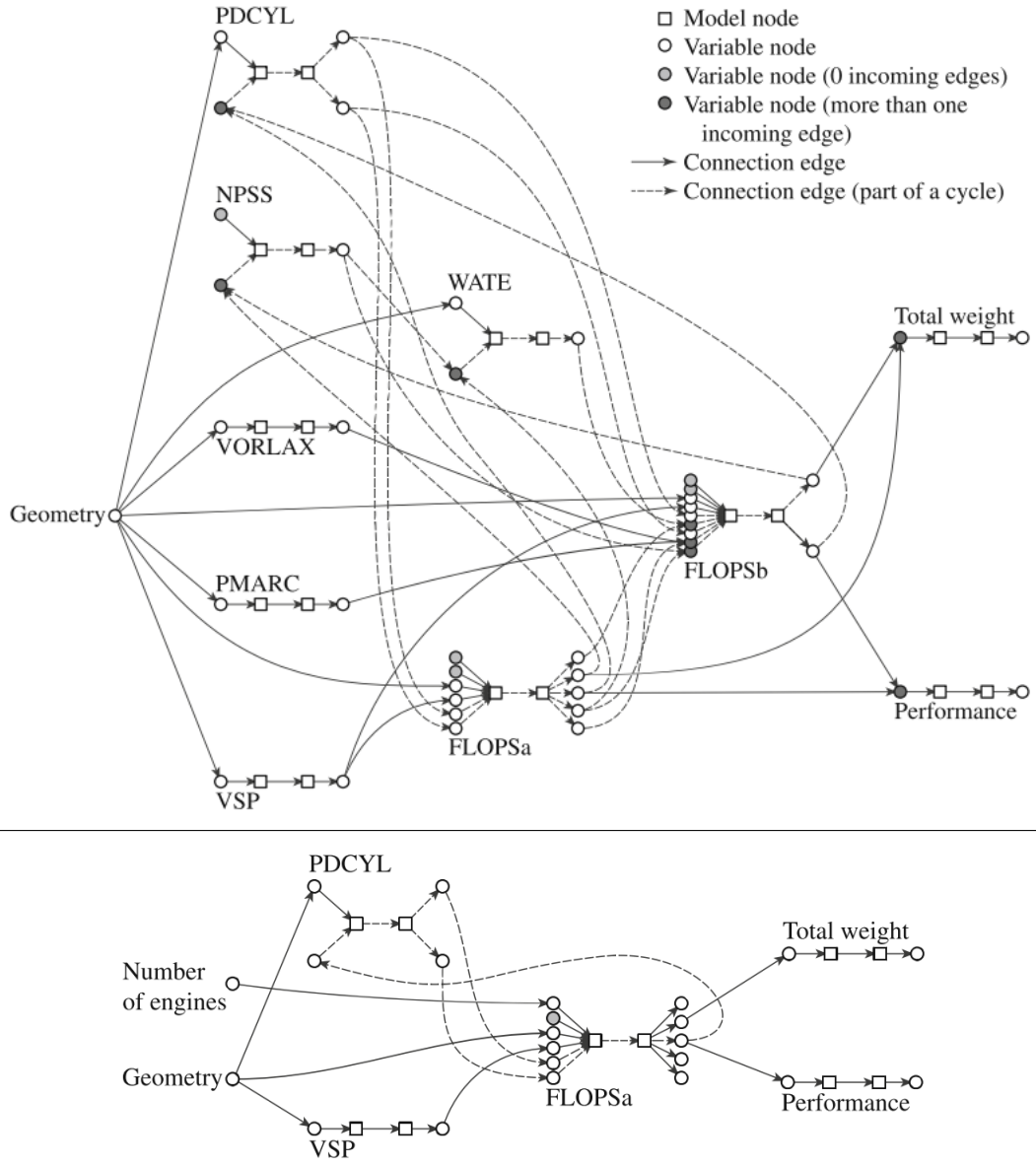


Figure 2.4: MCG (top) and possible FPG (bottom) for a subsonic transport aircraft example problem (Pate et al., 2014).

After performing a set of algorithms on the MCG, in which blocks that are unnecessary for the analysis are removed, the *Fundamental Problem Graph* (FPG) is established. The FPG represents the optimization problem without the implication of a solution strategy. For the same example, the FPG with the fewest number of cycles, or feedback connections, can be determined to calcualte the objective. This is represented by the FPG on the bottom of Fig. 2.4, where only those tools that minimize feedback couplings are used.

In a last step the *Problem Solution Graph* (PSG) is created using the FPG, where a solution strategy is applied according to a predetermined MDO architecture. Here, the optimizer (driver) blocks are coupled to the elements in the FPG that perform the optimization.

One of the strengths of this approach lies in the fact that it is based on directed graphs, or digraphs, which are a useful way to illustrate human-understandable workflows, as the shown in Fig. 2.4, and can easily be translated into machine-readable relationships. This opens up opportunities to apply computational algorithms on complex problems with a large amount of disciplinary interconnections to arrive at efficient MDO systems. The close relationship of the graph-based approach to the XDSM representation is also advantageous, since the XDSM can equivalently be expressed using directed graphs. Many problem decomposition and sequencing mechanisms use adjacency matrices such as the DSM to perform workflow optimizations, as summarized by Meier et al. (2007), which can easily be integrated into the graph-based system to generate workflows. Another benefit of this approach is its simplicity in the implementation. No additional frameworks or reasoning engines are required to make use of this method, and due to the simplicity to model graph-based relationships (directed links, or edges, between two nodes), there is a lot of freedom in the choice of the programming environment.

KADMOS adapts this approach to generate MDO workflows since it provides a convenient way to store, analyze, and represent tool and process data. Graph networks in combination with CPACS, which is introduced in the next section, are used to generate MDO formulations according to the described mechanism: first, a graph containing information on all tools in the database is created, before selecting the appropriate tools for the solution of the given MDO problem. In the last step, an MDO architecture is applied to the chosen tools according to the desired solution strategy. In this research, however, only the first two steps are relevant in which the *Fundamental Problem Graph* is determined. The application of the solution strategy is out of scope.

To finish this section, it should be noted that all graph manipulation and analysis tasks are performed using the *NetworkX* package in *Python*. *Python*, specifically, is used as the language of choice for the *Object-Oriented Programming* in this research due to its widespread use and rich library collection.

2.3. CPACS

The 3rd generation of MDO, as is shown in Figure 1.4, attempts to enable a more collaborative design process with a closer cooperation between decentralized and heterogeneous partners. As optimization problems grow in size, they often get partitioned into smaller subproblems according to certain disciplines or specialties, and then analyzed by the appropriate team of specialists.

In such a collaborative project, the communication and information exchange between engineers plays a crucial role in its efficiency, calling for a standardized data model. Nagel et al. (2012) argue that a standardized data model has a significant impact on the efficiency of collaborative projects in aircraft design, and therefore propose the use of the *Common Parametric Aircraft Configuration Schema* (CPACS).

CPACS allows multiple design teams to use the same data structure across disciplines and fidelity levels, reducing issues when models are passed on in later design phases (Rizzi et al., 2012). That way, design teams working on the aerodynamics and structural integrity of an aircraft, for instance, do not have to develop their own parametric model, but can adhere to one standardized schema that includes data sets that relate to each discipline, and thereby increasing the efficiency of interdisciplinary collaboration (Nagel et al., 2012). An example of this schema can be seen in Figure 2.5.

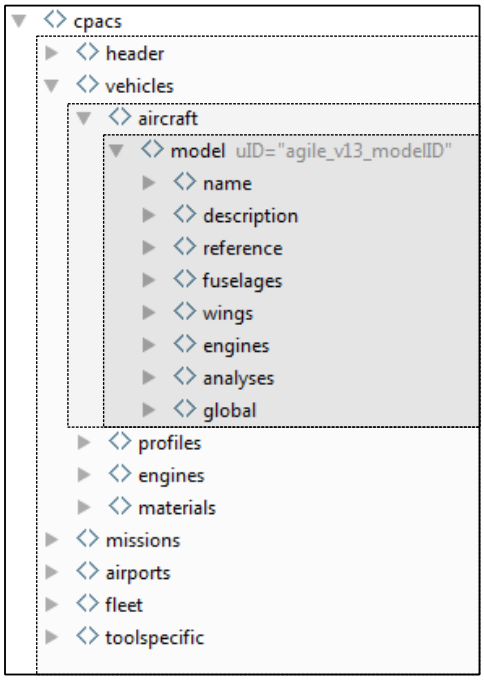


Figure 2.5: XML structure and element hierarchy imposed by CPACS.

The CPACS data model becomes the main node of communication between analysis tools and decreases the amount of interfaces needed to ensure an appropriate data exchange amongst them, as shown in Figure 2.6. This allows segregated disciplines to communicate with each other through one node without the need to know about data structures of other disciplines, and is referred to as the **central model approach** (Böhnke, D. ,Nagel, B., 2011).

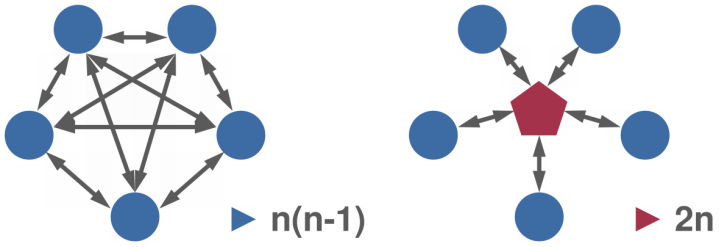


Figure 2.6: The Central Model Approach in CPACS (DLR, 2016).

CPACS becomes an important factor in this research with regard to the automatic generation of MDO workflows by coupling disciplinary tools and optimizers. Without CPACS, all interfaces between each discipline will have to be set up in order to ensure communication between all disciplines, which is time consuming, tedious, and impractical. This is especially problematic when new tools are added to the knowledge base, as the specialist will have to know each disciplinary tool that the added tool communicates with (or will do so in future). CPACS removes this barrier and allows for a quick integration of disciplinary tools into the data exchange network.

Because CPACS is an **XML schema** written in the *Extensible Markup Language* (XML) that is extensively used in this research, an overview of XML is given in the following section.

2.4. Extensible Markup Language (XML)

XML is a data format that follows *World Wide Web Consortium* (W3C) recommendation (W3C, 2016) and provides a human-readable and machine-readable structure for data storage. The main reason for using XML lies in its convenient format that allows humans to easily interact with data while retaining machine-readability. The syntax of XML is based on an intuitive tree-layout, with a root node containing all other nodes in the XML tree, as seen in Fig. 2.7.

```
<?xml version="1.0" encoding="UTF-8"?>
<book category="cooking">
   <title lang="en">Everyday Italian</title>
   <author>
      <name>
         <first>Giada</first>
         <last>De Laurentiis</last>
      </name>
      <birthdate>
         <day>22</day>
         <month>August</month>
         <year>1970</year>
      </birthdate>
    </author>
   <year>2005</year>
   <price>30.00</price>
</book>
```

Figure 2.7: Example of an XML structure describing a book (W3Schools, 2017).

The example shows the simplicity in storing data this way. By creating a tree structure, information can be "compartmentalized" and easily be understood by humans, making it a suitable data format for working even with large data sets. Each element in the XML tree is referred to as a **node** and must have a closing tag (such as <*price*>*30.00*</*price*>). The (single) node at the top of the XML tree is referred to as the **root**. Each node has exactly one parent node, but can have any number of children. Nodes that do not have any child-nodes must have a **value** and are referred to as **leaf-nodes** (W3C, 2016). Each node can carry attributes that can carry additional information about that node. A possible attribute is a "uID" attribute which extensively used CPACS. This attribute provides a unique identity to a node in order to distinguish it from other nodes, as will be presented later in the report.

Another feature that plays an important role in the generation of graph networks in KADMOS is the application of **XPaths**. Referring to the XML tree in the example, the XPath for the author's first name can be expressed as */book/author/name/first*, as it is basically following a specific branch in the tree to express the node in **string-format**. This way, nodes sharing a name such as <*year*> can easily be identified uniquely by */book/year* and */book/author/birthdate/year*. These identifiers are referred to as **XPaths** and are an important concept in the generation of graph networks from data that is stored in XML files.

The syntactic rules that define a correctly structured XML file can be summarized as follows (W3C, 2016):

- It must begin with the XML declaration
- It must have one unique root element
- Start-tags must have matching end-tags
- Elements are case sensitive
- All elements must be closed

- All elements must be properly nested
- All attribute values must be quoted
- Entities must be used for special characters

An XML file is said to be *well-formed* if and only if it follows all rules listed above. XML parsers often throw errors when parsing an invalid XML file, which is why these rules *must* be followed, even when inconvenient at times. A well-formed file structure, nonetheless, does not guarantee file-validity. Validity refers to the set of rules imposed on an XML file via an XML schema, which in this research is provided by CPACS.

2.5. Terminology

In this section, some common terms that used throughout this report are summarized to give the reader a better understanding of their meaning in the discussion ahead.

Tool Software applications are referred to as tools. When using tools in the context of graph structures, the term "function" or "function node" is used since it refers to a category of nodes that contains tools, objective functions and constraint functions.

Tool Configuration A selection of tools that is used in the *Fundamental Problem Graph* and all other graphs that are derived from that FPG. May also be referred to as "tool combination".

Aircraft Description An aircraft description refers to the elements of an XML file, since the elements are used to define the aircraft (or parts of it). A "complete" description refers to an XML file that contains all tool input and output elements present in the database.

System Inputs Tool input nodes that are not written to by any other tool. These inputs must be provided by the system (or user) in order for the tool to be able to execute.

MDO Formulation Inexectuable MDO workflow. Inexecutable refers to the fact that it is visualized in the form of graph networks, which can not be directly used to run the software tools that they describe.

MDO Workflow Refers to an executable MDO Formulation, or graph networks that were translated into scripts that can be executed to run software applications according to the solution strategy described by the MDO formulation.

3

Repository Components and Structure

As has been discussed previously, KADMOS is concerned with the "Formulation Phase" of the MDO-based development process shown in Fig.1.5, whose goal is the generation of an *MDO formulation* according to a problem definition. The MDO formulation is a representation of the information exchange between software tools, and is modeled using graph networks. Graphs have the advantage of representing large and complex data relationships for their analysis and manipulation, but are not very suitable as a user interface due to its unstructured nature. A better way to represent data that can easily be interpreted by both human and machine is found in the use of XML files, which can contain large data structures such as a full description of an aircraft, and still be easily be read and understood by humans. In this chapter, the repository components and their structure is discussed, giving insight into the requirements and necessities for the generation of graph networks.

The chapter starts with the introduction of tool input and output files and how they map to graph networks. Next, the Basefile is introduced, which is an important component of the repository. Following this, the validity of the tool files is discussed using a XML schema, before introducing the reader to the Info file and execution modes. Finally, an overview of the repository is given with a top-level figure showing structure of the KADMOS tool repository.

3.1. Tool Data Definition

In order to generate graph structures that model the information exchange between software tools, knowledge about the inputs and outputs of each of these tools is required. In the graph representation of a system of software tools, two distinct node types must be represented: variable nodes and function nodes, as seen in Fig. 3.1. **Function nodes** are referred to as such because they not only represent software tools, but also Objective Functions and Constraint Functions, which can all be expressed as functions requiring certain inputs and producing certain outputs. Each function is treated as a black box, and the system does not differentiate between them. **Variable nodes**, on the other hand, represent the inputs and outputs of such functions. It should be noted that throughout this paper, only *directed* graphs are used to establish data networks, since the flow of information is, in most cases, one-directional. For more information on the characteristics of directional graphs, refer to Chapter 2.

Although these graph structures are very convenient for the analysis of complex systems and the relationships that lie within, they are unsuitable for the storage of information that makes up these systems since its structure impedes humans in the handling and interpretation of its data. This

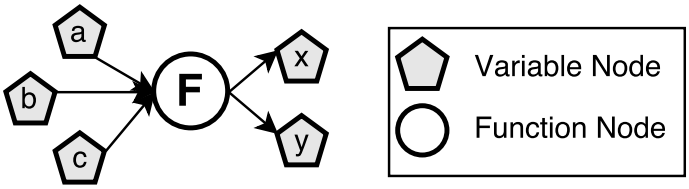


Figure 3.1: Example of a simplified graph structure using variable and function nodes.

has to do with the fact that graphs are commonly expressed using objects like dictionaries, lists, and tuples, which although can be read by humans, are difficult to understand when complexity increases. In order to provide a system that enables an easier access to and a better understanding of this information, XML files are used as a means of storing the data that make up those systems.

XML specifically has been chosen for two reasons, the first being that XML is a popular and widely-used format that has many implementation accross many programming languages, including *Python*. The second reason is that *CPACS* is written as an XML schema, which makes the use of XML highly convenient. Although alternative languages exist, using CPACS would require to "translate" the XML schema into the appropriate language, which is time consuming and unnecessary.

3.1.1. Input and Output Files

Since the graph representation relies on the input and output variable nodes of a function node for the construction of connection networks in a system of software tools, the requirement for input and output XML files becomes apparent. These XML files can separately be used to describe all input and output nodes by a function in a manner as shown in Fig. 3.2.

```
<input>
   <a>1</a>
   <b>2</b>
   <c>3</c>
</input>
```

```
<output>
   <x>10</x>
   <y>20</y>
</output>
```

Figure 3.2: Equivalent XML file structure for function *F* in the simplified graph of Fig 3.1.

To be able to set up an *MDO formulation* using specific functions, or software applications, each function in the graph must have exactly one input and one output XML file that describe the input and output variables for that specific function, respectively. One can observe that each variable node in the graph represents exactly the same leaf nodes that are present in the XML trees. Variable nodes in the input XML file have an outgoing edge towards the function node, whereas the nodes in the output XML file have an incoming edge from the function node. Both representations of the relationship between function and variable nodes are equivalent, as shown in Fig. 3.3.

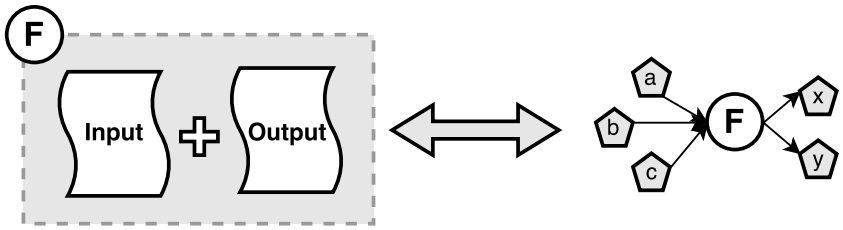


Figure 3.3: Full equivalence between tool XML files and graph representation.

The storage and division of data into input and output XML files allows the user to easily access,

understand, and modify information that generates the graph structures. Each of the tool-related XML files contains the exact nodes that are required for the execution of the application, as well as the exact nodes that it produces during execution. This implies that the nodes in these files are **case-specific**; they represent a specific aircraft description, and changes to that description require changes to the node structure and/or node values. All subsequent tool input and output definitions in this chapter represent case-specific aircraft descriptions.

```
<root>
   <input>
      <cpacs>
         ...
      </cpacs>
   </input>
   <output>
      <cpacs>
         ...
      </cpacs>
   </output>
</root>
```

Figure 3.4: Alternative storage of tool inputs and outputs in a combined XML tree.

It should be noted that input and output files are separated because of convenience. Input and output XML trees could be used in the same XML file, as shown in Fig. 3.4, but it requires the addition of a feature that enables KADMOS to distinguish input from output nodes. One way of implementing this could be in the use of extra root nodes that divide up the tree into two separate trees, where the *cpacs* root node becomes a child of an additional node, such as "*/root/input/wings/wing/~*". This, however, leads to many complications in the direct comparison of trees, for instance, and adds significant roadblocks to a fast and effective use of the XML files.

Input *F1*

```
<variables>
   <a> 1 </a>
   <b> 2 </b>
   <c> 3 </c>
</variables>
```

Input *F2*

```
<variables>
   <y> 8 </y>
   <z> 7 </z>
   <g> 12 </g>
</variables>
```

Output *F1*

```
<variables>
   <x> 7 </x>
   <y> 8 </y>
   <z> 7 </z>
</variables>
```

Output *F2*

```
<variables>
   <s> 8 </s>
   <t> 9 </t>
</variables>
```

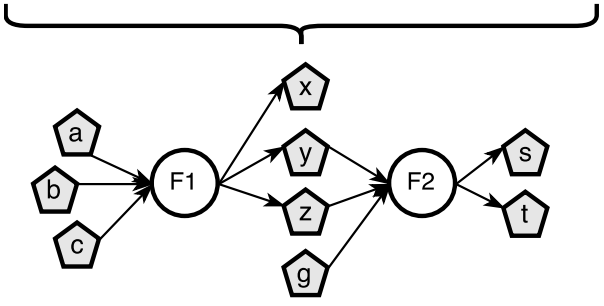


Figure 3.5: The shared input and output nodes of the two tools result in a coupling in the graph network.

By adding more design tools to the database, the amount of input and output XML files is expanded, which allows for the generation of more complex graph networks than the one seen in Fig. 3.3. A network of connections is established when variable nodes from different functions match, or in other words, when the input node of one tool match the output nodes of another tool. This possible information exchange is referred to as a **coupling** between two tools. Fig. 3.5 shows how two tools with a shared variable node in their respective input and output XML trees create a network in the graph structure with the direction of the exchanged node data indicated by the edge direction.

Now that the input and output XML files have been identified as the main ingredients for the generation of graph networks, the need arises that the nodes contained in each file *must* refer to the same node of the same aircraft description in order to establish a *valid* network. As has been shown through Fig. 3.5, graph structures only become networks if the nodes shared between functions match each other. However, consider the case shown in Fig. 3.6. It can be seen that the nodes contained in the input XML tree match both nodes in the output XML tree, highlighting the need to make the nodes *unique*. This is achieved through the use of nodes attributes in nodes with (possible) multiplicity which contain a unique identifier in order to prescribe them a unique identity. Using these so-called *uID-attributes*, matching nodes can be determined explicitly.

Output *F1*

```
<cpacs>
   <aircraft>
      <wing>
         <spar>
            <length> 17 </length>
            <width> 0.4 </width>
            <height> 0.3 </height>
         </spar>
         <spar>
            <length> 17 </length>
            <width> 0.4</width>
            <height> 0.3 </height>
         </spar>
      </wing>
   </aircraft>
</cpacs>
```

Input *F2*

```
<cpacs>
   <aircraft>
      <wing>
         <spar>
            <length> 17 </length>
            <width> 0.4</width>
            <height> 0.3 </height>
         </spar>
      </wing>
   </aircraft>
</cpacs>
```

Figure 3.6: Multiplicity in XML tree prevents an explicit match between input and output nodes of two functions.

Fig. 3.7 demonstrates how the presence of the uID-attributes in nodes with multiplicity ensures the correct and definitive coupling between functions, since the only way to be able to differentiate between those nodes is by assigning them unique identifiers. The importance of these unique identifiers in the generation of graph networks is discussed in further detail in Chapter 4.

Another important issue that relates to the validity of graph networks is the need to ensure that all nodes in each input and output XML file relate to the same aircraft description and use case. When looking at a system of functions that each take inputs and produce outputs, it is important to ensure that the nodes that are used to execute a software application actually refer to the intended aircraft description. Fig. 3.8 illustrates how, although having the same node structure and uID, both *spar*-elements have different values and therefore relate to different designs or use cases.

Since tool couplings in a graph network are based on the node structure and node *uIDs* within the input and output XML files, the graph network will provide the appropriate couplings between two tools based on these. However, it will not take into account the distinct values stored in the nodes. This means that values that are present in a specific node with a specific *uID* in one XML file could potentially have a different value than the same node in a different tool file. Although the system

Output *F1*

```
<cpacs>
  <aircraft>
    <wing>
      <spar uID="frontSpar">
        <length> 17 </length>
        <width> 0.4 </width>
        <height> 0.3 </height>
      </spar>
      <spar uID="rearSpar">
        <length> 17 </length>
        <width> 0.4</width>
        <height> 0.3 </height>
      </spar>
    </wing>
  </aircraft>
</cpacs>
```

Input *F2*

```
<cpacs>
  <aircraft>
    <wing>
      <spar uID="frontSpar">
        <length> 17 </length>
        <width> 0.4</width>
        <height> 0.3 </height>
      </spar>
    </wing>
  </aircraft>
</cpacs>
```

Figure 3.7: Use of uID-attribute allow node-matching explicitly.

```
<cpacs>
  <aircraft>
    <wing>
      <spar uID="frontSpar">
        <length> 17 </length>
        <width> 0.4 </width>
        <height> 0.3 </height>
      </spar>
    </wing>
  </aircraft>
</cpacs>
```

```
<cpacs>
  <aircraft>
    <wing>
      <spar uID="frontSpar">
        <length> 20 </length>
        <width> 0.5 </width>
        <height> 0.4 </height>
      </spar>
    </wing>
  </aircraft>
</cpacs>
```

Figure 3.8: Two XML trees referring to different wing spars due to differences in values.

will generate a valid coupling between the two tools, the resulting graph becomes inconsistent due to the potential clash in node values. The current implementation of KADMOS does not account for node values in the generation of function graphs, which can lead to inconsistencies in the MDO formulation, especially in a manual set up of the use case. It should be accounted for in future implementations.

To summarize, a valid and consistent graph network between provided software tools must fulfill the following three requirements for the relevant input and output XML files:

- XML node structure among tools must be identical
- XML node uID's among tools must be identical
- XML node values must be identical

In order to avoid inconsistencies among input and output XML files, an independent file, the *Base-file*, is introduced into the repository structure that carries the complete aircraft description for a specific use case. The Basefile used as a validator for all input and output XML files, ensuring that each node in these files carries the appropriate element structure, uID (if applicable), and node value referring to the relevant aircraft description and use case.

3.1.2. Basefile

The introduction of the Basefile serves two objectives in the generation of graph networks. Its first and major purpose was outlined above and is concerned with the validation of the aircraft description used in each input and output XML file. Since the Basefile contains the complete aircraft description, it provides a convenient platform from which to extract the relevant input and output nodes for each tool.

This prevents a "chicken-or-the-egg" problem in which the issue would arise whether to first generate input and output XML files in order to represent the input and output nodes of each software application, or to first generate a complete aircraft description, store it in the form of a Basefile, and subsequently extract the relevant tool files from it.

Since the Basefile already serves as a validator for each tool-related file as discussed above, it is convenient to use it as the starting point in the generation of a case-specific database, as shown in Fig 3.9.

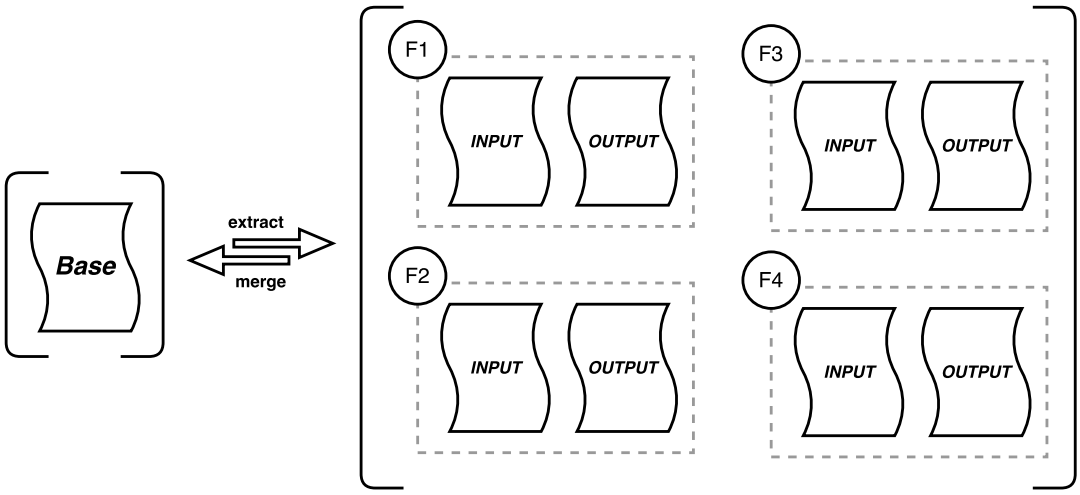


Figure 3.9: The Basefile is used to derive Input/Output XML files. Alternatively, merging all tool files results in the Basefile.

The other function of the Basefile is to provide a central data exchange point during the execution of software applications. Since it contains all nodes that are read by or written to by each function in the simulation workflow, it is convenient to utilize it as the read/write-file the workflow, as shown in Fig. 3.10.

This implies that it also serves as the initial design point in the MDO worklfow. Since the Basefile contains the accumulation of all relevant and applicable nodes of each tool, it contains a complete aircraft description, which is used as a *starting point* for an optimization or analysis. Conversely, this also means that the complete description is required in order to establish function couplings, generate graph networks and ultimately execute the tools that require particular nodes for execution.

To summarize, the Basefile has the following two purposes in the generation in KADMOS:

1. Present a validation and extraction platform for all I/O-files.
2. Serve as the central file for data exchange.
3. Provide an initial design point.

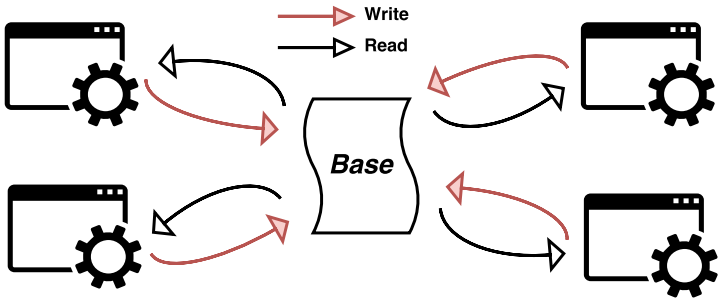


Figure 3.10: Basefile serves as the central point for data exchange.

It can be seen that the Basefile, as the focal point in the database, faces one hard requirement: it must contain a complete aircraft description, i.e. it must contain *all* nodes that are required or produced by the functions in the tool database. This opens up the question on where the complete description comes from, how it is generated, or how consistent it is. This, and other related issues and their solutions, are presented and discussed in Chapter 5.

It is now clear that in order to ensure the validity and consistency of the input and output XML files in the database, a Basefile is required for their validation, containing the complete aircraft description and therefore every node that is present in the input and output XML files. However, the Basefile itself is also subject to questions about its validity and consistency. Section 3.2 discusses this issue.

3.2. File Validity and the XML Schema

In order to be able to perform a meaningful and effective comparison between design cases and aircraft descriptions that are defined by the Basefile, a universal "language", or *XML schema* is needed, which dictates the structure of the Basefile and that can be used to validate it. An XML schema is, as introduced in Chapter 2, a set of rules imposed on an XML file in order to ensure a certain structure or characteristics to be present. It defines the legal elements in an XML file (and therefore also illegal ones) and regulates their properties such as attributes or values.

An example of an XML schema that is extensively used in KADMOS is the *Common Parametric Aircraft Configuration Schema*, or CPACS, which attempts to entirely describe an aircraft design using a hierarchical structure as seen in Fig. 2.5. By imposing CPACS on the Basefile of a use-case, the aircraft description must follow the structure that is forced upon it. A deviation from this structure results in an invalid XML file, which notifies the designer that the generated description does not follow the rules. In accordance with the discussion in Section 3.1, only the Basefile needs to be checked for validity since input and output XML files for each tool are extracted from it, and therefore follow its structure.

Because a prescribed structure is required to be present in the Basefile, the separate disciplinary tools are bound to the same "dictionary" of the aircraft design, and therefore use the same elements to describe it, providing an efficient way of standardizing the aircraft design and centralizing its data exchange. This is a crucial feature in the desgin of a repository that enables the coupling of independent software applications.

However, using XML schemas such as CPACS also has its drawbacks. Required nodes for the execution of certain tools must always be described by the schema, or determined through the use of already existing nodes that are defined in the schema. This can lead to inflexibilities in the addition

of tools to the repository since the XML schema is a **centralized set of rules**, and any extensions to the rules must go through a centralized screening process, which is relatively slow to adapt new structures (as compared to simply adding the missing tool nodes to the Basefile, for example). This also leads to the importance of correct versioning of the schema in the generation of Basefiles, since any changes in the schema that are not implemented in the Basefile will render it invalid.

Another weakness in the use of XML schemas, specifically CPACS, can be an overly *strict* application of its rules. The schema often imposes elements to be present in the aircraft description that may not actually be required or are not known at the time the description is generated. Imposed elements may also include descriptive elements that do not add any substance to the aircraft design, such as "description" or "name", and are only present for convenience, as shown in Fig. 3.11. Furthermore, the sequence of child-elements is frequently imposed on a parent element, which on one hand improves the manual comparison of two XML trees when performed, but on the other hand increases the difficulty to create valid aircraft description manually, even when the required nodes are known.

Valid (imposed) description

```
<cpacs>
   <aircraft>
      <wing>
         <spar uID="frontSpar">
            <name>Front Spar</name>
            <length> 17 </length>
            <width> 0.4 </width>
            <height> 0.3 </height>
         </spar>
      </wing>
   </aircraft>
</cpacs>
```

Invalid description

```
<cpacs>
   <aircraft>
      <wing>
         <spar uID="frontSpar">
            <height> 0.3 </height>
            <length> 17 </length>
            <width> 0.4 </width>
         </spar>
      </wing>
   </aircraft>
</cpacs>
```

Figure 3.11: Example of an imposed structure and an invalid node structure. The tree on the right in rendered invalid due to missing *name*-node and incorrect sequence.

At the moment, there exist implementations in the used *Python* packages in KADMOS that perform XML file validation according to a given schema. These, however, apply a strict validation as dictated by the schema. In order to get a file validation, a *soft* validation algorithm should be used that only checks for the "general" structure, and ignoring other rules such as the required children of a node, their sequence, etc.

A good example of nodes that are currently not described by CPACS, but used extensively within *AGILE*, are the *toolspecific* node and its descendants. The *toolspecific* node contains all tool settings for each present tool in the MDO formulation and are therefore necessary for both the integration and execution of software tools with the Basefile. Semantically, however, they are in a separate domain since they do not describe the aircraft design in itself, but instead provide the necessary information to execute the tools in a simulation workflow, such as the one as shown in Fig. 3.12.

By having a *toolspecific* node in the Basefile that does not correspond to the applied schema, any validation results in a negative outcome. This, naturally, is an unwanted effect since it negates the use of the XML schema. At the moment, no solution has been implemented that deals with this issue, but the following proposals are given to circumvent this problem.

The most obvious option would be to simply add *toolspecific* to the schema to ensure its validation. This, however, has a drawback since changes to the software tools occur more frequently than changes to the schema, as mentioned above, leading to inflexibilities in the addition of new tools or the adjustments of present tools. To overcome these inflexibilities, *toolspecific* could be added to

```
<cpacs/>
   <toolspecific>
      <eMWET>
         <wingUID>MainWing_wingID</wingUID>
         <loadcaseUID>Design-point_2.5g_MTOM_VMO_cruiseHeight</loadcaseUID>
         <preferences>
            <spar_pref>model</spar_pref>
            <fueltank_pref>given</fueltank_pref>
            <rib_pref>model</rib_pref>
         </preferences>
         <spars>
            <model_front_spar_uid>wing_Spar_FS</model_front_spar_uid>
            <model_rear_spar_uid>wing_Spar_RS</model_rear_spar_uid>
         </spars>
         <fueltank>
            <fueltank_start_y>0.0</fueltank_start_y>
            <fueltank_end_y>0.7</fueltank_end_y>
         </fueltank>
         <stringer_efficiency>0.96</stringer_efficiency>
         <display>0</display>
      </eMWET>
   </toolspecific>
<cpacs/>
```

Figure 3.12: Example of the currently implemented use of the *toolspecific*-node and its descendants for the tool *EMWET*.

the schema in a generalized form to contain child nodes that are not "tool dependent", and communicate the necessary tool settings though their attributes, which are not dictated by the schema, as shown in Fig. 3.13.

```
<root>
   <cpacs/>
      <aircraft/>
   <cpacs/>
   <toolspecific>
      <eMWET>
         <wingUID>MainWing_wingID</wingUID>
         ...
      </eMWET>
   <toolspecific>
</root>
```

```
<cpacs/>
   <toolspecific>
      <tool uID="EMWET">
         <setting uID="emwet_wingUID">
            MainWing_wingID
         </setting>
         ...
      <tool/>
   <toolspecific>
<cpacs/>
```

Figure 3.13: Two proposed alternatives for the application of tool settings in KADMOS.

Another option could lie in the separation of tool settings and aircraft description by adding a new root node. Instead of having *toolspecific* as a child node of *cpacs*, it could be implemented as its sibling, each sharing a parent that represents the new root node, as show in Fig. 3.13. The CPACS validation can then be adjusted to only affect the descendants of the *cpacs* node.

An alternative approach to the ones mentioned could be the use of separate files, and the introduction of a "settings" file which is schema-independent. This would make a lot of sense semantically, since the tool settings are the only nodes the design engineer should be concerned with (and access) in the set-up of the workflow when no manual changes to the aircraft description are necessary.

To summarize, the validation of the Basefile as well as the input and output XML files is an important task in the generation of graph networks in order to ensure a standardized aircraft description and a consistent information exchange. In order to be able to create graph networks, additional data about the available tools in the repository is required, which is stored in the *Info-File*. The structure and use of a tool *Info-File* is discussed in the next section.

3.3. Tool Metadata

As has been mentioned, one of the main goals of the AGILE program is to improve the collaboration among heterogeneous design teams that contribute to MDO workflows with their expertise and software applications. By coupling different disciplinary tools from partners in a project, information is exchanged between the applications.

Since software tools that are used in simulation workflows often fall under proprietary restrictions and may be generally unavailable to the design engineers directly, the tools added to the KADMOS repository are considered **black boxes** and are treated as such.

This means that the software tools that are coupled in an MDO workflow must receive and produce information according to a known structure, resulting in the necessity for the input and output files that are validated by a common XML schema. However, in order to be able to analyze and trade-off MDO workflows using certain tool configurations, data *about* the tools themselves must be available. In KADMOS, this data is stored in the **Info-File**. The Info-File contains all metadata about the software tool itself, storing information such as tool name, version, description, and so on.

```
{
   "general_info": {
      "name": "EMWET",
      "version": 1.0,
      "creator": "A. Elham",
      "description": "Calculates the structural weight of a wing."
   },

   "execution_info":
   [
      {
         "mode": "Main",
         "description": "The main mode in EMWET.",
         "prog_env": "MATLAB",
         "toolspecific": "eMWET",
         "runtime": 20,
         "fidelity": "L1",
         "precision": 0.05
      }
   ]
}
```

Figure 3.14: Example of an *Info-File* for the structural analysis tool *EMWET*.

The information stored in the Info-File is separated into two domains. One describes data such as the one mentioned above and is referred to as "General Info", whereas the other domain describes its "Execution Info". "Execution Info" provides KADMOS with the ability to perform meaningful analyses of these graph networks since tool properties such as *Runtime*, *Fidelity*, or *Precision* can highly affect the choice of the tool configuration for the MDO formulation. To ensure a flawless operation of the system, both the "General Info" and the "Execution Info" *must* be provided when a tool is added to the KADMOS tool repository.

3.4. Execution Modes

Referring back to Fig. 3.14, it can be seen that the "Execution Info" contains a list of dictionaries that are headed by the key *"mode"*. The "mode" info describes the **execution mode** of a tool, and at least one execution mode must be present in the Info-File. Execution modes describe the different sets of tool settings in which the tool can be executed. Since software tools often provide the opportunity to be executed under varying settings, different input nodes may be required between the execution

modes, and possibly producing different outputs.

Since the different execution modes may require and produce different nodes, each execution mode would require its own input and output file in order to unambiguously describe the input and output nodes for the tool under the defined settings. This, however, can lead to a congestion of files in the repository for tools that have many different execution modes, or settings under which they can be executed.

Instead of defining input and output files for each execution mode of a tool, a different approach is implemented in KADMOS. A *"modes"*-attribute is added to the tool files to describe which nodes are affected by which execution mode. Consider the example in Fig. 3.15 for tool *"F"*.

Input *F*

```
<cpacs>
   <aircraft>
      <wing>
         <spar uID="frontSpar">
            <length> 17 </length>
            <width> 0.4 </width>
            <height> 0.3 </height>
         </spar>
      </wing>
      <materials>
         <material uid="aluminium2024">
            <density modes="m1">
               2700
            </density>
            <unitcost modes="m2">
               1000
            </unitcost>
         </material>
      </materials>
   </aircraft>
</cpacs>
```

Output *F*

```
<cpacs>
   <aircraft>
      <wing>
         <spar uID="frontSpar">
            <weight modes="m1">
               2000
            </weight>
            <cost modes="m2">
               5400
            </cost>
         </spar>
      </wing>
   </aircraft>
</cpacs>
```

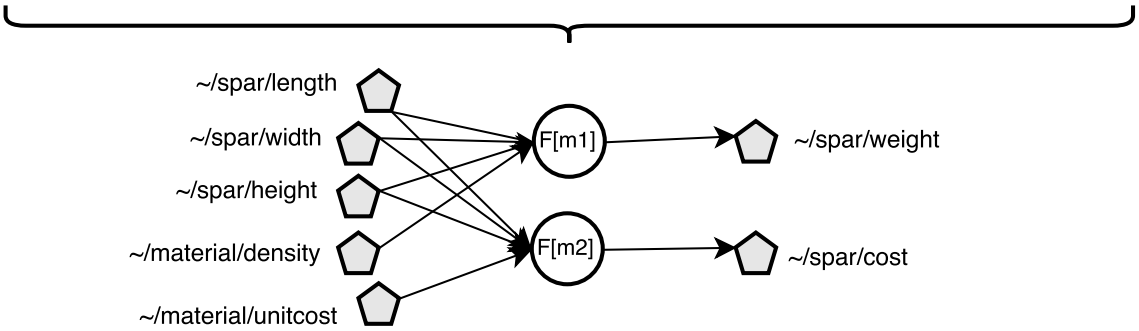


Figure 3.15: Example of two execution modes *m1* and *m2* for function *F*, leading to two separate function nodes referring to the relevant execution mode.

Two execution modes *m1* and *m2* are present in the tool, computing the spar weight and spar cost, respectively. By adding the *"modes"*-attribute to the tool files, the engineer can indicate which execution mode applies to which input and output nodes, discarding the need for additional tool files to describe the inputs and output per execution mode.

In the example in Fig. 3.15, it can be seen that while the *spar*-dimensions in the input file do not have a *modes*-attribute present, both the *density* and *unitcost* nodes carry the *modes*-attribute according to its execution mode. The absence of the *modes*-attribute in a node indicates to the system that the node is used in *all* execution modes.

An important detail in the generation of graphs is that each execution mode is represented by a separate function node related to the tool. It can be seen in Fig. 3.15 that both execution modes are treated as completely separate function nodes in the graph, which facilitates trade-offs with regard to the tools that are used in the MDO workflow, and the modes that they are executed in.

The use of *modes*-attributes in the tool files results in a cleaner repository where only *one* set of input and output files has to be present per tool. This prevents cluttering of large repositories where many tools with many different execution settings are present, simplifying the tool data collection and management.

3.5. Overview

This section treated the most important elements that are required to establish graph networks, which can subsequently be analyzed and processed into simulation workflows and used for Multi-disciplinary Design Optimizations. Fig. 3.16 gives a summarized overview of the required elements.

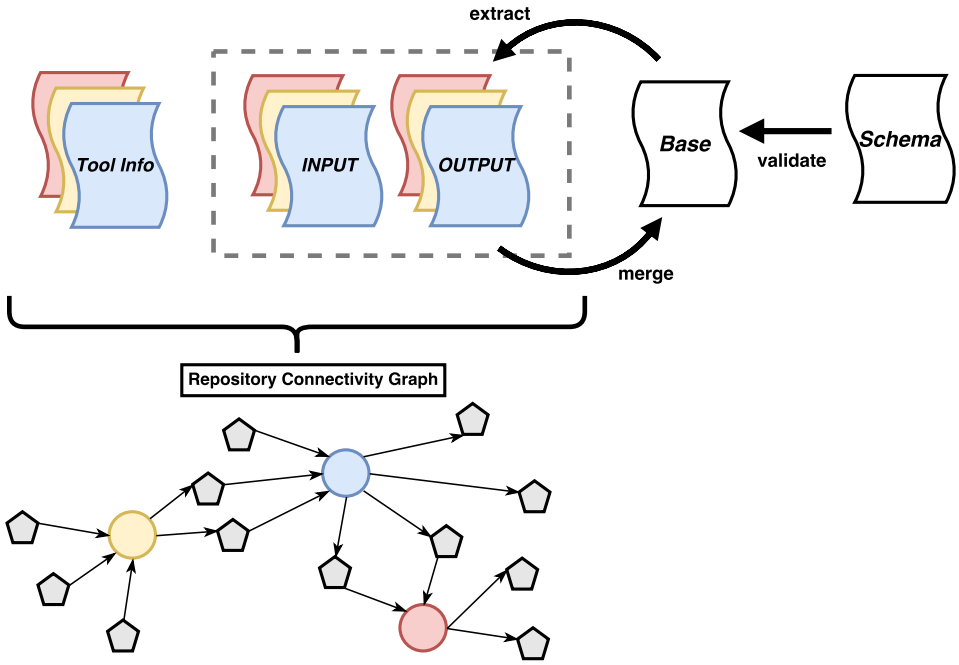


Figure 3.16: Top level view of the graph generation workflow.

Here, it can be seen that in order to be able to generate the *Repository Connectivity Graph*, input and output XML file as well as Info-Files are required for each tool in the repository. The information contained in these files is loaded into KADMOS, which transforms it into a graph network, as will be presented and discussed in Chapter 4.

Two possible ways exist to set up the use case. Either the input and output files are externally provided to KADMOS and are merged into the Basefile, or the Basefile is provided and the relevant tool input and output files are extracted from it.

Since the Basefile compiles all present nodes in the tool files into one complete description, it is used as a validator for the tool files, while itself being validated by the applied XML schema. Because it contains all tool nodes, the Basefile also serves as the main point of data exchange between the tools in the MDO workflow.

In Table 3.1, an overview is given for each file type and its function in the creation of graph networks using KADMOS.

Table 3.1: Overview of all files and their functions in KADMOS.

File	Function
Info File	• contains general tool information • defines tool execution modes
Input/Output File	• contains complete tool input and output description
Basefile	• contains complete aircraft description • validates input and output files • central point for data exchange
XML schema	• standardizes aircraft description • validates use case description • enables comparison between use cases

Now that the structure of the tool repository in KADMOS is known, the process of generating graph networks from the information contained in the repository can be discussed in the upcoming chapter.

4

Generating Graph Networks

The previous chapter dealt with the structure and composition of a tool database that allows *KADMOS* to process its data and create graph networks of disciplinary software applications that model the exchange of information. The mechanism of generating these graph structures and thereby creating the foundation on which the executable simulation workflows are built, is discussed in this chapter.

It should be noted that this chapter is concerned with the creation of the *Repository Connectivity Graph (RCG)* that represents the connectivity and relationships between *all* tools in the provided tool database for a specific use case. The formulation of the *Fundamental Problem Graph* (FPG) is discussed in Chapter 5, whereas the *MDO Data Graph* and *MDP Process Graph* are out of scope for this research, as mentioned in Chapter 2.

The chapter discusses the process of transforming XML nodes into graph structures in Section 4.1, going into more detail on the procedure in Section 4.2. Section 4.3 summarizes the class structure of KADMOS in the context of the types of generated graphs.

4.1. From XML Files to Graph Networks

As presented in Section 3.5 of the previous chapter, several files are required to be present in the database in order to generate complete graph networks between software tools. These files include the input and output XML files for each tool, the Basefile, the Info-File, and the XML schema. An activity diagram of the workflow that transforms the information contained in these files into graph structures is presented in Fig. 4.1, showing how these files are utilized. What follows is a step-by-step discussion of this figure.

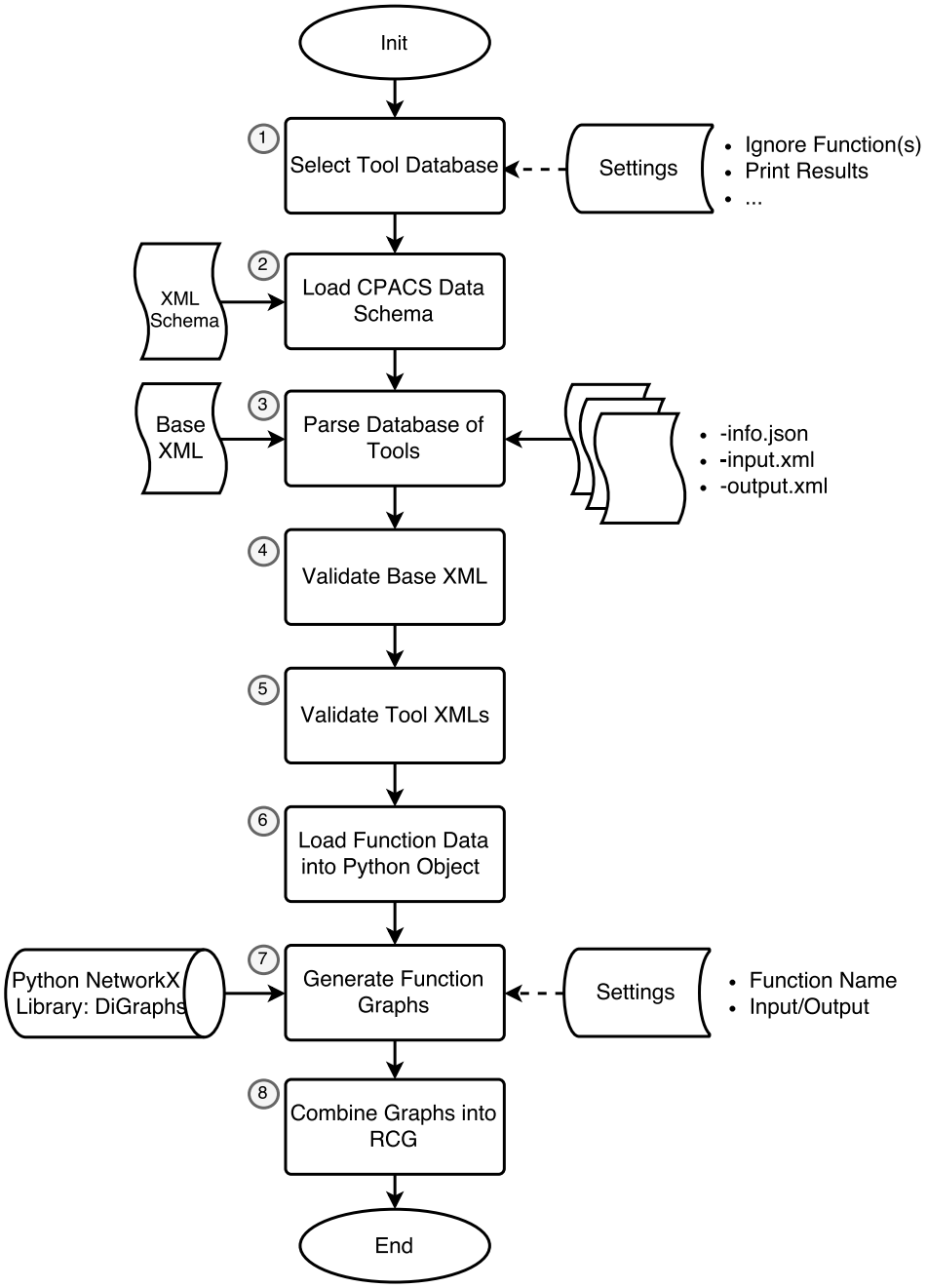


Figure 4.1: Activity diagram showing the generation of graph networks using the tool database, summarized in 8 steps.

It can be observed that the generation of graphs can be divided up into two parts. The first part includes steps ① through ⑥ and is concerned with parsing, validating, and processing the relevant files before loading them into a *Python*-object that summarizes all required data.

In step ① through ③, the design engineer selects the relevant use case and parses the applicable file into the system. Certain settings can be applied to exclude certain tools from loading, for example, as would be necessary in the case of temporary unavailability. Depending on the parser, the loaded XML and JSON files are automatically ensured to be well-formed and therefore follow their respective syntax. A detailed description of the parsed files and why they are required for *KADMOS* is discussed in Chapter 3.

After loading all required files, the Basefile and all input and output XML files are validated in steps ④ and ⑤. As previously described in Chapter 3, the validation of the Basefile is according to the loaded XML schema, whereas the input and output XML files are validated through a comparison with the Basefile. After a successful validation, the information in the parsed files is loaded into the *Python*-object *function-data*, which captures all relevant data according to each function for further processing, taking a "snapshot" of the use case. Any changes that occur to the database after it has been loaded will not appear in the graph structures, and therefore also also not taken into account in the simulation workflows. The structure of the *function-data* object is shown in Fig 4.2.

```
[
   {
      "info":{
         "general_info": {
            "name": str,
            "version": float,
            "creator": str,
            "description": str
         },
         "execution_info":[
            {
               "mode": str,
               "runtime": int,
               "precision": float,
               "fidelity": str
            },
            ... # list of all execution mode attributes
            ]
      },
      "input":{
         "leafNodes":[
            {
               "xpath": str,
               "tag": str,
               "attributes": dict,
               "value": str,
               "level": int
            },
            ... # list of all input node attributes
         ]
      },
      "output": {
         "leafNodes":[
            {
               "xpath": str,
               "tag": str,
               "attributes": dict,
               "value": str,
               "level": int
            },
            ...# list of all output node attributes
         ]
      }
   },
   ...
]
```

Figure 4.2: Function-data object used to collect all relevant information in tool database, shown for one tool.

The second part of the activity diagram in Fig. 4.1 includes steps ⑦ and ⑧ and deals with the generation of graph structures from the parsed data. In step ⑦, the *Python*-library *NetworkX* is utilized to generate separate graphs for each function in the tool database. Settings can be used here to create graphs using specified function, or only inputs or outputs for those functions, in order to allow the design engineer to inspect functions separately. The abstracted workflow for this step is as follows.

For each function in the database:

1. Create empty graph.
2. Get function name and mode
3. Get input and output leaf nodes according to function and mode
4. Create input and output edges using function and leaf nodes
5. Add edges to graph
6. Add node and graph attributes

The node attributes added to the graph in the last point are mostly concerned with the visualization of the graph and include the node *category* (such as "function" or "variable"), *label*, and *level*. The node *level* describes the "depth" of the leaf element and can be useful in the visualization of graph couplings. The leaf element in */cpacs/aircraft/wing/spar/length*, for example, has the node-level 4, whereas the root counts as level 0.

In the last step of the of the activity diagram of Fig. 4.1, all graphs are combined into one single graph network that captures all relationships between each tool, as shown in Fig 4.3. This graph network represents the *Repository Connectivity Graph* and forms the foundation for all other graphs in the creation of the simulation workflows using *KADMOS*.

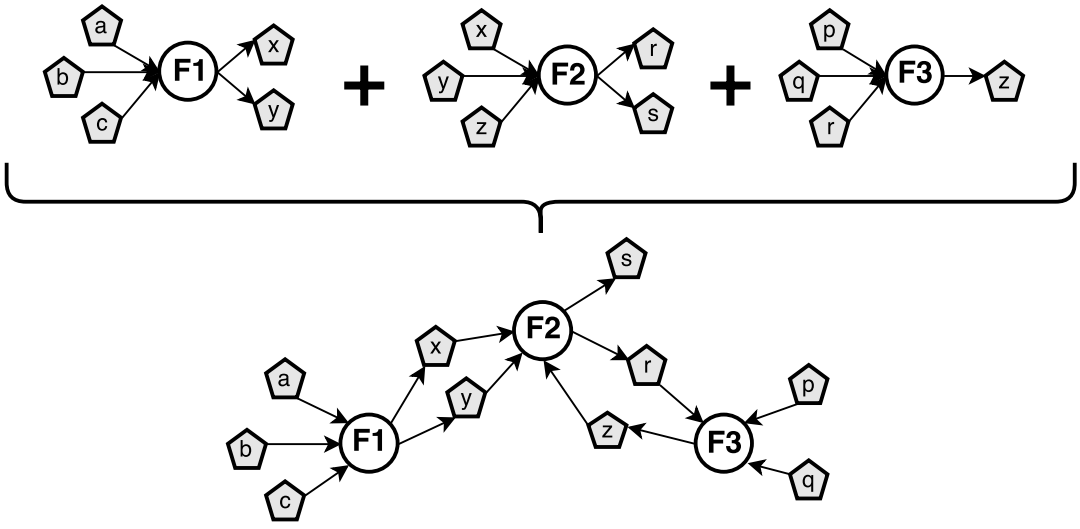


Figure 4.3: Separate graphs for each function are combined into one graph network.

4.2. Transforming XML into Graph Nodes

Many of the previously shown figures, such as Fig. 4.3, contain variable nodes that are labeled according to the leaf node in the associated XML tree. The name of the leaf node, however, is an unsuitable descriptor for variable nodes in a graph network due to the possibility of mismatches between similarly named nodes that should otherwise not result in tool couplings.

Output *F1*

```
<cpacs>
   <aircraft>
      <wing>
         <spar uID="frontSpar">
            <length> 17 </length>
            <width> 0.4</width>
            <height> 0.3 </height>
         </spar>
      </wing>
   </aircraft>
</cpacs>
```

Input *F2*

```
cpacs>
   <aircraft>
      <fuselage>
         <length> 38 </length>
         <width> 3.9 </width>
         <height> 4.1 </height>
      </fuselage>
      <wing>
         <spar uID="frontSpar">
            <length> 17 </length>
            <width> 0.4 </width>
            <height> 0.3 </height>
         </spar>
      </wing>
   </aircraft>
</cpacs>
```

Figure 4.4: XML Trees of two tools with identical leaf nodes in separate branches.

Consider Fig. 4.4, where the *spar*-node contains identical leaf nodes as the *fuselage*-element. The corresponding graph structure of this XML tree, using the leaf element as the node descriptor, can be seen in Fig. 4.5. It can clearly be seen that the use of leaf elements as the node descriptor is not appropriate, since it does not reflect the couplings described by the XML files because the graph structure merges similarly named nodes.

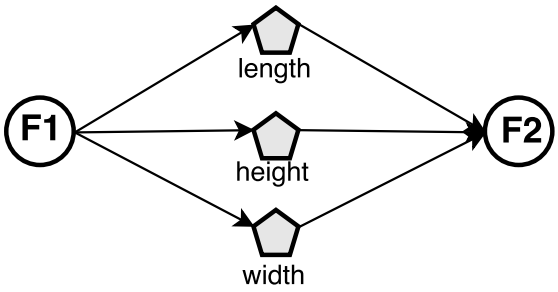


Figure 4.5: Merged variables nodes due to inappropriate node descriptors representing the XML trees in Fig. 4.4.

As explained in Chapter 2, the couplings established in graph networks must occur through matching branches in the input and output XML trees of a tool database, instead. A suitable alternative for a node descriptor, therefore, is found in the use of the *XPath*. As mentioned in the introduction of its concept in Chapter 2, the XPath of a node describes the "path" between the root of the XML tree and that node, and allows to express its "location" in form of a *string*. This characteristic can be used to describe variable nodes in graph networks to make them unique and thereby avoiding the merge of variable nodes, ensuring appropriate couplings between tools, as seen in Fig. 4.6.

One issue that arises with the application of XPaths lies in their use of indeces when describing the path of nodes that have siblings with the same name. The XML tree seen in Fig. 3.7, for instance, would have XPaths */cpacs/aircraft/wing/spar[1]/length* and */cpacs/aircraft/wing/spar[2]/length* representing the *length*-nodes of each spar.

By disregarding the uID's in the ancestry of a variable node, there is no guarantee for an accurate coupling of graph networks. Therefore, the use of the **uID-XPath** is implemented in KADMOS, converting each "regular" XPath to a uID-XPath and use it as a descriptor for all variable nodes. The above XPaths are represented as */cpacs/aircraft/wing/spar[frontSpar]/length* and */cpac-*

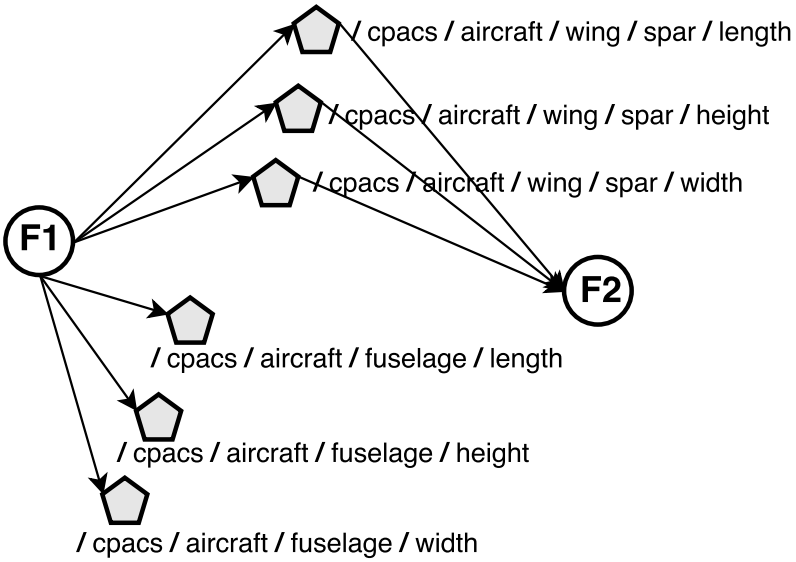


Figure 4.6: Accurate representation of the XML trees in Fig. 4.4 using XPaths as node descriptors.

s/aircraft/wing/spar[rearSpar]/length using the uID-XPath. It is therefore highly important for the coupling of nodes to not only assure that the node structure is correct so that the XPaths of coupled nodes match, but also that their uID's match, as these are also compared when generating graph structures.

4.3. KADMOS Class Structure

As has been discussed in the previous sections, KADMOS aims at generating simulation workflows using a specific tool database, which is achieved with the use graph networks. In order for the system to be able to convert a graph network into a executable simulation workflow with an applied solution strategy, the *Repository Connectivity Graph* generated using the tool database must undergo several "stages" in which the tool configuration, execution sequence, and MDO architecture are determined, essentially following the approach introduced by Pate et al. (2014). To keep graphs that represent the separate stages comparable, compatible, and consistent, a universal class structure is necessary in KADMOS. This class structure is summarized in form of a UML diagram in Fig. 4.7.

The graphs generated in KADMOS rely on two parent classes, *KnowledgeBase* and *KadmosGraph*. The *KnowledgeBase*-class serves as the initiator for KADMOS and controls the workflow shown in the activity diagram of Fig. 4.1. Its instance represents the complete information that is found in the tool database for the chosen use-case.

The other main parent class in KADMOS is *KadmosGraph*. As can be seen in Fig 4.7, *KadmosGraph* encompasses a convenient class inheritance mechanism in which the *NetworkX*-graph class serves as the base-class. This allows *KadmosGraph*-instances and inheriting child-classes to perform the same functions as *NetworkX*-graphs, making the examination and analysis of graph networks in *each* graph type highly effective. The *KadmosGraph*-class provides many useful capabilities to its child-classes, such as the ability to iterate, handle and modify graph nodes and edges, among others.

The direct successor of the *KadmosGraph*-class is the *RepositoryConnectivityGraph*-class which, as the name implies, provides an abstract definition for any RCG that is generated using the tool database provided by the *KnowledgeBase*-class. It can be seen that the RCG presents the basis for all other graph types in KADMOS because it contains all data that is available in the system. In order to

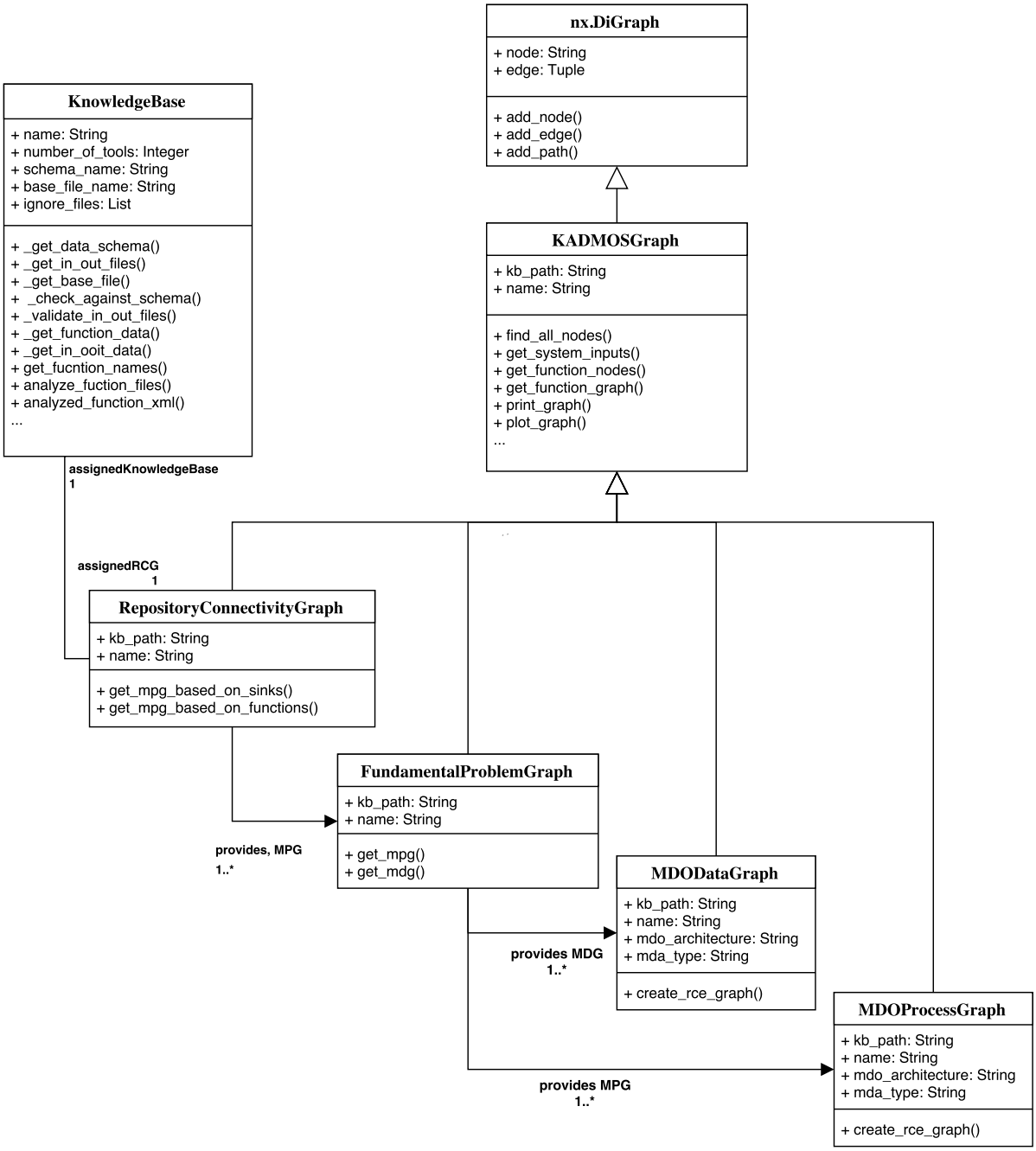


Figure 4.7: Class structure in KADMOS.

acquire any of the subsequent graphs, the RCG must always be generated first.

The RCG, however, does not provide any information on the tool configuration or execution sequence, and instead describes the system of tools as a whole. The tool configuration for the simulation workflow is determined using the RCG as a starting point, and results in the *Fundamental Problem Graph*, or FPG, which is further discussed in Chapter 5. It can be stated, however, that the FPG does not encompass the data and process flow, or the solution strategy of the workflow. Instead, it contains the collection of selected tools for the computation of chosen objective(s) without a predetermined execution sequence.

The solution strategy, or the way that the various tools are coupled and their information is exchanged, is represented by the *MDO Data Graph* and *MDO Process Graph*, which are abstracted through the *MdoDataGraph* and *MdoProcessGraph* classes, respectively. The generation of these two graphs is out of scope for this research, but it can be summarized that these two graphs determine the MDO architecture of the MDO system given a certain configuration of tools.

Looking at the overview of the KADMOS system shown in Fig. 1.5, it can be seen that the first step in the MDO developemnt process is completed by generating the *Repository Connectivity Graph*. Since the RCG contains all repository information, it represents the foundation upon which the MDO formulation is build. The information, however, is case-specific and refers to a single use case. This means that for each new use case, a new repository must be set up and the tool inputs and outputs configured towards the new aircraft description. This issue prevents an agile set up of MDO workflows and does not enable information reuse. In the next chapter, the issues with the case-specific repository are discussed in more detail, and an approach to solve them is presented.

5

Building an Agile Tool Database

In the previous chapters, a methodology to generate *Repository Connectivity Graphs* using given tool repositories was provided. The generated RCG serves as the basis for the creation of MDO formulations that are ultimately transformed by KADMOS into executable MDO workflows. The set up of the RCG as described in the foregoing chapters, however, has some significant drawbacks when it comes to the agility of the set up process. To counter these issues, an extension to the case-specific repository was developed and is discussed in this chapter.

The chapter structure is as follows. First, the limitations of the case-specific repository are presented in Section 5.1. Next, the abstract tool repository is introduced and its differences with the existing implementation are discussed in Sections 5.2 and 5.3. Before presenting the generation of the Basefile is presented in Section 5.5, the *Fundamental Problem Graph* is treated in Section 5.4. At last, the extraction of the tool files is presented in Section 5.6 and an overview of the existing case-specific and the extended repository is given in Section 5.7.

5.1. Limitations of Case-Specific Approach

The methodology to generate graph networks for a given aircraft description was presented in the previous chapters and can briefly be summarized as follows. First, the input and output XML files for each tool in the tool database are extracted from the Basefile containing the complete aircraft description. Together with the tool data file of each corresponding tool, the input and output nodes in the XML files are transformed into graph structures, and ultimatelty combined into a graph network.

The resulting web of tools now describes a network limited to the specific use case described by the Basefile and the utilized tools in the database, and although the graph can be transformed into an executable MDO workflow, two major issues arise in its generation which restrict its usability and reusability. The first issue relates the workload that the design engineer faces in the set up of new use cases or modifications to existing ones ones. The second issue concerns accuracy of the supplied aircraft description with regard to the tools in the repository. Both issue are presented in the following discussion.

5.1.1. File and Schema Modification

The first issue concerns the static nature of a case-specific aircraft description, where *any* changes to any of the files in the tool database require to be adapted in all other files due to the reciprocal relationships among Basefile, input and output XML files, and the XML schema, as seen in Fig. 5.1.

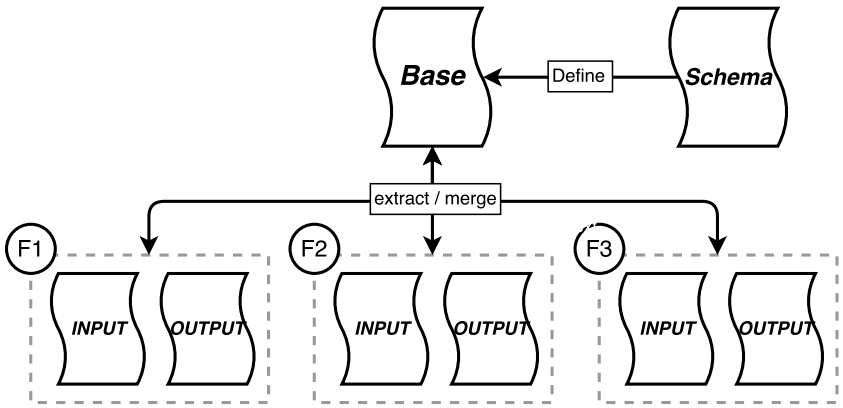


Figure 5.1: Reciprocal relationship between Basefile and tool input/output files. The basefile is either created by merging all tool input and output files, or the tool files are extracted using the Basefile. Both ways of setting up the use case are equivalent.

Consider a valid set-up in which multiple tools are used to generate a workflow, with each tool being interconnected through a set of shared nodes. Each of these tools is also "connected" to the Basefile since it contains all existing nodes and is used to validate the tool files. If in this case *any* changes occur to the tool files in the database, all other files, including the Basefile, must be adjusted to integrate these changes due to their interconnection. Failing to do so leads to the dissimilar node structures in the tool files, which in turn leads to loss of couplings among tools where these couplings should exist.

Since software tools regularly undergo changes and are often in development, any use case set-up that utilizes these tools must keep up with the modifications of said tools. These changes can include added or eliminated input nodes and output nodes, modification of execution modes etc., which can, especially in large use cases with many coupled tools, lead to a significant overhead and tedious adjustments of coupled tools to accomodate these changes in the tool database. This is especially true for the distributed nature of the tools in the *AGILE* program, where tool data and expert knowledge are not always immediately available to designer.

Similarly, small changes in the use case, or Basefile, can entail many adjustments to the tool files in the database. If the aircraft description is modified to carry two wings spars instead of one, as shown in Fig. 5.2, this change in the design must be propagated through all tools in the database that affect or are affected by the wing structure. Although these changes seem to be small and easily fixable, many small changes with a large tool database tend to accumulate and cause tedious, repetitive, non-creative work.

Lastly, modifications to the XML schema must be taken into account similarly to modifications in the design. Fig. 5.3 shows how small changes to the XML schema can change the required node structure in the Basefile and therefore all tool files. Similarly to the previous case, small changes can accumulate in large tool databases and result in considerable workload for the engineer.

Original Wing Design

```
<cpacs>
   <aircraft>
      <wing>
         <spar uID="frontSpar">
            <length> 17 </length>
            <width> 0.4</width>
            <height> 0.3 </height>
         </spar>
      </wing>
   </aircraft>
</cpacs>
```

Modified Wing Design

```
<cpacs>
   <aircraft>
      <wing>
         <spar uID="frontSpar">
            <length> 17 </length>
            <width> 0.4 </width>
            <height> 0.3 </height>
         </spar>
         <spar uID="rearSpar">
            <length> 17 </length>
            <width> 0.4</width>
            <height> 0.3 </height>
         </spar>
      </wing>
   </aircraft>
</cpacs>
```

Figure 5.2: Small changes in the use case, such as the addition of a wing spar, can cause many adjustments to the tool database since all affected tool files must be adjusted.

Original Schema

```
<cpacs>
   <aircraft>
      <wing>
         <spar uID="frontSpar">
            <length> 17 </length>
            <width> 0.4</width>
            <height> 0.3 </height>
         </spar>
      </wing>
   </aircraft>
</cpacs>
```

Altered Schema

```
<cpacs>
   <aircraft>
      <wing>
         <spar uID="frontSpar">
            <dimensions>
               <length> 17 </length>
               <width> 0.4 </width>
               <height> 0.3 </height>
            </dimensions>
         </spar>
      </wing>
   </aircraft>
</cpacs>
```

Figure 5.3: Any changes in the XML Schema must be adopted to maintain validity.

The changes shown in the foregoing discussion represent simple and approachable examples. Realistically, changes can affect tens or hundreds of nodes through simple adjustments in the database, leading to many work hours spent by the engineer to identify and implement the affected changes in the tool database. Therefore, an **automatic system** must be implemented that simplifies and accelerates the implementation of modifications to the files in the database, and thereby relieve the engineer of this tedious workload.

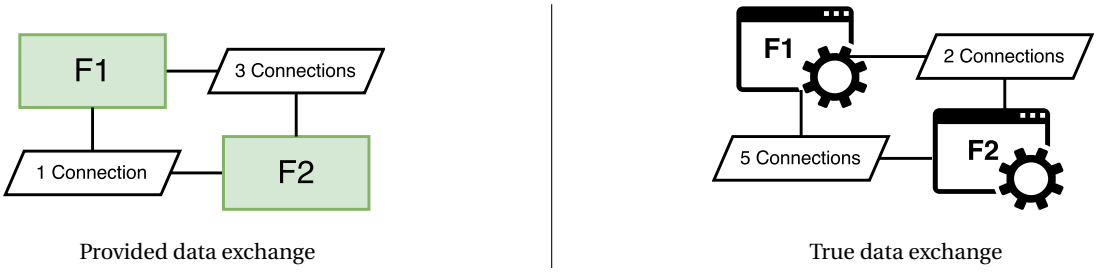
5.1.2. Use Case Accuracy

The second main issue with the set-up of the use case has to do with the consistency and accuracy of the aircraft description that is used to generate the graph networks. As has been mentioned in Chapter 3, the creation of the tool files in the current approach is based on the assumption that the aircraft description *completely and correctly* represents all input nodes that each tool requires and all output nodes that each tool produces.

This, however, is a very questionable assumption in that it is very difficult to correctly describe the output (and input, to some extent) of a tool if no knowledge about its intrinsics is provided. It has been stated that each tool is treated as a *blackbox*, which is the reason why metadata about the

respective tool is needed in order to implement it into a graph network. This metadata, however, does not give any indication on the tool *behaviour*, or the nodes and values that the tool produces when it receives a given set of inputs.

Because of the fact that it can be difficult to predict the output of a tool without executing it, deriving its inputs and outputs from an externally provided definition may result in an inaccurate representation of the data exchange within the system of tools. This may lead to potentially inefficient optmization workflows, as shown in the example of Fig 5.4.



Provided data exchange

True data exchange

Figure 5.4: Assumed and actual data exchange between two tools.

Here, tools *F1* and *F2* are assigned a sequence in a workflow that follows the minimization of feedback loops. Tool *F1* has three provided feed-forward connections and one feedback, which, depending on the applied solution strategy, may yield the most computationally effecicient optimization workflow.

However, in reality, the provided connections may not represent the *actual* information exchange when the tools are executed, leading to a loss in efficiency in cases where the amount of feedback connections is higher than previously assumed. The difference in *provided* and *true* information exchange can be caused by a number of reasons such as tool settings or modifications to the tool itself, but the result is the same: by not being able to define the aircraft description using the tools in the database, and instead rely on an external supply of the aircraft description, the *true* information exchange can not be modeled with complete confidence.

The core of the issue is that an externally provided aircraft description, or use case, can not be guaranteed to represent tool inputs and outputs without executing the tools using appropriate tool settings and comparing its *true* inputs and outputs to the provided description. Therefore, the approach described in the past chapters is enhanced by implementing the **generation of the aircraft description through execution of the tools** that are used in the optimization workflow, and not rely on an external supply of the use case. This guarantees that KADMOS can work with the true exchange of data in the system of tools. Optimally, the extended repository implementation will provide engineers with the ability to modify existing use cases or generate new descriptions through modifications of the tool settings only, and without having to reconfigure the complete set up.

5.2. Schematic Tool Repository

In order to assert that **(1)** the workload stays manageable when changes to any file in the tool repository occur, and **(2)** that the aircraft description for a use case represents the tool inputs and outputs in the MDO formulation, the top-level approach depicted in Fig. 3.16 in Chapter 3 is extended by adding a **schematic layer** to the tool repository, as shown in Fig. 5.5.

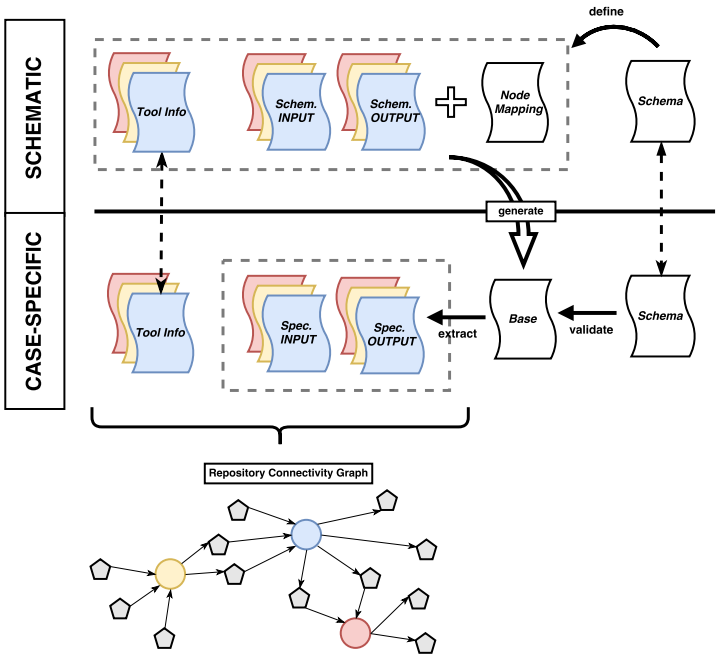


Figure 5.5: Layered approach using schematic data to generate a case-specific aircraft description.

The schematic representation does not indicate the exact information that a tool requires or produces, but instead depicts the abstract state of information. To visualize how the schematic information is structured, consider Fig. 5.6 which compares the case-specific description of a tool input to its schematic counterpart.

It can be seen that the schematic representation does neither include values or attributes, nor does it provide information about the multiplicity of nodes in the XML tree. This means that the only knowledge that is required to be present in the schematic tool database is the description of the tool inputs and outputs based on the definition imposed by the applied XML schema.

Using this concept, graph networks can be generated in a similar manner as described in Section 4. Tool couplings are established on the basis of the node structure, indicating the **abstract couplings** among tools, which only show the type of information that is exchanged between tools, but neither value nor amount can be deducted.

The idea is to use a schematic, or **case-independent** tool input and output description to establish a graph network and analyze the *abstract* exchange of information. Using this graph network, a schematic MDO formulation can be defined by selecting the a **tool configuration** (a combination of tools) appropriate for the MDO Problem and solution strategy, and execute these tools to derive the use-case, as will be shown in the following example.

Consider the case given in Fig. 5.7 where the graph shows a schematic network of the three tools *HANGAR*, *Q3D* and *EMWET* listed in Table 5.1. The tools are are coupled through the variable groups describing the "Wing Geometry" and "Lift Distribution", and *EMWET* produces the variable "Wing Weight".

Since this schematic graph is case-independent, there is no information on any values or amount of data that is being exchanged between the tools; the only information available to the system is that there exists *some* data exchange based on the node structure of the input and output files. Looking at the sample input data for the lift distribution of *EMWET* in Fig. 5.8, it can be seen that variable nodes such as *cfz* and *cmy* are exchanged for each strip of a wing segment between *Q3D* and

Case-Specific Description

```
<cpacs>
   <aircraft>
      <wing>
         <spar uID="frontSpar">
            <materialUID>
                  aluminum7075
               </materialUID>
            <length> 17 </length>
            <width> 0.4 </width>
            <height> 0.3 </height>
         </spar>
         <spar uID="rearSpar">
               <materialUID>
                  aluminum2024
               </materialUID>
            <length> 17 </length>
            <width> 0.4</width>
            <height> 0.3 </height>
         </spar>
      </wing>
      <materials>
         <material uID="aluminum2024">
            <modulus> 7.6e10</modulus>
         </material>
         <material uID="aluminum7075">
            <modulus> 7.2e10</modulus>
         </material>
      </materials>
   </aircraft>
</cpacs>
```

Schematic Description

```
<cpacs>
   <aircraft>
      <wing>
         <spar>
            <materialUID/>
            <length/>
            <width/>
            <height/>
         </spar>
      </wing>
      <materials>
         <material>
            <modulus/>
         </material>
      </materials>
   </aircraft>
</cpacs>
```

Figure 5.6: Comparison between a case-specific and schematic aircraft description.

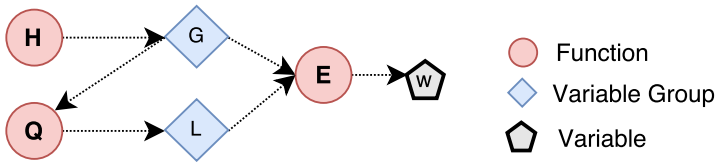


Figure 5.7: Schematic graph with three tools *HANGAR* (**H**), *Q3D* (**Q**), *EMWET* (**E**) coupled through Variable Groups "Wing Geometry" (**G**) and "Lift Distribution" (**L**), with *EMWET* producing Variable "Wing Weight" (**w**).

Table 5.1: Required and produced node variables by each tool.

Tool	**Requires**	**Produces**
HANGAR	-	Wing Geometry
Q3D	Wing Geometry	Lift Distribution
EMWET	Wing Geometry, Lift Distribution	Wing Weight

EMWET. However, the amount of strips that are actually generated for the use case is completely dependent on the provided wing geometry and the tool settings for *Q3D*, which means that it can only be determined after *HANGAR* and *Q3D* are executed.

One possible execution sequence for an MDO formulation with the variable "Wing Weight" as the objective can be seen in Fig. 5.9. The workflow shows how *HANGAR* is first executed to take an *empty* aircraft description in ① and provide the "Wing Geometry" variables that are written to the file in ②. The "Wing Geometry" variables completely depend on the tool settings in *HANGAR*, and only

```
<cpacs>
   <vehicles>
      <aircraft>
         <model>
            <analyses>
               <loadAnalysis>
                  <loadCases>
                     <aeroDataSetsForLoads>
                        <aeroDataSetForLoads>
                           <wings>
                              <wing>
                                 <wingUID/>
                                 <coefficients>
                                    <cfz/>
                                    <cmy/>
                                 </coefficients>
                                 <segments>
                                    <segment>
                                       <strip>
                                          <cfz/>
                                          <cmy/>
                                       </strip>
                                       <segmentUID/>
                                    </segment>
                                 </segments>
                              </wing>
                           </wings>
                        </aeroDataSetForLoads>
                     </aeroDataSetsForLoads>
                  </loadCases>
               </loadAnalysis>
            </analyses>
         </model>
      </aircraft>
   </vehicles>
</cpacs>
```

Figure 5.8: Sample of the schemtic input for the lift distribution of *EMWET*.

after its execution the *case-specific* nodes become known, which can subsequently be used by other tools to perform their analysis. Since the file in ② now contains the "Wing Geometry", *Q3D* can be executed using its nodes and write the calculated "Lift Distribution" to the file in ③. Since the file in ③ now has both the "Wing Geometry" and "Lift Distribution" provided by the previous two tool executions, *EMWET* can now use this information to determine the objective "Wing Weight".

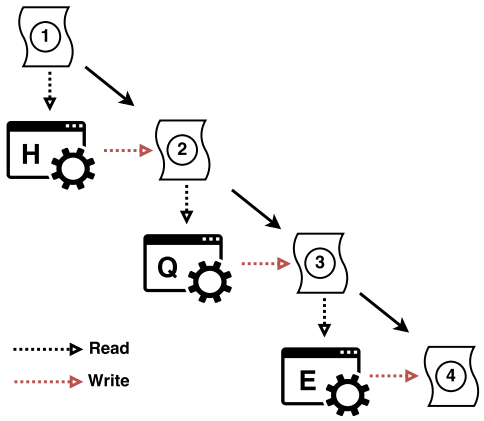


Figure 5.9: Possible execution sequence for the example case in Fig. 5.7.

From the previous discussion it became clear that in order to create a *case-specific* MDO formu-

lation, the exact inputs and outputs for the tools must be known. The exact inputs and outputs, however, can only be established after the executing the tools, which in turn would require the set-up of a case-specific MDO formulation, creating a "chicken-or-the-egg" problem. This problem is circumvented by establishing a *schematic* MDO formulation and using it as a basis to execute the appropriate tools to generate the use case. This basic principle allows KADMOS to derive a use case by executing tools based on *schematic* couplings that are provided through the schematic input and output files.

The creation of a case-specific aircraft description using the schematic tool inputs and outputs ultimatley allows for a more agile generation of MDO formulations. Scheamtic tool files must be set up only once, since they represent an abstract tool description, and can be used and reused in any use case without the need to reconfigure the files.

5.3. Schematic Workflow

As presented in Fig. 5.5, the schematic layer of the tool repository has a similar structure as the specific layer, with the major difference being the absence of the Basefile and the presence of the Node Mapping. Node Mapping plays an important part in the generation of the Basefile as well as in the extraction of case-specific tool input and output files, and is discussed in Section 5.6.

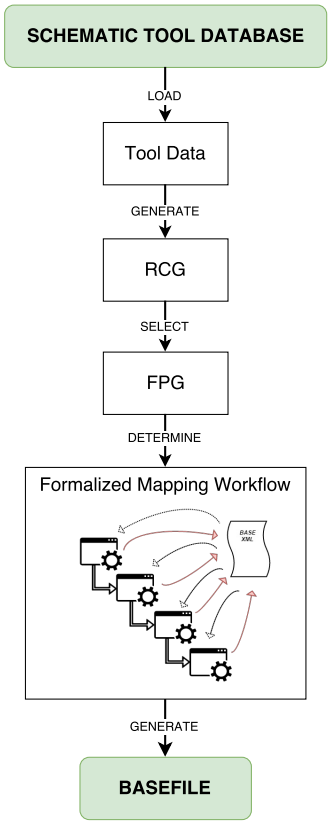


Figure 5.10: Schematic workflow to generate Basefile.

All other components, such as the tool input, output files and information files, as well as the XML schema, are still present and have the same roles as in the case-specific layer. The tool input, output and information files are required in the generation of the schematic MDO formulation, whereas the XML schema is used to define their node structure to ensure a standardized aircraft description.

These files are used in a schematic MDO formulation to derive a case-specific MDO formulation as shown in Fig 5.10. This workflow shows how, similarly to the case-specific workflow, the schematic tool data is collected and transformed into a graph structure represented by the RCG.

In order to create a case-specific Basefile, however, the schematic RCG must be reduced to the appropriate *Fundamental Problem Graph*, or FPG, before a formalized workflow can be applied in which the software tools can be executed. This formulation of the FPG and it importance to the workflow are discussed in the following section.

5.4. Fundamental Problem Graph

As was discussed in Chapter 4, the *Repository Connectivity Graph* represents the complete information on the tools that are present in the tool repository. All tool inputs and outputs for each tool execution mode, as well as all couplings between these tools are provided by the RCG, presenting a complete descrption of the repository. However, not each tool is required in an MDO formulation that follows a certain MDO Problem, which is why the RCG is **reduced** to the *Fundamental Problem Graph* before a solution strategy is applied.

Consider the example RCG on the top of Fig. 5.11. Here, the complete tool repository is represented with five present tools and the "variable of interest", depicted in red. It can be seen that in order to determine the variable of interest, *not all* tools are required, or even necessary in a workflow. Therefore, the RCG on top can be *reduced* to the tools appropriate for the solution strategy of the MDO formulation by removing certain tools from the graph.

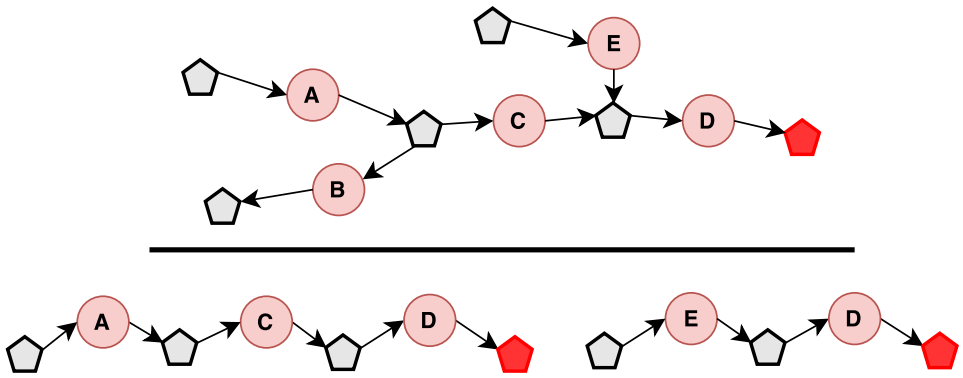


Figure 5.11: Two possible *Fundamental Problem Graphs* (bottom) for a variable of interest (red) based on the *Repository Connectivity Graph* (top).

This can be done in many ways, as seen on the bottom of the figure where two possible FPGs are shown. For more complex repositories, however, the amount of possible FPGs increases significantly which leads to issues with computational performance. Large graphs can have several million possible combinations of tools, or **tool configurations**, for a specific MDO Problem, and analyzing each one in order to determine its suitability is often infeasible. Therefore, a strategy to reduce the amount of analyzed tool combinations has been developed and presented in the upcoming sections.

5.4.1. Graph Sinks and Sources

The "variable of interest" in Fig 5.11 represents a **graph sink** and is a convenient approach to determine possible tool configurations that can be used in an FPG. By imposing a sink on a function or variable in a graph, an MDO Problem can directly be implemented in the MDO formulation. The

MDO Problem can, for instance, be the search for the optimal wing weight in a wing optimization. By defining the wing weight as the sink, all *simple paths* to the sink can be investigated to determine which tools directly affect that node. Tools that have no influence on the sink are not interesting for an MDO workflow and can be dismissed, such as function node *B* in Fig. 5.11.

Another approach that was considered for the generation of FPGs is the use of a **graph source** in combination with a graph sink. A graph source would usually represent tools that establish the geometry of an aircraft or aircraft wing using top-level requirements, such as the *Initiator Design and Engineering Engine* for a blended wing body (Vos and Dommelen, 2012). The graph source initiates paths towards the sink that could possibly be used for an FPG.

This *source-to-sink* approach, however, was discarted since it requires a categorization of tools into possible sources and non-sources, which would limit the flexibitly of the system in cases where the use of the "source-tools" would not be necessary. Instead, each tool in the repository acts as a potential source, enabling a more flexible analysis of their paths or combination of paths towards the sink.

To summarize, the *Fundamental Problem Graph* is established by analyzing all possible tool paths, or combinations of tools, to the sink. The desired tool combination is then selected from all the possible combinations and serves as the FPG. An activity diagram of this process as implemented in KADMOS is shown in Fig. 5.12.

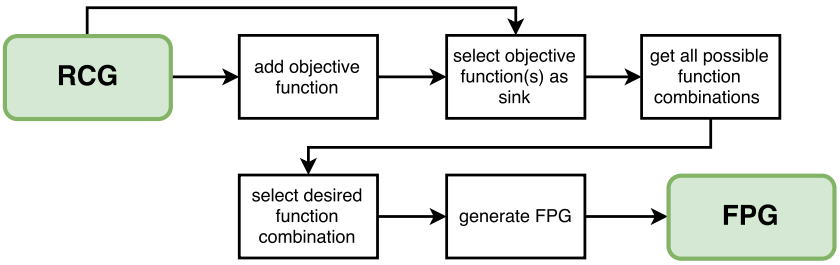


Figure 5.12: Activity diagram for the generation of a *Fundamental Problem Graph*.

In order to simplify the analysis of possible paths towards the sink, the RCG is transformed into a *function graph*. An example of a function graph is given in Fig. 5.13, where the variable nodes of the original graph are converted to edges to maintain the coupling between the tools. Although variable information is lost, this representation allows for a more computationally effective estimation of possible paths towards a sink due to the lack of parsed variable nodes. Since all variables are removed in the function graph, KADMOS can use any defined function as the sink, usually the **Objective Function**.

It is important to note that the use of function graphs in this context is limited to finding an appropriate tool configuration. Once the desired tool configuration is selected, the *Fundamental Problem Graph* is derived from the *Repository Connectivity Graph* with all relevant variable nodes present.

5.4.2. Tool Configurations

The function graph of an RCG enables KADMOS to perform an efficient analysis of all **function paths** towards a pre-defined sink, as shown in Fig. 5.14. Since each function in the function graph has an influence on that sink, the possibility arises to consider *any* combination of the tools in the graph for the FPG.

This means that in order to decide which tool configuration to use for the FPG, criteria such as

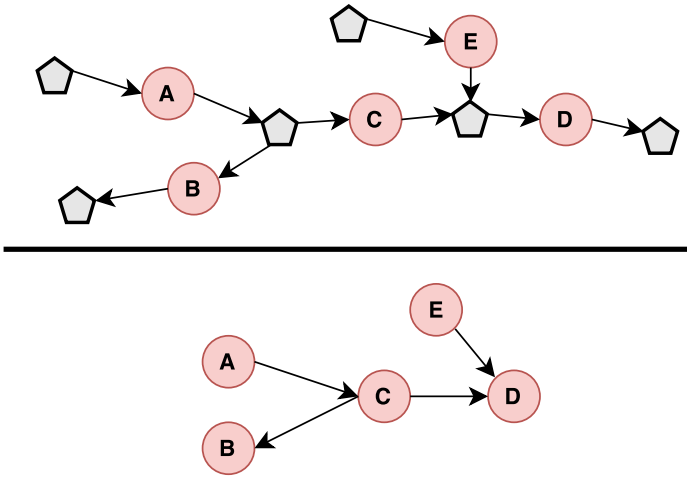


Figure 5.13: The function graph (bottom) derived from an ordinary graph (top).

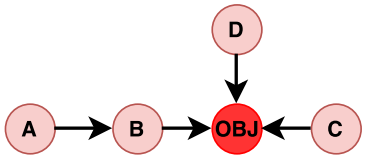


Figure 5.14: Function graph containing a sink (red).

the amount and type of tools, the amount of system inputs, the amount tool couplings etc., that are present in the graph of a given configuration need to be analyzed. To support the engineer in choosing a certain tool configuration over another, and to perform a meaningful **trade-off** between the tool configurations, each configuration must be evaluated using these criteria to determine the most suitable configuration for the MDO formulation.

A "brute force" approach towards finding all possible tool paths to the sink could be done by simply analyzing *all* possible combinations of tools. Considering the function graph shown in Fig. 5.14, this would result in a total of

$$\binom{4}{1}+\binom{4}{2}+\binom{4}{3}+\binom{4}{4}=15 \tag{5.1}$$

evaluated tool combinations. The 15 combinations result from the fact that no combinations can be duplicate and that the order does not matter. The order in phase of the selection process does not matter because the tool sequence, as will be discussed later, is not defined in the *Fundamental Problem Graph*. The FPG merely describes the tool configuration for the solution of the MDO problem.

This "brute force" approach, however, quickly becomes infeasible due to the exponential growth of total tool configurations. When considering that the relationship

$$\sum_{0}^{n}\binom{n}{k}=2^n, \tag{5.2}$$

stands (Spivey, 2007), a graph containing 15 functions, for instance, that have a path to the Objective Function, will result in

$$\sum_{k=1}^{15}\begin{pmatrix}15\\k\end{pmatrix}=\sum_{k=0}^{15}\begin{pmatrix}15\\k\end{pmatrix}-\begin{pmatrix}15\\0\end{pmatrix}=2^{15}-1=32,767 \tag{5.3}$$

possible combinations, with each extra function adding considerable computational effort.

To alleviate this issue, an algorithm using **Combination Subsets** was developed that avoids the analysis of tool configurations which include paths that have no influence on the Objective Function, as shown in the example of Fig. 5.15.

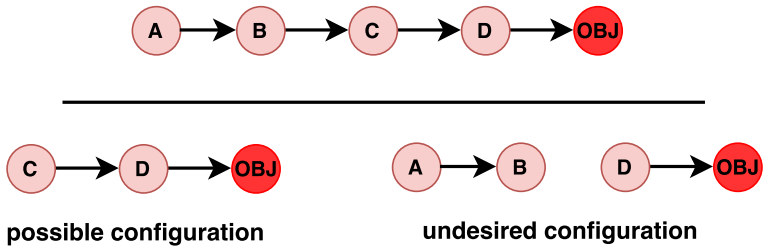


Figure 5.15: Possible and undesired tool configurations (bottom) for an Objective Function of a given RCG (top).

Here, it can be seen that the undesired configuration includes tools A and B that, when executed, produce output variables that are not utilized by any function in the workflow since there are no outgoing couplings to any other tool. These configurations should be avoided since they add computational effort without any benefit for the MDO system. A "brute force" analysis would lead to $2^n-1=15$ inspected configurations in this case, although only 4 *desired* configurations exist, namely $(ABCD)$, (BCD), (CD), and (D).

By applying *Combination Subsets*, the algorithm groups all tool paths into subsets according to the **longest tool path**, which in this example leads to a single subset $[(ABCD),(BCD),(CD),(D)]$. Subsequently, all combinations between the subsets are determined while avoiding undesired configurations as the one shown in Fig. 5.15.

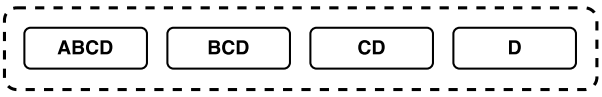


Figure 5.16: The only combination subset for the graph in Fig. 5.15.

The advantage of applying Combination Subsets lies in the division of tool paths into groups of paths that are subsets of each other. Since combinations between paths where one includes the other do not lead to any new combinations, they do not have to be considered when inspecting the possible tool combinations for the FPG. The combinations in this example would lead to only four inspected configurations since only one subset exists.

It should be noted that, in the case that configuration (CD) is selected, the inputs of tool C become *System Inputs* and have to be provided externally, illustrating the necessity of using trade-offs in the creation of *Fundamental Problem Graphs*. The application of trade-offs between tool configurations is presented in Chapter 6.

Going back to the example in Fig. 5.14, three subsets exist: $[(AB),(B)],[(C)],[(D)]$. In order to consider *all* combinations between these subsets, the algorithm adds an *emtpy* path to each matrix. The resulting matrices are then multiplied using *matrix chain multiplication* to determine all combinations of tools, such that:

$$[(AB),(B),()]\,x\,[(C),()]\,x\,[(D),()] = R \tag{5.4}$$

where R is the resulting matrix of all possible tool configurations. Table 5.2 shows the combinations in the resulting matrix and compares them to the brute force approach.

Table 5.2: Inspected ("x") and uninspected("o") tool configurations for trade-off of example graph of Fig. 5.14.

Combination	(A)	(B)	(C)	(D)	(AB)	(AC)	(AD)	(BC)
Brute Force	x	x	x	x	x	x	x	x
Combination Subsets	o	x	x	x	x	o	o	x
Combination	(BD)	(CD)	(ABC)	(ABD)	(ACD)	(BCD)	(ABCD)	
Brute Force	x	x	x	x	x	x	x	
Combination Subsets	x	x	x	x	x	o	x	

The application of *Combination Subsets* has been investigated on a variety of graphs with a differing amount of nodes and complexity in order to determine the performance gain of this approach. In this context, the measure for graph complexity was defined as the ratio between the amount of edges and the amount of nodes in the graph, or e/n. Fig. 5.17 shows how an increase in the amount of nodes in a graph leads to more inspected tool configurations, as expected, since more tools lead to more combinations among them. Similarly, the complexity e/n of the graph tends to increase the amount of configuration analyses, which is also according to expectations since the increase in graph edges results in an increase in possible paths towards the sink.

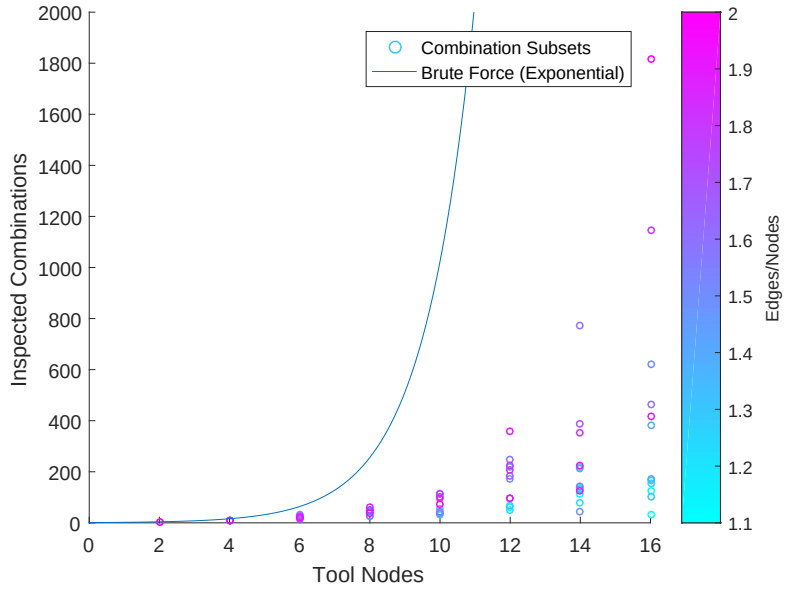


Figure 5.17: Analyzed configurations versus number of nodes per graph for "Brute Force" and "Combination Subsets".

An important fact that is illustrated by this graph is how much improvement this approach offers as compared to the "brute force" approach. By rejecting all undesired tool combinations and using *Combination Subsets*, KADMOS must inspect significantly fewer combinations, which is especially noticable when more than 6 nodes are present in the graph, leading to a better system performance in large graphs.

Despite the fact that the amount of inspected tool combinations increases significantly in larger graphs with high complexity, the *Combination Subsets* will always lead to the same or fewer inspections than the "brute force" approach by definition because it disregards any combinations that lie on the same path.

Although *Combination Subsets* result in a drastic reduction in inspected tool combinations, Fig. 5.17 shows that large graphs lead to a considerable amount of inspections that this approach can not alleviate. This is why another mechanism is necessary in the creation of the *Fundamental Problem Graph*: Pre-Filtering.

Pre-Filtering, as the name implies, is a way to filter the tools in the *Repository Conectivity Graph* by including or excluding certain tools before the inspection of possible tool combinations. This decreases the amount of function nodes and graph complexity in the function graph, leading to a reduction in inspections and an increase in performance. An example for a filter is the definitive exclusion of certain tools from the FPG, which leads to the dismissal of all possible paths that contain these tools.

Although only this filter is applied in KADMOS at the moment, the following list is recommended for the implementation in KADMOS:

Include Tools/Modes. Exclude all tool combinations that do not contain the indicated tools and/or exection modes.

Exclude Tools/Modes. Exclude all tool combinations that contain the indicated tools and/or exection modes.

Minimum Tools. Exclude all tool combinations that have less than the indicated amount of tools.

Maximum Tools. Exclude all tool combinations that have more than the indicated amount of tools.

Objective Variable System Inputs .Exclude all tool combinations where any objective variables are not produced as an output of the present tools.

Objective Variable Collisions. Exclude all tool combinations where the same objective variables are written by more than one tool.

Some of the proposed implementations above require the creation of iterator functions which can take significant development time, but it is highly recommended to implement these in order to keep KADMOS operational even for large and highly interconnected repositories.

This section treated the creation of the *Fundamental Problem Graph* by performing the selection of an appropriate set of tools for the MDO workflow. Following the process depicted in Fig 5.10, the creation of the case-specific Basefile using the tools in the FPG is discussed in the next section.

5.5. Basefile Generation

In Sections 5.1.2 and 5.2, the need for the generation of the Basefile for the case-specific tool repository was introduced. By executing the workflow according to a schematic *Fundamental Problem Graph*, a use-case can be generated which can be used to derive the case-specific couplings among the tools in the MDO workflow. This ensures that the nodes and values that exist in the use-case

files represent the nodes that are actually read and written by the utilized tools when executeed, and guarantees consistency between the information exchange modeled by the MDO formulation and the true information exchange in a simulation workflow.

In the previous section, a method to extract a *Fundamental Problem Graph* was described in order to define a tool configuration that provides a solution to the MDO Problem. This configuration is now put into a sequence and executed in the given order, writing all outputs to the Basefile, as seen in Fig. 5.18. The choice for the sequence of the tool configuration has not been performed in this research as it was part of an active research in the same project (Schuurman, 2017), and a manual selection of the execution sequence is implemented instead.

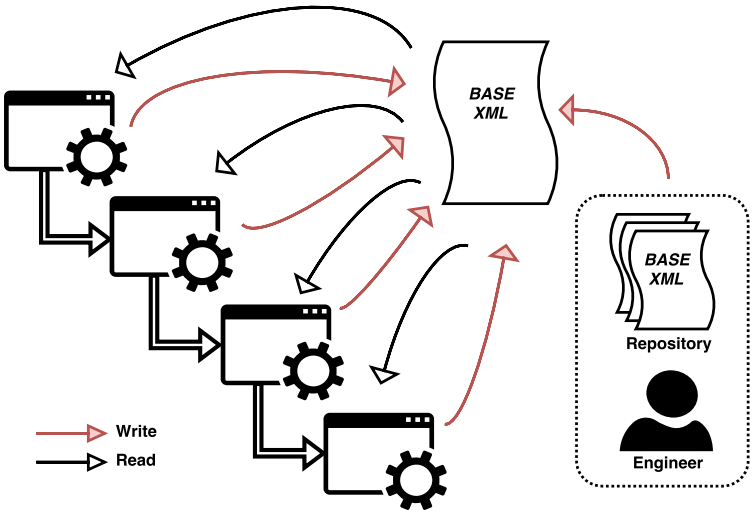


Figure 5.18: Generation of the Basefile by executing a tool configuration according to a selected sequence and storing the outputs.

By executing the tools in a given sequence, the required inputs are read from the Basefile and the produced outputs are written to it. This way, the Basefile is *enriched* with the case-specific nodes that each tool writes to the file, enabling the following tools in the sequence to use these nodes as inputs.

The resulting Basefile contains a complete, consistent and case-specific description of the aircraft, since each tool input is based upon the nodes and values produced by the other tools in the workflow. Once the complete sequence is executed, the case-specific input and output nodes for each tool in the tool configuration can be extracted from the Basefile, and can subsequently be used the determine the case-specific MDO formulation.

During the execution of a given tool in the workflow, three possibilites arise regarding its required input nodes:

1. The Basefile contains all nodes necessary to run the tool.
2. The Basefile does not contain all nodes necessary to execute the tool, but the required nodes are provided later in the execution. This can occur in the presence of feedback couplings.
3. The Basefile does not contain all nodes necessary to execute the tool, and no tool in workflow is capable of producing these nodes

Case 1 In the first case, there is no need to do anything except of running the tool. The exact values and tool behaviour are properly described by the tool output with all input nodes present in the Basefile.

Case 2 The second case requires more attention in the execution of the tool. Due to the predefined order in which the tool is executed, one or more required nodes are not available for execution *at the moment*. In this case, the tool needs to "borrow" nodes temporarily in order for it to be executed. The "borrowed" nodes can either be defined by the user **manually**, or be **automatically** loaded from a previously defined Basefile of a different use case. In the former case, the user is given a **template** to fill in, which is then merged into the Basefile for tool execution. If the user wishes to load the required missing node(s) from an existing repository, previously generated Basefiles (from other use cases) are scanned for nodes that match the node structure and *uIDs*. The utilization of previously generated Basefiles is a good example of **knowledge-reuse**, and can lead to significant time savings in the generation of the Basefile if similar use cases already exist in the repository. An example of the manual and automatic addition of required nodes is presented in Chapter 6.

It should be noted that by adding any missing required nodes to the Basefile, the tool can be executed using the nodes made available, and the execution workflow can be continued. However, since the tool does not utilize nodes that are *produced by the tools* in the workflow, **multiple** iterations are recommended for the execution of the workflow in order to guarantee Basefile consistency. By iterating the execution of the tool sequence, tools that had missing nodes in the first iteration can in the following iterations run on nodes that were written by the tools in the workflow, and so on.

Case 3 Similar to the second case, the required nodes for tool execution may not be present in the Basefile at the time of tool execution in the third case. The difference is, however, that none of the tools in the sequence provide the necessary inputs for that tool, which means that the inputs must be provided externally and are regarded as *System Inputs*. The missing nodes are provided exactly as described above in *Case 2*, and are written to the Basefile after a manual or automatic addition of the nodes. Basefile consistency does not play a role here due to the fact that there is no tool output that replaces the externally provided nodes.

By considering the three cases above, each tool can be executed in the given sequence to generate the Basefile containing a case-specific description of the aircraft. The Basefile now contains all input and output nodes related to the tools in the selected workflow and pertaining to a specific use case. In order to create a case-specific *Repository Connectivity Graph* according to Fig. 5.5, the input and output files must be extracted from the Basefile. Although this can be done manually, an automatic implementation for this task has been developed for KADMOS, and is presented in the following section.

5.6. Tool File Extraction and Node Mapping

The *manual extraction* of tool input and output files from a defined Basefile containing the use case can take a considerable amount of time. As was discussed in more detail in Chapter 3, each tool requires one input and one output file in order to fully describe the required and produced nodes by that tool. An aircraft description, however, may contain several thousand nodes which are *not* related to that tool, requiring the engineer to scan through the description and remove all the redundant nodes for each specific file. For large use cases containing many tools, this task can take a multiple hours to complete, especially when the engineer is not familiar with the tool. Any newly

generated use case, or changes in the use case, will require the engineer to repeat this process or perform revisions to each tool file. This issue has been identified in Section 5.1.1 as one of the main problems to solve with regard to the set up of the tool repository.

Fig. 5.19 shows an overview of the extraction process for a given Basefile. It can be seen that both the schematic inputs and outputs, as well as the node mapping files play a role in the input and output extraction, pertaining to the **two essential steps** involved in the *extraction process* for each tool and file. First, the schematic description is applied to dismiss all irrelevant nodes for the respective tool file, and second, Node Mapping is used to check references between nodes to remove unreferenced nodes.

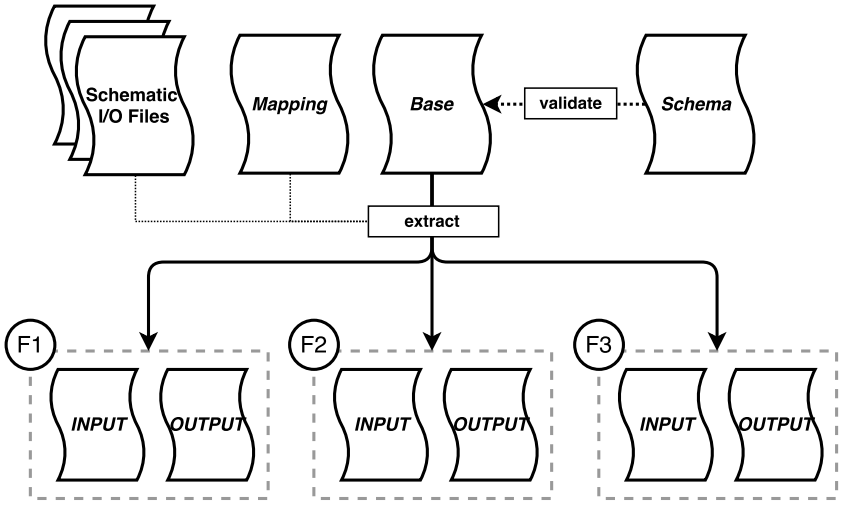


Figure 5.19: Automatic extraction of the Basefile is done by using the schematic description of the tool inputs and outputs, and a node mapping file that enables to identify links between nodes in the Basefile.

Schematic Matching In the first step, the schematic file nodes are compared to the nodes in the Basefile by matching their node structure. All matching nodes remain in what is now the case-specifc file, while the others are removed, leaving only the nodes that are defined by its schematic counterpart. Consider the example case for the extraction of the input file for the tool "Example-Tool", as shown in Fig. 5.20.

The Basefile in this example contains all nodes that are described by the schematic input file, but in addition to that has the node */cpacs/aircraft/fuselage* that is not defined in the schematic file. Since there is no match in the structure for this node between the two files, it is dismissed from the case-specific file resulting in the XML tree seen on the left in Fig 5.21.

It can be observed here that the nodes in the case-specific file now correspond to the schematic file in terms of node structure. However, there are still redundant nodes present in the case-specific file that do not correspond to the input of the tool according to its tool settings. As can be seen in the node */capcs/toolspecific/exampleTool/wingUID*, the example tool is calibrated to analyze the wing with the *uID "main_wing"*. The file, however, still contains a wing description for the wing with the *uID "horizontal_tailplane"*, which is not used in the tool execution and therefore should be removed from the input file.

Node References In the second step of the file extraction process, node references within the case-specific file are checked in order to remove all redundant nodes that are not considered by the tool.

Basefile

```
<cpacs>
   <aircraft>
      <fuselage>
         <dimensions>
            <length> 44.51 </length>
            <width> 3.95 </width>
            <height> 4.14 </height>
         </dimensions>
      </fuselage>
      <wings>
         <wing uID="main_wing">
            <spar uID="main_wing_spar">
               <materialUID isLink="True">
                  aluminum7075
               </materialUID>
               <length> 17 </length>
               <width> 0.4 </width>
               <height> 0.3 </height>
            </spar>
         </wing>
         <wing uID="horizontal_tailplane">
            <spar uID="ht_spar">
               <materialUID isLink="True">
                  aluminum2024
               </materialUID>
               <length> 17 </length>
               <width> 0.4</width>
               <height> 0.3 </height>
            </spar>
         </wing>
      </wings>
      <materials>
         <material uID="aluminum2024">
            <modulus> 7.6e10</modulus>
         </material>
         <material uID="aluminum7075">
            <modulus> 7.2e10</modulus>
         </material>
      </materials>
   </aircraft>
   <toolspecific>
      <exampleTool>
         <wingUID isLink="True">
            main_wing
         </wingUID>
      </exampleTool>
   </toolspecific>
</cpacs>
```

Schematic Input File

```
<cpacs>
   <aircraft>
      <wing>
         <spar>
            <materialUID/>
            <length/>
            <width/>
            <height/>
         </spar>
      </wing>
      <materials>
         <material>
            <modulus/>
         </material>
      </materials>
   </aircraft>
   <toolspecific>
      <exampleTool>
         <wingUID/>
      </exampleTool>
   </toolspecific>
</cpacs>
```

Figure 5.20: Comparison between a generated Basefile and the schematic input for tool "ExampleTool".

Node references represent the dependencies between so-called link-items and uID-items, and serve as a way to indicate a relationship between two nodes in the XML tree. A **link-item** is a node that carries an *"isLink"*-attribute and refers to one specific **uID-item**, a node that carries a *uID*-attribute. By "applying" a link-item to the tree, all potentially related nodes that do not match the *uID* value of the link-item are removed from the tree. In order to check which nodes are referred to by the link-item, the presence of the *Node Mapping* file is required, which contains all mappings between each *link-item* and its corresponding *uID-item*. The required mappings for the example case are shown in Fig. 5.22.

Node Mapping is applied on two levels: the XML schema and the tool level. While the mapping must strictly correspond to the prescribed description of the XML-schema, the engineer must also

Input file after first step

```
<cpacs>
   <aircraft>
      <wings>
         <wing uID="main_wing">
            <spar uID="main_wing_spar">
               <materialUID isLink="True">
                  aluminum7075
               </materialUID>
               <length> 17 </length>
               <width> 0.4 </width>
               <height> 0.3 </height>
            </spar>
         </wing>
         <wing uID="horizontal_tailplane">
            <spar uID="ht_spar">
               <materialUID isLink="True">
                  aluminum2024
               </materialUID>
               <length> 17 </length>
               <width> 0.4</width>
               <height> 0.3 </height>
            </spar>
         </wing>
      </wings>
      <materials>
         <material uID="aluminum2024">
            <modulus> 7.6e10</modulus>
         </material>
         <material uID="aluminum7075">
            <modulus> 7.2e10</modulus>
         </material>
      </materials>
   </aircraft>
   <toolspecific>
      <exampleTool>
         <wingUID isLink="True">
            main_wing
         </wingUID>
      </exampleTool>
   </toolspecific>
</cpacs>
```

Input file after second step

```
<cpacs>
   <aircraft>
      <wings>
         <wing uID="main_wing">
            <spar uID="main_wing_spar">
               <materialUID isLink="True">
                  aluminum7075
               </materialUID>
               <length> 17 </length>
               <width> 0.4 </width>
               <height> 0.3 </height>
            </spar>
         </wing>
      </wings>
      <materials>
         <material uID="aluminum7075">
            <modulus> 7.2e10</modulus>
         </material>
      </materials>
   </aircraft>
   <toolspecific>
      <exampleTool>
         <wingUID isLink="True">
            main_wing
         </wingUID>
      </exampleTool>
   </toolspecific>
</cpacs>
```

Figure 5.21: Example input file after the application of the first step (left) and second step (right) in the file extraction process.

```
{
  "/cpacs/toolspecific/exampleTool/wingUID" :"/cpacs/aircraft/wings/wing",
  "/cpacs/aircraft/wings/wing/spar/materialUID" :"/cpacs/aircraft/materials/material"
}
```

Figure 5.22: Required contents of the Node Mapping file for the example Basefile in Fig. 5.20.

ensure that the link-items in the *toolspecific* nodes are included in the mapping file. This should always occur upon tool integration into the repository in order for the mapping to work for the added tool.

Starting with the tool settings, which are stored in the node */cpacs/toolspecific* for the corresponding tool, the reference *uIDs* of each link-item are **recursively** applied onto the XML tree. As seen in Fig. 5.22, the node */cpacs/ toolspecific/ exampleTool/ wingUID* refers to the node */cpacs/ aircraft/ wings/ wing*. This means that all ~*/wing/wing*-nodes are checked for the *uID "main_wing"*, and only

the one that this *uID* applies to remains in the case-specific file. This process is applied recursively, checking all descendants of the "applied" uID-item for other link-elements in order to ensure that "references-of-references" remain in the tree. This is necessary as seen in the example of the node */cpacs/ aircaft/ materials/ material*, where only the *material*-node with the *uID "aluminium2024"* is referenced by the relevant *wing*-node. Without any recursion, both *material*-nodes would remain in the file with only one being strictly required by the tool, possibly leading to wrong couplings within an MDO formulation.

It should be noted that node references are checked extensively in the process of generating the Basefile as described in Section 5.5, but were omitted for the sake of clarity. In order to ensure that the correct tool inputs exist and are referred to in the tool settings, node references are just as necessary as described in this example.

An important fact to mention in the use of a Node Mapping is that it completely depends on the XML schema that is applied in the use case, and the structure of the tool setting nodes. Changes in the node structure that are dictated by the XML schema need to be adopted in the Node Mapping in order to ensure that the correct nodes are searched in the mapping process. A faulty mapping definition will always result in a negative outcome for references since none of the present nodes can be matched. For the same reason, changes in the node structure of the tool settings must be accounted for.

The automatic extraction of the case-specific input and output files concludes the application of a *dynamic* tool repository in KADMOS. The next section provides a discussion on the differences, improvements, and the relationship between the schematic and case-specific component of the tool repository, and wraps up the discussion on the developed methodology in this report.

5.7. The big picture: Schematic vs. Case-Specific Repository

Chapters 3 and 4 discussed the creation of MDO formulations based on a provided use case, or aircraft description. The process is straightforward: extract the input and output files for each tool, and generate graph networks based on their input and output information. These graph structures can now be used to quickly and efficiently analyze the couplings between the tools in the repository, enabling engineers to make decisions on MDO workflows that are to be created for a specific use case.

Two main issues were found using this approach, as stated in Section 5.1. The first issue concerns the time and effort it takes the engineer to synchronize changes in the repository when modifications to any of the files occur. The second issue relates to the externally provided aircraft description, which may not represent the *true* inputs and outputs of the tools that are being used in the repository for the supplied use case.

To solve these problems, a schematic layer has been added to the MDO development process that uses an abstracted description of tool inputs and outputs to create abstracted MDO formulations, which in turn can be used to generate aircraft descriptions by executing the tools in the formulation. By using the tools in the repository to generate use cases, the aircraft description will be represented by the exact tool inputs and outputs, and any changes to the description simply requires a re-execution of the tools under the appropriate tool settings.

An overview of the workflow of the complete MDO development process is presented in Fig. 5.23. Here, it can be seen that both the schematic and case-specific workflow are independent processes, meaning that it is not strictly required to define a schematic tool description in order to start the MDO development process. However, having a defined set of schematic tool input and output files

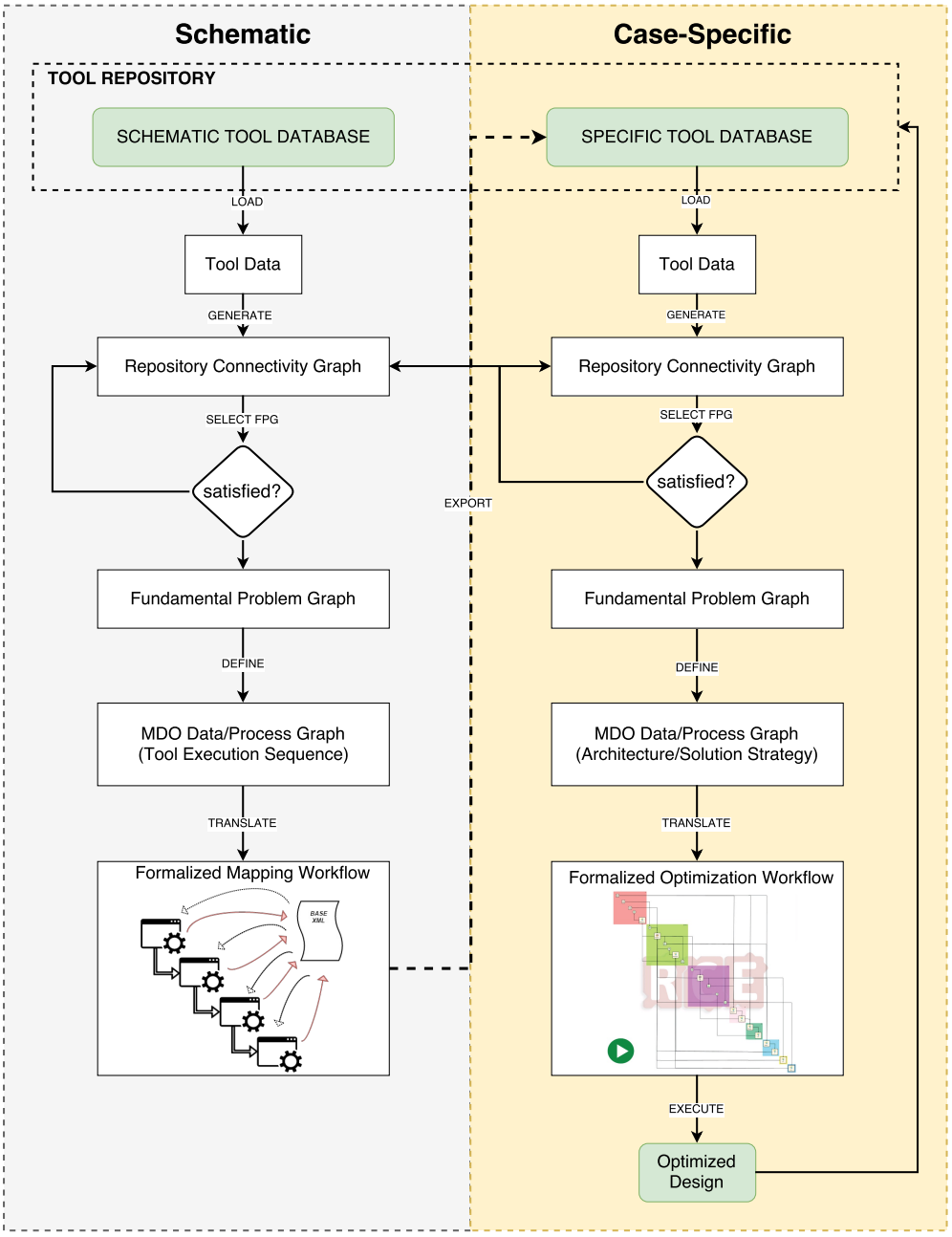


Figure 5.23: Complete workflow containing the schemtic as well as the case-specific tool repositories.

may help the engineer in the extraction of case specific tool inputs and outputs. If a use case is supplied externally and the tools for the MDO formulation are known, the case-specific files can be extracted automatically using the schematic tool files along with the Node Mapping, similar to the process described in Section 5.6.

In order to generate the case-specific tool database for a new use case, the schematic input and output files for each tool must be defined. The advantage here is that these files have to be defined only once since they are case-independent and only shows the abstracted input and output tool data.

Using the schematic files, an abstracted *Repository Connectivity Graph* is generated that can be used to derive the schematic *Fundamental Problem Graph*. Since this FPG is based on the schematic

definition of tools, it does not indicate the amount of nodes that are exchanged between the tools and is completely based on the type of information that is exchanged, or the node structure of the tools. The *amount* of data that is ultimately transferred between the tools can not be estimated until the use case is completely determined, which means that the engineer's knolwedge and experience plays an important role in an efficient *case-specific* generation of MDO workflows without using many iterations going back to the schematic MDO formulation.

Once the use case is established, the case-specific RCG and FPG can be created based on the generated aircraft description. Since the *true* tool couplings become known here, the engineer can now evaluate whether the tool configuration and tool settings are appropriate for the intended MDO workflow. If this is not the case, the engineer can either go back to the schematic phase and re-generate a use case based on a more preferred tool configuration and/or different tool settings. When minor changes to the tool configuration have to be performed, the engineer can simply apply these modifications to the case-specific RCG manually without having to re-generate the use case, and continue with the workflow.

Every generated use case is stored in the tool repository and can potentially help the engineer in setting up future use cases. This knowledge reuse is especially helpful during the generation of the Basefile, where required nodes may be missing during an execution, forcing the engineer to supply these either manually, or select them automatically from existing cases.

In order to set up a use case based on the schematic workflow, the following must be defined in the tool repository:

- An XML schema must be present that standardizes the product description.
- Tool inputs and outputs must be defined in an abstract manner and must follow the XML schema. This also means that the tools must be calibrated so that they read from and write to the appropriate nodes in a product description.
- Tool information, or metadata, must be filled out and consistent with the input and output files. All execution modes must be properly defined and indicated in the tool files.
- A *Node Mapping* must exist that identifies the links between nodes in the product description.

Once an optimized design has been acquired, as shown at the end of the case-specific workflow in Fig. 5.23, the obtained results can be analyzed and evaluated. If any modifications to the MDO workflow need to be implemented, the engineer can go back to the the tool repository and restart the MDO development process from both the schematic or the case-specific tool database. Changes in the initial design or the tool configuration will likely require modifications to the Basefile, which means that its re-generation using the schematic workflow are recommended. Changes that apply to the solution strategy, however, can be implemented by following the case-specific workflow, since no changes to the tool input and output files or the Basefile are necessary. By providing engineers with the possibility to quickly set up use cases and apply changes to generated MDO formulations without having intrinsic knowledge about the tools involved, the MDO development process becomes more accessible to non-experts and more agile in its application.

6

Case Study: MDO Wing Optimization

In this chapter, the case study for the implementation of the KADMOS tool repository is presented in order to give the reader a better understanding and visualization of the developed methods. Appendix A provides a code sample for the presented case study.

Starting with Section 6.1, the use case is introduced and the problem description is given, along with a definition of the utilized software tools. The manual use case configuration is described in Section 6.2 to illustrate the tasks that are required in the definition and modifcation of a use case. In Section 6.3, the schematic approach is described, demonstrating its capabilities in the automation of the configuration tasks. Its advantages in the set up of new and unrelated use cases are highlighted in Section 6.4, before a short reflection is given on the developed approach in Section 6.5.

6.1. MDO Problem Description

This case study discusses the set-up and implementation of an MDO wing optimization problem using a KADMOS repository. The initial design point for the wing optimization study can be seen in Fig. 6.1, where the wing description is extracted from the conventional aircraft used in the first *AGILE Design Campaign*.

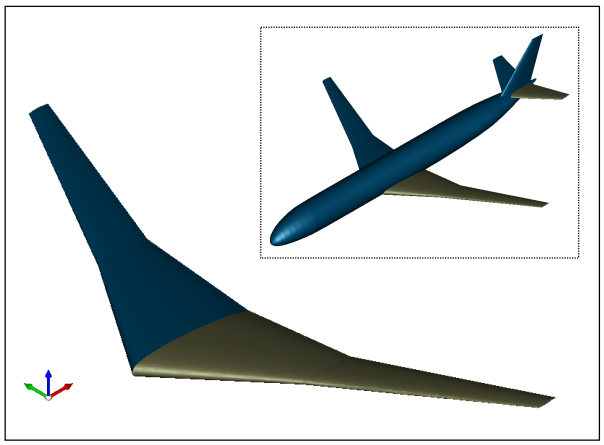


Figure 6.1: Initial design point for the wing analysis case study.

The goal of this optimization is to find an optimal wing design with respect to its weight, while satis-

fying constraints regarding (minimum) fuel volume and (maximum) wing loading. Although being necessary in an executable MDO workflow, the constraint functions are not included in the MDO formulation due an incomplete implementation of *sink* nodes in the graph network. A recommendation on this issue is given in Chapter 7.

In order create an MDO workflow for this optimization problem, KADMOS is used to establish the tool couplings and to determine the information exchange between the separate tools. The tools and their integration in the MDO workflow are pre-defined by an expert, meaning that there is no need to inspect the "best" sequence or configuration in this case. This, however, is one of the many strengths of KADMOS, as it allows engineers to inspect and compare different tool configurations, enabling trade-offs between them.

The tools used in this case study are shown in Table 6.1 and represent a realistic set of tool for a conceptual design and optimization of a wing. Some of the tools have multiple execution modes that are taken into account in the *Repository Connectivity Graph*, as will be shown in the next section.

Table 6.1: Software Tool used in the case study.

Tool	Description
HANGAR	Loads a use case file which has been initiated by a design initialization software tool
SCAM	Simplifies wing morphing by using only few selected parameters
GACA	Provides geometrical values that are not explicitly stored in CPACS files (wing span, wing reference area, wing fuel tank volume)
Q3D	Performs aerodynamic analysis of the wing Can be executed in two modes: FLC (inviscid) and VDE (viscous)
EMWET	Calculates the structural weight of a wing
SMFA	Estimates the required mission fuel based on the Breguet equation
MTOW	Calculates maximum take-off weight and zero-fuel mass
OBJ	Represents objective function

6.2. Manual Use Case Implementation

As has been described in detail in Chapters 3 and 4, the generation of graph networks such as the *Repository Connectivity Graph* and the *Fundamental Problem Graph* require the presence of several types of files in the KADMOS tool repository. These files include the input and output XML-files for each tool, the Basefile which contains the complete description of the aircraft, an Info-File for each tool and the applied XML-schema, in this scenario CPACS.

The aircraft description is provided externally and is contained in the Basefile, which carries all nodes that are used by the tools in the repository. The skeleton Basefile for this use case can be seen in Fig. 6.2.

In order to be able to generate graph networks that represent the couplings among the utilized tools, the input and output files must be extracted from the Basefile. This must be done using expert knowledge about the tool since each tool is treated as a *black box* and relevant tool nodes are not evident to non-experts. When considering the fact that KADMOS is a platform that supports the cooperation among heterogeneous teams that are not only distributed by discipline, but also geographically, this task becomes more difficult to perform in a timely manner. Bottlenecks include scheduling issues and unavailability of experts, iterations in the case of mistakes, and the familiar-

Basefile *EMWET*-input

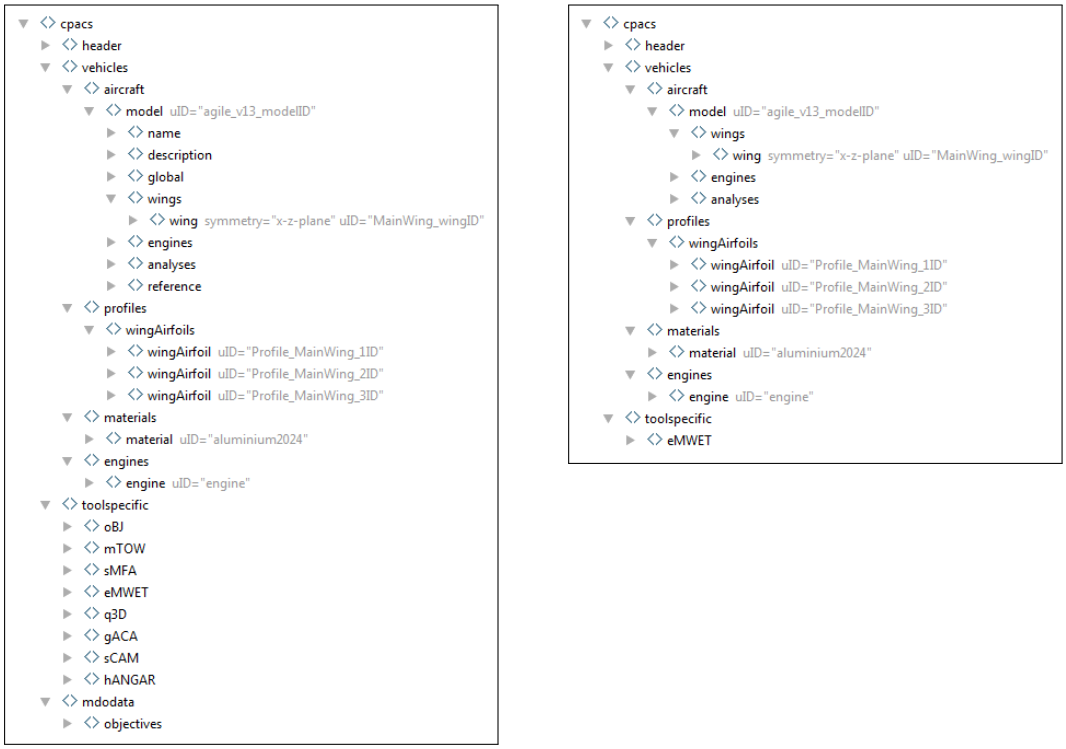


Figure 6.2: Basefile of the provided aircraft description (left) and the extracted *EMWET* input file (right).

ization of the disciplinary expert with the methodology. When modifications are applied to the use case, it may again become necessary to consult the disciplinary expert in order to ensure a correct and consistent extraction of tool files. These reasons can lead to long set-up times of large projects, which is why an automated extraction of tool files is implemented in KADMOS, as will be shown later in the discussion.

Once the tool files are extracted, the repository is complete and KADMOS can generate the *Repository Connectivity Graph*. The tool repository for this use case can be seen in Fig. 6.3.

By launching KADMOS, the graph generation workflow is initiated as shown in the activity diagram of Fig. 4.1. KADMOS converts the information stored in the XML-files into graph networks where the couplings between each tool can be visualized. A *circular view* of the *Repository Connectivity Graph* for this use case is presented in Fig 6.4.

The circular view shows the couplings between all tools in the repository in each execution mode that has been defined in each tool's Info-File. Since the RCG contains the complete information of the tool repository, it can be used to analyze the present tools and the couplings among them. Information such as the amount of input nodes, output nodes, neighboring tools and couplings can be quickly determined for each tool, or a group of tools, by using the appropriate functions. This way, an appropriate *Fundamental Problem Graph* can be determined, if not already known, in order to complete the MDO formulation.

In this use case, the tool configuration is predetermined, which means the relevant tools and execution modes are simply selected to generate the *Fundamental Problem Graph*. The FPG represents a certain configuration of tools, or a subset of tools, and is always derived from the complete *Repository Connectivity Graph*. Fig. 6.5 shows the FPG for this use case, with the tools on the diagonal and

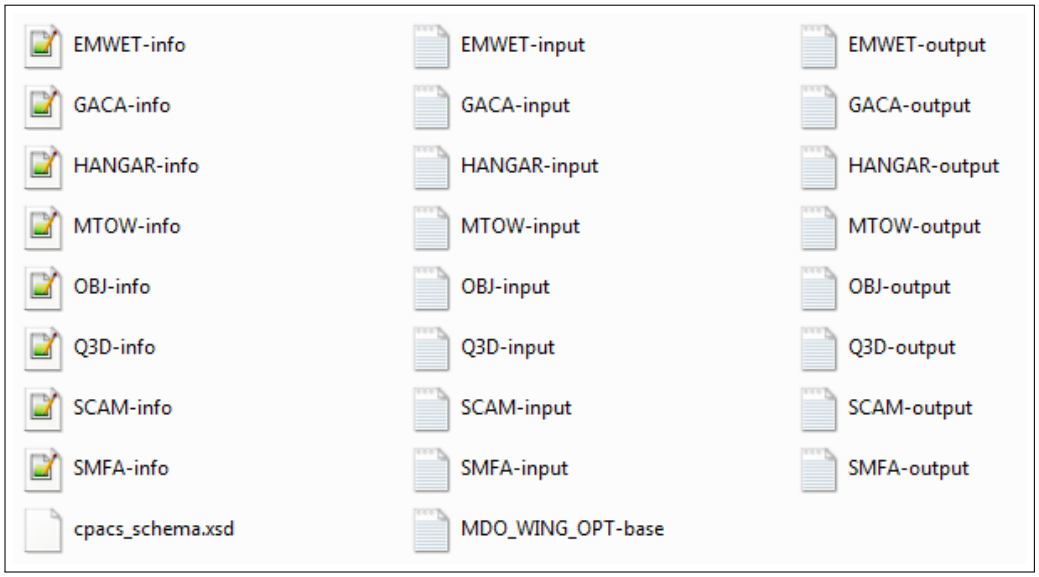


Figure 6.3: Complete tool repository for the use case.

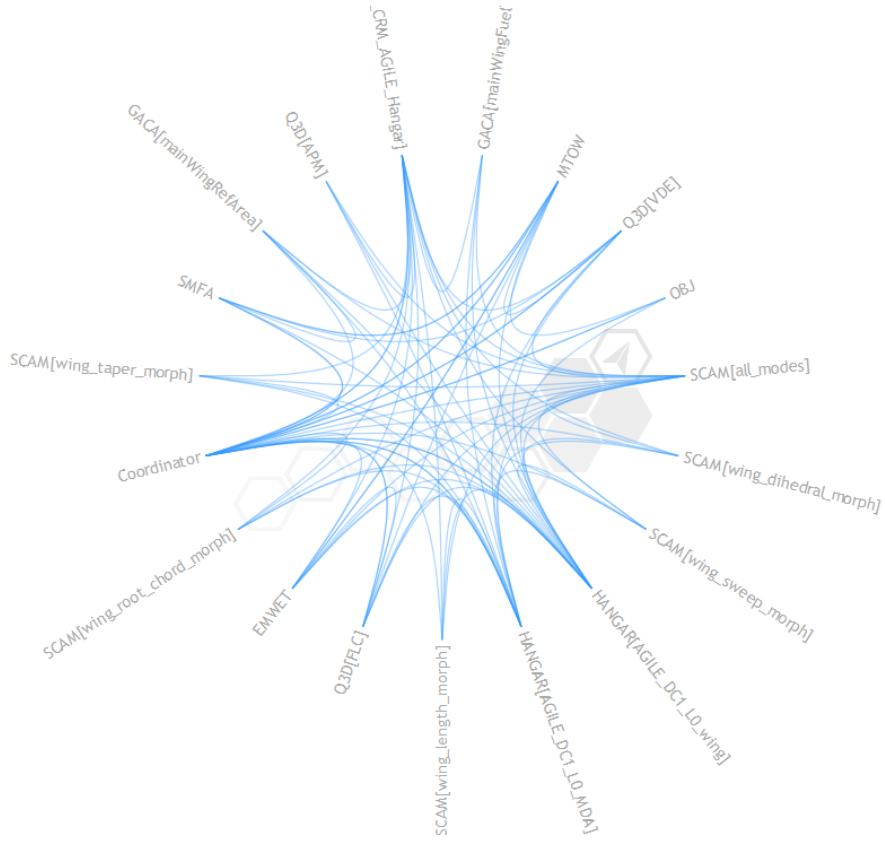


Figure 6.4: Circular view of the *Repository Connectivity Graph* using the visualization package provided by Van Gent et al. (2017).

their couplings in the grey boxes.

Here, it can be seen that modes like *Q3D[APM]* or *SCAM[wing_dihedral_morph]* are not included in the FPG, since they are not deemed necessary in the optimization of the wing. It should be noted

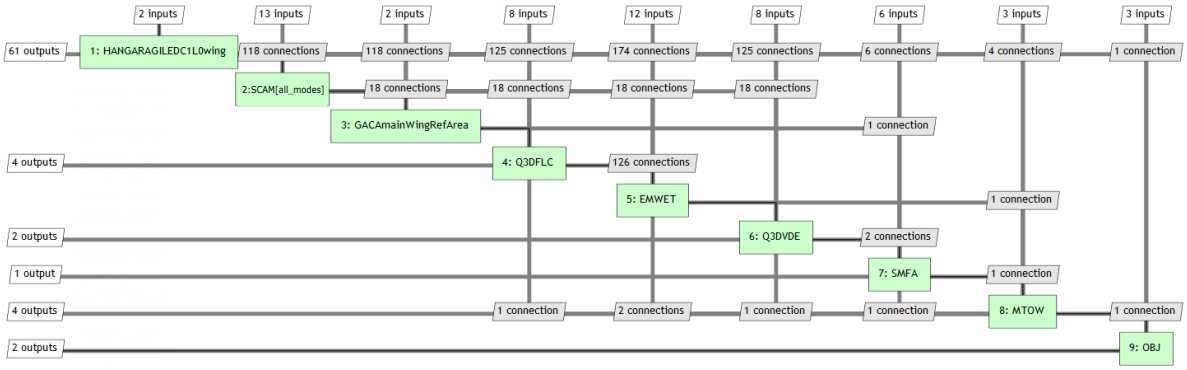


Figure 6.5: XDSM of the *Fundamental Problem Graph* for the wing optimization use case.

that the *SCAM*-mode *"all_modes"* summarizes all execution modes into one mode, and is represented as a separate function node which is used in the FPG.

The FPG shown in Fig. 6.5 forms the basis for the application of an MDO architecture and solution strategy to the selected set of tools. Although this is out of scope for the research described in this report, the Fig. 6.6 is shown to give an impression of a possible MDO formulation in KADMOS before its translation into an executable simulation workflow.

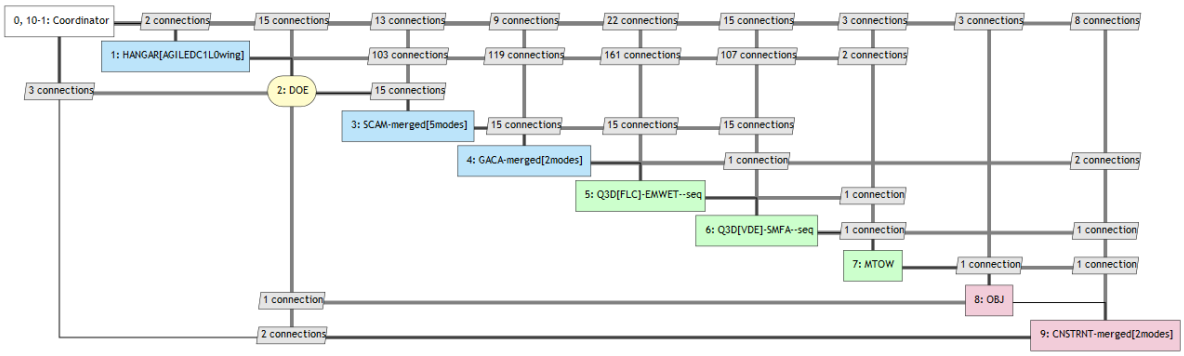
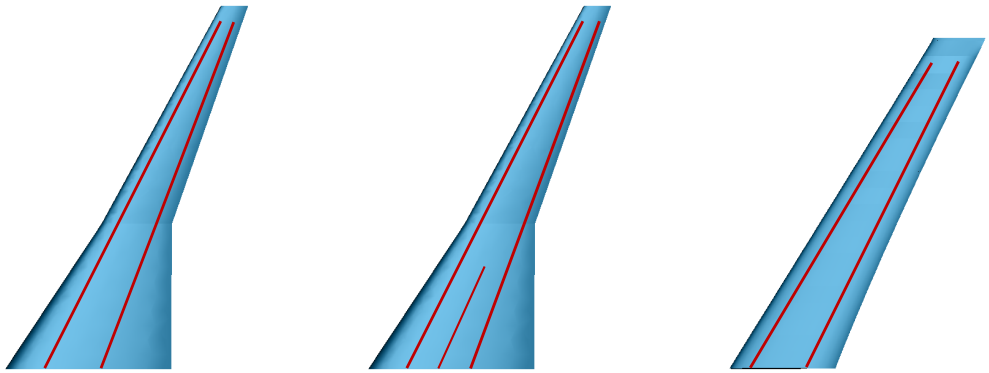


Figure 6.6: Possible implementation of a *Design of Experiments* for a similar use case as the one shown in the *Fundamental Problem Graph* of Fig. 6.5.

Returning to the *Fundamental Problem Graph* in Fig. 6.5, an example is provided that shows the importance of the correct definition of input and output nodes in the tool files of the repository. Consider the application of minor changes in the initial design the wing. This could for example be an addition of a wing spar or the removal of the kink in the wing planform, as shown in Fig. 6.7.

The wing geometry in this use case is regulated by *HANGAR* and as shown in Fig. 6.8, it affects the inputs of *SCAM*, *GACA*, *Q3D* (both modes) and *EMWET*. If even slight changes to the output of *HANGAR* are made, such as the addition of a spar in the initial design, all nodes that are affected by the changes have to be adapted in the corresponding files. Performing the modifications in one file but not the others leads to inconsistencies and **loss of couplings**, as seen in Fig. 6.8.

In this example, the wing nodes concerning the component segments (which contain the spar description) of the wing are modified, leading to a loss in connections between *HANGAR* and each tool that requires these nodes as input. In addition, since the changes were only performed in the *HANGAR* output file, the nodes do not correspond to the description as defined by the provided Basefile, leading to an **inconsistent** and **invalid** use case.



Original wing Added spar Removed wing kink

Figure 6.7: Example modifications in the initial wing design (top view).

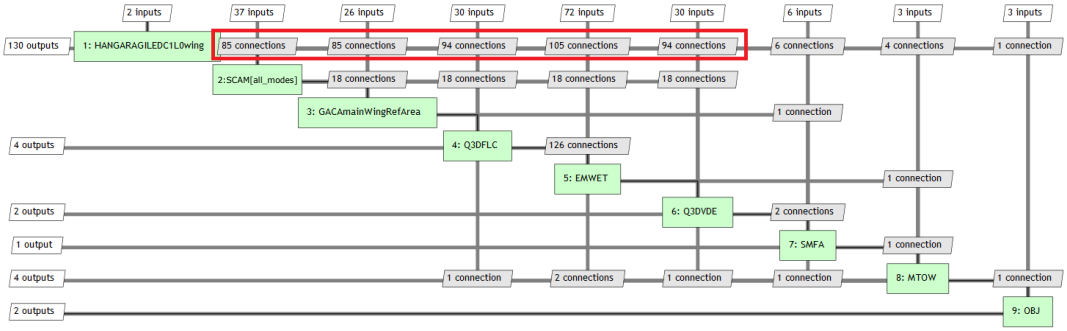


Figure 6.8: Modified wing exhibits loss in connections due to change in node structure, values and attributes.

It can be seen that carrying out even minor changes to the use case after it has been set up requires a lot of work since the engineer must follow up on each modified node in each tool file that is affected, leading to significant overhead in large use cases where many nodes are shared. As described above, the distributed nature of these systems adds bottlenecks that increase the time required to perform these changes confidently with the use of tool experts.

This is identified as one of the two major issues, as described in Chapter 5, that concern the set up of the tool repository. The second issue relates to the "correctness" of the provided aircraft description with respect to how precisely it represents the *true* tool inputs and outputs. Since the aircraft description in the Basefile is provided externally, it may or may not represent the actual input and output information of the tools in the repository. When extracting tool files from the Basefile, one can *assume* that the nodes in the description represent the nodes that are used by the tool and are generated by it, but it is only an assumption. Without executing the tool to read inputs and write outputs that are then used for the description, one can not determine the *true* data exchange. The reason why knowing the exact information exchange is important is because the entire premise of KADMOS is to generate suitable MDO workflows for a given MDO problem. If the exact data exchange between the separate tools is not known precisely, KADMOS can create suitable MDO workflows only based on faulty information, resulting in workflows that deviate from the intended formulation.

6.3. Schematic Tool Repository

Introducing a *schematic* description of the tool inputs and outputs, KADMOS attempts to solve the stated issues by decoupling the tool input and output description from the use case. As discussed in detail in Chapter 5, schematic tool files are abstracted versions of the case-specific tool files that have been presented in the previous section. An example of a schematic file is shown in Fig. 6.9 for the tool output for *EMWET*.

```
<?xml version='1.0' encoding='UTF-8'?>
<cpacs>
   <vehicles>
      <aircraft>
         <model>
            <analyses>
               <massBreakdown>
                  <mOEM>
                     <mEM>
                        <mStructure>
                           <mWingsStructure>
                              <mWingStructure>
                                 <massDescription>
                                    <mass/>
                                 </massDescription>
                              </mWingStructure>
                              <massDescription>
                                 <mass/>
                              </massDescription>
                           </mWingsStructure>
                        </mStructure>
                     </mEM>
                  </mOEM>
               </massBreakdown>
            </analyses>
         </model>
      </aircraft>
   </vehicles>
</cpacs>
```

Figure 6.9: Schematic output tree for *EMWET*.

When it comes to the case-specific and schematic repository structure, not many differences exist between the two. Both the XML-schema and the tool Info-Files remain exactly as they are, since they both are required for their respective purpose in each repository. The major difference in the schematic repository is the absence of the Basefile since an abstracted tool description does not require a use case. Instead, the Node Mapping file must be present in order to allow KADMOS to check references and dependencies between nodes. This is especially necessary in the generation of the Basefile as well as in the automatic extraction of tool input and output files from the generated Basefile, as laid out in detail in Sections 5.5 and 5.6. A summary of both repository structures can be seen in Fig. 5.5. In the following discussion, it is shown how a *schematic* tool repository can be used to generate a case-specific repository in an *agile* manner, in order to solve the two issues of the manual repository set up discussed in the previous section.

Although the schematic tool files do not describe a specific use case, they can be used in their **abstracted** form to generate *Repository Connectivity Graphs* to inspect the possible information exchange between software applications without the knowledge of an actual product description.

Similar to the case-specific RCG, the schematic RCG is used to derive the schematic *Fundamental Problem Graph*. In the use case given in this chapter, the tool configuration and sequence is prescribed. However, this may not always be the case.

Consider the same case where the tool configuration is not predetermined. In order to derive a *Fundamental Problem Graph*, it would be desirable to analyze different possible tool configurations and perform a trade-off on them. This is facilitated by KADMOS in order to let the engineer select an appropriate FPG for the MDO formulation. By defining a graph sink from all possible tools in the *Repository Connectivity Graph*, as seen in Fig. 6.10, KADMOS finds all tool paths towards that sink and determines all possible tool configurations that "calculate" the sink (at least one tool writes to the inputs of the sink).

```
The following function nodes were found in graph:

  #    Inputs    Outputs    Couplings  Function
---  --------  ---------  -----------  ------------------------------
  0        58          1          179  SCAM[wing_sweep_morph]
  1        57          1          179  SCAM[wing_dihedral_morph]
  2        56          1          169  GACA[mainWingRefArea]
  3        57          1          179  SCAM[wing_length_morph]
  4        67          2          178  GACA[mainWingFuelTankVol]
  5         6          2            5  CNSTRNT[wingLoading]
  6       109          2          239  EMWET
  7         4          2            3  OBJ
  8        15          2           14  SMFA
  9         7          2            4  CNSTRNT[fuelTankVolume]
 10        58          3          203  SCAM[wing_taper_morph]
 11        57          3          203  SCAM[wing_root_chord_morph]
 12        68          4          175  Q3D[VDE]
 13        65          6          169  Q3D[APM]
 14         7          6           16  MTOW
 15        67          6          239  SCAM[all_modes]
 16        68         15          184  Q3D[FLC]
 17         2         72          523  HANGAR[NASA_CRM_AGILE_Hangar]
 18         2        125          710  HANGAR[AGILE_DC1_L0_wing]
 19         2        323          693  HANGAR[AGILE_DC1_L0_MDA]

Please select Objective(s):
```

Figure 6.10: Selection prompt for the "Objective Function" from all tools in the schematic *Repository Connectivity Graph*.

Choosing the tool *OBJ* (representing the Objective Function) as the sink in this use case, each tool configuration can be plotted, for instance, with respect to the amount of tools and system inputs as shown in Fig. 6.11. The amount of system inputs is one possible criterion for the selection of an appropriate tool configuration, but many other criteria (not implemented) such as the minimum/maximum tool fidelity, minimum/maximum precision, etc, or a combination of those, could be used as criteria. The trade-off completely depends on the preferences of the engineer and the availability of implemented criteria, both of which are out of scope of this research.

Scatter plots such as the one in Fig. 6.11 can be used to find the most suitable tool configuration. In this example, it can be seen that regions develop according the amount of System Inputs that are present for each configuration. From this it can be deduced that the tool configurations in the top regions are less desired since the system (or user) must provide many variables to the MDO workflow. This is due to the fact that tools such as *HANGAR* that provide the aircraft geometry are not included in those configurations. They may not be desired for an MDO formulation, providing the engineer with an impression on which tool configurations to filter out.

In order to generate the tool configuration for the schematic *Fundamental Problem Graph*, KADMOS prompts the user to select the desired configuration, as shown in Fig. 6.12. In this example, the possible tool configurations are ranked according to the most tools in the configuration. Future implementations in KADMOS, however, should allow for more sophisticated ranking analyses based on a wider selection of criteria. Ranking the functions according to prescribed criteria can provide significant support to the engineer in the decision-making for the choice of the most suitable tool configuration.

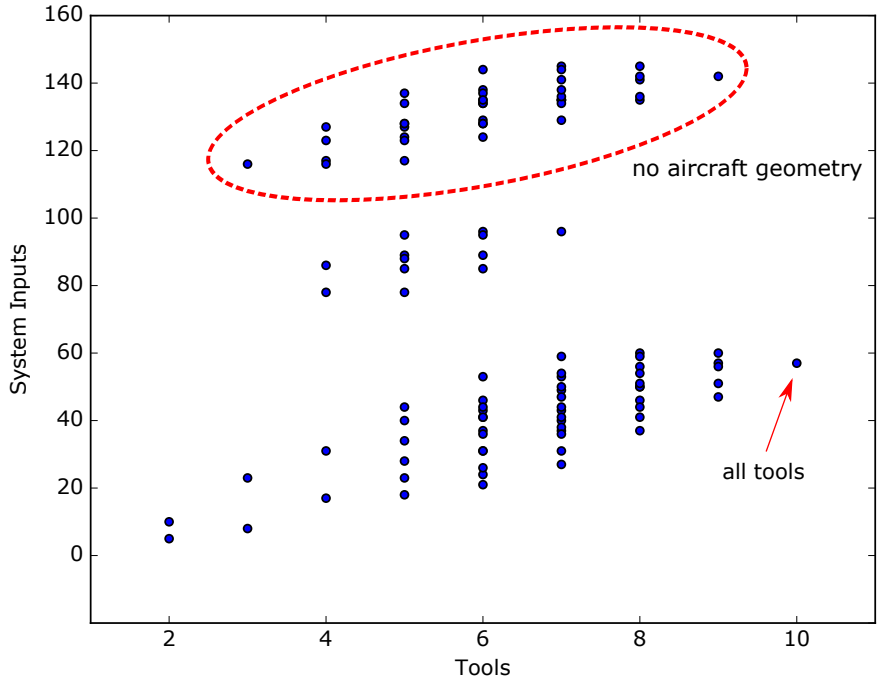


Figure 6.11: All possible tool configurations plotted with respect to the amount of tools and the corresponding required System Inputs.

```
NOTE: Function configurations are considered for Objective(s): [OBJ]

The following tool configurations were found in the graph:

  #    functions    nodes    edges    couplings    system_inputs  Configuration
---  -----------  -------  -------  -----------  ---------------  ------------------------------------------------------------
  0           10      219      619          455               57  ['HANGAR[AGILE_DC1_L0_wing]', 'Q3D[VDE]', u'OBJ', ...
  1            9      218      559          377               57  ['HANGAR[AGILE_DC1_L0_wing]', 'Q3D[VDE]', u'OBJ', ...
  2            9      208      546          362               47  ['HANGAR[AGILE_DC1_L0_wing]', 'Q3D[VDE]', u'OBJ', ...
  3            9      189      492           76              142  ['Q3D[VDE]', u'OBJ', 'Q3D[FLC]', 'SCAM[all_modes]', ...
  4            9      216      562          391               56  ['HANGAR[AGILE_DC1_L0_wing]', 'Q3D[VDE]', u'OBJ', ...
  5            9      208      547          383               51  ['HANGAR[AGILE_DC1_L0_wing]', u'OBJ', 'Q3D[FLC]', ...
  6            9      206      536          374               60  ['HANGAR[AGILE_DC1_L0_wing]', 'Q3D[VDE]', u'OBJ', ...
  7            8      176      409           55              145  ['Q3D[VDE]', u'OBJ', 'SCAM[all_modes]', u'SMFA', ...
  8            8      186      435           66              141  ['Q3D[VDE]', u'OBJ', 'Q3D[FLC]', 'SCAM[all_modes]', ...
  9            8      205      489          304               46  ['HANGAR[AGILE_DC1_L0_wing]', 'Q3D[VDE]', u'OBJ', ...
 10            8      204      486          293               44  ['HANGAR[AGILE_DC1_L0_wing]', 'Q3D[VDE]', u'OBJ', ...
 11            8      205      476          299               60  ['HANGAR[AGILE_DC1_L0_wing]', 'Q3D[VDE]', u'OBJ', ...
 12            8      178      419           37              135  ['Q3D[VDE]', u'OBJ', 'Q3D[FLC]', u'SMFA', u'MTOW', ...
 13            8      215      502          316               56  ['HANGAR[AGILE_DC1_L0_wing]', 'Q3D[VDE]', u'OBJ', ...
 14            8      181      425          276               37  ['HANGAR[AGILE_DC1_L0_wing]', 'Q3D[VDE]', u'OBJ', ...
 15            8      203      479          310               59  ['HANGAR[AGILE_DC1_L0_wing]', 'Q3D[VDE]', u'OBJ', ...
 16            8      205      490          319               50  ['HANGAR[AGILE_DC1_L0_wing]', u'OBJ', 'Q3D[FLC]', ...
 17            8      188      432           52              142  ['Q3D[VDE]', u'OBJ', 'Q3D[FLC]', 'SCAM[all_modes]', ...
 18            8      195      463          287               50  ['HANGAR[AGILE_DC1_L0_wing]', 'Q3D[VDE]', u'OBJ', ...
 19            8      178      420           64              136  [u'OBJ', 'Q3D[FLC]', 'SCAM[all_modes]', u'SMFA', ...

Please select a tool combination from the list above:
```

Figure 6.12: Top 20 of all possible tool configurations for Objective Function *OBJ* based on the *Repository Connectivity Graph*, ranked by amount of tools in configuration in ascending order.

Once the desired tool configuration is selected, KADMOS generates the FPG using the chosen tools, as presented in Fig. 6.13. Comparing the schematic *Fundamental Problem Graph* to its case-specific counterpart seen in Fig. 6.5, it can be observed that the tools are coupled by less existing connections.

This has to do with the fact that the schematic FPG is based on the schematic description of tool inputs and outputs, which does not take into account the multiplicity of nodes as the case-specific description does. Therefore, the schematic couplings do not indicate the correct amount of infor-

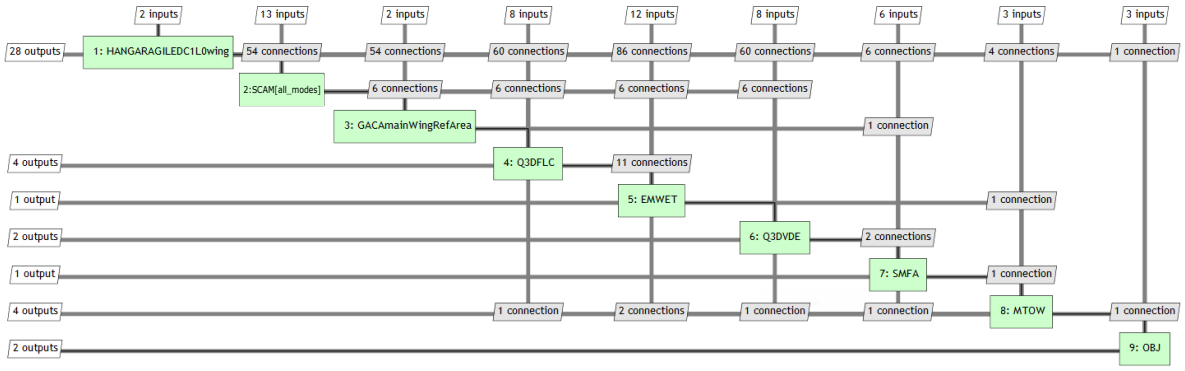


Figure 6.13: *Design Structure Matrix* of the schematic *Fundamental Problem Graph* applying the same tool configuration as in Fig. 6.5.

mation that would be exchanged in an executable simulation workflow using a certain tool configuration.

Since the selected configuration in the schematic *Fundamental Problem Graph* is executed to determine the use case in the Basefile, the generated use case is "tied" to that set of tools. Changing the tools after generating the Basefile implies that the *true* tool inputs and outputs are only assumed, which was one of the described issues of the repository.

The case-specific FPG, whose tool configuration and couplings are based on the generated use case, may turn out to not be the most suitable one when considering its exchange of information. This, however, can only be determined *after* the use case is generated using the schematic *Fundamental Problem Graph*.

Thus, it depends on the engineer's experience and knowledge of the system to *efficiently* choose a suitable tool configuration for the generated use case based on the schematic description. This circumstance also demonstrates the importance of a more sophisticated implementation of the selection of a tool configuration.

Once the tool configuration has been chosen for the schematic FPG, an execution sequence must be applied to the selected tools in order to execute them and generate the Basefile. This is done manually at the moment, but should be applied methodically in the future by possibly using system recommendations such as in the tool selection process described above. The selection of a sequence is important in the the generation of the Basefile because the *Fundamental Problem Graph* itself does not imply an execution sequence, which is needed to run the tools. However, since this use already prescribes a tool configuration and sequence, the order shown on the diagonal of Fig. 6.13 is applied.

Once the tool sequence is determined, the tools in the *Fundamental Problem Graph* can be executed to write Basefile. The principle is simple: each tool in the given order takes its inputs from the Basefile and writes its outputs to the same file. This way, the use case that is fully described by the Basefile is generated piece-by-piece. An illustration of this process is shown in Fig. 5.18.

The difficulties with this process arise when the input nodes are missing in the Basefile at the moment of execution of a given tool. Missing nodes can result from a multitude of reasons, such as a faulty tool calibration or erroneous tool execution of any of the preceding tools in the sequence. Another reason may be the selected sequence where required nodes for a given tool are only written to the Basefile at a later stage, and are therefore unavailable to the tool at the moment. Missing nodes that are not written by any tool in the configuration are referred to as System Inputs and must also

be provided in the Basefile generation process.

Consider a faulty set up of the *HANGAR* tool that results in the absence of the load case description that is required input of other tools such as *Q3D[FLC]* to estimate the loads on the wing. Since KADMOS has the schematic description of the tool inputs available in the tool repository, it is able to scan the Basefile and notify the user if any required input nodes are missing, as shown in Fig. 6.14 for this example.

```
Tool Execution: Q3D[FLC]
------------------------

The following nodes are missing for execution of Q3D[FLC]:
---
/cpacs/vehicles/aircraft/.../flightLoadCase/state/atmosphericConditions/density
/cpacs/vehicles/aircraft/.../flightLoadCase/state/atmosphericConditions/speedOfSound
/cpacs/vehicles/aircraft/.../flightLoadCase/state/atmosphericConditions/temperature
/cpacs/vehicles/aircraft/.../flightLoadCase/state/attitudeAndMotion/translation/acceleration/wDot
/cpacs/vehicles/aircraft/.../flightLoadCase/state/attitudeAndMotion/translation/velocity/u
---

Continue?
[0] No
[1] Yes
```

Figure 6.14: Missing nodes for tool execution for *Q3D[FLC]*.

Since the absence of these nodes prevents *Q3D[FLC]* from executing normally, they must be provided either by the user (manually) or by the system (automatically). Four methods for the addition of nodes are considered in KADMOS as shown in Table 6.2, of which the first two are presented.

Table 6.2: Various methods to add missing nodes to the Basefile.

Method	Description
Manual	System writes a template with missing nodes for user to fill in
Full-Automatic	Existing repositories are checked for existence of missing nodes. Nodes are matched by structure and uIDs; first match is returned.
Select from Repository	Similar as above, but all matches are returned for selection. (not implemented)
Match Node Structure	Only node structure is matched, but no uIDs. All matched nodes are returned for selection. (not implemented)

In the manual addition of missing nodes, a template is created by KADMOS that builds the node structure according to the schematic description of the nodes. Subsequently, it checks the existing nodes in the Basefile as well as the tool setting nodes for node references, as discussed in Section 5.6, and adds the appropriate *uIDs* to the relevant nodes. An example template file is shown in Fig. 6.15. Here, it can be seen that the appropriate *uIDs* are correctly indicated, and that the user must fill in the missing values himself. After completing the template, its nodes are merged into the Basefile which can subsequently be used for the execution of the tool.

In the automatic addition of nodes, KADMOS scans the existing files in the repository for a match in node structure and *uIDs* of the missing nodes. If a similar use case has been set up in the past featuring any of missing nodes, the system matches these and adds them to the Basefile. In the example of Fig. 6.16, KADMOS finds all matches for the missing nodes in the Basefile named *"MDO_AGILE_DC1"* of another use case.

By adding missing nodes to the Basefile using the developed methods, the tool configuration can be

```
<?xml version='1.0' encoding='UTF-8'?>
<cpacs>
  <vehicles>
    <aircraft>
      <model uID="agile_v13_modelID">
        <analyses>
          <loadAnalysis>
            <loadCases>
              <flightLoadCases>
                <flightLoadCase uID="Design-point_2.5g_MTOM_VMO_cruiseHeight">
                  <state>
                    <atmosphericConditions>
                      <speedOfSound>fill_in_here</speedOfSound>
                      <temperature>fill_in_here</temperature>
                      <density>fill_in_here</density>
                    </atmosphericConditions>
                    <attitudeAndMotion>
                      <translation>
                        <acceleration>
                          <wDot>fill_in_here</wDot>
                        </acceleration>
                        <velocity>
                          <u>fill_in_here</u>
                        </velocity>
                      </translation>
                    </attitudeAndMotion>
                  </state>
                </flightLoadCase>
              </flightLoadCases>
            </loadCases>
          </loadAnalysis>
        </analyses>
      </model>
    </aircraft>
  </vehicles>
</cpacs>
```

Figure 6.15: Example of a template file for the manual addition of the missing nodes shown in Fig. 6.14.

executed in the selected sequence to generate the complete aircraft description. This way, the Basefile contains all nodes that are read and written by the tools in the repository, ensuring consistency between the use case formulation and the true exchange of information during an optimization.

```
The following nodes were added to the Basefile:

 #  XPath                                                                                                  Value  Basefile
--  -----------------------------------------------------------------------------------------------  ---------  --------------------
 0  /cpacs/vehicles/.../flightLoadCase[Design-point_2.5g_MTOM_VMO_cruiseHeight]/state/.../speedOfSound      295.1  MDO_AGILE_DC1-base.xml
 1  /cpacs/vehicles/.../flightLoadCase[Design-point_2.5g_MTOM_VMO_cruiseHeight]/state/.../temperature    216.584  MDO_AGILE_DC1-base.xml
 2  /cpacs/vehicles/.../flightLoadCase[Design-point_2.5g_MTOM_VMO_cruiseHeight]/state/.../density      0.31093  MDO_AGILE_DC1-base.xml
 3  /cpacs/vehicles/.../flightLoadCase[Design-point_2.5g_MTOM_VMO_cruiseHeight]/state/.../acceleration/wDot  14.715  MDO_AGILE_DC1-base.xml
 4  /cpacs/vehicles/.../flightLoadCase[Design-point_2.5g_MTOM_VMO_cruiseHeight]/state/.../velocity/u    230.178  MDO_AGILE_DC1-base.xml
```

Figure 6.16: Addition of missing nodes of Fig. 6.14 using the automatic method.

The generated Basefile now contains the use case and can be used to extract the case-specific tool input and output files. This is done automatically by KADMOS using the schematic description of the tool inputs and outputs, as well as the Node Mapping, in order to relieve engineers from this task for the reasons stated in the previous section.

The case-specific tool inputs and outputs contained in the extracted files can now be used to generate the case-specific *Repository Connectivity Graph* and *Fundamental Problem Graph*, the same that have been obtained through the manual implementation shown in Figures 6.4 and 6.5.

The generated FPG using the extracted files is shown in Fig 6.17 and represents the same tools and tool couplings that were established manually as described above. This demonstrates that the ap-

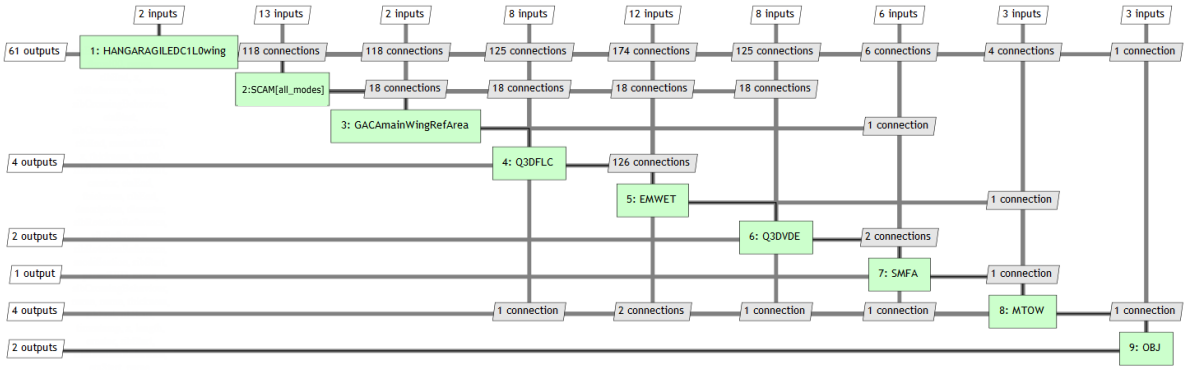


Figure 6.17: The same *Fundamental Problem Graph* as generated manually, created using the schematic tool repository.

proach using the schematic description of the tool input files, together with the appropriate mapping definitions, can be used to accelerate the MDO formulation process by automating most of the repetitive tasks related to the configuration of tool files for a given use case.

6.4. Changing the Use Case

To show the agile capabilities in the set up of MDO formulations for new use cases, a *Fundamental Problem Graph* containing the same tools and tool sequence is created featuring the wing of the *ATR-72* as an initial design point, as shown in Fig 6.18.

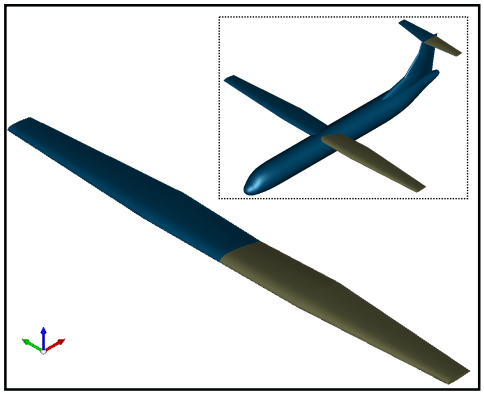


Figure 6.18: Initial design point using the wing of an *ATR-72*.

After changing the tool settings for *HANGAR* to generate the appropriate wing configuration (the ATR-72 wing), and adjusting the other tools in the repository to the relevant *uIDs*, the process described in the previous section is repeated. The changes to the tools can be seen in Fig.6.19 and 6.20 for *HANGAR* and *Q3D*, respectively.

The remainder of the process remains the same. First, the Objective Function, or graph sink, is selected in order to find the suitable tool configuration for the MDO formulation. The only difference in the selection of the tool configuration for this use case is the different *HANGAR* mode that needs to be run in order to create the ATR-72 wing geometry.

Once the schematic FPG and its sequence is defined, the system executes the tools in the given order and generates the Basefile for the new use case. In order to then create the case-specific *Repository Connectivity Graph*, the case-specific tool files are extracted from the Basefile and subsequently used to build the case-specific *Fundamental Problem Graph*, which is shown in Fig. 6.21.

```
<cpacs>
   <toolspecific>
      <hANGAR>
         <loadCpacsFile>
            ATR72_AGILE_Hangar_wing.xml
         </loadCpacsFile>
         <mode>
            ATR72_AGILE_Hangar_wing
         </mode>
      </hANGAR>
   </toolspecific>
</cpacs>
```

Figure 6.19: Different use case requires changes in the tool settings of *HANGAR*.

```
<capcs>
   <toolspecific>
      <q3D>
         <FLC modes="FLC">
            <typeOfRun>flightLoadCase</typeOfRun>
            <modelUID>ATR72</modelUID>
            <wingUID>wing</wingUID>
            <n_afp>100</n_afp>
            <AVL.nchord>8</AVL.nchord>
            <AVL.nspan>20</AVL.nspan>
            <Q3D.n_wing>8</Q3D.n_wing>
            <flightLoadCaseUID>
               Design-point_2.5g_MTOM_VMO_cruiseHeight
            </flightLoadCaseUID>
         </FLC>
         <VDE modes="VDE">
            <typeOfRun>viscousDragEstimation</typeOfRun>
            <modelUID>ATR72</modelUID>
            <wingUID>wing</wingUID>
            <n_afp>100</n_afp>
            <AVL.nchord>8</AVL.nchord>
            <AVL.nspan>20</AVL.nspan>
            <Q3D.n_wing>8</Q3D.n_wing>
            <flightLoadCaseUID>
               Design-point_1.0g_MTOM_Vcr_cruiseHeight
            </flightLoadCaseUID>
         </VDE>
      </q3D>
   </toolspecific>
</capcs>
```

Figure 6.20: Tool settings in *Q3D* adjusted for wing and aircraft model *uIDs*.

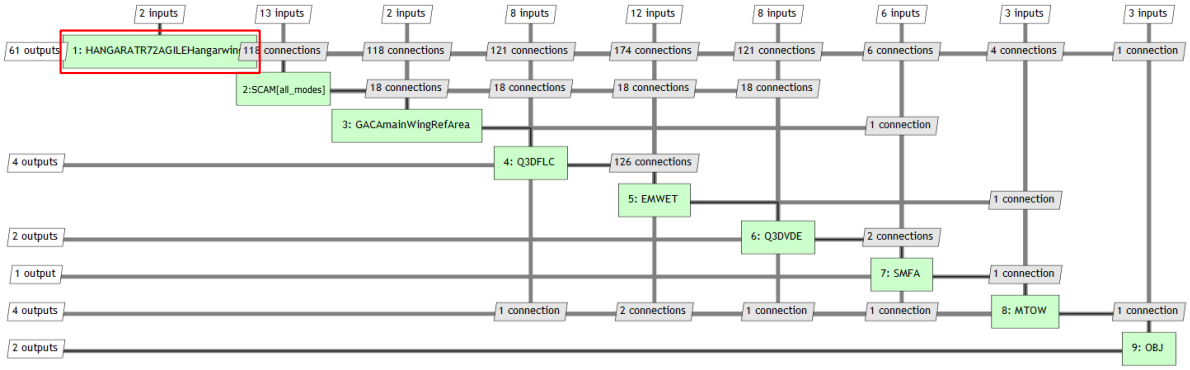


Figure 6.21: XDSM of the *Fundamental Problem Graph* using the initial design of the *ATR-72* wing.

6.5. Critical Reflection

Using the schematic description of the tools in the repository, many of the manual steps such as the extraction of tool files from the Basefile or the synchronization of node modifications between all relevant files have been automated. This is done to such an extent that the user simply has to change the tool settings of the relevant tools in order to implement changes to the use case or to set up a completely new case, without having to access the use case description in the Basefile at all.

The advantage of the schematic implementation is clear: tool files can be created once using the "abstract" node structure that is imposed by the applied XML-schema, and can be reused for any subsequent use case. By choosing the desired tools in the repository and specifying the tool settings, engineers can generate initial aircraft descriptions and use these to create MDO formulations according to the specific MDO problem. Most of the manual steps in the set up are removed or almost entirely automated, which means that less time is spent on repetitive tasks and less expert knowledge on the utilized tools is required to set up a use case, bringing KADMOS closer towards an agile approach in the configuration of MDO workflows.

7

Conclusion and Recommendations

7.1. Conclusion

In this research paper, the development of a tool repository that facilitates an agile generation of MDO workflows was described. Based on the research objectives stated in Chapter 1, the conclusions on each of the objectives are presented in this section.

- **Objective 1:** *Establish a tool repository structure that enables the automatic generation of MDO formulations according to an arbitrary MDO problem definition through the application of knowledge rules and knowledge reuse*

An approach for the structure of a tool repository was developed that uses the *schematic* definition of tool inputs and outputs together with a set of knowledge rules to generate case-specific MDO systems. The knowledge rules specify relationships between the nodes in an aircraft description and are defined by the applied XML schema. Since the system is dependent on an XML schema such as CPACS, but not restricted to a specific one, the system retains its validity while staying versatile in its application.

The repository structure has two distinct application levels. The schematic level defines the "abstract" description of tool inputs and outputs, but does not carry information on actual, or case-specific input and output. By selecting a schematic tool configuration for the solution of a defined MDO Problem and executing this configuration in a certain sequence, a use case is generated and stored in a Basefile. The second level of the tool repository encompasses case-specific tool input and output, and is used to generate the MDO Process representing the tool network for the specific use case.

- **Objective 2:** *Develop a methodology that facilitates the swift and uncomplicated addition of design tools and knowledge rules to the knowledge-enabled tool repository*

Since the graph networks that are used by KADMOS to describe the information exchange among software application are not well-suited as an interface with humans, the repository structure was based on XML, enabling an easy handing of files.

The engineer can interact with the tool repository on both the schematic and the case-specific level. On the schematic level, the engineer can add or remove tools, adjust tool execution settings and

easily modify the tool input and output structure in order to adapt it to the application. On the case-specific level, the user can inspect the true information exchange between tools, manually adjust details to inspect their impact on the network, or re-use the knowledge from previously generated use-cases to analyze the MDO system.

- ◇ **Objective 3:** *Extend KADMOS with the knowledge-enabled tool repository and integrate it with other KADMOS modules to allow for a rapid generation of MDO formulations*

A class structure was developed for KADMOS that defines the hierarchy of the separate graph types following the approach described by Pate et al. (2014). The class structure defines a top-level KADMOS-graph class that is used as the parent class for all graph types, thereby providing a common foundation for the analysis and manipulation of graphs structures in KADMOS. By dividing KADMOS graphs into separate graph types, each step in the generation of the MDO formulation can be visualized, inspected, and analyzed independently, thereby reducing the complexity of the system.

To allow for a flawless application of the information contained in the tool repository, the repository was integrated into the KADMOS class structure as a separate class, providing the required data for generation of the *Repository Connectivity Graph*. This graph serves as the starting point for the operation of graphs, and serves as the basis for the generation of MDO workflows.

The developed system was tested in a realistic MDO wing optimization scenario using a variety of separate tools and execution modes. The complexity in the configuration of MDO Processes was considerably reduced because the system takes over the majority of work in the set-up of tool networks, confining the role of the engineer to the set-up of the tool repository and the management of the workflow configuration.

Many repetitive and tasks in the system set-up were removed such as:

- Generating valid and consistent aircraft description files (Basefiles)
- Defining input and output files according to use case
- Determining and ensuring the correctness and consistency of tool couplings
- Ensuring tool inputs and outputs are valid and represent the actual tool behavior

The most important benefit of the developed system is its capability to rapidly redefine and reorganize MDO formulations when adding or removing disciplinary tools, changing tool settings and execution modes, or adjusting the initial aircraft design or top level requirements.

While any of these modifications previously led to a tedious reconfiguration of the MDO system, the developed approach automates most of the time-consuming tasks and allows for an *agile* redeployment of the system.

Although leaving room for improvement, the system has shown the potential to significantly reduce time and effort in the generation of complex MDO workflows, allowing engineers to apply their focus on the interpretation of the results that these workflows produce.

7.2. Recommendations

In this section, the most important recommendations for improvements in the developed tool repository and its application in KADMOS is presented. The recommendations are categorized according to "Conceptual" (Section 7.2.1) and "Implementation" (Section 7.2.2) .

7.2.1. Conceptual

Objective Functions The objective function is added to the graph as a sink, preventing constraint functions to be part of the workflow. The current implementation should be extended to allow the user to pick constraint functions based on the selected objective functions. Another issue with the current implementation of objective functions and sink is that only one sink can be implemented in the workflow. This could be extended in order to allow a combination of objective functions be selected as sinks.

FPG Generation The current implementation has many shortcomings in the selection of tool configurations as it was not the main goal of this reserch. Improvements should include more selection/analysis criteria for tool configurations, as well as a more refined ranking system that could be based on a combination of criteria to perform better trade-offs.

In the search for possible tool combinations, the current approach using "Combination Subsets" can be greatly improved by writing more sophisticated iterator functions that allow for more efficient filtering of configurations without running into performance issues. The recommended pre-filters should be implemented as these are expected to greatly decrease the inspected combinations, improving performance significantly.

Tool sequence is selected manually and without any support from the system in the current implementation. This should be improved, possibly by implementing the work of Schuurman (2017).

MDO Formulation In its current state, KADMOS creates simulation workflows based on the initial design point. It takes into account all tool couplings in the system of tools and can establish the most efficient workflow based on the initial knowledge about the used tools and aircraft description. However, certain tools may change their input and output structure dynamically during an optimization due to changes in nodes and node values. This may lead to significant changes in the tool couplings, which in turn may require adjustments in the MDO workflow to retain an optimal configuration.

7.2.2. Implementation

Schema Validation Although CPACS is a well-defined schema, it has some weaknesses with regard to the validation of files. By prescribing the presence and order of nodes, aircraft descriptions that do not follow the schema completely can not be validated. A possibility would be to include a *soft* validation that does not take into account missing nodes or node order, as suggested in Section 3.2.

Tool Settings At the moment, tool settings are stored in the *toolspecific*-node in the aircraft description and are not strictly "regulated" by CPACS. Following the discussion in Chapter 3, tool settings should either be separated from the nodes that are defined in CPACS, or be "CPACSized".

Execution Modes Execution modes in its current state are implemented according to the various tool settings for a given tool. However, difficulties have been encountered with the execution of tools in different "analysis cases", such as two different flight load cases. The current implementation treats each "analysis case" of a tool as a different setting, and therefore creates different function nodes for each case. A decision should be made in KADMOS to strictly define an "execution mode", and whether to separate "analysis cases" from execution modes or not.

User Interface The current user interface of KADMOS is console-based and may not represent the most efficient way to analyze and interact with graph networks. Since graph networks are not only computationally efficient, but also illustrate objects in a visually intuitive way, more effort could be spent on implementations that allow for a more user-friendly way to analyze and modify graphs. The current visualization package is a good start, but the user interaction is limited to the visualization of graph networks.

File Management At the moment, changes to any tool file, be it a schematic or case-specific file, must be validated for accuracy and consistency manually. An automated file management system could simplify the creation of tool files, validate changes in the tool files, and alert the user whether nodes follow the XML-schema definition before KADMOS is executed. KADMOS should perform these checks on the files before they are transformed into graphs.

Tool Input It would be useful to have a proper indication of which input nodes are actually read by each tool. In the current implementation, the tool inputs are provided according to the best knowledge of the engineer; however, it is not guaranteed that the indicated input nodes are actually used by the tool. This can naturally lead to faulty graphs, which ultimately leads to inefficiencies in the simulation workflow.

Node *uIDs* Node *uIDs* tend to be unique only throughout a specific use case. It is advisable to apply unique element *uIDs* throughout the repository so that differing descriptions do not result in a "match" when applying Node Mapping in the generation of the Basefile.

Node Values As discussed in Section 3.1, function graphs are generated based on the node structure and node *uIDs* of the nodes present in a tool file. In order to avoid inconsistencies, especially during a manual set up of a use case, node values should be checked to be consistent with the aircraft description.

Node Mapping The Node Mapping in the Basefile generation process and the tool files extraction does not work flawlessly due to experienced implementation issues and time constraints. It would be advisable to finish this implementation and "clean it up" in order to be able to apply the schematic workflow with confidence and make it more accessible for future enhancements (such as adopting new CPACS nodes, etc).

References

Agte, J., de Weck, O., Sobieszczanski-Sobieski, J., Arendsen, P., Morris, A., and Spieck, M. 2010. MDO: Assessment and direction for advancement-an opinion of one international group. *Structural and Multidisciplinary Optimization*, 40(1-6).

Belie, R., Martin, L., and Company, A. 2002. Non-Technical Barriers to Multidisciplinary Optimization in the Aerospace Industry. *9th AIAA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*.

Böhnke, D. ,Nagel, B., V. G. 2011. An approach to multi-fidelity in conceptual aircraft design in distributed design environments. *Aerospace Conference, 2011 IEEE*, pages 1–10.

Bowcutt, K. G. 2003. A Perspective on the Future of Aerospace Vehicle Design. *12th AIAA International Space Planes and Hypersonic Systems and Technologies Conference*.

DLR. CPACS – A Common Language for Aircraft Design, April 2016. URL `https://software.dlr.de/p/cpacs/home/`.

Flager, F. and Haymaker, J. 2007. A comparison of multidisciplinary design, analysis and optimization processes in the building construction and aerospace industries. *24th W78 Conference on Bringing ITC knowledge to work*.

van Gent, I. and La Rocca, G. 2017. Composing MDO symphonies : graph-based problem formulation to enable automated execution for large MDO systems. pages 5–9.

van Gent, I., Ciampa, P. D., Aigner, B., and Jepsen, J. 2017. Knowledge architecture supporting collaborative MDO in the AGILE paradigm. pages 5–9.

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

Martins, J. R. R. A. and Lambe, A. B. 2013. Multidisciplinary Design Optimization: A Survey of Architectures. *AIAA Journal*, 51(9).

Meier, C., Yassine, A. A., and Browning, T. R. 2007. Design Process Sequencing With Competent Genetic Algorithms. *Journal of Mechanical Design*, 129(6):566.

Moerland, E., Becker, R. G., and Nagel, B. 2015. Collaborative understanding of disciplinary correlations using a low-fidelity physics-based aerospace toolkit. *CEAS Aeronautical Journal*, 6(3): 441–454.

Nagel, B., Böhnke, D., Gollnick, V., Schmollgruber, P., Rizzi, A., La Rocca, G., and Alonso, J. J. 2012. Communication in Aircraft Design: Can We Establish a Common Language? *28th International Congress of the Aeronautical Sciences*.

Nagel, B. and Ciampa, P. D. 2014. Aircraft 3rd Generation MDO for Innovative Collaboration of Heterogeneous Teams of Experts. *European Comission, Horizon 2020*, page 4.

Pate, D. J., Gray, J., and German, B. J. 2014. A graph theoretic approach to problem formulation for multidisciplinary design analysis and optimization. *Structural and Multidisciplinary Optimization*.

Rizzi, A., Zhang, M., Nagel, B., Boehnke, D., and Saquet, P. 2012. Towards a Unified Framework using CPACS for Geometry Management in Aircraft Design. *50th AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition*.

Schmit, L. A. and Farshi, B. 1974. Some Approximation Concepts For Structural Concepts For Structural Synthesis. *AIAA Journal*, 12(5).

Schuurman, M. 2017. A Methodological Approach for Enabling the Analysis and Assessment of Multidisciplinary Design Workflows. *TU Delft Faculty of Aerospace Engineering*.

Sobieszczanski-Sobieski, J. and Haftka, R. T. 1996. Multidisciplinary Aerospace Design Optimization: Survey of Recent Developments. *American Institute of Aeronautics and Astronautics*, 70(3).

Spivey, M. Z. 2007. Combinatorial Sums and Finite Differences. *Discrete Mathematics*, 307(24): 3130–3146.

Vandenbrande, J. H., Grandine, T. A., and Hogan, T. 2006, The search for the perfect body : Shape control for multidisciplinary design optimization. In *44th AIAA Aerospace Sciences Meeting and Exhibit*, number January, 2006.

Venkataraman, S. and Haftka, R. 2002. Structural Optimization: What has Moore's Law Done for Us? *43rd AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*.

Vos, R. and Dommelen, J. V. 2012. A Conceptual Design and Optimization Method for Blended-Wing-Body Aircraft. *53rd AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics and Materials Conference*, pages 1–14.

W3C. Extensible Markup Language (XML), October 2016. URL `https://www.w3.org/XML/`.

W3Schools. XML Tutorial, January 2017. URL `https://www.w3schools.com/xml/`.

A

Code Sample: Case Study

In this appendix, the code sample for the case study in Chapter 6 is presented, following the same steps that are shown in the discussion.

KADMOS

This notebook demonstrates some of the capabilities of KADMOS in the dynamic generation of case-specific tool repositories.

System Set-Up

```
import os
import sys

kadmosPath = r'C:\Users\aMakus\Programming\Repositories\kadmos'  # specify path
sys.path.append(kadmosPath)  # add KAMODS path to system paths
```

```
# from pyKADMOS.scripts.Andreas_development_scripts.KB_init_test import initiate_schematic_kb
from pyKADMOS.sample.initiation import KnowledgeBaseInitiator
from pyKADMOS.sample.knowledgebase import KnowledgeBase
from pyKADMOS.sample import BASE_file_generation as BG
from pyKADMOS.sample import IO_file_generation as IO
```

Settings

```
# directories
working_dir = r'C:\Users\aMakus\Programming\Repositories\kadmos\pyKADMOS\scripts\Andreas_development_scripts\DEMO_KNOWLEDGE_BASE'  # use raw string
schema_dir = "MDO_WING_OPT_DEMO_1"
use_case_dir = "MDO_WING_OPT_DEMO_1"
```

Initiate KB schema and build RCG (schematic)

This workflow starts from the schematic tool database and creates the case-specific database.

```
init = KnowledgeBaseInitiator(working_dir, schema_dir, use_case_dir)  # create data object
RCG = init.get_MCG()  # full RCG
RCG_functionGraph = RCG.get_function_graph(keep_objective_variables=False)  # function graph
;
```

```
The provided directory name 'MDO_WING_OPT_DEMO_1' already exists in:
'C:\Users\aMakus\Programming\Repositories\kadmos\pyKADMOS\scripts\Andreas_development_scripts\DEMO_KNOWLEDGE_BASE\SPECIFIC_BASE'.

Would you like to overwrite its contents?
[0] No
[1] Yes
1

################################################################
Initiating KADMOS for Knowledge Base Schema:  MDO_WING_OPT_DEMO_1
################################################################

NOTE: Writing to C:\Users\aMakus\Programming\Repositories\kadmos\pyKADMOS\scripts\Andreas_development_scripts\DEMO_KNOWLEDGE_BASE\SPECIFIC_BASE\MDO_WI
NG_OPT_DEMO_1.
```

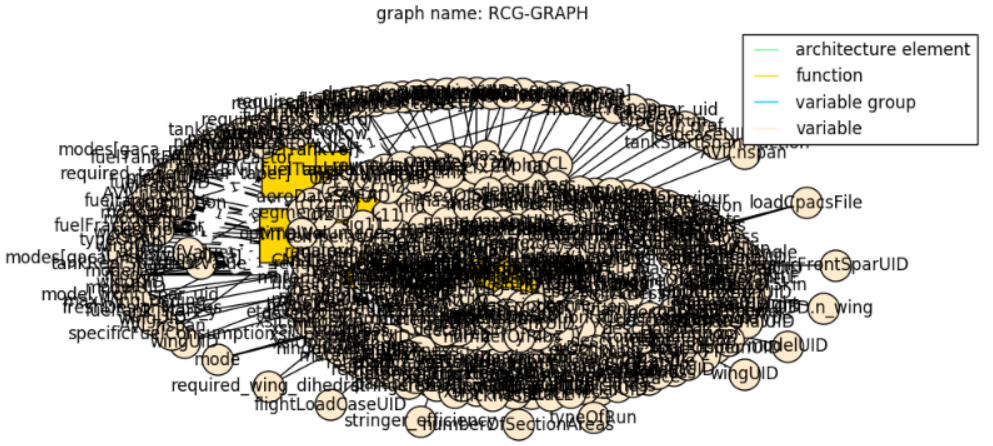
Plot RCG

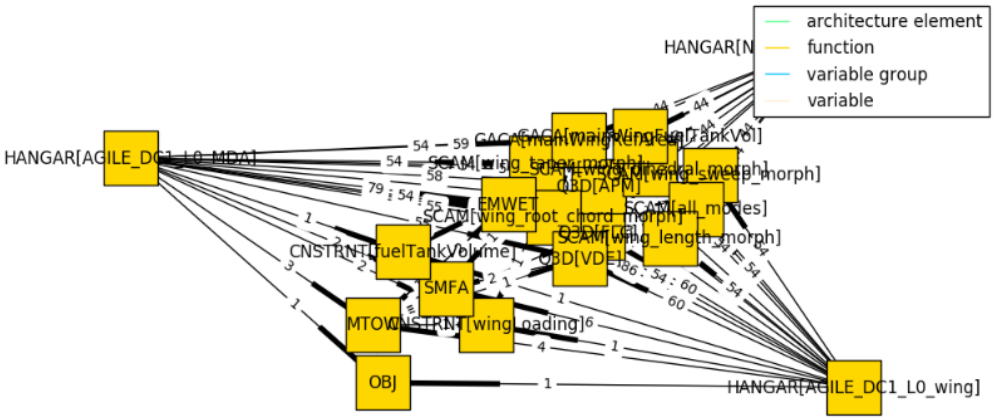
```
plotFullGraph = True  # T,F
plotFunctionGraph = True  # T,F
contractionLevel = None  # None or integer

# contract MCG
if isinstance(contractionLevel, (int, long)):
    plot_RCG = RCG.get_contracted_graph(contractionLevel)
else:
    plot_RCG = RCG

# plot graphs
fig_size_laptop = (13, 6)
if plotFullGraph:
    print "\nBuilding RCG... please have patience..."
    plot_RCG.plot_graph("Full RCG", fig_size=fig_size_laptop, show_now=True, edge_label='coupling_strength')
if plotFunctionGraph:
    print "\nBuilding RCG... please have patience..."
    RCG_functionGraph.plot_graph("Function RCG", fig_size=fig_size_laptop, show_now=True, edge_label='coupling_strength')
```

```
Building RCG... please have patience...
```





Get Objective Function(s)

```
# if multiple OBJs added, select
objs = RCG.select_objectives_from_graph()
```

```
The following function nodes were found in graph:

  #    Inputs    Outputs    Couplings  Function
---  --------  ---------  -----------  ----------------------------
  0        58          1          179  SCAM[wing_sweep_morph]
  1        57          1          179  SCAM[wing_dihedral_morph]
  2        56          1          169  GACA[mainWingRefArea]
  3        57          1          179  SCAM[wing_length_morph]
  4        67          2          178  GACA[mainWingFuelTankVol]
  5         6          2            5  CNSTRNT[wingLoading]
  6       109          2          239  EMWET
  7         4          2            3  OBJ
  8        15          2           14  SMFA
  9         7          2            4  CNSTRNT[fuelTankVolume]
 10        58          3          203  SCAM[wing_taper_morph]
 11        57          3          203  SCAM[wing_root_chord_morph]
 12        68          4          175  Q3D[VDE]
 13        65          6          169  Q3D[APM]
 14         7          6           16  MTOW
 15        67          6          239  SCAM[all_modes]
 16        68         15          184  Q3D[FLC]
 17         2         72          523  HANGAR[NASA_CRM_AGILE_Hangar]
 18         2        125          710  HANGAR[AGILE_DC1_L0_wing]
 19         2        323          693  HANGAR[AGILE_DC1_L0_MDA]

Please select Objective(s):

7

The following was selected:
7
```

FPG Generation Settings

```
# print/plot
print_configs = True  # T,F
plot_configs = False  # T,F
print_details = True  # T,F
conf_limit = 30  # display amount of (sorted) configs
# tool config settings
min_funcs = 3  # minimum functions in config

# ignore one of the two
if schema_dir == "MDO_WING_OPT_DEMO_1":
    ignore_HANGAR = 'HANGAR[AGILE_DC1_L0_MDA]'
else:
    ignore_HANGAR = 'HANGAR[AGILE_DC1_L0_wing]'

ignoreFuncs = [ ignore_HANGAR,
                "HANGAR[NASA_CRM_AGILE_Hangar]",
                "SCAM[wing_length_morph]",
                "SCAM[wing_taper_morph]",
                "SCAM[wing_dihedral_morph]",
                "SCAM[wing_sweep_morph]",
                "SCAM[wing_root_chord_morph]"
              ]
print "ignore:",ignoreFuncs
```

```
ignore: ['HANGAR[AGILE_DC1_L0_MDA]', 'HANGAR[NASA_CRM_AGILE_Hangar]', 'SCAM[wing_length_morph]', 'SCAM[wing_taper_morph]', 'SCAM[wing_dihedral_morp
h]', 'SCAM[wing_sweep_morph]', 'SCAM[wing_root_chord_morph]']
```

FPG Generation

```
# retrieve all possible function combinations (workflow configuration) for MPG
MPGpaths = RCG.get_function_paths_by_objective(*objs, ignore_funcs=ignoreFuncs, min_funcs=min_funcs)

# compare function combinations and rank them in table
selCombo = RCG.select_function_combination_from(*MPGpaths, n_limit=conf_limit, print_combos=print_configs, plot_combos=plot_configs)

# get FPG using the selected function combination
FPG = RCG.get_FPG_by_function_nodes(*selCombo)
```

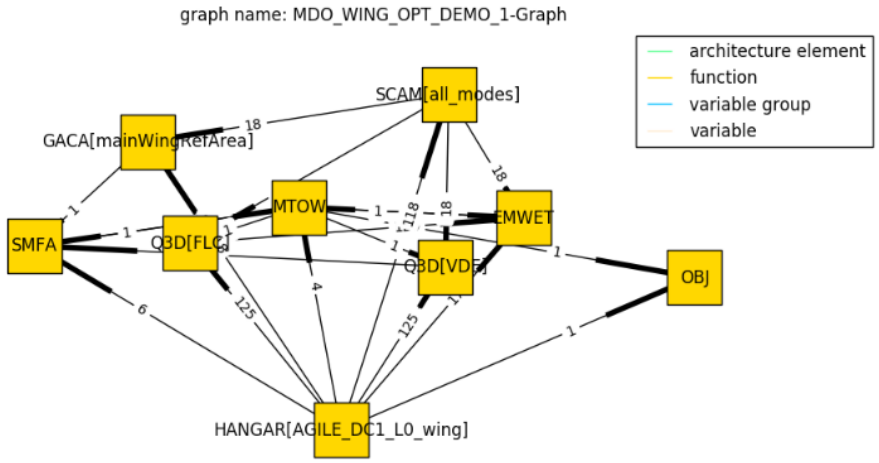
```
NOTE: Function configurations are considered for Objective(s): [OBJ]

The following tool configurations were found in the graph:

  #    functions    nodes    edges    couplings    system_inputs  Configuration
---  -----------  -------  -------  -----------  ---------------  ----------------------------------------------------------------------------------------
-------------------------------------------------
  0            9      457     1271          898               57  ['HANGAR[AGILE_DC1_L0_wing]', 'Q3D[VDE]', u'OBJ', 'Q3D[FLC]', u'MTOW', u'SMFA', 'SCA
M[all_modes]', u'EMWET', 'GACA[mainWingRefArea]']
  1            8      444     1008          628              175  ['HANGAR[AGILE_DC1_L0_wing]', 'Q3D[VDE]', u'OBJ', u'MTOW', u'SMFA', 'SCAM[all_mode
s]', u'EMWET', 'GACA[mainWingRefArea]']
  2            8      443     1122          690               44  ['HANGAR[AGILE_DC1_L0_wing]', 'Q3D[VDE]', u'OBJ', u'MTOW', u'SMFA', 'Q3D[FLC]', u'EM
WET', 'GACA[mainWingRefArea]']
  3            8      454     1150          761               56  ['HANGAR[AGILE_DC1_L0_wing]', 'Q3D[VDE]', u'OBJ', 'Q3D[FLC]', u'MTOW', u'SMFA', 'SCA
M[all_modes]', u'EMWET']
  4            8      394     1019          227              222  ['Q3D[VDE]', u'OBJ', 'Q3D[FLC]', u'MTOW', u'SMFA', 'SCAM[all_modes]', u'EMWET', 'GAC
A[mainWingRefArea]']
  5            8      446     1134          752               51  ['HANGAR[AGILE_DC1_L0_wing]', u'OBJ', 'Q3D[FLC]', u'MTOW', u'SMFA', 'SCAM[all_mode
s]', u'EMWET', 'GACA[mainWingRefArea]']
  6            7      433      871          482              169  ['HANGAR[AGILE_DC1_L0_wing]', u'OBJ', u'MTOW', u'SMFA', 'SCAM[all_modes]', u'EMWET',
'GACA[mainWingRefArea]']
  7            7      441      887          491              174  ['HANGAR[AGILE_DC1_L0_wing]', 'Q3D[VDE]', u'OBJ', u'MTOW', u'SMFA', 'SCAM[all_mode
s]', u'EMWET']
  8            7      305      695          433               37  ['HANGAR[AGILE_DC1_L0_wing]', 'Q3D[VDE]', u'OBJ', u'MTOW', u'SMFA', 'SCAM[all_mode
s]', 'GACA[mainWingRefArea]']
  9            7      432      985          562               38  ['HANGAR[AGILE_DC1_L0_wing]', u'OBJ', u'MTOW', u'SMFA', 'Q3D[FLC]', u'EMWET', 'GACA
[mainWingRefArea]']
 10            7      443     1013          615               50  ['HANGAR[AGILE_DC1_L0_wing]', u'OBJ', 'Q3D[FLC]', u'MTOW', u'SMFA', 'SCAM[all_mode
s]', u'EMWET']
 11            7      440     1001          571               43  ['HANGAR[AGILE_DC1_L0_wing]', 'Q3D[VDE]', u'OBJ', u'MTOW', u'SMFA', 'Q3D[FLC]', u'EM
WET']
 12            7      383      882          206              216  [u'OBJ', 'Q3D[FLC]', u'MTOW', u'SMFA', 'SCAM[all_modes]', u'EMWET', 'GACA[mainWingRe
fArea]']
 13            7      430      859          438              162  ['HANGAR[AGILE_DC1_L0_wing]', 'Q3D[VDE]', u'OBJ', u'SMFA', u'MTOW', u'EMWET', 'GACA
```

Plot FPG

```
# get function graph for MCG, FPG
FPGfunctionGraph = FPG.get_function_graph(keep_objective_variables=False)
fig_size_laptop = (13, 6)
FPGfunctionGraph.plot_graph("Plot", fig_size=fig_size_laptop, show_now=True, edge_label='coupling_strength')
```



Define Tool Sequence (Manually)

```
if schema_dir == "MDO_WING_OPT_DEMO_1":
    HANGAR = 'HANGAR[AGILE_DC1_L0_wing]'
else:
    HANGAR = 'HANGAR[AGILE_DC1_L0]'

execSequ = [ HANGAR,
             'SCAM[all_modes]',
             'GACA[mainWingRefArea]',
             'Q3D[FLC]',
             'EMWET',
             'Q3D[VDE]',
             'SMFA',
             'MTOW',
             'OBJ'
             ]
```

Generate Basefile

```
schema_files_dir_path = init.schema_files_dir_path
spec_files_dir_path = init.specific_files_dir_path

basef_gen = BG.Base_file_generator(FPG, schema_files_dir_path, print_details=print_details)
basefile_path = basef_gen.generate_basefile(execSequ, spec_files_dir_path)
```

```
NOTE: XML-schema already exists in target paths.


~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Basefile generation initiated for tool sequence: ['HANGAR[AGILE_DC1_L0_wing]', 'SCAM[all_modes]', 'GACA[mainWingRefArea]', 'Q3D[FLC]', 'EMWET', 'Q3D[V
DE]', 'SMFA', 'MTOW', 'OBJ']
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Checking mapping-dict for completeness...

WARNING - The following link paths are not present in 'mapping dict':
[0]  /cpacs/vehicles/aircraft/model/wings/wing/componentSegments/componentSegment/structure/lowerShell/stringer/stringerStructureUID
[1]  /cpacs/vehicles/aircraft/model/wings/wing/componentSegments/componentSegment/structure/upperShell/stringer/stringerStructureUID
[2]  /cpacs/vehicles/aircraft/model/engines/engine/parentUID

Reload?
[0] No
[1] Yes
0

NOTE: No System Inputs found in graph.


Tool Execution: HANGAR[AGILE_DC1_L0_wing]
-----------------------------------------

Running HANGAR in AGILE_DC1_L0_wing mode...

Tool execution completed; output successfully written to Basefile.


Tool Execution: SCAM[all_modes]
--------------------------------
WARNING: UID-item (link) path not found in Mapping-Dict: /cpacs/vehicles/aircraft/model/wings/wing/componentSegments/componentSegment/structure/lowerS
hell/stringer/stringerStructureUID
WARNING: UID-item (link) path not found in Mapping-Dict: /cpacs/vehicles/aircraft/model/wings/wing/componentSegments/componentSegment/structure/lowerS
hell/stringer/stringerStructureUID
WARNING: UID-item (link) path not found in Mapping-Dict: /cpacs/vehicles/aircraft/model/wings/wing/componentSegments/componentSegment/structure/upperS
hell/stringer/stringerStructureUID
WARNING: UID-item (link) path not found in Mapping-Dict: /cpacs/vehicles/aircraft/model/wings/wing/componentSegments/componentSegment/structure/upperS
hell/stringer/stringerStructureUID
```

Generate Input/Output Files

```
IO_gen = IO.IO_file_generator(MPG, basefile_path, schema_files_dir_path, print_details=print_details)
IO_gen.generate_io_files(execSequ, spec_files_dir_path)
```

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
I/O-file generation initiated for tool sequence: ['HANGAR[AGILE_DC1_L0_wing]', 'SCAM[all_modes]', 'GACA[mainWingRefArea]', 'Q3D[FLC]', 'EMWET', 'Q3D[V
DE]', 'SMFA', 'MTOW', 'OBJ']
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Checking mapping-dict for completeness...

WARNING - The following link paths are not present in 'mapping dict':
[0]  /cpacs/vehicles/aircraft/model/wings/wing/componentSegments/componentSegment/structure/lowerShell/stringer/stringerStructureUID
[1]  /cpacs/vehicles/aircraft/model/wings/wing/componentSegments/componentSegment/structure/upperShell/stringer/stringerStructureUID
[2]  /cpacs/vehicles/aircraft/model/engines/engine/parentUID

Reload?
[0] No
[1] Yes
0

Generating case-specific IO-files for HANGAR[AGILE_DC1_L0_wing]...
WARNING: UID-item (link) path not found in Mapping-Dict: /cpacs/vehicles/aircraft/model/wings/wing/componentSegments/componentSegment/structure/upperS
hell/stringer/stringerStructureUID
WARNING: UID-item (link) path not found in Mapping-Dict: /cpacs/vehicles/aircraft/model/wings/wing/componentSegments/componentSegment/structure/upperS
hell/stringer/stringerStructureUID
WARNING: UID-item (link) path not found in Mapping-Dict: /cpacs/vehicles/aircraft/model/wings/wing/componentSegments/componentSegment/structure/lowerS
hell/stringer/stringerStructureUID
WARNING: UID-item (link) path not found in Mapping-Dict: /cpacs/vehicles/aircraft/model/wings/wing/componentSegments/componentSegment/structure/lowerS
hell/stringer/stringerStructureUID
WARNING: UID-item (link) path not found in Mapping-Dict: /cpacs/vehicles/aircraft/model/engines/engine/parentUID
WARNING: UID-item (link) path not found in Mapping-Dict: /cpacs/vehicles/aircraft/model/engines/engine/parentUID

Generating case-specific IO-files for SCAM[all_modes]...
WARNING: UID-item (link) path not found in Mapping-Dict: /cpacs/vehicles/aircraft/model/wings/wing/componentSegments/componentSegment/structure/upperS
hell/stringer/stringerStructureUID
WARNING: UID-item (link) path not found in Mapping-Dict: /cpacs/vehicles/aircraft/model/wings/wing/componentSegments/componentSegment/structure/upperS
hell/stringer/stringerStructureUID
WARNING: UID-item (link) path not found in Mapping-Dict: /cpacs/vehicles/aircraft/model/wings/wing/componentSegments/componentSegment/structure/lowerS
hell/stringer/stringerStructureUID
WARNING: UID-item (link) path not found in Mapping-Dict: /cpacs/vehicles/aircraft/model/wings/wing/componentSegments/componentSegment/structure/lowerS
hell/stringer/stringerStructureUID

Generating case-specific IO-files for GACA[mainWingRefArea]...
```

Run Case-Specific Workflow

```
UC_dir_path = os.path.join(working_dir, "SPECIFIC_BASE")
UC_dir = use_case_dir + ''
print os.path.join(UC_dir_path, UC_dir)

print execSequ
```

```
C:\Users\aMakus\Programming\Repositories\kadmos\pyKADMOS\scripts\Andreas_development_scripts\DEMO_KNOWLEDGE_BASE\SPECIFIC_BASE\MDO_WING_OPT_DEMO_1
['HANGAR[AGILE_DC1_L0_wing]', 'SCAM[all_modes]', 'GACA[mainWingRefArea]', 'Q3D[FLC]', 'EMWET', 'Q3D[VDE]', 'SMFA', 'MTOW', 'OBJ']
```

```
UC_obj = KnowledgeBase(UC_dir_path, UC_dir)
RCG = UC_obj.get_MCG()
FPG = RCG.get_FPG_by_function_nodes(*execSequ)
```

```
 INPUT CHECKS
 ------------
Knowledge base 'MDO_WING_OPT_DEMO_1' found.
Reading files in the knowledge base...  Complete.
All files in the knowledge base were included.
XML Schema 'cpacs_schema.xsd' found.
Knowledge base files are in order.
Base file MDO_WING_OPT_DEMO_1-base.xml found.
WARNING: 'toolspecific' nodes found in EMWET-input.xml
WARNING: 'toolspecific' nodes found in GACA-input.xml
WARNING: 'toolspecific' nodes found in HANGAR-input.xml
WARNING: Multiple 'modes' attributes found in ancestry of element /cpacs/vehicles/aircraft/model/analyses/massBreakdown/mOEM/mWA/mass in HANGAR-outpu
t.xml; lowest one is applied.
WARNING: 'toolspecific' nodes found in MTOW-input.xml
WARNING: 'toolspecific' nodes found in OBJ-input.xml
WARNING: 'toolspecific' nodes found in Q3D-input.xml
WARNING: 'toolspecific' nodes found in SCAM-input.xml
WARNING: 'toolspecific' nodes found in SMFA-input.xml

Composing RCG...  Complete.
```

Plot FPG

```
plot_func_FPG = True
if plot_func_FPG:
    funcMPG = MPG.get_function_graph()
    print "\nBuilding plot... please have patience..."
    fig_size_laptop = (13, 6)
    funcMPG.plot_graph("Full RCG", fig_size=fig_size_laptop, show_now=True, edge_label='coupling_strength')
```

```
Building plot... please have patience...
```