Oskar Hvalstad-Nilsen

YAML Based Input File Format For Finite Element Analysis

Master's thesis in Engineering and ICT
Supervisor: Associate professor Bjørn Haugen
June 2019

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Engineering
Department of Mechanical and Industrial Engineering

Oskar Hvalstad-Nilsen

YAML Based Input File Format For Finite Element Analysis

Master's thesis in Engineering and ICT
Supervisor: Associate professor Bjørn Haugen
June 2019

Norwegian University of Science and Technology
Faculty of Engineering
Department of Mechanical and Industrial Engineering

NTNU
Norwegian University of
Science and Technology

Abstract

The finite element method (FEM) can be used to perform finite element analysis (FEA) of any given physical phenomenon, and is widely used in the fields of structural and mechanical engineering. Since FEA is computationally laborious it is best performed by a computer, and the FEM model is typically conveyed to the solver software by input files. These input files are generally formatted for specific FEA software, hence interoperability between them is an intricate matter.

YAML 1.2 is an openly accessible data serialization language for all programming languages. Among its design goals are human readability, data portability, expressiveness, and easy implementation. Its portability and expressiveness combined with the fact that well-defined YAML parsers already exist, makes it an interesting choice for FEA input files.

The objectives of this thesis are to:

I. Propose a YAML based input file format for finite element analysis of structural problems, and

II. Implement a prototype parser for the proposed format using C++ and available open-source tools.

This thesis proposes a basic YAML based input file format for FEA of structural problems. The format is based on Bjørn Haugen's Cfem input file format for FEA, and bears close resemblance to USFOS' structural file format, UFO, in terms of FEM data record definition. The format was developed by defining a top-level data structure in YAML and populating it with Cfem data record definitions, converted into the YAML language. Hence, the proposed format can represent a subset of the data types in Cfem. The format is less restrictive than many existing formats, as it supports arbitrary ordering of both data records and attributes. However, it lacks complex functionality featured in mature input file formats for FEA.

The prototype parser was implemented in (object-oriented) C++, using *yaml-cpp* to parse the YAML character stream. The prototype parser successfully constructs an internal C++ representation of the data in the character stream. Thus, it can be used in connection with format verification.

Both the proposed format and the prototype parser are characterized by being in their infancy. They are minimalistic and scarce, but they are extendible and function as a proof of concept: They prove that a YAML based input file format for FEA of structural problems is possible, and they form a basis for it.

Sammendrag

Elementmetoden kan brukes til å utføre datasimuleringer (FEA) av fysiske fenomener og er mye brukt innen konstruksjon og maskinteknikk. Siden FEA innebærer arbeidskrevende beregninger utføres det best av en datamaskin, hvor elementmodellen vanligvis overføres til løsningsprogramvaren via inputfiler. Disse inputfilene har som regel formater som er utviklet for én spesifikk FEA-programvare, slik at å sikre kompabilitet mellom dem er komplisert.

YAML 1.2 er et fritt tilgjengelig dataserialiseringsspråk for alle programmeringsspråk. Blant designmålene for YAML finner man mennesklig lesbarhet, dataportabilitet, uttrykkbarhet og enkel implementering. Disse egenskapene, kombinert med at det allerede eksisterer veldefinerte YAML-parsere, gjør dette til et interessant språk for inputfiler til FEA.

Målet med denne oppgaven er:

1. Foreslå et YAML-basert inputfilformat for FEA av konstruksjoner, og
2. Implementere en prototypeparser for formatet i C++ ved hjelp av tilgjengelige verktøy med åpen kildekode.

Denne oppgaven legger frem et grunnleggende YAML-basert inputfilformat for FEA av konstruksjoner. Formatet er basert på Bjørn Haugen's inputfilformat for FEA, Cfem. Det har også likhetstrekk med USFOS' strukturelle filformat, UFO, med hensyn til å definere datatyper. Formatet ble utviklet ved å definere en overordnet datastruktur, og ved å legge inn Cfem-datatyper, oversatt til YAML. Dermed kan det foreslåtte formatet representere en delmengde av datatypene i Cfem. Formatet er mindre restriktivt enn mange eksisterende formater, da det støtter vilkårlig rekkefølge blant både dataposter og attributter. Det mangler imidlertid komplisert funksjonalitet som finnes i mer modne inputfilformater for FEA.

Prototypeparseren ble implementert i (objektorientert) C++, og bruker *yaml-cpp* til å lese YAML-filene. Den lykkes i å lage en intern C++ representasjon av dataene i YAML-filen. Dermed kan prototypeparseren brukes til å verifisere det YAML-baserte inputfilformatet.

Både det foreslåtte formatet og prototypeparseren er preget av at de er i begynnelsen av utviklingsfasen. De er minimalistiske og knappe, men de kan utvides. Dermed fungerer formatet og prototypeparseren som et konseptbevis: De viser at et YAML-basert inputfilformat for FEA av konstruksjoner er mulig, og de danner et grunnlag videre utvikling.

Preface

This masters thesis was written in the spring of 2019 at the Department of Mechanical and Industrial Engineering (MTP) as part of the study programme Engineering and ICT (MTING) at the Norwegian University of Science and Technology (NTNU) in Trondheim. The project was carried out in the spring of 2019 and a pre-thesis project was conducted during the automn of 2018. The results of the pre-thesis project is embedded in this masters thesis.

This thesis aims at defining a good YAML based input file format for finite element analysis of structural problems. Additionally, a prototype parser for the proposed format is implemented in C++ using the open-source library yaml-cpp.

I would like to express gratitude to my supervisor Bjørn Haugen who has played an instrumental role in the work with this masters thesis. He has been incredibly helpful with both technical and practical issues, and provided invaluable feedback throughout the work with the project.

Trondheim, 2019-06-17

Oskar Hvalstad-Nilsen

Table of Contents

Abstract i

Sammendrag ii

Table of Contents vii

Code listings x

List of Tables xi

List of Figures xiii

Abbreviations xiv

1 Introduction 1

1.1 Background and motivation 1

1.1.1 Input files for FEA of structural problems 1

1.1.2 YAML as a basis for FEA input files 2

1.1.3 OOCfem and Focus Konstruksjon 3

1.2 Objectives and scope 4

1.2.1 Project deliverables 4

1.3 Structure of the thesis 5

2 Theoretical background **7**

2.1 Key terminology 7

2.2 Languages 8

2.2.1 YAML 1.2 8

2.2.2 A brief introduction to JSON 14

2.2.3 yaml-cpp 16

2.3 Existing input file formats for FEA 18

2.3.1 Abaqus input files (.inp) 18

2.3.2 Nastran's input file format for FEA 24

2.3.3 USFOS and the UFO structural file format 27

2.3.4 OOCfem and the CFEM input file format 30

3 Proposed solution **33**

3.1 Proposed YAML based input file format for FEA 33

3.1.1 Overview 33

3.1.2 Detailed description 34

3.2 Prototype parser for the proposed format 40

3.2.1 Overview 40

3.2.2 Detailed description 41

4 Evaluation and discussion **51**

4.1 The proposed YAML based input file format for FEA 51

4.1.1 Overview 51

4.1.2 Assessment of requirements 52

4.1.3 Use of YAML 53

4.1.4 Data hierarchy and dependency levels 54

4.1.5 Comparison with Abaqus, Nastran and UFO 55

4.1.6 Format verification using the prototype parser 57

4.1.7 Intended use of the proposed format 59

4.2 The prototype parser for the proposed format 59
4.2.1 Overview 59
4.2.2 Grouping of FEM data instances 60
4.2.3 Performance assessment 60
4.2.4 The converter/emitter module 62
4.2.5 Intended use of the prototype parser 62

5 Conclusion **63**

6 Future work **65**

Bibliography **69**

Appendices **72**

A Documentation of the YAML based format for FEA **73**
A.1 Native Input File Commands 76
A.1.1 Nodal data 76
A.1.2 Element data 80
A.1.3 Material data 85
A.1.4 Cross-sectional data 86
A.1.5 Load commands 89

B YAMLParser - Developer's guide **91**

C YAMLParser - How to run **95**

Code listings

2.1 Dynamic typing in Python . . . 8
2.2 Static typing in C++ . . . 8
2.3 Block style and flow style presentation of the same serialization tree . . . 11
2.4 Shopping list example in YAML block style . . . 11
2.5 Integer tags . . . 14
2.6 Floating points tags . . . 14
2.7 Miscallaneous tags . . . 14
2.8 Two YAML documents in one stream . . . 14
2.9 Example of complex mapping structure . . . 14
2.10 Simple *yaml-cpp* parsing example . . . 16
2.11 Conversion from `YAML::Node` to C++ primitive with yaml-cpp . . . 17
2.12 Example of yaml-cpp standard overload operator << . . . 18
2.13 The Abaqus input file (.inp) for the simple beam cantilever . . . 22
2.14 A simple beam element spanned by grid points in NASTRAN bulk data format 27
2.15 Example of legal and illegal use of the whitespace character is USFOS . . . 28
2.16 A simple beam element spanned by nodal points in UFO . . . 30
3.1 Anatomy of the STRUCTURE node, presented in YAML block style. . . . 35
3.2 Generic FEM data record . . . 35
3.3 Example of a NODE record with only mandatory attributes specified. . . . 36
3.4 Example of a fully specified NODE record. . . . 36
3.5 BEAM record with mandatory attributes . . . 37
3.6 COORDSYS specified with separate scalar fields . . . 38
3.7 COORDSYS specified with vectors presented as sequences . . . 38
3.8 Simple beam cantilever formulated in the proposed format using YAML flow style. . . . 39
3.9 Implementation of `Parser::parseDependencyLevelNull()` . . . 42
3.10 Logic for checking the existence of a specific FEM node in the structure . . . 45
3.11 Logic for checking the existence of a specific FEM element in the structure . . 45
3.12 Prototype parser assignment of optional attributes for a NODELOAD record . . 47
3.13 Populating `tokens2D` . . . 49
4.1 NODE records without aliasing. . . . 54
4.2 NODE records with aliasing for boundary codes. . . . 54
4.3 ISOMATERIAL record with optional attributes commented out. . . . 55
4.4 A NODE record in UFO . . . 56

4.5 A NODE record in YAML 56
4.6 Simple beam cantilever formulated in the proposed format using YAML flow style (reproduced). 58
4.7 Excerpt from Parser.cpp, modified to remove nodes from the YAML representation after they have been parsed. 61

List of Tables

2.1 Continuation Example of a PBAR entry in Nastran 26

4.1 Format file size comparison 57

List of Figures

2.1 YAML processing overview 9
2.2 YAML kind/style combinations 13
2.3 Composite structure composed by three plies. 21
2.4 Deflection of a simple beam cantilever analyzed by Abaqus/CAE 24
2.5 Nastran input file organization 24
2.6 Warehouse modelled in Focus Konstruksjon 31
2.7 Ski jump modelled in Focus Konstruksjon 32
2.8 Visualization of deflection 32
2.9 Assessment of parameters wrt. the Eurocode 32

3.1 Highlevel structure of the hierarchy in the proposed YAML based format for FEA. 34
3.2 Highlevel program flow of the prototype parser 40
3.3 Generic highlevel program flow for the *Parse dependency level* tasks in Figure 3.2 40
3.4 Highlevel flow chart of program flow in `Parser::Parse()` 42
3.5 Program flow for a generic parse FEM data type function 43
3.6 UML diagram of the Structure class 46
3.7 Inheritance diagram for Identifiable 48

4.1 Prototype parser output for the simple beam cantilever presented in Code listing 3.8 58

Abbreviations

API	Application Programming Interface
BC	Boundary condition
CAE	Computer-aided engineering
DoF	Degree of freedom
FE	Finite element
FEA	Finite element analysis
FEM	Finite element method
ID	Identifier (Identification number)
JSON	Javascript object notation
UFO	User-friendly structural file FOrmat
uuid	Universally unique identifier
URI	Uniform Resource Identifier
YAML	YAML Ain't Markup Language

Chapter 1

Introduction

1.1 Background and motivation

Finite Element Analysis (FEA) is widely used in the field of mechanical and structural engineering. All the physical phenomena in engineering mechanics are modelled by differential equations, and the finite element method (FEM) is a numerical approach that can be applied to solve general differential equations in an approximate manner [1]. According to an article in the IOP Conference series [2], FEM applies to industries such as machinery manufacturing, aerospace, automobile, ship, electronic, Engineering and so on.

FEM and FEA have had a significant impact on the engineering industry. In a technology blog post for SimScale, a software platform for Computer-aided engineering (CAE), the author cites[1] that some of the core benefits of FEM include ”increased accuracy, enhanced design and better insight into critical design parameters, virtual prototyping, fewer hardware prototypes, a faster and less expensive design cycle, increased productivity, and increased revenue” [3].

1.1.1 Input files for FEA of structural problems

Since FEA is computationally laborious it is best performed by a computer, and the FEM model is typically conveyed to the solver software by use of input files. A FEM model is formulated by a set of data entries typically specifying properties for geometry, mechanical behaviour, material behaviour, load situations, constraints, and connectivity [4]. An input file for FEA of structural problems must convey this information unambiguously with a well-defined syntax that can be interpreted by a software program. Examples of input file formats for FEA are Abaqus/CAE input file format, Nastran bulk data format, and UFO structural file format. These formats will be described in chapter 2.

[1] Secondary source: Hastings, J.K. Judes, M.A. Brauer, J.R. (1985) Accuracy and Economy of Finite Element Magnetic Analysis. 33rd Annual National Relay Conference

Serialization is used when large amounts of data have to be stored in flat files and retrieved at a later stage [5]. The complex structures modelled with FEM are time-consuming to make and their substructures often participate in other models, thus they must be storable and retrievable. If the model's data integrity is violated, it could have severe consequences. Faulty designs could cause failure of the structure, which would lead to enormous costs and, in the worst case, loss of lives - examples of engineering disasters are numerous [6]. Even if the error is uncovered early on, resolving it might still be costly. Hence, data serialization is a primary concern when designing an input file format for FEA of structural problems.

Input file formats for FEA are typically developed for specific FEA programs (e.g. Abaqus input file format is designed specifically for Abaqus). Thus, interoperability between the programs and portability of their input files are often scarce, so exchanging data between different FEA programs can be an intricate matter. This is a known problem that has been addressed by researchers [7]. Since the input files are not based on a common, openly accessible format the task of writing software to parse them is complicated. If, on the other hand, an input file for FEA was based on a well-defined open source language for data serialization, parsers could be written in any programming language that supports that data serialization language and utilize existing parsers for it.

The task of defining requirements for a good input file format for FEA of structural problems is complex. Nevertheless, this thesis will emphazise the following functional requirements for the proposed YAML based input file format for FEA:

1. The input file format must be sufficiently expressive to represent every data type comprised in a FEM structural model.
2. The input file format must be readable by computers, i.e. a computer must be able to parse the input file.
3. The input file format must be serializable (and deserializable).

The first requirement is extensive, as a FEM model typically consists of a vast number of data types, as seen by documentation of existing input file formats for FEA [8][9]. This requirement ensures that the format is adequately expressive to represent all information about geometry, material and mechanical behaviour, load situations, constraints, connectivity et cetera. The second requirement is essential because the very nature of FEA revolves around solving differential equations using computer software. The third requirement ensures that the data maintains its integrity.

1.1.2 YAML as a basis for FEA input files

YAML 1.2 is a human friendly, Unicode based data serialization language for all programming languages. It is designed for high human readability, data portability, expressiveness, and easy implementation [10].

YAML is interesting as a basis for FEA input files for several reasons. Firstly, it is openly accessible and portable between a wide variety of programming languages [10]. In general,

FEA programs can be written in any well-defined, expressive programming language, so a well-documented input file format that can be parsed in different programming languages could enhance portability of FEM data. Moreover, many FEA programs support scripting, often written in dynamic programming languages (e.g. Abaqus/CAE supports Python scripts), opening for possible benefit from using an input file format designed around the same data structures as these languages. The YAML 1.2 specification [10] states that YAML is designed to match the data structures of agile languages (a link between the term *agile programming languages* and dynamic programming languages is drawn in chapter 2.1) and has features beyond regular data serialization, making YAML an interesting choice for FEA input files.

The emphasis on human readability in YAML could also be beneficial for troubleshooting purposes. Oftentimes, troubleshooting involves that the user must inspect and edit the input files manually, in which case human readability and ease of use would be desired.

1.1.3 OOCfem and Focus Konstruksjon

In a meeting with supervisor Associate prof. Bjørn Haugen on 04.06.2019, he stated that Cfem was a software for FEA originating from the work in his ph.d. thesis [11]. Later, Cfem developed into an object-oriented program and its name was updated to OOCfem to account for this update. OOCfem is currently used for structural analysis in Focus Konstruksjon, a software for modelling and analysis of structural problems developed by Focus Software AS.

OOCfem features functionality to parse input files on the USFOS UFO format [12] and a designated input file format for OOCfem, named Cfem input file format, is under development. During the meeting, Associate prof. Haugen said that Cfem input file format aims to be a superset of the UFO format, with some alternative commands and an extended library of FEM data types. A key issue pursued by the Cfem input file format is the ability to pool material and cross-sectional data into one record type to be used for composite definitions.

As time has passed, the demand for a new input file format has arisen with the demand of utilizing cloud computing for the analysis. According to Associate prof. Haugen, Focus Software wishes to make a transition from the current API-based system to a file based one, to accommodate cloud computed analyses. With a file based system, entire input files could be sent to a cloud server to be analyzed, an option that is impossible with the API-based system. Having an input file format based on YAML is of high interest because of openly available parsers implemented in different languages. Moreover, it is a well-defined, expressive format with extensive documentation and features beyond regular data serialization [10].

1.2 Objectives and scope

The objectives of this project is to:

I. Propose a YAML based input file format for finite element analysis of structural problems, and

II. Implement a prototype parser for the proposed format using C++ and available open-source tools.

The purpose of the prototype parser is to verify that the format is possible to parse with C++ and available open-source tools, and to evaluate complexity of doing so. Hence, it must be able to consistently construct a C++ representation from example input files. Due to time limitations, the project scope is limited to defining a representative subset of a complete input file format; not to implement a broad library of data types.

1.2.1 Project deliverables

This thesis is the main deliverable for the project. In addition, source code for the prototype parser, documentation for the proposed YAML based input file format for FEA, and example files are delivered alongside this thesis. Due to the length of the input files, they are not included in Appendices. The files are found in the .zip file delivered alongside the thesis. The .zip file includes:

- The C++ source code for the prototype parser. The C++ project name is *YAMLParser*.
- Doxygen [13] documentation for the prototype parser.
- Example Cfem files, and corresponding YAML files. These files are found in the C++ project folder for YAMLParser. Additionally, example Cfem files from Focus Konstruksjon are found in the Cfem_example_models folder.

The Doxygen documentation is a collection of interactive HTML pages. The documentation for YAMLParser is found in the folder Doxygen_documentation, and the HTML start page is named *index.html*. This documentation is very useful to check classes, functions, relationships, inheritance diagrams etc.

The appendices in this thesis include the documentation of the proposed format (Appendix A), a developer's guide for YAMLParser (Appendix B), and a guide on how to run the YAMLParser program (Appendix C).

1.3 Structure of the thesis

This thesis will establish a theoretical foundation to design an input file format for FEA and present the proposed input file format and prototype parser.

Chapter 2 establishes the technical foundation of the project by presenting key aspects of the YAML language and the tools for parsing YAML in C++. Further, it presents the syntax and structure of well-defined input file formats for FEA to form a theoretical basis for evaluating the proposed YAML based input file format for FEA. When the technical foundation is established, the proposed solution is presented in chapter 3.

The nature of developing a file format and a software project implies that the deliverables contain source code and format documentation. These deliverables are not presented in their entirety in chapter 3, as documentation is comprised in the source code and in stand-alone documents. Thus, chapter 3 will present an overview of key elements in the implementation and structure of the proposed solution, and the reader is advised to consult the Doxygen documentation and format documentation (Appendix A) for further details. Chapter 3 is divided into two sections: section 3.1 concerning the proposed YAML based input file format and section 3.2 concerning the prototype parser.

Chapter 4 evaluates the proposed solution and discusses key issues in its design. Like chapter 3, this chapter is also divided into one part concerning the proposed YAML based input file format an one part concerning the prototype parser. After the evaluation, a conclusion is drawn in chapter 5 and objectives for future work are presented in chapter 6.

The reader of this thesis is expected to be familiar with FEM, object oriented programming, and C++. Additionally, the reader is assumed to have at least basic knowledge of data structures.

Chapter 2

Theoretical background

2.1 Key terminology

Native data structures of agile languages

According to the YAML 1.2 specification [10], one of the highest prioritized design goals of YAML is that it should match the native data structures of agile languages. However, the specification does not specify what an agile programming language is. Rather, it merely states that the following programming languages are examples of agile ones: Perl, Python, Ruby, and Javascript.

Agile Alliance is a nonprofit member organization dedicated to promoting the concepts of agile software development as outlined in the agile manifesto [14]. They state that ”Agile software development is an umbrella term for a set of frameworks and practices based on the values and principles expressed in the Manifesto for Agile Software Development and the 12 Principles behind it” [15]. The Manifesto for Agile Software Development [16] is concerned with strategies and working methodologies to uncover better ways of developing software. The manifesto does not discuss specific programming languages or language characteristics. Hence, the notion of *agile programming languages* is unclear, as it conflates programming languages and working methodologies for software development. This thesis will treat the term based on the following considerations:

Native data structures/types of a programming language are the simplest, most basic data types available in the language. Typically they are numerical and logical values (integers, floating points, booleans etc.) and characters. These types could again be wrapped in simple containers such as lists and mappings, provided in the language [17][18].

Despite the fact that the YAML 1.2 specification does not specify the term ”agile programming language”, the listed examples have an important commonality: They are all dynamically typed [19][20]. A study of dynamically typed languages conducted by Tratt [19] describes static and dynamic typing as follows: ”Statically typed languages are those which define and enforce types at compile-time. [...] Dynamic typing, at its simplest level, is when type checks are

left until run-time." Whereas statically type checked languages perform type checking during compilation, insisting that variable types must be unchanged throughout the run time of the program, languages with dynamic type checking allows variables to change their types during run time. Code listing 2.1 presents code in a dynamically typed language, and its opposition is illustrated in Code listing 2.2. Notice that for static typing, the type of var must be explicitly declared as an int, while in the Python example this is implicit.

Code listing 2.1: Dynamic typing in Python

```
var = 10
print(var)
var = "Hello World!"
print(var)
```

Output:

```
10
Hello World!
```

Code listing 2.2: Static typing in C++

```
int var = 10;
std::cout << var;
var = "Hello World!";
std::cout << var;
```

Output:

```
error: invalid conversion from 'const char*' to 'int':
var = "Hello World!";
```

The YAML 1.2 specification states that all data structures can be adequately represented with the following three basic primitives: Mappings (hashes/dictionaries), sequences (arrays/lists) and scalars (strings/numbers). In a dynamically typed language, such a scalar can contain numerical values, string values, booleans etc. Sequences and mappings have similar properties as they are not limited to containing one data type.

On basis of this, the *native data structures of agile languages* are considered to be the following three:

- **Scalars**: Atomic quantities that can hold one value at a time.
- **Sequences**: Data types that represent a countable number of non-unique, ordered values.
- **Mappings**: Associative memories/arrays constituted by unordered key/value-pairs, where each key is unique.

2.2 Languages

2.2.1 YAML 1.2

YAML (recursive acronym for "YAML Ain't Markup Language") is a human friendly, Unicode based data serialization language for all programming languages. It emphasizes data portability between programming languages, matching with native data structures of agile languages, having a consistent data model to support generic tools, and being expressive while easy to implement and use [10]. YAML is a language that has evolved over time. Only the most recent version, YAML 1.2 [21], will be described in this thesis[1].

YAML is an extensive language and this section will provide a detailed description of its underlying structures, syntax, and process flow. Some important YAML basics are summarized in the following bullet point list. The information is gathered from the YAML 1.2 specification [10].

[1]The prototype parser presented in chapter 3 only supports YAML 1.2.

- By convention, the extension for YAML files is **.yaml**.
- YAML is case sensitive.
- The native data structures of YAML are **scalars**, **sequences**, and **mappings**.
- YAML data can be presented in two styles: Block style and Flow style.
- YAML structures information by indentation of whitespace. A very important note is that tabs *cannot* be used for indentation.
- All data structures are represented as *YAML Node*s in the YAML representation.

The fact that tabs cannot be used for indentation in YAML prohibits possible ambiguities [10]. According to the YAML 1.2 specification the number of whitespaces used in a tab indentation could differ between programming environments. Consider a case where a YAML serialization has been stored as a character stream by an environment that uses four whitespace characters for each tab indentation. If the serialization is parsed by an environment that uses three whitespaces to represent a tab indentation, the environment may not recognize the indentation possibly causing a faulty YAML serialization in the load process. YAML maintains portability by requiring that tabs should not be used for indentation [10]. It is noteworthy that most modern editors supports user-defined configurations of the tab key such that tabs are substituted by a user-defined number of whitespace characters. However, this does not imply that the use of the ASCII tab character for indentation is allowed in YAML.

The YAML specification states that YAML is both a text format and a method for representing any native data structure in that format. On basis of this, it introduces two concepts: a class of data objects called YAML *representations* and a syntax for presenting the YAML representations as a character stream, called a YAML *stream*. The YAML representations are consumed by a computer program - an *application*, while the text format is designed for easy human consumption (readability). The tool for converting between these views is the YAML *processor* [10]. Figure 2.1 displays the stages of conversion between the native data structures and the character stream.

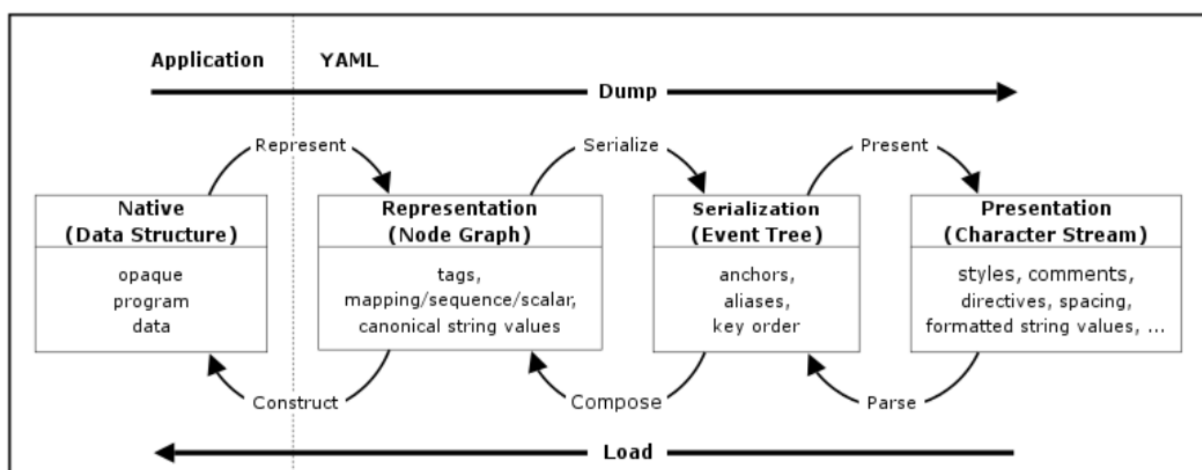


Figure 2.1: "[*YAML*] Processing Overview" [10].

As seen in Figure 2.1, the process of converting a native data structure to a character stream is called *dumping*. Its inverse, i.e. conversion from a character stream to native data structures, is called *loading*. Additionally, there are two intermediary stages between the native data structure and the character stream: A node graph representation and an event tree for serialization.

The YAML representation graph

The representation graph is the YAML representation of native data structures. It consists of mappings, sequences, scalars, tags, and canonical string values. The representation graph is a rooted, connected, directed graph of tagged YAML *nodes*. Important properties of the representation graph is that all nodes are reachable from the root node, the graph may contain cycles (it is *not* a directed acyclic graph, DAG), and the nodes, apart from the root node, have one or more incoming edges. Each YAML node represents a single native data structure - that is, its content is either scalar, sequence or mapping. Additionally, each node has a simple identifier, a *tag*, associated with it. The tag associates meta information with the node; it places a restriction on what values the node can have. Tags can be defined globally across all applications as unique URIs, or locally as application specific tags which are not expected to be unique. Since the tag dictates what values a node can have it plays an important role in node comparison. In particular, two nodes with the same tag and content are considered *equal* [10].

The YAML serialization tree

The serialization tree is a tree structure where each node has an ordered set of children, imposing order on mappings and indicating a subsequent occurrence of a previously encountered node [10]. The reason for producing a serialization tree is that it can be traversed to produce a series of event calls for one-pass processing of YAML data.

Since a node can have multiple incoming edges in the representation graph, i.e. it may be part of more than one collection, a way of referencing a node is necessary to serialize the graph. This is done by *anchors* and *aliases*. On the first occurrence of a node, it is identified by an anchor, and every subsequent occurrence is serialized as an alias node which refers back to that anchor. Anchors do not need to be unique within a serialization, nor do they need to have an alias node referring to it [10].

The key order of mapping nodes is a serialization detail that is imposed upon construction of the serialization tree and can not be overridden. Hence, all cases where node ordering is significant must be formulated as sequences. However, it is possible for the application to impose order on the keys. This is done by representing the mapping as a sequence of mappings, where each mapping is a single key:value pair [10].

The YAML presentation stream

As stated in the YAML 1.2 specification, the complementary view to the YAML representations is the presentation stream, *YAML character stream*. It is a stream of Unicode characters

aiming to present a YAML serialization in a human readable way. A YAML character stream can contain several serialization trees contained in independent documents in the stream. However, since the documents are independent, a YAML node cannot be member of more than one document.

The perhaps most significant versatility in the YAML presentation stream is the possibility to express YAML in two different styles: Block style and flow style. Block styles use indentation to denote structure, while the flow style use explicit indicators [10]. Code listing 2.3 shows the same serialization tree presented in block style and in flow style. The serialization tree represents a sequence of mappings named `musicians`.

Code listing 2.3: Block style and flow style presentation of the same serialization tree

A sequence presented in block style:
```
musicians:
  - Johann Sebastian Bach: classical
  - Dave Grohl: rock
  - AC/DC: rock
```

And in flow style:
```
musicians: [{Johann Sebastian Bach: classical}, {Dave Grohl: rock}, {AC/DC: rock}]
```

The YAML block style essentially mimics the way humans organize data to make it more human readable by using indentation-based scoping and a wide range of different YAML node styles in the YAML presentation (character stream) [10]. Moreover, an argument could be made that mappings and sequences are presented similarly to the way humans structure associative data and lists. Consider the example in Code listing 2.4. This way of structuring a shopping list is comparable to the way a human being might do it.

Code listing 2.4: Shopping list example in YAML block style

```
Shopping list:
  - milk
  - bread
  - honey
  - breakfast cereal
```

Data types

As previously stated, YAML is designed around the common native data types of agile programming languages, i.e. scalars, sequences and mappings [10], defined as follows:

- A **scalar** represents atomic data types, presented as zero or more Unicode characters. Examples of scalar content are numerical values (integers, floating points), characters, character strings, booleans, dates, and the `NULL` type.
- A **sequence** is an ordered series of indexed entries, analogous to a list in Python or a std::array in C++.
- A **mapping** is a container with unordered associations of unique keys to values, analogous to a Python dictionary or a std::map in C++.

The only data structure that represents an atomic data structure is the scalar. The other two are collections of data. Hence, the YAML 1.2 specification states that it is sometimes useful to regard sequences and mappings together as **collections**.

YAML syntax

The YAML syntax is a set of rules and practices governing character streams [10]. The two style partitions, block style and flow style, have many commonalities, but differ fundamentally in terms of presenting collections. The block style is the conventional YAML format and aims to satisfy the primary design goal of YAML - high human readability. In contrast, the flow style is more compact and can be considered as a natural extension of the JSON syntax [10]. This section will describe the YAML syntax for both block and flow styles, as described by the YAML 1.2 specification. Unless specifically stated, the syntax presented applies to both presentation styles.

- **Comments** are denoted by an octothorpe (#). They cannot appear within scalars, but can otherwise occur anywhere in the YAML character stream. All line content between a comment sign and the end of the line is considered a comment and will not affect the YAML model. YAML only supports single-line comments, thus, for comments spanning multiple lines each line must begin with an octothorpe.

Collections adhere to the following syntax:

- The YAML block style uses indentation (implicit typing) to denote scope and each entry begins with a new line.
  - A block sequence entry begins with a dash and a space (-).
  - A block mapping begins with a colon and a space (:), on the form *key: value*.
- The YAML flow style denotes structure by explicit type indicators.
  - A flow sequence is surrounded by brackets ([,]). The entries within the brackets are separated by comma (,).
  - A flow mapping is surrounded by curly braces ({, }). Each entry is a key: value pair, and the entries are separated by comma (,).

There are several styles in which to present **scalar** content in YAML. The following list summarizes the description of scalars specified in the YAML 1.2 specification:

- In block notation, scalar content can be written using a literal style where all line breaks are significant, that is new lines are preserved. The indicator for the scalar literal style is a vertical bar (|).

- Alternatively, it can be written with a folded style where each line break is folded to a whitespace character unless it ends an empty or a more indented line. The indicator for the scalar folded style is a greater-than sign (>).
- In flow notation, scalars include the plain style and two quoted styles (double quoted (" ") and single quoted (' ')).
- All flow scalars can span multiple lines and line breaks are always folded.
- The double quoted style must be used if escape sequences are needed in the scalar character stream.
- The single quoted style can be used when escaping is not required.

Figure 2.2 is reproduced from the YAML 1.2 specification and illustrates the possible style combination for YAML nodes.

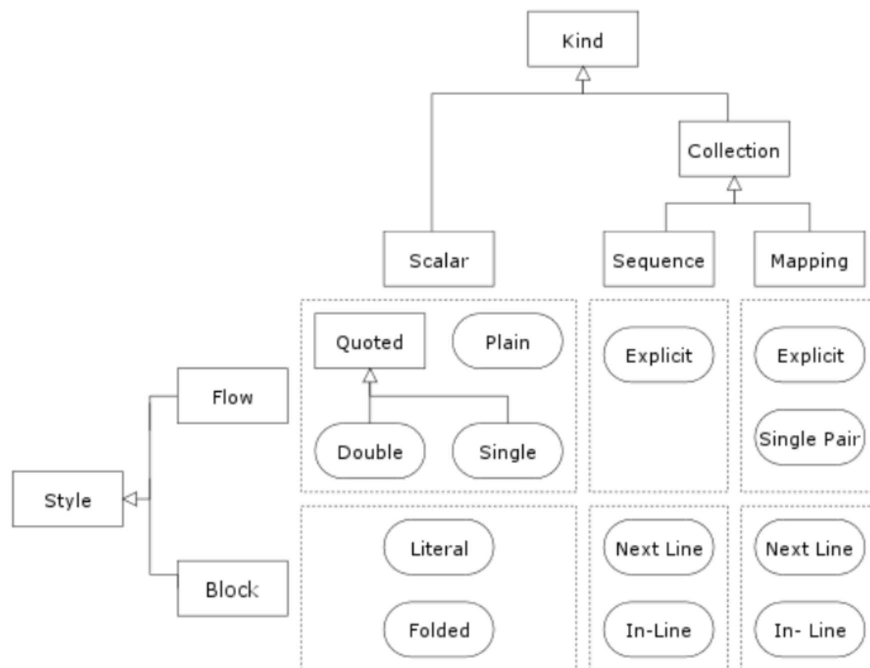


Figure 2.2: "[*YAML*] Kind/Style Combinations" [10]).

Tags associates meta information with a YAML node, placing a restriction on what values the node can have. Important syntax for tags are described beneath and some examples of tags are presented in Code listings 2.5 to 2.7[2].

- Untagged nodes are given a type by the application in the load process. The tags are resolved according to a defined YAML schema, e.g. the the *failsafe schema*. The YAML 1.2 specification defines YAML schemas as follows: A YAML schema is a combination of a set of tags and a mechanism for resolving non-specific tags. [10].

[2] The examples are modifications of the examples in provided in the YAML 1.2 specification [10]

- Explicit typing is denoted with a tag using the exclamation mark (!).
- Global tags are defined by globally unique URIs.

Code listing 2.5: Integers tags

```
canonical: 15
decimal: +15
dexadecimal: 0xF
```

Code listing 2.6: Floating points tags

```
canonical: 1.234e+5
exponential: 200e+06
infinity: .inf
not a number: .NaN
```

Code listing 2.7: Miscallaneous tags

```
null:
booleans: [ true, false ]
string: '0123'
```

YAML scalars and collections are stored in *documents*, such that a YAML file consists of one or more documents. Syntactically the YAML data is structured using the following rules for **structures** [10]:

- Documents are separated using three dashes (---) to denote the beginning of a document, and three dots (...) to denote the end of a document.
- Repeated nodes within a document are first identified by an anchor (marked with an ampersand (&)). Then new occurrences of the nodes are aliased by referencing with an asterisk (*).
- A complex mapping is structured by explicitly indicating the key and value fields of the mapping. A complex mapping key is indicated by a question mark and whitespace character (?), and a complex value is indicated by a colon and whitespace character (:) like in regular mappings.

Code listing 2.8: Two YAML documents in one stream

```
---
#Purchase
date: 2019-05-17
items: [hot dog, ice cream]
sum: 59
...
---
#Purchase
date: 2019-05-20
items: [milk, bread, ham]
sum: 73
...
```

Code listing 2.9: Example of complex mapping structure

```
#Mapping with a multi-line
#literal style key
? |
  This is a literal style key
  spanning multiple lines
: This is its value
```

2.2.2 A brief introduction to JSON

The YAML 1.2 specification states that "[...] YAML [*therefore*] can be viewed as a natural superset of JSON" [10]. Specifically, the YAML flow style aims at resembling the JSON syntax. JSON (JavaScript Object Notation) is a lightweight data-interchange format, easily processed by both humans and machines. It is language independent, but follows conventions from programming languages such as C, C++, Java, JavaScript, Perl, and Python [22].

JSON is built on two basic, universal data structures [22]:

1. A collection of name/value pairs. In various languages, this is realized as an *object*, record, struct, dictionary, hash table, keyed list, or associative array.
2. An ordered list of values. In most languages, this is realized as an *array*, vector, list, or sequence.

With these data structures, JSON can take on five different forms [22][23]:

- A **value**, which can be an object, array, number, string, true, boolean value, or null.
- An **object**, which is an unordered set of name/value pairs surrounded by curly braces ({, }). Syntactically, a name is a string and it is not required to be unique. The name and the value in an object is separated by a colon (:), and an object's value is separated from the following name by a single comma (,).
- An **array**. This type is an ordered collection of values, surrounded by brackets ([,]) and separated by comma.
- A **string**, represented as a sequence of zero or more Unicode characters wrapped in double quotes (", "). All Unicode symbols can be part of a string, but some special ones must be escaped by placing a backslash (\) in front of the symbol. The special symbols are listed in ECMA-404 The JSON Data Interchange Standard [23].
- A **number**, which is a sequence of digits with an optional plus (+) or minus (-) sign. Numbers must be a sequence of digits, hence numerical values such as `Infinity` and `NaN` (not a number) is illegal. JSON is agnostic about the type of numbers (integer, float, double float etc.) and leaves the task of assigning such a type to the application upon parsing.

Even though YAML is practically a superset of JSON, there is an important detail that complicates this relation. Most importantly, JSON requires that mapping keys *should* be unique, while YAML insists they *must* be [10]. As a result, JSON files with non-unique mapping keys are not considered valid YAML files.

JSON and YAML differ in their design goals. JSON is designed for simplicity and universality, making it trivial to generate and parse, at the cost of human readability [22]. YAML, on the other hand, is primarily designed for human consumption, introducing structures that enhance readability but also complicate the processes of machine parsing and generating [10].

JSON is not as expressive as YAML, because it lacks features such as node referencing (anchors), and support for comments and non-digit numerical values. Consequently, writing a parser for YAML is more complex than writing a JSON parser. Thus, the trade-off between the simplicity of JSON and the expressiveness of YAML should be considered when choosing serialization format.

2.2.3 yaml-cpp

yaml-cpp is a YAML parser and emitter library for C++ matching the YAML 1.2 specification. It is an open-source software accessible on Github, licensed under the MIT License, and can be built on various platforms with CMake. Currently, the most recent version is *yaml-cpp 0.6.0*, which requires C++11 and, unlike earlier versions, do not depend on Boost source libraries [24]. Older versions (*yaml-cpp 0.5.x* and older) have an old API and thus it will not work with the C++ implementation proposed in chapter 3.

Parsing with yaml-cpp

The parse functionality in yaml-cpp relies on performing a YAML load operation (cf. Figure 2.1). It parses the character stream to a serialization tree, and composes the YAML representation graph which must be traversed by the application to construct the native data structures. When parsing a node, yaml-cpp opens an input stream and creates an instance of yaml-cpp's NodeBuilder class. The NodeBuilder handles information to set anchor, alias, tag and content. When the whole set of nodes in the YAML file is built, that is, a YAML representation graph has been composed, the application can access the nodes to obtain the native data types [24][25].

The YAML 1.2 specification states that each YAML node represents a single native data structure. yaml-cpp accommodates this by creating `YAML::Nodes` and populating them with the data from the character stream at the core of the parsing process. The YAML nodes constitute a graph, naturally adhering to the YAML 1.2 specification. The `YAML::Nodes` can either be built from scratch, or they can be asserted a type based on the data that populate it [25]. These two ways of composing a YAML node is illustrated in Code listing 2.10. Code listing 2.10 is reproduced from (excerpts of) examples presented in the yaml-cpp tutorial [25].

Code listing 2.10: Simple *yaml-cpp* parsing example

Building a YAML::Node from scratch:

```
YAML::Node node;  // starts out as null
node["key"] = "value";  // it now is a map node
node["seq"].push_back("first element");  // node["seq"] automatically becomes a sequence
node["seq"].push_back("second element");

node["mirror"] = node["seq"][0];  // this creates an alias
node["seq"][0] = "1st element";  // this also changes node["mirror"]
node["mirror"] = "element #1";  // and this changes node["seq"][0] - they're really the
                                // "same" node

node["self"] = node;  // you can even create self-aliases
node[node["mirror"]] = node["seq"];  // and strange loops
```

The resulting node is:

```
&1
key: value
&2 seq: [&3 "element #1", second element]
mirror: *3
self: *1
*3 : *2
```

An example of automated YAML::Node type assertion:

```
YAML::Node node = YAML::Load("[1, 2, 3]") // Populate with sequence data
assert(node.IsSequence()); // Check if the node represents a sequence
```

The assert statement will return `true` in this case.

To construct native data types from the representation graph, yaml-cpp has built-in functionality for conversion to an from most built-in data types [25]. yaml-cpp can convert to and from three special C++ standard library (STL) types: `std::vector`, `std::list`, and `std::map`. Conversion from YAML scalars to C++ primitives (Integers, characters, booleans, floating points, and double floating points) is supported by helper functions defined in *impl.h* in yaml-cpp. These helper functions are defined as C++ templates with respect to a `YAML::Node` so it is called as a member function of the node (see example in Code listing 2.11). The helper functions convert the type to the desired data type if the conversion is allowed. Otherwise, they throw a `BAD_CONVERSION` exception, defined in *exeption.h* in yaml-cpp.

yaml-cpp also supports conversion of user-defined types (defined by user-made classes). Such functionality must be implemented by the user in the `YAML::convert<>` template class [25].

Code listing 2.11: Conversion from `YAML::Node` to C++ primitive with yaml-cpp (reproduced from yaml-cpp Tutorial [25]).

```
YAML::Node node = YAML::Load("{pi: 3.14159}");

// Conversion from Node to double
double pi = node["pi"].as<double>();

----------------------------------------------------------
.as<>() is the template helper function and double is the primitive to convert to.
```

Emitting with yaml-cpp

In yaml-cpp, emitting is the process of writing YAML data to file [26], i.e. to perform a YAML dump operation. According to the yaml-cpp guide How To Emit YAML [26], the model for emitting YAML is `std::ostream` manipulators. A `YAML::Emitter` object acts as an output stream, and its output can be retrieved through the `c_str()` function.

As stated by the yaml-cpp guide How To Emit YAML, a `YAML::Emitter` object acts as a state machine, and a manipulator is used to move it between states. The manipulators generally affects the next output of the stream, but may also be set permanently for the lifetime of the `YAML::Emitter` object. Examples of desirable lifetime manipulators are setting the YAML style (block/flow) for collections, or setting a fixed, user-specified number of whitespaces for indentation. Throughout the lifetime of the `YAML::Emitter` object, it is switched back and forth between states depending on the structure of the data to emit. The `YAML::Emitter` object has different states for different YAML structures such as list, map, literal, block, flow, alias and anchor (section 2.2.1 describes the YAML structures).

yaml-cpp provides functionality to overload the C++ operator <<. By default *yaml-cpp* does the overload for three members of the C++ STL: `std::vector`, `std::list` and `std::map`. The consequence of this is that these data types can easily be emitted as YAML structures - the `std::vector` and `std::list` are emitted as YAML sequences, and the `std::map` is emitted as a YAML mapping [26]. An example of use is shown in Listing 2.12. It is also possible to do this overload for other, user-defined types. Moreover, operator << is overloaded for YAML::Node, enabling the emitter to output into existing nodes.

Code listing 2.12: Example of yaml-cpp standard overload operator <<

C++ code:
```
std::vector<int> primes;
primes.push_back(2);
primes.push_back(3);
primes.push_back(5);
primes.push_back(7);

std::map<std::string, std::string> headmasters;
headmasters["NTNU"] = "Gunnar Bovim";
headmasters["Hogwarts"] = "Albus Percival Wulfric Brian Dumbledore";

YAML::Emitter out;
out << YAML::BeginSeq;
out << YAML::Flow << squares;
out << ages;
out << YAML::EndSeq;
```

Emitter output:
```
- [2, 3, 5, 7]
-
  NTNU: Gunnar Bovim
  Hogwarts: Albus Percival Wulfric Brian Dumbledore
```

2.3 Existing input file formats for FEA

This section will describe the basics for some well-defined input file formats for FEA. The formats are selected on the basis of wide industrial use and relation to OOCfem. Central concepts and syntax will be attended to establish a basis for proposing a YAML based input file format for FEA of structural analysis.

2.3.1 Abaqus input files (.inp)

Abaqus Unified FEA is a product suite for finite element analysis and computer aided engineering used in industries such as automotive, aerospace, and industrial production. Abaqus/CAE (Complete Abaqus Environment) is a product in this suite, used for modelling, analysis, job management, and results visualization of Abaqus analyses [27].

The Abaqus input file is an ASCII data file that can be written in a simple text editor, but is usually generated by a graphical preprosessor such as Abaqus/CAE. In an Abaqus input file, the following sections constitute a FE model: A heading, a model data section, assemblies, and history data [28]. A description of these sections, collected from the Abaqus analysis user's manual [28][29], is presented in the following.

Model data defines the nodes, elements, materials, initial conditions etc. [28]. To define a FE model, geometry and material data must be defined in this section. The nodes, elements, and cross-sections constitute the model geometry. Most parts of the geometry (such as beams, shell elements etc.) are associated with materials. Thus, material definitions are core parts of the model. Material definitions include properties such as Young's modulus, Poisson's ratio, thermal data and behaviour (elasticity/plasticity, isotropy etc.). Additionally, the model data may include information about parts, initial conditions, boundary conditions, interactions, and

more. This data is not required by the Abaqus solver, but proves necessary to analyze complex structures.

FE models of large, complex structures can be composed of *parts* connected by **assemblies**. This makes the model more manageable and neat, but perhaps most importantly it allows reuse of part definitions. For instance, a steel structure connected by bolts could have a vast number of equal bolts. Defining the bolt as a part and reusing its definition saves a significant amount of time and greatly enhances the maintainability of the model. A part is an idealization of an object that can be instantiated any number of times in a model. Each part instance is a member of an assembly and is characterized by the part definition. In other words, an assembly is a collection of positioned part instances. Data such as boundary conditions, constraints, interactions and a loading history is defined with respect to an assembly [30].

History data comprises a number of *steps* carrying information about an analysis procedure, and external loads and/or non-equilibrium initial conditions. A sequence of one or more steps constitutes the history definition of the analysis [28].

The Abaqus input file defines data by combinations of keyword lines and data lines. In addition, Abaqus has a special line type for comments; comment lines. All comment lines are ignored by Abaqus and any information on these lines have no effect after parsing.

Keyword lines introduce options and often have parameters. An option is a set of data describing a part of the problem definition [29], for example the the collection of nodes in the problem is denoted by the `*NODE` keyword. The keywords are followed by zero or more parameters defining the behaviour of the option. The parameters are specific to the option, some are required and some are optional. The `*NODE` option have no required parameters but have several optional ones. For example the NSET parameter used to associate the node with a node set.

Most keyword lines are followed by **data lines** used to provide numeric or alphanumeric entries. The data lines populate the option with (alpha)numeric values that are processed in the analysis. Data lines are dependent on a preceding keyword line and cannot exist on their own.

The **heading** is used to define a title for the analysis and can span an arbitrary number of lines. This description will appear at the top of the output result files to clarify which input file the output is related to. The heading is optional, so it may be omitted. However, it is helpful in order to describe the input file to a reader. A heading is identified by the `*HEADING` keyword [28].

Syntax for Abaqus input files

Comments in an Abaqus input file can only occur on special lines, called comment lines. The Abaqus input syntax rules [29] state that any line that begins with stars (asterisks) in columns 1 and 2 (**) are considered comment lines. These lines can be placed anywhere in the file and will be ignored by Abaqus.

The syntax for keyword lines and data lines is quite extensive, thus only an excerpt will be presented here. For more information, consult the Abaqus Analysis User's guide [29].

Keyword lines adhere to the following syntax:

- The first non-blank character of each keyword line must be a star (*)[3].
- The keyword must be followed by a comma (,) if any parameters are given.
- Parameters must be separated by commas.
- Blanks on a keyword line are ignored.
- A line can include no more than 256 characters, including blanks. However, it can be continuated on the next line by inserting a comma as the last character on the line. This is necessary in situations where there are a high number of parameters or the parameter values contain many characters.
- Keywords and parameters are not case sensitive.
- If a parameter has a value, the equal sign (=) is used. The value can be an integer, a floating point number, or a character string, depending on the context.
- The same parameter should not appear more than once on a single keyword line.

An excerpt of the syntax for **data lines** [29] is presented in the following:

- A data line can include no more than 256 characters, including blanks. Trailing blanks are ignored.
- All data items must be separated by commas. An empty data field is specified by omitting data between commas. Abaqus will use values of zero for any required numeric data that are omitted unless a default value is specified.
- A line must contain only the number of items specified.
- Empty data fields at the end of a line can be ignored.
- Floating point numbers can occupy a maximum of 20 spaces including the sign, decimal point, and any exponential notation.
- Integer data items can occupy a maximum of 9 digits.
- Character strings can be up to 80 characters long and are not case sensitive.

Abaqus imposes an ordering on the data lines, based on the relationship between the variables. Whenever one variable is a function of another, the data must be given in the proper order so that Abaqus can interpolate for intermediate values correctly [29].

[3] In the Abaqus Analysis User's Manual, an asterisk is referred to as a *star*.

Representing composites in Abaqus - Abaqus sections

A *section* in Abaqus contains information about the properties of a part or a region of a part. The information required in the definition of a section depends on the type of region in question [31]. Sections provide useful features for defining a wide range of properties, such as pooling of material and cross section for use in composites, and possibilities to assign electromagnetic, fluid and acoustic properties. The Abaqus reference guide [31] divides sections into five categories: Solid sections, shell sections, beam sections, fluid sections and other sections. In turn, these categories have one or more sub types, many of which are past the scope of this project. This section will give a brief description of section types concerning composites, based on the Abaqus analysis user's manual [31][32][33].

Facilitating functionality to represent composites is important in the YAML based input file format proposed in chapter 3, as composites are common in structural problems. Hence, Abaqus' approach to represent composites are of high interest. There are two types of sections concerning composite representation. The first type is **composite *solid* sections**, consisting of layers of materials [31]. This type of composite section is used in connection with two-dimensional, three-dimensional and/or axisymmetric solid regions. Each layer (ply) is specified in the input file by a material name, a thickness, and an orientation [31]. The orientation property concerns the orientation of the material (direction of the fibers). Additionally the number of plies (integration points) must be specified [33]. These properties adequately defines a (solid) composite.

The second type is **composite *shell* sections**, consisting of layers of materials, a section Poisson's ratio, and optional rebar layers [32]. The parameters for defining this section type is similar to the parameters needed to define a solid composite section - a material name, a thickness, and an orientation must be specified for each ply (layer) [31]. Each ply affects composite behaviour in terms of the shell section's response to stretching, bending, shear and torsion [32]. The two types of composite sections have many commonalities, and composite solid elements are primarily intended for modelling convenience; they usually do not carry any benefits over composite shell sections in terms of solution accuracy [33].

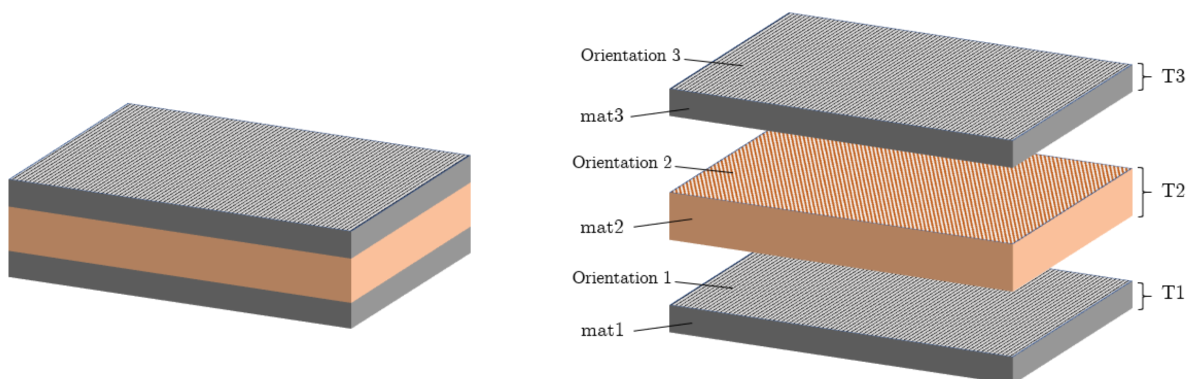


Figure 2.3: Composite structure composed by three plies. Each ply is specified by a material, a thickness, and a material orientation.

Abaqus sets

Abaqus offers functionality to define node and element *sets*. Sets are named by a character string of length 1 to 80 characters. This name is used to reference all the members of the set. Element sets are useful for associating data and operations on groups of nodes/elements. For instance, boundary conditions such as symmetry can be assigned to a set of nodes [29]. The Abaqus Analysis User's Manual describe sets as the basic reference throughout Abaqus.

Example input file

The following example presents a simple beam cantilever modelled and analyzed in Abaqus/-CAE. The beam element is of type B21, meaning it is linear and planar [34]. The beam is comprised of eleven nodes and ten beam elements, gathered in a node set and an element set, respectively. The material is elastic steel with Young's modulus 200 MPa and Poisson's ratio 0.27. The beam is fixed in all DoFs on its left end, specified in the data records associated with the `*Boundary` heading in the input file. A concentrated force of 10 kN is applied at the rightmost end of the beam, specified in the step section of the input file. Lastly, field output and history output are requested for visualization. In this example, the requests are simple preselected values. The Abaqus input file for the analysis is presented in Code listing 2.13 and an illustration of the deflection computed by Abaqus can be seen in Figure 2.4.

Code listing 2.13: The Abaqus input file (.inp) for the simple beam cantilever

```
*Heading
** Job name: Job-2 Model name: BeamModel
** Generated by: Abaqus/CAE 2017
*Preprint, echo=NO, model=NO, history=NO, contact=NO
**
** PARTS
**
*Part, name=BeamPart
*Node
      1,           0.,           0.
      2,          0.5,           0.
      3,           1.,           0.
      4,          1.5,           0.
      5,           2.,           0.
      6,          2.5,           0.
      7,           3.,           0.
      8,          3.5,           0.
      9,           4.,           0.
     10,          4.5,           0.
     11,           5.,           0.
*Element, type=B21
 1,  1,  2
 2,  2,  3
 3,  3,  4
 4,  4,  5
 5,  5,  6
 6,  6,  7
 7,  7,  8
 8,  8,  9
 9,  9, 10
10, 10, 11
*Nset, nset=BeamSet, generate
  1,  11,   1
*Elset, elset=BeamSet, generate
  1,  10,   1
```

```
** Section: BeamSection  Profile: Profile-1
*Beam Section, elset=BeamSet, material=Steel, poisson = 0.27, temperature=GRADIENTS, section=I
0.2, 0.3, 0., 0.1, 0., 0.05, 0.05
0.,0.,-1.
*End Part
**
**
** ASSEMBLY
**
*Assembly, name=Assembly
**
*Instance, name=Part-1-1, part=BeamPart
*End Instance
**
*Nset, nset=FixedNode, instance=Part-1-1
 1,
*Nset, nset=FreeEnd, instance=Part-1-1
 11,
*End Assembly
**
** MATERIALS
**
*Material, name=Steel
*Elastic
 2e+11, 0.27
**
** BOUNDARY CONDITIONS
**
** Name: FixedEnd Type: Displacement/Rotation
*Boundary
FixedNode, 1, 1
FixedNode, 2, 2
FixedNode, 6, 6
** ----------------------------------------------------------------
**
** STEP: StaticStep
**
*Step, name=StaticStep, nlgeom=NO
*Static
1., 1., 1e-05, 1.
**
** LOADS
**
** Name: PointForce  Type: Concentrated force
*Cload
FreeEnd, 2, -10000.
**
** OUTPUT REQUESTS
**
*Restart, write, frequency=0
**
** FIELD OUTPUT: F-Output-1
**
*Output, field, variable=PRESELECT
**
** HISTORY OUTPUT: H-Output-1
**
*Output, history, variable=PRESELECT
*End Step
```

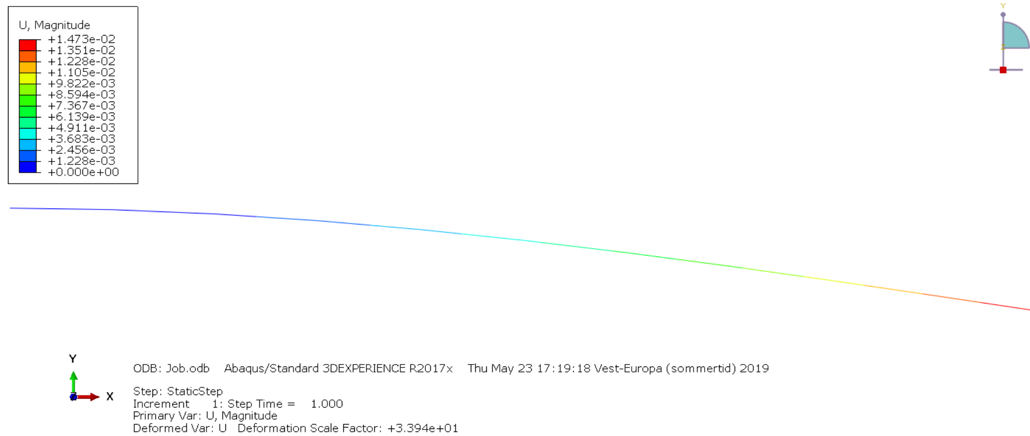


Figure 2.4: Deflection of a simple beam cantilever analyzed by Abaqus/CAE

2.3.2 Nastran's input file format for FEA

Nastran is a general-purpose structural analyzer initially developed by MSC Software and NASA, and it has developed into an industry-standard tool for Computer-Aided Engineering (CAE) [35]. The Nastran input file requires three sections: The *Executive Control Section* describing statements that select a solution sequence and various diagnostics; the *Case Control Section* which selects loads and constraints, requests printing, plotting and/or punching of input and output data, and defines the subcase structure for the analysis; and lastly the *Bulk Data Section*, which specifies the model geometry, element connectivity, element and material properties, constrains, and loads. It has two optional sections as well: The *Nastran Statement* and the *File Management Section*. Their respective tasks is to specify values for certain Executive System operational parameters, and to attach and initialize Database sets and FORTRAN files [36]. The Nastran input file organization is illustrated in Figure 2.5.

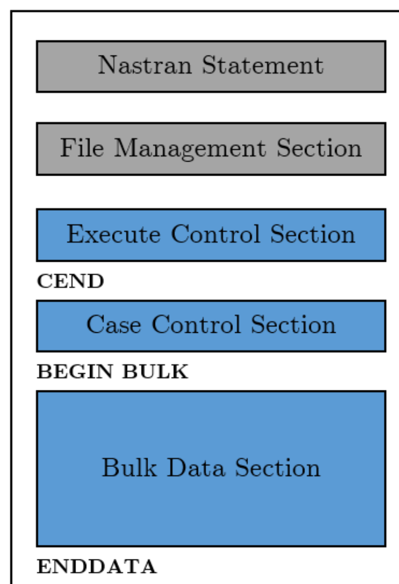


Figure 2.5: Nastran input file organization

The Nastran manual[36] states that a Nastran bulk data entry can be regarded as a row with fields containing input data. Each data entry consists of ten fields, the first of which contains the name of the Bulk data entry. The last field is used for optional continuation information. Fields in between contain the input data for the bulk data entry. A bulk data entry can be given in one of three formats: Free Field Format, Small Field format, or Large Field Format. The data entries must be one of the following three types:

- **Integer**. Integer data entries are numerical values that cannot contain a decimal point.
- **Real**. Real data entries must contain a decimal point. They can be formulated on both decimal form (e.g. 9.0) and scientific form (e.g. 9.E+0).
- **Character**. Character data entries can be alphanumeric, but should begin with an alpha character. The Nastran user manual states that legal characters are the English ones: A-Z and a-z, while the numeric characters are 0-9. Use of characters with special meaning in Nastran, such as $, &, *, =, comma (,) etc. should not be used, in order to avoid ambiguities for the Nastran bulk data interpreter.

Nastran data entries are identified by numerical values, unique to their class, called Identifiers [36]. Among the most common Identifiers are identification numbers for Grids (ID), elements (EID), properties (PID), materials (MID), and coordinate systems (CID). All identification numbers must be grater than zero. These identification numbers are essential, because they provide the means of unambiguously recognizing and accessing a particular data entry.

In **Free Field Format**, data fields are separated by commas (optionally blanks) and must start on column 1. The data fields can contain maximum eight characters of information, thus integer of character fields with more than eight characters will cause a fatal error. Real numbers, on the other hand, will be rounded off to eight characters if the original content surpasses the eight character limit. The Free Field Format denies embedded blanks (i.e. whitespace characters within an entry). Hence, two or more succeeding commas must be used to skip one or more fields [36].

The **Small Field Format** separates a Bulk Data entry into ten equal eight-character fields. Alike the Free Field Format, integer and character entries with more than eight characters will cause a fatal error, and real number entries are rounded off to eight characters, losing some precision. Another commonality is that embedded blanks are denied in Small Field Format as well [36].

For high-accuracy purposes the **Large Field Format** must be used. This format spans at least two lines (rows) for each entry, and the added fields are used to increase the number of significant digits. To make multiline data entries, continuations are required to denote that the subsequent line is a continuation of the previous one. This is done by marking the parent entry (the first line) with a continuation entry in the last field, and the subsequent line begins with a continuation entry. These two entries tell the Nastran bulk data interpreter that the second line should be interpreted as a continuation of the first [36]. The following example is reproduced from the Nastran Quick reference guide and illustrates how continuation entries are denoted for a Large Field Format of the PBAR simple beam property entry.

Table 2.1: Continuation Example of a PBAR entry (reproduced from the Nastran Quick guide [36]). The upper table is the template for a PBAR entry and continuation example is presented in the lower table.

PBAR	PID	MID	A	I1	I2	J	NSM		
	C1	C2	D1	D2	E1	E2.	F1	F2	
	K1	K2	I12						

PBAR	39	6	2.9	1.86	2.92	.48			+PB1
+PB1	0.	0.	0.	1.	1.	1.	1.	0.	+PB2
+PB2	.86	.86							

In Nastran, **comments** are denoted by a dollar sign ($) and may appear anywhere within the input file. All line content between a comment sign and the end of the line is considered a comment and will be ignored by the program [36].

Example of a simple beam model in Nastran Bulk Data

For this example, all information regarding commands and definitions are collected from chapter 5 in MSC. Nastran Quick Reference Guide [36]. In Nastran a nodal point is named a grid point, called with the GRID command. A grid point is defined as follows[4] (fields with an asterisk (*) are required; the others are optional):

1	2	3	4	5	6	7	8	9	10
GRID*	ID*	CP	X_1*	X_2*	X_3*	CD	PS	SEID	

Where the fields contain:

- **ID**: Grid point identification number
- **CP**: Identification number of the coordinate system in which the location of the grid point is defined.
- **X_1, X_2, X_3**: Location of the grid point coordinate system CP.
- **CD**: Identification number of coordinate system in which the displacements, DOFs, constraints, and solution vectors are defined at the grid point.
- **PS**: Permanent single-point constraints associated with grid point (any of the digits 1-6 with no embedded blanks).
- **SEID**: Super element identification number.

A beam element can be defined in NASTRAN using the CBEAM command:

[4] See the Nastran manual [36] for more detail.

1	2	3	4	5	6	7	8	9	10
CBEAM*	EID*	PID*	GA*	GB*	G0/X_1	X_2	X_3		

Where the fields contain:

- **EID**: Element identification number.
- **PID**: Property identification number of a PBEAM, PCOMP or PBEAML entry.
- **GA, GB**: Grid point identification numbers of connection points.
- **X_1, X_2, X_3**: Components of vector, from GA, in the displacement coordinate system at GA.
- **G0**: Grid point identification number to optionally supply X1, X2, and X3. Direction of orientation vector is GA to G0

So a beam can be modelled in NASTRAN by defining two grid points and defining a beam element between them. Listing 2.14 shows an example of NASTRAN bulk data for two grid points and a beam element. For simplicity, this example will assume that a PBEAM property with PID = 1 is defined, and it does not show the rest of the NASTRAN input file.

Code listing 2.14: A simple beam element spanned by grid points in NASTRAN bulk data format

```
$ Grid point definitions
$ GRID    ID    CP    X_1    X_2   X_3    CD    PS    SEID    $$
GRID      1     $     0      0     0      $     $     $
GRID      2     $     1      1     0      $     $     $

$ Beam element definition
$ CBEAM   EID   PID   GA     GB     X_1    X_2    X_3    $$    $$
CBEAM     1     1     1      2      $      $      $
```

2.3.3 USFOS and the UFO structural file format

USFOS is a leading computer program for nonlinear static and dynamic analysis of space frame structures used for integrity assessment, collapse analyses and accidental load analyses. Its software is developed by SINTEF and the Norwegian University of Science and Technology (NTNU), and is maintained and supported by USFOS A/S, a spin-off from the R&D institutions at SINTEF [37]. USFOS has its own structural file format called UFO (User-firendly structural file FOrmat) used in connection with modelling of framed structures [9]. UFO is of interest in this project because Haugen's Cfem input file format is closely related to it. In turn, the format proposed in this project (Chapter 3) is based on Haugen's Cfem format.

USFOS is used by leading oil companies and engineering consultants all over the world, and has particular strengths for use in integrity assessment of structures. It is used for a wide range of analyses, from accidental loads to design and reassessment of structures in offshore and civil engineering applications [37].

The USFOS Input Description [9] states that the input is organized in records, each starting with an identifier of four to eight characters. The records consist of integer or real data and may be given in an arbitrary order. A record may consist of several lines of data, each line being at most 132 characters long. The contents of this section is based on the USFOS Input Description [9] and the user's manual for the USFOS graphical user interface [38].

UFO structural file format

UFO is designed for high usability in connection with modelling of framed structures. The input may be given in one file or distributed among several. All data referencing is done via user defined external numbers (IDs), hence there are no internal numbers assigned to the data items (data records). The UFO file(s) must contain one, and only one, HEAD record identifying the analysis. Further, it contains nodal data, element data, cross-sectional data, material data, and/or miscellaneous data. These groups of data is comprised by a number of data types, where each type must have **i)** a definition and **ii)** fields for assigning optional properties [9].

Every data record has at least one mandatory parameter, and may have one or more optional parameters. The mandatory field(s) must all be defined for the record to be valid. The optional parameters, on the other hand, may be omitted. For omitted optional parameters, the USFOS software follows a set of rules for assigned default values to those parameters. The rules for assigning default values to omitted fields are specific to each data record. They can be found in the documentation of each data record in chapter 6.5 in the USFOS user manual [9]. Detailed descriptions of data types are not provided in this thesis, as UFO is extensive. The reader is referred to the USFOS user manual [9] for format details.

UFO format details and syntax

The UFO data items are separated by one or several whitespace characters [9]. Since the whitespace character is used for item separation, values have to be specified explicitly - they can (generally) not be left blank. For instance, in Nastran it is allowed to leave some (non-required) fields blank when using the Large Field Format and the Nastran Bulk data interpreter will still process the file without error [36]. This is not allowed in USFOS because of the fact that the whitespace character is used as a delimiter. However, there is an important exception to this rule: If all succeeding fields also are left blank, then the USFOS software will succeed in processing the data. The reason for this is that there is no more line content to separate [9]. Code listing 2.15 illustrates this limitation. Another evident consequence of using whitespaces for item separation is that all digits, letters and special symbols in a data item must be given consequently without blanks.

Code listing 2.15: Example of legal and illegal use of the whitespace character is USFOS

NODE record definition:
```
NODE NodeID x y z ix iy iz irx iry irz
```

USFOS data item with illegal embedded whitespace character to denote that `ix` **is omitted:**
```
NODE NodeID x y z    iy iz irx iry irz
```

USFOS data item with legally omitted fields at the end of the line:
```
NODE NodeID x y z
```

Important UFO syntax, collected from the USFOS User's Manual [9], is summarized in the following bullet points:

- **Comments** are denoted by a comment sign and can appear anywhere in the input file. All line content that occurs after a comment sign is ignored by USFOS, so inline comments are allowed. USFOS treats the following characters as comment signs: `'` `*` `#` `%` `!`
- **Alphanumeric data items** consist of at least one character. The first character must be a letter, while the remaining ones can be letters, digits or special symbols. This is the only limitation to the number of characters in an alphanumeric data item.
- **Integer** data items must be digits and the first digit may be preceded by a plus (+) or minus (-) sign.
- **Real number** data items consist of up to three components: An integer part, a decimal part and an exponent part.
- **Text strings** consist of one or more alphanumeric characters. Allowed alpha characters are A-Z and a-z, and allowed digits are 0-9. They may also have special symbols, with some exceptions because of their special functions of interpretation. The excepted characters are: `/` `$` `&`, and *blank* (whitespace character).
- Simple numerical operations on the input are supported by the input reader. Allowed operations are addition, subtraction, multiplication, division, and trigonometric calculations.

Example of a simple beam formulated in UFO

In UFO, nodal points are defined by the NODE command, and a beam between two nodes can be defined by the BEAM command [9].

The NODE command is defined as follows (fields with an asterisk (*) are required; the others are optional):

NODE* NodeID* x* y* z* ix iy iz irx iry irz

Where the parameters represent

- **NodeID**: User defined (external) node number.
- **x, y, z**: X, Y and Z coordinate of the node.
- **ix**: Boundary condition code for X-direction of the actual coordinate system used at the node. 0:Free, 1:Fixed.
- **iy, iz, irx, iry, irz**: Similar boundary conditions for the remaining 2 translational degrees of freedom and 3 rotational degrees of freedom.

The BEAM command is defined as:

BEAM* ElemID* node1* node2* material* geom* L_coor Ecc1 Ecc2

Where the parameters represent

- **ElemID**: User defined (external) element number
- **node1, node2**: Nodes 1 and 2 of the beam is connected to the NodeID of node1 and node2 respectively.
- **Material**: User defined material number defining the material properties of the element.
- **Geom**: User defined geometry number defining the geometry of the element.
- **L_coor**: User defined unit vector defining the local coordinate system of the element. If omitted, a default local coordinate system will be used.
- **Ecc1, Ecc2**: Nodes 1 and 2 of the beam have an eccentricity defined by Ecc1 and Ecc2 respectively.

Assume, for simplicity, that a material with MatID = 1 and a geometry with GeoID = 1 are defined. Then a simple beam can be modelled in UFO as illustrated in Listing 2.16.

Code listing 2.16: A simple beam element spanned by nodal points in UFO

```
#       id  x  y  z  ix  iy  iz  irx  iry  irz
NODE    1   0  0  0  1   0   1   0    0    0
NODE    2   1  1  0  1   1   1   0    0    0

#       ElemID  node1  node2  material  geom  l_coor  Ecc1  Ecc2
BEAM    1       1      2      1         1
```

2.3.4 OOCfem and the CFEM input file format

OOCfem is a simulation software for FEA applications. It was originally not object oriented and its name was Cfem. Cfem originated from the work in Bjørn Haugen's ph.d. thesis [11], developed into OOCfem, and is now in use industrially by Focus Software AS.

OOCfem is used in Focus Konstruksjon [12], a software for analysis based of the finite element method developed by Focus Software AS. Focus Konstruksjon is used to model simple and complex structures in steel, wood, and/or concrete and analyze them in accordance with the Eurocode [39]. A structural model in Focus Konstruksjon is comprised of beams, rods, shells, load cases combinations and more. The solver (OOCfem) performs a wide range of static analyses with foundation in linear and non-linear theory, buckling theory etc. [39].

Both UFO and the Cfem input file format are supported by OOCfem [40]. Cfem input file format is designed by Bjørn Haugen for use in OOCfem. It is closely related to USFOS' UFO input files and can represent data for nodes, materials, elements, cross-sections, load situations,

commands for solver algorithms, and output control [41]. The Cfem input file format is currently a subset of UFO, with some alternative commands, but Cfem aims at being an extension of UFO adjusted to cooperate well with OOCfem[5]. The proposed YAML based input file format for FEA presented in chapter 3 is based on the Cfem format.

The power of Focus Konstruksjon with OOCfem and corresponding input file formats is illustrated in the following examples.

Example 1: Warehouse

This example shows a model of a warehouse structure in steel. Key elements in the model are nodes, beams, elastic materials (steel), unit vectors, master/slave couplings among nodes, and load combinations of concentrated loads. The analysis specified in the input file is a linear static analysis and the solver is OOCfem's Eigen solver. The model is visualized in Figure 2.6. The corresponding input file is delivered alongside this report, named *Lagerhall_Rasmussen.fem*[6].

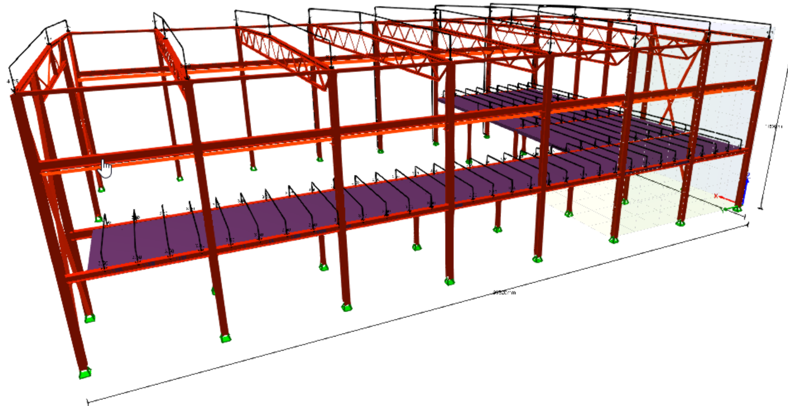


Figure 2.6: Warehouse modelled in Focus Konstruksjon

Example 2: Ski jump

The model presented in this example is a ski jump. The entire structure consists of nodes, beam cross-sections, beams, concentrated loads combined in load combinations, unit vectors denoting beam orientations, and elastic materials. A linear static analysis based on the load combinations is conducted with the OOCfem Eigen solver, testing for singularities. The model is visualized in Figures 2.7 to 2.9 and the corresponding input file is delivered alongside this report[6] (*Skibakke.fem*).

[5]Reference: Associate prof. Bjørn Haugen, supervision meeting, 04.06.2019.

[6]Due to the length of the input file, it is not included in Appendices. The file is found in the .zip file delivered alongside the thesis.

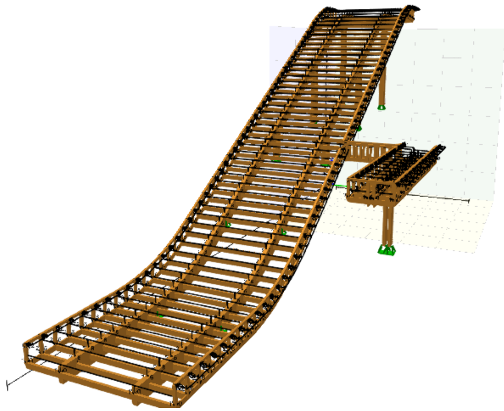


Figure 2.7: Ski jump modelled in Focus Konstruksjon

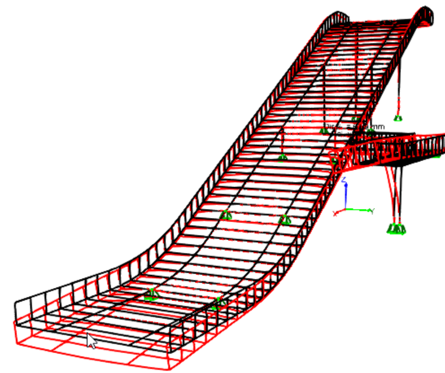


Figure 2.8: Visualization of deflection

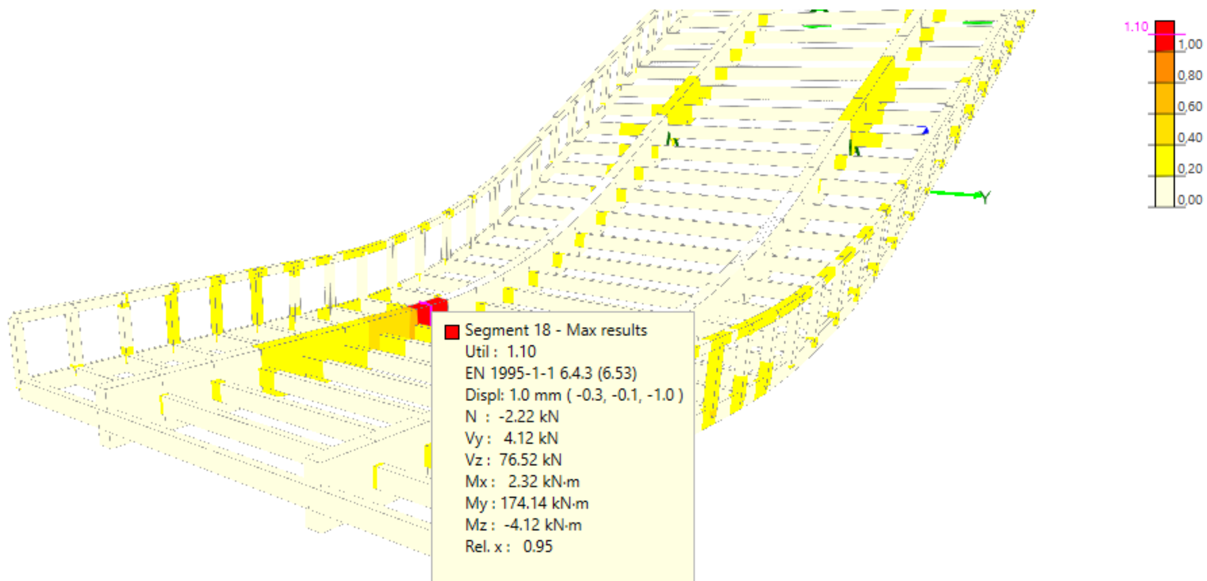


Figure 2.9: Assessment of parameters wrt. the Eurocode

Figures 2.7 and 2.8 show standard features that are provided in most FEA software. The first simply displays the model, while the latter shows the resulting deflection when subject to the load situation defined in the input file. Lastly, Figure 2.9 presents a feature for assessing whether the structure, when subject to the specified load case, satisfies the requirements set in the Eurocode [42]. The moment in the red beam element exceeds the Eurocode limit by a factor 1.10. This feature is very useful in connection with structural design because the Eurocode specifies critical requirements for structures built in Europe.

Chapter 3

Proposed solution

This chapter will present the proposed YAML based input file format for FEA and the prototype parser for that format, implemented in C++. Section 3.1 presents the structure, data types and syntax of the proposed format, while section 3.2 shows key aspects of the implementation and program flow for the prototype parser.

3.1 Proposed YAML based input file format for FEA

3.1.1 Overview

The proposed YAML based input file format for FEA aims to define the FEM data records used to compose expressive, unambiguous FEM models. It is built around the same native data structures as YAML and adheres to the YAML 1.2 syntax. Thus, the input file character stream can be parsed by any parser capable of performing a YAML load operation. Moreover, it can be presented in any YAML presentation style making the format versatile with regard to data presentation.

The proposed format arranges the FEM data in a hierarchical, nested structure. The top-level entity of the structure is a YAML mapping identified by the the literal string STRUCTURE as its key. The corresponding value is an arbitrarily long YAML sequence with one FEM data record in each entry. Hence, STRUCTURE is the container of all the FEM data comprised in the model. This mapping associates the FEM data records with the structure, analogous to the way a structural problem is formulated by a set of points, elements, materials, geometrical definitions, loads etc.

A FEM data record represents one FEM data type. It specifies a small part of the total structure, for example an element, a material or a load situation. A FEM data record is composed of a FEM keyword and a set of attributes, and its content depends on its type. For instance, an element contains data describing orientational, geometrical, and material properties. The highlevel relationship between FEM data nodes and the STRUCTURE node (in the YAML representation every data structure is represented by a YAML node) is visualized in Figure 3.1.

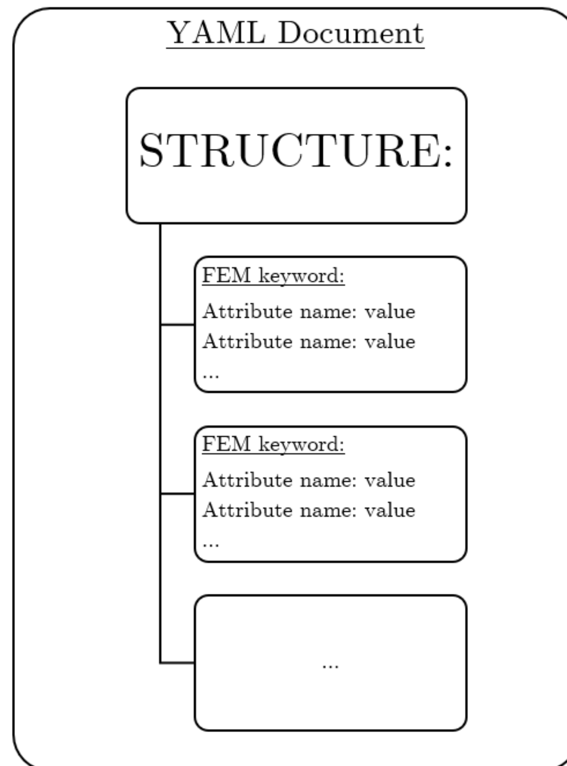


Figure 3.1: Highlevel structure of the hierarchy in the proposed YAML based format for FEA.

3.1.2 Detailed description

The proposed format aims to define a FEM model in an unambiguous, human readable way as a structure adhering to the restrictions imposed by the YAML language. By the fact that it has YAML as a basis, its data is structured in mappings, sequences and scalars. Additionally, it can be presented in two styles: YAML block style and YAML flow style. It is also possible to construct files that present the data in some arbitrary combination of the two, given that the YAML restrictions are satisfied.

The proposed format is a hierarchical, nested structure. The root of every FEM model expressed in the format is the STRUCTURE node. This node is the top level representation of the model, constituted by a number of FEM data records. The combination of these data records holds all the FEM data comprised in the model.

The STRUCTURE node

This is the root of the FEM model and, when parsed, it is the only node reached directly from the YAML root node (i.e. the node that represents the YAML document). This node is a mapping with the string literal ”STRUCTURE” as its key, and a YAML sequence as its value. Each entry in the sequence is a FEM data record. The anatomy of the STRUCTURE node is shown in Code listing 3.1. Documentation of the FEM data types in the proposed format can be found in Appendix A.

Code listing 3.1: Anatomy of the STRUCTURE node, presented in YAML block style.

```
STRUCTURE: #Mapping key
  #Begin sequence of data records (Mapping value)
  - FEM data record
    # Attributes
  - FEM data record
    # Attributes
  - FEM data record
    # Attributes
# - More elements...
  #End sequence of data records (Mapping value)
```

The FEM data records

A FEM data record consists of a keyword identifying the FEM data type of the record and several unique attributes. The attributes are composed by a value that is mapped to an attribute name. Attribute names are text strings describing what the value represents, and must be unique within each record. The value related to the attribute name holds the actual attribute value. The content of the value field can be any of the native YAML data structures (scalars, sequences or mappings). Similarly to the attribute names, the keyword identifiers are character strings holding a name denoting the FEM data type of the record.

The data for each FEM record is held in a nested structure composed of two mappings. The inner mapping contains the record's attributes, with the attribute names as keys and the attribute values as (mapping) values. These attributes are wrapped by the value part of the outer mapping and associated with the FEM keyword for the record. This is illustrated in Code listing 3.2. The structure relating the data records to the STRUCTURE node is shown in Figure 3.1.

Code listing 3.2: Generic FEM data record

```
FEM_KEYWORD:                #Record type identifier
     Attribute name: value  #Unique attribute name with mapped value
     Attribute name: value  #Unique attribute name with mapped value
#    ...
```

Every FEM data record has one or more **mandatory attributes**. The mandatory attributes represent the minimal set of data fields required to properly define the record. Hence, if one or more mandatory attributes are undefined or erroneously defined (i.e. assigned an illegal type and/or value) the data record is faulty.

In addition to the mandatory attributes, a FEM data record has zero or more **optional attributes**. These attributes specifies data that is not required to define the record, but carry extra information. For instance, a NODE record only requires attributes to specify the ID and its location in terms of x, y and z coordinates. However, the data record may also specify translational and rotational boundary conditions (BCs) as well as a reference to a local coordinate system with respect to which the boundary conditions apply. If these optional attributes are omitted, the NODE is considered free in all six degrees of freedom (DoFs) applied with respect to a global coordinate system. In contrast to UFO and Cfem, there is no restriction on the combination of which optional attributes may be omitted. Both UFO and Cfem require that when an attribute

is omitted, all succeeding attributes must also be omitted. The input file format proposed here does not impose this restriction.

The course of action to take when parsing a FEM data record with omitted attributes varies between the FEM data types. The format documentation (Appendix A) specifies the correct/-expected action for the individual FEM data types and attributes.

Code listing 3.3: Example of a NODE record with only mandatory attributes specified.

```
NODE:
  #Mandatory attributes
  id: 1
  x: 0.0
  y: 1.0
  z: 0.0
```

Code listing 3.4: Example of a fully specified NODE record.

```
NODE:
  #Mandatory attributes
  id: 2
  x: 1.0
  y: 0.0
  z: 0.0
  #Optional attributes
  ix: 1
  iy: 1
  iz: 1
  irx: 1
  iry: 1
  irz: 0
  rotID: 2
```

Code listings 3.3 and 3.4 both define valid NODE records. Code listing 3.3 shows a node defined with the minimal set of attributes required, while Code listing 3.4 specifies all the optional ones as well.

Upon parsing of the FEM data record specified in Code listing 3.3 the boundary conditions are assigned to zero, meaning that the node is free in all DoFs. Even though it might be considered superfluous to associate a coordinate system with only free DoFs, the parser should assign the global reference frame (coordinate system). The reason for this recommendation is that after the character stream is parsed, any operation could be performed on the data. If, for instance, a boundary code is assigned, leaving the node fixed in one of the DoFs, the reference frame for that BC is known to the application ensuring that the NODE record is still valid.

Since the NODE record presented in Code listing 3.4 is fully specified, there are no additional values to assign upon parsing. The data record is properly defined provided that the input file also contains a data record defining a local coordinate system with rotID equal to two. Observe that the record explicitly states that the node is free to rotate about the Z-axis (`irz` is assigned to zero). This attribute could have been omitted, in which case its value should be assigned to zero by the parser software. Nonetheless, such an explicit specification could enhance readability and clarify that the node is indeed free with respect to this DoF to an audience that is unfamiliar with the format documentation.

Data hierarchy and dependency levels

The YAML representation of the FEM model is a hierarchical structure, as described in section 3.1.1. This structure is sensible with regards to the natural hierarchical structure of data in the finite element method. Consider, for example, a beam element. It can be modelled from information about start and end points (nodes), material data, and a cross-sectional geometry.

Consequently, the beam element could be considered above nodes, materials, and cross-sections in the FEM hierarchy, because these FEM types carry information required to define the beam.

Despite its hierarchical structure, the proposed format does not impose a specific order on the FEM data records. Rather, the format requires the presence of all the data necessary to define each FEM data record, but leaves the task of ordering the data to the parser software. However, every FEM data record associated with the same structure must be a member of the sequence mapped to the STRUCTURE node.

A convenient way of managing this hierarchical structure is to introduce the concept of *dependency levels*. The dependency level of an FEM data type denotes its ordering in the FEM data hierarchy. In the proposed format, a dependency between two FEM data types is defined as follows: Consider two FEM data types T1 and T2. T1 is dependent on T2 if one of its attributes has a reference to an instance of T2. With this as basis, the dependency level of a data type is determined according to the following rules:

1. FEM data types that are independent of other FEM data types have dependency level *null*.
2. The dependency level of a FEM data type is equal to the highest dependency level of its attributes plus one.

Hence, data types that depend on one or more data types with dependency level null have dependency level *one*, data types that depend on one or more data types with dependency level one have dependency level *two* and so on. FEM data types on dependency level *null* have no references to other FEM data types; their attributes are exclusively comprised of numeric or alphanumeric attributes.

In the proposed format, a beam element is identified by the BEAM record[1]. By revisiting the beam element discussed above, it is seen that the highest dependency level among its attributes originates from the attributes referencing NODE records, who have dependency level *one* (because rotID in NODE attributes references an independent data record). Therefore, a BEAM record has dependency level equal to *two*.

Code listing 3.5: BEAM record with mandatory attributes

```
BEAM:
  elemID: int        # User defined external element number
  node1: int         # Reference to external node identifier
  node2: int         # Reference to external node identifier
  material: int      # Reference to external material identifier
  geoID: int         # Reference to external profile geometry identifier
```

Syntax

As the proposed format is a pure YAML format, it supports both presentation styles provided by YAML (block style and flow style). Naturally, the YAML syntax described in Section 2.2.1 governs the proposed format. Additional syntax for proposed format is summarized in the following bullet points:

[1] See Appendix A for documentation

- Each entry in the sequence associated with the STRUCTURE node contains exactly one FEM data record.
- Every FEM data record in the model must be a member of the STRUCTURE (sequence) node.
- Each element in the STRUCTURE (sequence) node is a mapping with the FEM keyword, represented as a string, as its key.
- Attributes are represented as mappings on the form attribute_name: value. The attribute name must be a string and the mapping value can be of any native YAML data type.
- The FEM data records can appear in any arbitrary order in the sequence associated with the STRUCTURE.
- In general, any combination of omitted and specified optional attributes is allowed. This rule is void if some optional attributes are dependent on other optional attributes.
- Record identifiers (IDs) should have a non-zero positive value to avoid collisions with default FEM data records.
- Default attributes have ID equal to 0 or -1.

The proposed format adheres to the YAML syntax for indentation, hence the attributes must be indented with at least one whitespace character more than the record's FEM keyword in the block style presentation.

YAML aims to be easily readable by humans, hence its syntax is rather liberal. This property allows a high degree of versatility in terms of presentation style. By using a hybrid between the YAML block and flow styles, attributes that typically operate in groups can be gathered in a compact presentation style. Some attributes, such as collections of nodes, coordinates, boundary conditions, materials, etc., occur in groups and could therefore be represented by a YAML collection rather than separate scalars. For example, a coordinate system is defined by three vectors specifying the X-, Y-, and Z-axes. Code listings 3.6 and 3.7 show two ways of presenting a COORDSYS record - the first with the components presented as separate scalar fields and the second with the vector components comprised in YAML sequences.

Code listing 3.6: COORDSYS specified with separate scalar fields

```
COORDYS:
  rotID: 1
  Xx: 1
  Xy: 0
  Xz: 0
  Yx: 0
  Yy: 1
  Yz: 0
  Zx: 0
  Zy: 0
  Zz: 1
```

Code listing 3.7: COORDSYS specified with vectors presented as sequences

```
COORDSYS:
  rotID: 1
  X: [1, 0, 0]
  Y: [0, 1, 0]
  Z: [0, 0, 1]
```

One could argue that the presentation in Code listing 3.7 is better in terms of mathematics, because it clearly groups the vector components in a way that is common in the mathematical field. Nonetheless, this is a matter of preference and the format supports both presentations.

Example

The example in Code listing 3.8 shows the simple beam cantilever presented in Abaqus in chapter 2.3.1, now formulated in the proposed format. The output requests from the Abaqus example have been omitted because there are currently no support for such functionality in the proposed format. YAML flow style is chosen for two reasons: **i)** to show the flow style for the proposed format, and **ii)** to limit the length of the example. The input file is delivered alongside this report (named *cantilever_example_flow.yaml*) and a corresponding YAML block format formulation can be found in *cantilever_example_block.yaml*.

The input files starts by defining the NODE records in the problem. Most optional attributes are omitted, and that their coordinates are grouped (into the `xyz` attribute) instead of being specified as separate attributes. Next, the BEAM records are specified, omitting all optional attributes since they are all default in this case. The steel material is specified in the ISOMATERIAL record, and the beam profile is specified as a T-section. Finally, a concentrated load applied to the rightmost NODE (with id = 11) is specified in the NODELOAD record, and a trivial load combination is defined to complete the loading situation. The example also utilizes that comments placed inside YAML collections are allowed.

Code listing 3.8: Simple beam cantilever formulated in the proposed format using YAML flow style.

```
#Example resembling Abaqus example
{STRUCTURE: [
  #Nodes
  {NODE: {id: 1, xyz: [0, 0, 0], ix: 1, iy: 1, irz: 1}},
  {NODE: {id: 2, xyz: [0.5, 0, 0]}},
  {NODE: {id: 3, xyz: [1, 0, 0]}},
  {NODE: {id: 4, xyz: [1.5, 0, 0]}},
  {NODE: {id: 5, xyz: [2, 0, 0]}},
  {NODE: {id: 6, xyz: [2.5, 0, 0]}},
  {NODE: {id: 7, xyz: [3, 0, 0]}},
  {NODE: {id: 8, xyz: [3.5, 0, 0]}},
  {NODE: {id: 9, xyz: [4, 0, 0]}},
  {NODE: {id: 10, xyz: [4.5, 0, 0]}},
  {NODE: {id: 11, xyz: [5, 0, 0]}},
  #Beam elements
  {BEAM: {elemID: 1, nodes: [1, 2], material: 1, geoID: 1}},
  {BEAM: {elemID: 2, nodes: [2, 3], material: 1, geoID: 1}},
  {BEAM: {elemID: 3, nodes: [3, 4], material: 1, geoID: 1}},
  {BEAM: {elemID: 4, nodes: [4, 5], material: 1, geoID: 1}},
  {BEAM: {elemID: 5, nodes: [5, 6], material: 1, geoID: 1}},
  {BEAM: {elemID: 6, nodes: [6, 7], material: 1, geoID: 1}},
  {BEAM: {elemID: 7, nodes: [7, 8], material: 1, geoID: 1}},
  {BEAM: {elemID: 8, nodes: [8, 9], material: 1, geoID: 1}},
  {BEAM: {elemID: 9, nodes: [9, 10], material: 1, geoID: 1}},
  {BEAM: {elemID: 10, nodes: [10, 11], material: 1, geoID: 1}},
  #Material
  {ISOMATERIAL: {matID: 1, type: elastic, Emod: 2.0e+11, poisson: 0.27}},
  #Cross-section
  {TSECTION: {geoID: 1, H: 0.3, T_web: 0.05, Width: 0.1, T_top: 0.05}},
  #Load data
  {NODELOAD: {loadCaseID: 1, nodeID: 11, fy: -10000}},
  {LOADCOMB: {loadCombID: 1, factors: {1: 1.0}}}
]}
```

3.2 Prototype parser for the proposed format

3.2.1 Overview

The prototype parser is an object-oriented software, written in C++ using the open-source library yaml-cpp to perform a YAML load operation. yaml-cpp parses the character stream of the input file into a graph of `YAML::Nodes` (yaml-cpp syntax denoting a YAML node) where each node corresponds to a FEM data record, with the exception of the STRUCTURE node. The STRUCTURE nodes wraps the data records, as described in chapter 3.1.2. When yaml-cpp has parsed the character stream, the prototype parser traverses the resulting graph and instantiates objects for the FEM data types populated with data from the graph. Each object is added to a corresponding container in the structure class, connecting them into one structure. To verify that the data is correctly parsed and all class instances are properly instantiated, the parser simply prints the attributes for all objects to the console after all the logic for parsing has been executed.

The high-level program flow of the prototype parser is shown in Figure 3.2. The figure shows three repeating tasks, one for each dependency level. The content of these tasks, for an arbitrary dependency level, is presented in Figure 3.3.

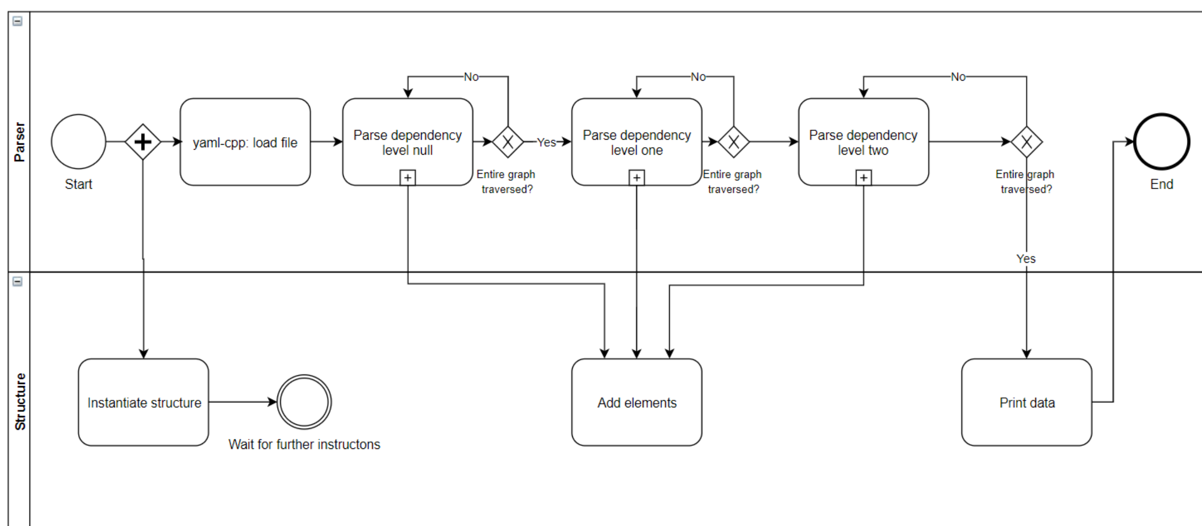


Figure 3.2: Highlevel program flow of the prototype parser

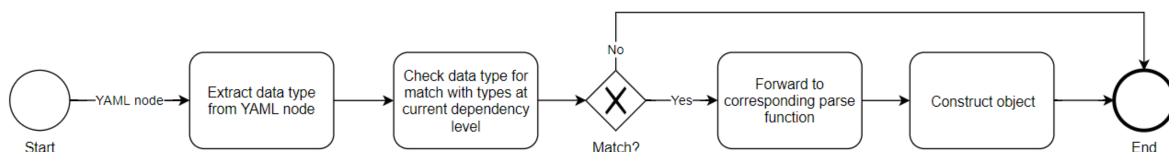


Figure 3.3: Generic highlevel program flow for the *Parse dependency level* tasks in Figure 3.2

Figure 3.2 states that an element is added in the structure for each iteration of the dependency level loops. This is a modelling simplification and is not what actually happens. Adding the element to the structure only happens if the FEM type in the YAML node actually matches

the dependency level, i.e. if the YAML node is forwarded to the parse function constructing a corresponding object. The actual behaviour is more precisely reflected in Figures 3.4 and 3.5.

In this project, Doxygen [13] was used to generate documentation. Doxygen can generate an interactive HTML page documenting classes, variables and functions, and it generates inheritance trees and include structures. For information beyond the scope of this chapter, the reader is referred to the Doxygen documentation.

3.2.2 Detailed description

The prototype parser is written using Microsoft Visual C++ 2017 (v141). It utilizes the third-party library yaml-cpp (described in chapter 2.2.3) to perform a YAML load operation on the character stream in the input file. The result of the load operation is a YAML representation (graph of `YAML::Nodes`), which is traversed by the prototype parser to obtain the native data types.

The Parser class

The Parser class is the core part of the prototype parser. This class is responsible for accessing the YAML character stream using yaml-cpp, traversing the resulting graph of `YAML::Nodes` (i.e. the YAML representation), and creating instances of the FEM data type classes.

The Parser is instantiated by a function call to its constructor, with the name of the YAML input file as argument. The constructor starts with parsing the input file using yaml-cpp's `YAML::LoadFile(filename)` function and stores the resulting graph by its root. Next, it accesses the STRUCTURE node from the root. This is the entry point for the process of traversing the YAML representation. Lastly, it instantiates an object of the Structure class to which the FEM data instances will be assigned. When the constructor is finished, the process of traversing the YAML representation can begin.

The top-level function handling the traversal is `void Parser::Parse()`. This function ensures that the FEM data objects are instantiated in order of increasing dependency level by calling parse functions for each dependency level sequentially. The logic for actually calling the parse functions specific to each FEM data type is divided in intermediary functions, one for each dependency level.

The intermediary functions get their names from the dependency level they are responsible for parsing (`ParseDependencyLevelNull()`, `ParseDependencyLevelOne()` etc.). Each of these functions traverses the entire YAML representation once and checks whether the keyword in each node matches any of the FEM data types at its level of dependency. Whenever there is a match, they pass the node to the corresponding parse method specific to the FEM type. Code listing 3.9 presents the C++ code implemented in for parsing the FEM data records on dependency level null. The parse functions for the remaining dependency levels are implemented similarly.

Code listing 3.9: Implementation of `Parser::parseDependencyLevelNull()`

```
void Parser::parseDepenencyLevelNull() {
        for (int iterator = 0; iterator < structureNode.size(); ++iterator) {
                YAML::const_iterator it = structureNode[iterator].begin();
                std::string key = it->first.as<std::string>();

                if (key == "COORDSYS" || key == "NODTRANS" || key == "PCOORDSYS") {
                        nextNode = structureNode[iterator][key];
                        parseCoordSys(nextNode, key);
                }
                if (key == "ECCENT") {
                        nextNode = structureNode[iterator][key];
                        parseEccentricity(nextNode, key);
                }
                if (key == "ISOMATERIAL") {
                        nextNode = structureNode[iterator][key];
                        ParseIsoMaterial(nextNode, key);
                }
                if (key == "PIPE") {
                        nextNode = structureNode[iterator][key];
                        ParsePipe(nextNode, key);
                }
                if (key == "TSECTION") {
                        nextNode = structureNode[iterator][key];
                        parseTsection(nextNode, key);
                }
                if (key == "PLTHICK") {
                        nextNode = structureNode[iterator][key];
                        parsePLThick(nextNode, key);
                }
                if (key == "UNITVEC" || key == "ZVECTOR" || key == "YVECTOR") {
                        nextNode = structureNode[iterator][key];
                        parseVector(nextNode, key);
                }
        }
}
```

As seen in Code listing 3.9, the entire YAML representation is traversed and checked for keyword matches. Whenever there is a match, the corresponding data type parsing function (e.g. `parseCoordSys(nextNode, key)`) is called, forwarding the node (by reference) and the keyword. If the keyword does not match any of the data types associated with the dependency level, none of the condition checks are true, and the loop will proceed to the next iteration without calling any member function.

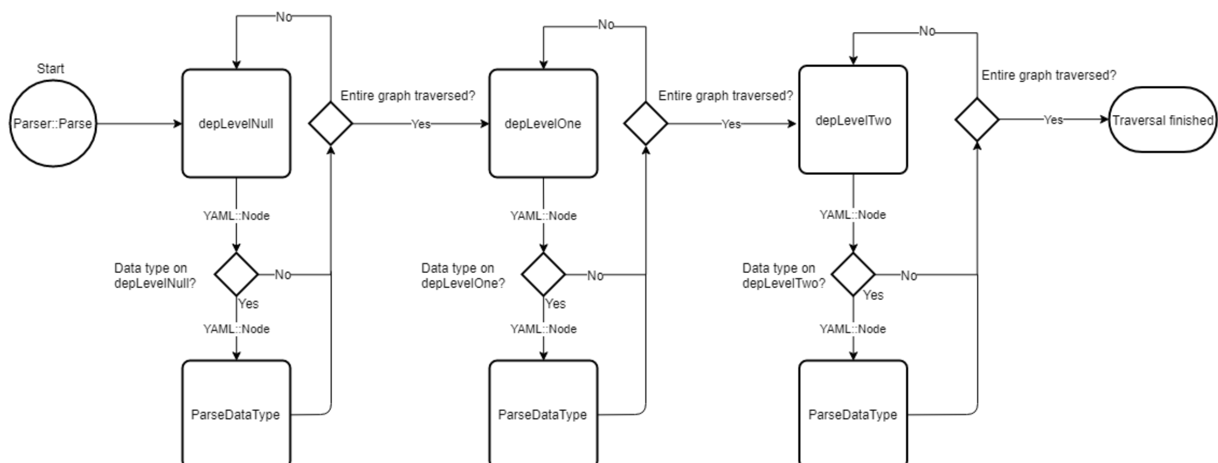


Figure 3.4: Highlevel flow chart of program flow in `Parser::Parse()`. The flow for *ParseDataType* is elaborated in Figure 3.5

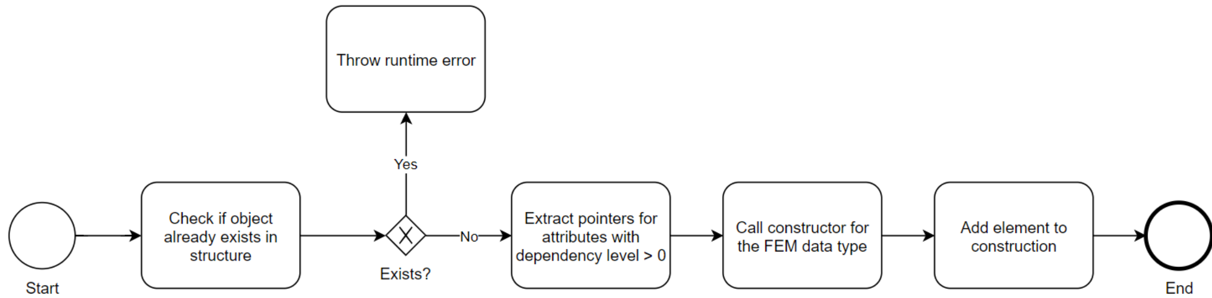


Figure 3.5: Program flow for a generic parse FEM data type function

Figure 3.4 shows a simplified flow chart of the program flow in the `Parser::Parse()` method. The abbreviations *depLevelNull*, *depLevelOne*, and *depLevelTwo* denotes the functions for parsing the corresponding dependency level. Moreover, *ParseDataType* represents the set of parse functions specific to each FEM data type at the current level of dependency. A flow chart for a generic parse FEM data type function is presented in Figure 3.5.

The Structure class

The Structure class represents the STRUCTURE node in the proposed format. It holds all the FEM data instances in the model, hence it is the top-level container of the FEM model. Since the proposed format insists that all FEM data records must be member of the STRUCTURE, there is an instance of this class during the runtime of the parser.

The FEM data objects are divided into separate groups in the Structure class. Each group has a list of `data_type*` (data type pointer) entries and a map of `std::string int, data_type*` pairs associated with it. The data type pointers point to the corresponding FEM data objects. The grouping is designed to match the `UsfosReader::input_data` grouping used in the USFOS input file reader (*input_usfos.h*) in Haugen's OOCFEM program [40] to ensure interoperability between this parser and OOCFEM. The groups included in the prototype parser are:

- Coordinate systems
- Cross-sections
- Composite sections
- Materials
- Nodes
- Elements
- Node loads
- Load combinations

The Structure class has functionality to perform three different operations on these groups: **i)** Check whether a specific object exists, **ii)** Fetch a specific object, and **iii)** Add a new object to a group. To fetch or check the existence of an object, the object is specified by its identification number (ID) and, in some cases, a type specification. The type specification is required when the ID is insufficient to unambiguously reference the object. For example, the group of elements contains several element types (FEBeam, FETrishell etc.). Functions for adding new object must have a pointer to the instance, passed as a function argument.

The exclusive or (XOR) gateway in Figure 3.5 involves a function call to one of the Structure member functions to check the existence of a FEM data object. Since the functions are associated with the grouping described above, this process consists of iterating over the map of `(std::string int, data_type*)` pairs, in search of an entry with ID equal to the function's input argument. If no such entry exists, the function returns false. Otherwise, it returns true. Code listing 3.10 shows the logic for checking the existence of a FEM node. The other groups have similar functionality, with one exception: `checkElementExistence(int id, std::string type)`. This exception is attended in the following.

Members of the FEM element group are not guaranteed to be unambiguously referenced by the ID alone. Therefore, a string literal denoting the FEM data type is used in combination with the ID to identify the object. Consider, for instance, a case where both a BEAM record and an eccentricity vector have ID equal to 1. This is possible, as the proposed format does not insist on unique IDs across these data types. In this case, an ID check is ambiguous. However, their IDs must be unique across the collection of records of equal types, so the combination of ID and type is able to identify the object correctly. This, however, raises a problem as C++ maps require unique keys. Hence, when adding a new data type with an ID that already exists in the map, the original entry will simply be overridden, leaving the map incomplete. It was chosen to iterate over the group's vector instead, as vectors does not have uniqueness requirements. The resulting logic for checking existence of a FEM object in the element list is shown in Code listing 3.11.

Code listing 3.10: Logic for checking the existence of a specific FEM node in the structure

```
bool Structure::checkNodeExistence(int id) {
        if (nodeMap.find(id) != nodeMap.end()) {
                return true;
        }
        return false;
}
```

Code listing 3.11: Logic for checking the existence of a specific FEM element in the structure

```
bool Structure::checkElementExistence(int id, std::string type) {
        std::vector<GenericFE*>::iterator it;
        for (it = elementList.begin(); it != elementList.end(); ++it) {
                if ((*it)->getID() == id && (*it)->getTypeAsString() == type) {
                        return true;
                }
        }
        return false;
}
```

In the constructor of the Structure class, a set of default data objects is instantiated. These objects are used whenever an optional attribute with dependency level one or higher is omitted, because these attributes are references to other data instances. For example, the `BEAM` record has the option of specifying two eccentricity vectors. If these attributes are omitted, the parser assigns references to a default eccentricity vector (zero eccentricity). The default data objects have ID equal to zero.

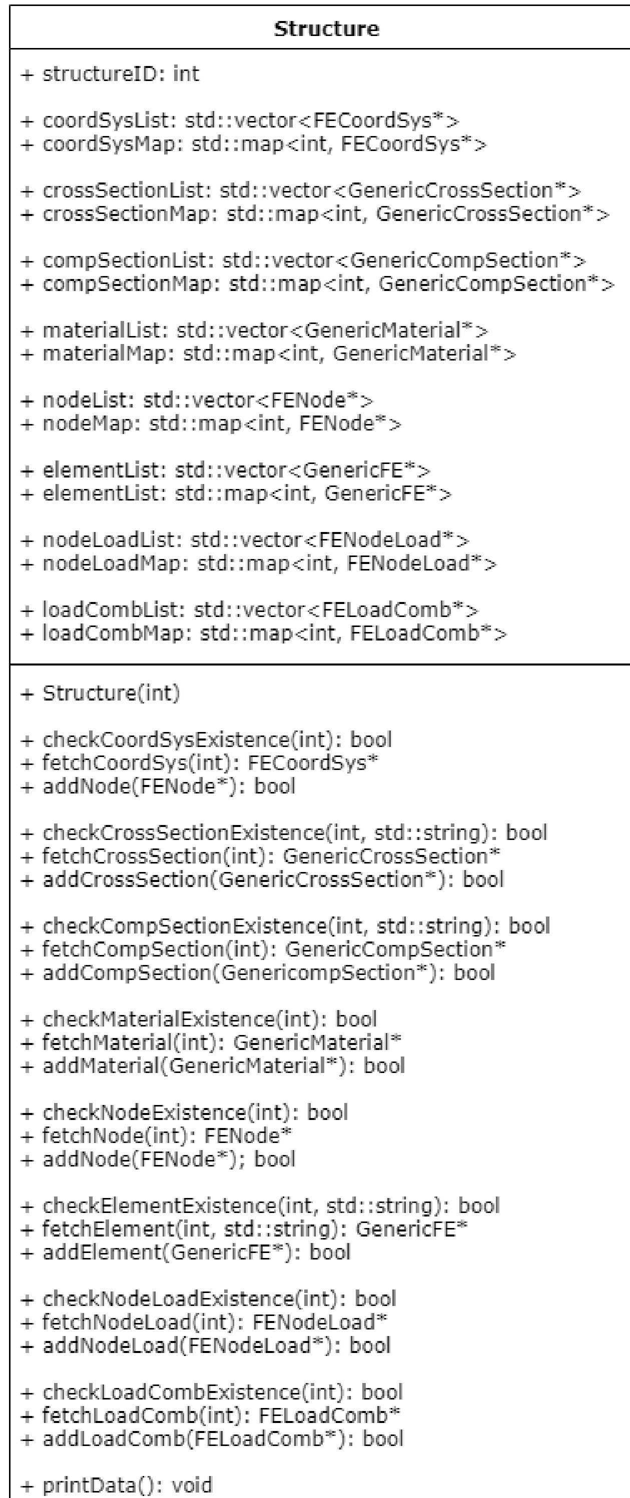


Figure 3.6: UML diagram of the Structure class

The FEM data type classes

The prototype parser implements the FEM data types in separate classes. Each class represents a FEM data structure, and most of the classes are associated with one FEM data record in the

proposed format. Yet, some FEM data type classes are associated with several records. An example is FECoordSys, which can be instantiated with data from one of the COORDSYS, NODTRANS, or PCOORDSYS records.

The classes generally contain identifiers (IDs) used to unambiguously identify the object. The FEM data records in the proposed format generally have an external explicitly user-defined identifier specified as a mandatory attribute. An assessment of identifier uniqueness must be performed upon parsing. Not all records in the proposed format have user-defined IDs. In these cases the prototype parser assigns an internal, usually autogenerated, identification number for the object. An example of such a record is the NODELOAD data type, for which the parser simply assigns an ID equal to 1 for the first record and increments by 1 for each succeeding NODELOAD record in the input file.

The proposed format structures the FEM data in records distinguished by keywords, where the data records have a specific set of attributes. When the prototype parser has classified a FEM data record according to its keyword, it can instantiate an object of the corresponding class. The attributes in the data record populates the instance.

The mandatory attributes constitute the minimal set of attributes required. Upon parsing of a `YAML::Node` from the YAML representation graph, the presence and validity of the mandatory attributes is assessed; a FEM data type object is instantiated if and only if all mandatory attributes are valid. Whenever one or more mandatory attributes are missing, the prototype parser program throws a runtime error printing an error message to the C++ console and to a log file. The program does not crash even though the FEM model may be corrupted, so it is very important that the user attends to these error messages.

A FEM data type object is not only populated by the mandatory values - the optional attribute values are also assigned. If the optional attributes are specified in the input file, the parser program simply assigns these values to the data type object on instantiating, provided that their data is valid. Omitted optional attributes are assigned default values according to the format documentation (Appendix A). An example of optional attribute assignment for a NODELOAD record is presented in Code listing 3.12. The value -1 for `eccID` is the identification number of the default eccentricity vector denoting zero eccentricity.

Code listing 3.12: Prototype parser assignment of optional attributes for a NODELOAD record

Data record specified with omitted fields

```
NODELOAD:
  loadCaseId: 1   #Mandatory
  nodeID: 3       #Mandatory
  fx: 4000.0      #Optional
# ...             Remaining optional attributes are omitted
```

Resulting (equal) data record after parsing

```
NODELOAD:
  loadCaseId: 1   #Specified in input file
  nodeID: 3       #Specified in input file
  fx: 4000.0      #Specified in input file
  fy: 0.0         #Assigned by parser program
  fz: 0.0         #Assigned by parser program
  mx: 0.0         #Assigned by parser program
  my: 0.0         #Assigned by parser program
  mz: 0.0         #Assigned by parser program
  eccID: -1       #Assigned by parser program
```

Class hierarchy

The FEM objects are divided in categories according to their type. The categories are represented by abstract classes providing the base type and functionality for the FEM data type classes. Every FEM data type class with an ID attribute for referencing is a subclass of *Identifiable*. This class is at the top of the hierarchy and represents an absolute basis. Identifiable contains one integer field (`id`) holding the reference number, and one enumerated type (enum) variable named `Type`. The enum variable contains a set of types for the FEM objects (e.g. NODE, COORDSYS, or NODELOAD) and each subclass of Identifiable has one of these types. The member functions of Identifiable is a getter and a setter for the identifier, and a getter for the enum value. The `id` and its setter function has protected visibility level, thus they are visible only to subclasses of Identifiable. Fields and functions with public visibility are `type` (instance of the `Type` enum) and `getType()`.

In addition to the top level category, several intermediary classes are also abstract. These classes represent a subcategory and inherit members from Identifiable, while introducing some extra functionality such as logic for assigning variable values from YAML::Nodes. The names of intermediary classes in the prototype parser begin with "Generic" (e.g. GenericCrossSection). Most functions in Identifiable are pure virtual functions, meaning that must be implemented for each subclass. The classes representing actual FEM data types are at the bottom level of this inheritance hierarchy - they are the leaves in the inheritance tree structure. The intermediary classes are mainly used for type management in connection with containers. The Structure class has containers for the FEM elements and these containers have the intermediary data types as their headers. An example is `materialList`, containing the materials defined in the structure. This container can hold a collection of any data type defined as a subclass of GenericMaterial.

As a last note, some data types are situated directly beneath Identifiable in the hierarchy. These classes have no intermediary class inducing its type on them. Figure 3.7 shows the inheritance diagram for Identifiable. Observe that the intermediary types have precedence between Identifiable (the root) and the leaves.

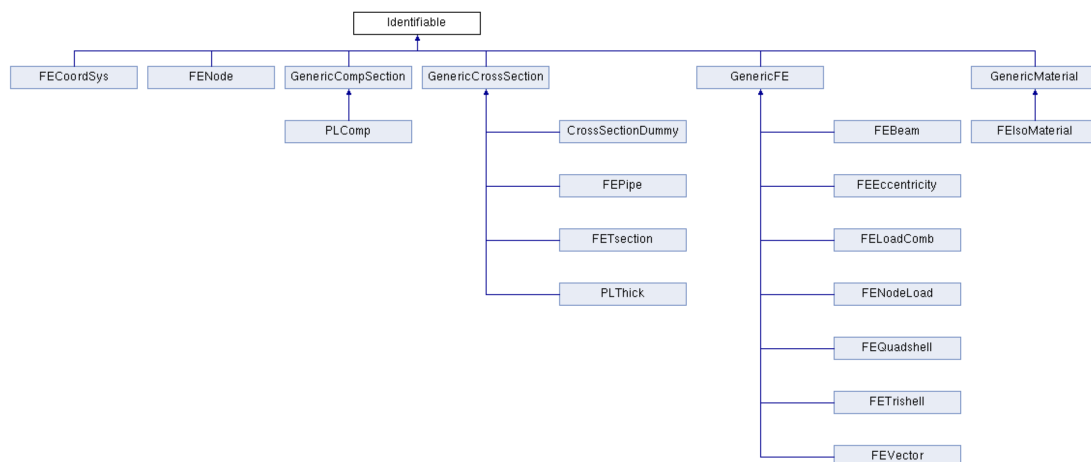


Figure 3.7: Inheritance diagram for Identifiable (Reproduced from the Doxygen documentation of YAMLParser)

The converter/emitter module

The main functionality of the prototype parser is, naturally, related to parsing FEM data types to construct an internal C++ representation that can be processed by a FEA solver software (e.g. OOCfem). However, YAMLParser has an additional module for converting UFO input files to the proposed YAML based format. The module was implemented in the pre-project for the masters thesis to generate example YAML files from UFO.

The converter module essentially tokenizes the UFO character stream into string tokens holding exactly one data record each. Then it utilizes the emitter functionality in yaml-cpp to perform a YAML dump operation of the tokens constructing a coinciding file adhering to the YAML syntax.

The module is comprised of three classes: The Writer, the FEAFormat, and the DataHolder. FEAFormat contains the templates for the FEM data records in the proposed format, represented by lists of attribute name strings. The templates are mapped to a string literal linking the template to the correct FEM data type. The string literal is checked by the Writer class to retrieve the correct template for each token. The DataHolder class reads the UFO file and creates an internal representation of the content (i.e. tokenizes the character stream). The tokens are stored in a two-dimensional vector (`tokens2D`), where one entry in the vector corresponds to one line. Each entry (line) is divided into string tokens, hence the second dimension of the vector. Code listing 3.13 illustrates how the two-dimensional vector is populated.

Code listing 3.13: Populating `tokens2D`

```
tokens2D = [line_1, line_2, ... , line_n]

where
line_i = [token_1, token_2, ..., token_m]

such that
tokens2D = [[token_1, token_2, ..., token_m], ..., [token_1, token_2, ..., token_m]]
```

The Writer class is responsible for serializing the content of `tokens2D` by dumping it to a YAML file. The essential method in the Writer class is `writeToYAMLFile(std::string filename)` which opens a C++ output file stream, looping through `tokens2D`, and checking that the data can be converted into the proposed format. Finally it dumps the resulting YAML character stream to file using the output file stream.

The converter module is not up to date with the proposed format. Several data types are not supported by the converter and the ability to convert sequences of arbitrary length is not pursued in this project, due to scope limitation. This is further discussed in chapters 4.2 and 6.

Chapter 4

Evaluation and discussion

This project aims at:

I. defining a good YAML based input file format for finite element analysis of structural problems, and

II. implementing a prototype parser for the format using C++ and available open-source tools.

This chapter will evaluate whether the project has been successful and to what extent. An assessment of advantages and disadvantages in both the proposed format and the prototype parser is included, and an attempt of addressing crucial issues is made. Moreover, the proposed format will be compared to the formats presented in chapter 2. This chapter is divided into two sections: section 4.1 will discuss the proposed YAML based input file format, while section 4.2 will address the prototype parser.

4.1 The proposed YAML based input file format for FEA

4.1.1 Overview

The proposed YAML based input file format for FEA meets the demand for data serialization presented in chapter 1.1.1. This is accomplished by representing the FEM model in YAML which is a data serialization language. The format produces a consistent FEM model which can be loaded and interpreted unambiguously by a computer program, and the YAML specification [10] describes a well-defined way of doing so.

Like in UFO and Cfem, the proposed input file format has external user-defined IDs for referencing. This is convenient with respect to readability, as autogenerated IDs are often long and cumbersome to work with. On the other hand, user-defined IDs leaves the responsibility of ensuring uniqueness to the user (or alternatively the application program assigning IDs

upon element creation). Most importantly, supporting user-defined IDs in the proposed format preserves the relation to UFO and Cfem.

Since the IDs are not guaranteed to be unique, their uniqueness must be assessed upon parsing to avoid ambiguities. This leads to increasing complexity in parsers for the proposed format. An alternative method for assigning IDs would be to use the universally unique identifier scheme (uuid). This scheme uses a 128 bits long number guaranteeing uniqueness across space and time[43]. The uuid is generally autogenerated so user-defined IDs are impossible (extremely labour-heavy) in this case. Most importantly, discarding the external IDs would reduce the compatibility with UFO and Cfem. On the other hand, the parser would be spared the labour of assessing ID uniqueness.

The proposed input file format uses the same strategy as UFO and Cfem input file format to define the FEM data records that make up the FEM model. That is, the records begin with a record identificator, e.g. NODE or TRISHELL, and the identificator is associated with a set of attributes fields. UFO and Cfem specify the attributes sequentially, separated by whitespace, after the identificator. The proposed format, on the other hand, associates them with the identificator by a key/value (mapping) relation, enabling the attributes to be specified in an arbitrary order. This strategy for defining a FEM data record has proved successful in both UFO and Cfem. Thus, the proposed format is assured to be extendible with regard to adding FEM data types to become equally expressive as UFO and Cfem. A negative consequence of resembling the structure of UFO and Cfem is that neither of them have a good strategy for representing composites. Therefore, possible strategies for representing composites within this data structure should be explored further, and the author considers examining Abaqus' *section* types a good place to start.

4.1.2 Assessment of requirements

Chapter 1.1.1 lists a set of requirements that the proposed format should satisfy in order to be considered well-functioning. Firstly, the format must represent every data type in a FEM structural model. This requirement is currently not satisfied, as the FEM data type library in the format is very scarce. However, the library is representative for most categories entailing the data records (cf. the Cfem documentation) so it can be expanded. A recommendation for extending the data type library is to use Haugen's Cfem format as a basis.

The second requirement is that the format must be readable by computers. Since the format is based on YAML, it can be parsed by any YAML parser. When the character stream is parsed, the application must process the YAML representation and convert them to language-specific data structures. The prototype parser uses yaml-cpp to load the data into a C++ program, but there also exist well-defined parsers for Python, Java, Ruby and other languages. Hence, this requirement is satisfied.

The third requirement is that the format must be serializable and deserializable. The fact that YAML is a data serialization language [10] ensures that this requirement is satisfied. The YAML 1.2 specification describes a precise way of formulating data to be serialized in YAML.

4.1.3 Use of YAML

An obvious consequence of basing the input file format on YAML is that the format inherits its properties. The highest prioritized design goal in YAML is human readability [10], and this property is inherited by the proposed input file format. This is helpful in terms of writing and troubleshooting the input file in a simple text editor. However, FEM models usually get complex and their input files get equivalently extensive so manual editing using a text editor is a cumbersome process despite readable input files. Nevertheless, an easily readable input file simplifies the task of examining specific data records during troubleshooting.

On the downside, human readability implies that the input file format is complex for computers to parse because it often leads to ambiguities. To parse a YAML file one must be able to interpret data in all the presentation styles in the YAML 1.2 specification, which can change arbitrarily throughout the character stream. For instance, a string in YAML can be presented in a literal style, a folded style, unquoted, single quoted and/or double quoted. The result is that a complex process is required for parsing general strings in YAML. This argument can be made with regards to the versatility between block and flow styles as well.

Even though developing a parser for YAML is complex, there exist well-defined YAML parsers implemented in different programming languages. Examples are *yaml-cpp* for C++, *PyYaml* for Python, and *Camel* for Java [21]. The software for these parsers is openly available and can be included in a programming project as third-party libraries. Easily accessible software libraries for parsing YAML is a strength to the proposed format because they make it possible to write parsers for it in different languages. Consequently, the format can be utilized to serialize and deserialize FE models in connection with FEA software implemented in any language with available YAML parsing software. It is, of course, also possible to write parser software for YAML in other languages as well, but that involves a significant workload.

Among the YAML features going beyond regular data serialization are anchors and aliases [44], a serialization detail used to reference previous occurrences of data in the YAML file. The proposed format does not utilize this functionality, as most FEM data records are not duplicate in nature (attribute values vary throughout the file). However, some properties may cause a lot of duplicate attributes. A situation where anchors and aliases could be useful is when many NODE records have the same set of boundary codes. All the boundary codes can be omitted if the node is free in all DoFs, but non-zero codes must be specified explicitly. Consider, for instance, a set of nodes all of which are fixed in the three translational DoFs, but with varying conditions for rotation. Explicitly specifying `ix`, `iy`, and `iz` causes many duplicate lines in the input file. By aliasing a mapping that specifies these attributes, the mapping could be referenced for each NODE with the possibility to add additional boundary codes. This strategy would both decrease the total number of lines and simplify modifications in the input file. Code listings 4.1 and 4.2 illustrate the example.

Code listing 4.1: NODE records without aliasing.

```
STRUCTURE:
 - NODE:
     id: 1
     xyz: [0, 0, 0]
     ix: 1
     iy: 1
     iz: 1
 - NODE:
     id: 2
     xyz: [1, 1, 0]
     ix: 1
     iy: 1
     iz: 1
     irx: 1
     iry: 1
```

Code listing 4.2: NODE records with aliasing for boundary codes.

```
#Create aliased mapping for
# translational DoFs:
trans_fixed: &trans_fixed
  ix: 1
  iy: 1
  iz: 1

STRUCTURE:
 - NODE:
     id: 1
     xyz: [0, 0, 0]
     <<: *trans_fixed #Reference aliased
     #DoFs
 - NODE:
     id: 2
     xyz: [1, 1, 1]
     <<: *trans_fixed #Reference aliased
     #DoFs
     irx: 1
```

4.1.4 Data hierarchy and dependency levels

The hierarchy of FEM data type reflects the very nature of a physical structure in the sense that, in structural mechanics, a structure is composed bottom up by entities with increasing level of complexity, beginning with the most basic ones. A file format designed to describe physical phenomena could benefit from resemblance to those phenomena. A positive consequence of such resemblance is that it is relatively straightforward to express the properties of the phenomena textually. And even though parsing is not straightforward because the hierarchy imposes an ordering on the creation of FEM data objects, the ordering is clear.

As previously stated, the data hierarchy forces parsers for the proposed input file format to process the data records in a specific order. This is a semi strict requirement as records with the same level of dependency can be parsed in an arbitrary order. From a parsing perspective randomly ordered records with a restriction on the processing order introduces a performance penalty because it involves multiple traversals of the entire YAML representation. However, the penalty can be significantly reduced by deleting the `YAML::Nodes` from the graph after they have been parsed. If the perspective is changed from parsing the format to emitting it, the possibility to randomly order the records yields high versatility. For example, the data could be structured according to their connection to a part making it easy to recognize which records compose which part.

The concept of dependency levels provides a simple, clear way of determining the placement of a FEM data record in the data hierarchy. It gives clear instructions as to when a data type should be parsed during the loading process of the YAML file. In addition to supplying valuable information to developers writing parser software for the format, the definition of dependency levels is easy to use when extending the format with new FEM data types.

4.1.5 Comparison with Abaqus, Nastran and UFO

This part of the report will draw attention to significant similarities and differences between the proposed input file format and the formats presented in chapter 2. Currently, the proposed format is very scarce compared to these formats, but an attempt at comparison is conducted nonetheless.

The **Abaqus input file format** has a more restrictive syntax than the proposed YAML based input file format. This is as expected, because versatility in presentation styles is a natural outcome of YAML's design goal for human readability. Abaqus imposes some ordering on the data records based on their affiliation to specific sections of the input files. For example, model data records and history data records cannot be interchanged in an Abaqus input file, while the proposed YAML based input file format allows records to appear arbitrarily.

In Abaqus comments may only appear on comment lines. By comparison, the proposed format allows comments anywhere in the input file, except within scalars. This distinction is noteworthy because is allows a better described input file. Moreover, it allows optional attributes to be commented out of data records in situations where deleting all traces of the original record is undesirable. This could simplify troubleshooting. An example of an ISOMATERIAL record (from the proposed format) where optional attributes are commented out are provided in Code listing 4.3:

Code listing 4.3: ISOMATERIAL record with optional attributes commented out.

```
ISOMATERIAL:
  id: 1
  type: elastic
  Emod: 2.0E+11
  poisson: 0.27
  #density: 8.5
  #thermX: 11.0E-6
```

More importantly, Abaqus supports functionality to represent sets of elements or nodes, and to define parts and assemblies. The ability to define element sets is widely used when applying conditions and performing operations on a group of elements. Because a typical FEM model consists of a large number of nodes and elements, and conditions rarely are defined for single elements, the set functionality is a feature used very frequently. In fact, one could argue that some kind of set functionality is required to create an adequately complex FEM model. Functionality to define node and element sets is not a feature in the proposed input file format. However, representing sets is perfectly possible within the scope of the YAML language, for instance by implementing sets as YAML sequences.

Another core feature in Abaqus is the possibility to define parts. Being able to reuse part definitions saves an enormous amount of work both with regards to defining and updating/changing the properties of repeating segments of the model. The ability to support part definitions is highly desired and is discussed further in chapter 6.

Chapter 2.3.1 presented Abaqus section definitions in connection with composite representations. Abaqus approaches the problem of composite representation by pooling cross-sectional geometry and material properties into sections that can be associated with an element. A similar approach is attempted in the proposed format with the PLCOMP record, inspired by USFOS

and Cfem. For now, support for composites is restricted to shell elements, but Abaqus' section functionality should be explored to develop composite sections for other elements as well.

The **Nastran bulk data section** has an intricate way of representing FEM data in its entries, comprised of three different formats. In sum, the three formats provide a balance between accuracy and performance, as trivial entries can be represented in a more compact way than entries with high performance requirements. On the downside, the categorization makes the format rather static. The proposed format, on the other hand, inherits the dynamicity of YAML. However, this does not come without a cost. Theoretically, the proposed format could have attribute values of any size, but programming languages parsing it must be able to represent them. For instance, if a numerical value is parsed as a double floating point, it must be stored in 8 bytes which corresponds to about 14 significant digits. An explicit limitation on the size of attribute values should be considered in the proposed format.

The proposed YAML based input file format has similar properties and structure as **UFO** and **Cfem**. In UFO and Cfem, every data record is specified by an identificator, at least one mandatory parameter, and it may have one or more optional parameters. Analogously, a FEM data type in the proposed format is composed by a FEM data keyword and a set of attributes, partitioned into mandatory and optional ones. Each FEM data record must have at least one mandatory attribute and may have zero or more optional ones. Moreover, external IDs are implemented in the proposed format based on the concept of external user-defined IDs used in UFO (and, in turn, Cfem).

The syntax of the proposed format differs significantly from UFO and Cfem. Where UFO and Cfem specify the attribute values sequentially after the record definition, the proposed format collects the attributes in a mapping and associates each attribute value with the corresponding attribute name. This difference enables the attributes to appear in an arbitrary order and explicitly stating attribute names in the proposed format.

Code listing 4.4: A NODE record in UFO

```
NODE 1 0.0 0.0 0.0 0 1 1 1 1 1
```

Code listing 4.5: A NODE record in YAML

```
NODE:
  id: 1
  x: 0.0
  y: 0.0
  z: 0.0
  ix: 0
  iy: 1
  iz: 1
  irx: 1
  iry: 1
  irz: 1
```

Code listings 4.4 and 4.5 display a NODE record with id equal to 1, located in the origin, and fixed in all DoFs except translation along the X-axis, in UFO and in the proposed YAML based format respectively. The YAML based format significantly enhances readability by explicitly associating the attribute values to their names. Positive outcomes of the YAML based input file format are readability, as the attribute names are explicitly specified, and versatility, as the attributes can have an arbitrary ordering. On the downside, the input files grow larger in the YAML based input file format than in UFO.

The reason why input files formulated on the proposed format grows larger is that it induces

boilerplate text (i.e. text that is used repeatedly) in the format. A model of a pipe joint, *Joint_model.fem*, can be found in the attached .zip-file. This file is 375 kB in size. The sizes of the corresponding YAML format files are shown in Table 4.1.

Format	Size	Increase
UFO	375 kB	
Proposed format in YAML block style	742 kB	97.9%
Proposed format in YAML flow style	567 kB	51.2%

Table 4.1: Format file size comparison

The results in this comparison is not conclusive, but gives an indication of the rate of growth when converting an input file from UFO to the proposed YAML based input file format for FEA. The increase in file size is not critical with the computational power of modern computers, but for large FEM models it is definitely noteworthy.

As a last note any optional attribute may be omitted in the proposed YAML based input file format. In contrast, UFO and Cfem insist that if an optional field is omitted, all subsequent fields in the record must be omitted as well.

4.1.6 Format verification using the prototype parser

Chapter 1.2 states that the purpose of the prototype parser is to verify that the format is possible to parse with C++ and available open-source tools, and to evaluate complexity of doing so.

The prototype parser is able to deserialize the YAML character stream and consistently create class instances from the data. Since the parser prints the data related to every object instance in the Structure object, parsing errors are uncovered by comparing the output to the content of the input file. This strategy for verification is not particularly scalable, as the amount of data grows too large to be effectively evaluated without automation. Nonetheless, it has proven effective in combination with relatively short input files to quickly assess whether the content of the internal data model in the parser program matches the input file. Hence, the prototype parser is considered successful as a tool to verify parsability and complexity in the proposed format.

Figure 4.1 shows the command window output from the prototype parser after it has parsed the YAML based input file presented in Code listing 3.8. The content of Code listing 3.8 is reproduced in Code listing 4.6 for comparison. Observe that Figure 4.1 shows some record that are not specified in Code listing 4.6. These are default attributes instantiated upon construction of the Structure class, used when no identifier is specified for optional attributes in FEM data types with dependency level one or higher. The default attributes are the ones with ID equal to 0 or -1.

```
Microsoft Visual Studio Debug Console
Please choose which part of the program to run (Parser = 0 | Converter = 1): 0
Parser chosen.
Enter file path: cantilever_example_flow.yaml
CoordSys:    id: 0
1       0       0
0       1       0
0       0       1
CrossSection:           id: -1type: CROSS_SECTION_DUMMY
Tsection:    id: 1, H: 0.3, T_web: 0.05, Width: 0.1, T_top: 0.05
IsoMaterial:   id: 0, type: elastic, Emod: 1, poisson: 1, yield: 0, density: 1, thermX: 1
IsoMaterial:   id: 1, type:, Emod: 2e+11, poisson: 0.27, yield: -1, density: -1, thermX: 0
FENode:   id: 1, x: 0, y: 0, z: 0, ix: 1, iy: 1, iz: 0, irx: 0, iry: 0, irz: 1, rotID: 0
FENode:   id: 2, x: 0.5, y: 0, z: 0, ix: 0, iy: 0, iz: 0, irx: 0, iry: 0, irz: 0, rotID: 0
FENode:   id: 3, x: 1, y: 0, z: 0, ix: 0, iy: 0, iz: 0, irx: 0, iry: 0, irz: 0, rotID: 0
FENode:   id: 4, x: 1.5, y: 0, z: 0, ix: 0, iy: 0, iz: 0, irx: 0, iry: 0, irz: 0, rotID: 0
FENode:   id: 5, x: 2, y: 0, z: 0, ix: 0, iy: 0, iz: 0, irx: 0, iry: 0, irz: 0, rotID: 0
FENode:   id: 6, x: 2.5, y: 0, z: 0, ix: 0, iy: 0, iz: 0, irx: 0, iry: 0, irz: 0, rotID: 0
FENode:   id: 7, x: 3, y: 0, z: 0, ix: 0, iy: 0, iz: 0, irx: 0, iry: 0, irz: 0, rotID: 0
FENode:   id: 8, x: 3.5, y: 0, z: 0, ix: 0, iy: 0, iz: 0, irx: 0, iry: 0, irz: 0, rotID: 0
FENode:   id: 9, x: 4, y: 0, z: 0, ix: 0, iy: 0, iz: 0, irx: 0, iry: 0, irz: 0, rotID: 0
FENode:   id: 10, x: 4.5, y: 0, z: 0, ix: 0, iy: 0, iz: 0, irx: 0, iry: 0, irz: 0, rotID: 0
FENode:   id: 11, x: 5, y: 0, z: 0, ix: 0, iy: 0, iz: 0, irx: 0, iry: 0, irz: 0, rotID: 0
Eccentricity:    id: 0, eX: 0, eY: 0, eZ: 0
Vector: id: 0, components (xyz): 1      0       0
FEBeam:    id: 1, node1: 1, node2: 2, material: 1, geoID: 1, ecc1: 0, ecc2: 0
FEBeam:    id: 2, node1: 2, node2: 3, material: 1, geoID: 1, ecc1: 0, ecc2: 0
FEBeam:    id: 3, node1: 3, node2: 4, material: 1, geoID: 1, ecc1: 0, ecc2: 0
FEBeam:    id: 4, node1: 4, node2: 5, material: 1, geoID: 1, ecc1: 0, ecc2: 0
FEBeam:    id: 5, node1: 5, node2: 6, material: 1, geoID: 1, ecc1: 0, ecc2: 0
FEBeam:    id: 6, node1: 6, node2: 7, material: 1, geoID: 1, ecc1: 0, ecc2: 0
FEBeam:    id: 7, node1: 7, node2: 8, material: 1, geoID: 1, ecc1: 0, ecc2: 0
FEBeam:    id: 8, node1: 8, node2: 9, material: 1, geoID: 1, ecc1: 0, ecc2: 0
FEBeam:    id: 9, node1: 9, node2: 10, material: 1, geoID: 1, ecc1: 0, ecc2: 0
FEBeam:    id: 10, node1: 10, node2: 11, material: 1, geoID: 1, ecc1: 0, ecc2: 0
FENodeLoad:    id: 1, loadCaseId: 1, node: 11, fx: 0, fy: -10000, fz: 0, mx: 0, my: 0, mz: 0, ecc: 0
LoadComb:    id: 1 factors: 1: 1.000000,

Some numbers:
Size of coordSysList: 1
Size of materials list: 2
Size of cross sections list: 2
Size of nodes list: 11
Size of elements list: 12
Size of NodeLoad list: 1
Size of loadCombList: 1
Total number of objects: 30

C:\Users\oskar\YAMLParser\x64\Debug\YAMLParser.exe (process 3840) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

Figure 4.1: Prototype parser output for the simple beam cantilever presented in Code listing 3.8

Code listing 4.6: Simple beam cantilever formulated in the proposed format using YAML flow style (reproduced).

```
#Example resembling Abaqus example
{STRUCTURE: [
  #Nodes
  {NODE: {id: 1, xyz: [0, 0, 0], ix: 1, iy: 1, irz: 1}},
  {NODE: {id: 2, xyz: [0.5, 0, 0]}},
  {NODE: {id: 3, xyz: [1, 0, 0]}},
  {NODE: {id: 4, xyz: [1.5, 0, 0]}},
  {NODE: {id: 5, xyz: [2, 0, 0]}},
  {NODE: {id: 6, xyz: [2.5, 0, 0]}},
  {NODE: {id: 7, xyz: [3, 0, 0]}},
  {NODE: {id: 8, xyz: [3.5, 0, 0]}},
  {NODE: {id: 9, xyz: [4, 0, 0]}},
  {NODE: {id: 10, xyz: [4.5, 0, 0]}},
  {NODE: {id: 11, xyz: [5, 0, 0]}},
  #Beam elements
  {BEAM: {elemID: 1, nodes: [1, 2], material: 1, geoID: 1}},
  {BEAM: {elemID: 2, nodes: [2, 3], material: 1, geoID: 1}},
  {BEAM: {elemID: 3, nodes: [3, 4], material: 1, geoID: 1}},
  {BEAM: {elemID: 4, nodes: [4, 5], material: 1, geoID: 1}},
  {BEAM: {elemID: 5, nodes: [5, 6], material: 1, geoID: 1}},
  {BEAM: {elemID: 6, nodes: [6, 7], material: 1, geoID: 1}},
  {BEAM: {elemID: 7, nodes: [7, 8], material: 1, geoID: 1}},
  {BEAM: {elemID: 8, nodes: [8, 9], material: 1, geoID: 1}},
  {BEAM: {elemID: 9, nodes: [9, 10], material: 1, geoID: 1}},
  {BEAM: {elemID: 10, nodes: [10, 11], material: 1, geoID: 1}},
  #Material
  {ISOMATERIAL: {matID: 1, type: elastic, Emod: 2.0e+11, poisson: 0.27}},
  #Cross-section
  {TSECTION: {geoID: 1, H: 0.3, T_web: 0.05, Width: 0.1, T_top: 0.05}},
```

```
  #Load data
  {NODELOAD: {loadCaseID: 1, nodeID: 11, fy: -10000}},
  {LOADCOMB: {loadCombID: 1, factors: {1: 1.0}}}
]}
```

An alternative strategy for format verification is to use the emitter module to perform a YAML dump operation, writing all the data back to a new file. In this case, the prototype parser initially loads the input file and instantiates objects from its content, before it dumps the content back into another YAML file used for comparison. The process of comparing the character streams of two files is relatively easy to implement with basic IO-functionality in C++. This strategy is easier to automate and streamline than the original strategy that simply prints all data to console. Moreover, it significantly decreases the probability of human error in the verification process. Although the manual comparison approach has been sufficient in the early phases of development conducted in this project, it is strongly recommended to implement the automated strategy for future work.

4.1.7 Intended use of the proposed format

The proposed format is an input file format for FEA of structural problems. Hence, it is concerned with representing geometry, material behaviour, loading situations, connectivity etc. Its intended use is restricted to structural analyses, and is not concerned with electromagnetism, computational fluid dynamics, acoustics, or other fields of science and engineering.

The proposed YAML based input file format for FEA is far from a complete format. In order to become one, its data type library must be extended, so the current purpose of the format is to be a proof of concept. The proposed format demonstrates how input data for FEA of structural analysis can be presented in YAML, by the fact that the FEM data types in the proposed format are successfully parsed by the prototype parser.

4.2 The prototype parser for the proposed format

4.2.1 Overview

The prototype parser successfully parses input files formulated in the proposed YAML based format for FEA. This is proved by the fact that it successfully constructs an internal C++ representation (i.e. class instances, pointers, primitives etc.) based on the input data without losing or adding information. In that sense the prototype parser is considered successful as it satisfies the requirement for format verification. Furthermore, it is implemented using only C++ and available open-source tool (yaml-cpp) to parse the input file format, which fulfills the second objective formulated in chapter 1.2.

The prototype parser has proven to be a valuable tool for developing the the proposed input file format. It has been used actively to verify that the format adheres to the YAML syntax and to ensure that the data is structured as intended. The FEM data type classes in the prototype

parser fully correspond to the FEM data records in the proposed format, and their instances are properly populated with the input data. This remark, however trivial, is essential because it implies that the parser is fully compatible with the proposed format.

4.2.2 Grouping of FEM data instances

As described in chapter 3.1.2, the FEM data instances are divided into separate groups in the Structure class. The division is based on similarity of elements and bears resemblance to the division in OOCfem's parser for the UFO format (*input_usfos*). This design enables the prototype parser to be compatible with OOCfem in the future.

4.2.3 Performance assessment

One of the design goals of YAML is to support one-pass processing [10], a property that decreases time needed to parse the character stream. While the character stream is effectively parsed by yaml-cpp, the succeeding traversal of the YAML representation, performed by the prototype parser, requires several iterations of the entire graph (see Figure 3.4 for illustration). This is a consequence of the dependency levels introduced in chapter 3.1 and originates from the natural hierarchy in the FEM data.

No alternative method to enhance the performance while conserving the order of object instantiation was uncovered in this project. Nonetheless, a way of minimizing this effect is to remove the `YAML::Node` from the graph after it is parsed. This is programmatically trivial to implement, as it simply involves deleting an entry in the YAML sequence representing the graph. According to a pull request in the yaml-cpp repository on Github [45], functionality for removing a `YAML::Node` from a sequence is implemented, but this update was not part of the source code in the version of yaml-cpp used in this project (even though the repository was cloned after the pull request was merged into the master branch).

In the yaml-cpp version used in this project, `YAML::Node`s have the member function `remove (const Key& key)`, where `Key` is the typename of the template. But when called on a `YAML::Node` of type sequence, no entry is removed. This is illustrated in Code listing 4.7.

Code listing 4.7: Excerpt from Parser.cpp, modified to remove nodes from the YAML representation after they have been parsed.

```
void Parser::parseDepenencyLevelOne() {
        for (int iterator = 0; iterator < structureNode.size(); ++iterator) {
                YAML::const_iterator it = structureNode[iterator].begin();
                std::string key = it->first.as<std::string>();

                if (key == "NODE") {
                        nextNode = structureNode[iterator][key];
                        parseNode(nextNode, key);
                        std::cout << "\nPre-removal: structureNode.size = "
                        << structureNode.size() << std::endl;

                        //Remove element from YAML secuence
                        structureNode.remove(iterator);

                        std::cout << "Post-removal: structureNode.size = "
                        << structureNode.size() << std::endl;
                }
                if (key == "PLCOMP") {
                        nextNode = structureNode[iterator][key];
                        parsePLComp(nextNode, key);
                }
                if (key == "LOADCOMB") {
                        nextNode = structureNode[iterator][key];
                        parseLoadComb(nextNode);
                }
        }
}
```

Output from std::cout statements:

```
Pre-removal: structureNode.size = 2
Post-removal: structureNode.size = 2

Pre-removal: structureNode.size = 2
Post-removal: structureNode.size = 2
```

Although the prototype parser does not remove a `YAML::Node` from the representation graph after parsing it, this functionality is desired. The matter is elaborated in chapter 6.

Object uniqueness

The problem with ambiguous IDs among element records, described in chapter 3.1.2, causes performance penalties and diminishes the purpose of the element map. The performance penalty is due to the fact that searching in a vector has linear time complexity (O(n)) [46], while searching in a map has logarithmic time complexity (O(log(n))) [47]. It is highly reasonable to expect the size of the element vector and map to be large, hence the penalty might be severe. One possible solution to this problem is to change the map into an `std::multimap`, which accepts duplicate keys and maintain logarithmic time complexity. Another is to change the map key from merely containing the ID to being comprised by an <ID, type> tuple. A third solution would be to update the proposed format to require all IDs to be unique across the data types that are members of the element group (i.e. all FEM data type classes that have `GenericFE` as base class).

Due to time limitations and the fact that this error was uncovered late in the project, it was chosen to avoid the problem by iterating over the vector and leave the map unattended. The reason for this is that high performance is not a critical requirement for a proof of concept. The prototype parser is satisfactory for format verification despite the subotimal performance.

4.2.4 The converter/emitter module

The converter module successfully emits a YAML character stream on the proposed format. The character stream passes validation when submitted to online YAML parsing services, such as Online YAML Parser[48]. Consequently, it is a valid YAML presentation, parsable by yaml-cpp and, in turn, the prototype parser developed in this project. The module is a valuable tool to generate files on the format proposed in chapter 3.1.

A considerable issue in the converter module is that it is not up to date with the proposed format. It lacks functionality to emit FEM data types comprising lists of arbitrary length (e.g. LOADCASE), and several data types are not supported by the converter module. Keeping the converter module up to date involves a significant workload and is beyond the scope of this project, as the converter is not part of the scope definition in chapter 1.2. Thus, it has not been prioritized for this thesis.

Throughout the work carried out in this project, the converter module has only been used in the initial phase to obtain a collection of comprehensive input files comprised of FEM data types that occur very frequently in a FEM model (such as NODE, BEAM, TRISHELL, UNITVEC etc.). As the work continued, keeping the converter updated was not prioritized because writing example input files to verify the new FEM data types was more effective than keeping the converter module updated.

4.2.5 Intended use of the prototype parser

The prototype parser is intended to be a tool for development of the YAML based input file format for FEA. Furthermore, it can be used in connection with testing of FEM solver algorithms as a means to make the YAML based input file format accessible. The prototype parser successfully deserializes a selection of the data records available in the YAML based input file format and should be used for exactly that purpose.

The prototype parser is an effective tool for practical testing of the YAML based input file format. Thus, when the format is extended with new data records, functionality for parsing the record should be implemented and tested in the prototype parser. This approach to format development enhances the credibility of the format in the sense that complications related to practical format applications can be uncovered when attempting to implement the parse logic.

Chapter 5

Conclusion

The main objective of this thesis is to propose a YAML based input file format for Finite Element Analysis of structural problems. The proposed format is not a full-scale input file format, but it comprises a collection of core data types needed to create FE models of structural problems. Furthermore, it builds on the same structures as UFO, where FEM data records are composed by a record identificator (keyword) and a set of mandatory and optional attributes. UFO has a broad data library, hence this structuring proves viable in terms of extendibility. Moreover, all FEM data types in the proposed format are similar to the types in Cfem, simplifying a transition from the Cfem input file format to the proposed format in OOCfem.

Three functional requirements for the YAML based input file format for FEA were listed in chapter 1:

1. The input file format must be sufficiently expressive to represent every data type comprised in a FEM structural model.
2. The input file format must be readable by computers, i.e. a computer must be able to parse the input file.
3. The input file format must be serializable (and deserializable).

Currently, the proposed format is not sufficiently expressive to represent every data type comprised in a FEM structural model. However, as the format is extendible, this requirement could be satisfied over time. The prototype parser successfully parses input files formulated in the proposed format, proving that the format is readable by computers. The proposed format also satisfies the requirement for data serializability, due to the fact that it is based on YAML which is a data serialization language.

The second part of the objective for this thesis is to implement a prototype parser for the format using C++ and available open-source tools. The prototype parser delivered alongside this thesis satisfies this objective with emphasis on being a generic prototype. It is not designed to be integrated directly into OOCfem, but the internal data structure in the proposed format resembles the structure in OOCfem's *usfos_reader* class. The internal data structure is easily accessible

through the Structure class of the prototype parser. Thus, the groundwork for possible integration with OOCfem is laid.

Both the proposed format and the prototype parser are characterized by being in their infancy. They are minimalistic and scarce, but function as a proof of concept: They prove that a YAML based input file format for FEA of structural problems is possible and they form a basis for it. On account of this, it is recommended that both the proposed format and the prototype parser are developed further.

Chapter 6

Future work

The work carried out in this project has resulted in a mere basis for a YAML based input file format for FEA, hence there is much work to be done in order to develop a worthy format. This chapter presents important issues that should be attended in further development, both with regard to the format and the prototype parser.

Extend the data type library

The perhaps most obvious weakness in the proposed format is that its library of FEM data types is scarce. The format should have all the data types comprised in Cfem input file format, as well as new data types must be designed, inter alia, to define composites. Since Cfem aims at capturing the functionality in UFO, the documentation of UFO should also be used when searching for and implementing new data types.

Representing composites

To better represent composites, *section*s, as defined in Abaqus, should be explored and implemented in the proposed YAML based input file format. The main issue with composites is that cross-sectional data and material data must be pooled into one record. The proposed format has functionality to do this for shell elements, represented by the PLCOMP record. However, analogous functionality for other cross-sections, such as pipes or I-profiles used in beams, composites are not supported in the current version of the format.

Support element and node sets

As discussed in chapter 4, the proposed format does not support sets of elements and nodes. Support for grouping data as in Abaqus is highly desired to apply initial and boundary conditions to collections of nodes and elements, and it is possible within the YAML language. Sets

could for instance be implemented as YAML sequences and conditions could then be applied through mappings.

Parts and assemblies

Parts and assemblies are valuable features in FEM modelling, as discussed in chapters 2.3.1 and 4.1. By introducing a PART layer between the STRUCTURE and the FEM data records in the proposed format, the proposed format would be able to divide the structure into parts. Moreover, parts could be reused with YAML anchors and aliases. This strategy, or an alternative, for representing parts as well as assemblies should be examined further.

Compatibility with OOCfem

In order to use the proposed YAML based input file format for FEA with OOCfem, compatibility between the prototype parser and OOCfem is a key issue. Since the proposed format is structured similarily as UFO and Cfem, the *usfos_reader* class in OOCfem should be examined, and the prototype parser for the proposed format should generate an internal data structure similar to the one produced by the *usfos_reader*.

Another concern when implementing the prototype parser into solver software is performance. The prototype parser is not optimized for high performance. Quite the contrary, rapid implementation and validation of new data types has been prioritized. For the prototype parser to function properly in cooperation with the OOCfem solver, its performance should be enhanced to decrease the loading time of a FE model.

Error messages in the prototype parser

The prototype parser would benefit from having more informative error messages. In the current version, the error messages provide information about the data type of failed records, but it does not tell the user where in the input file the faulty data record is located. yaml-cpp has functionality for detecting the line number of the faulty record in the input file, but the functionality is not designed to be accessible by yaml-cpp users. This functionality is found in the header file *mark.h* in yaml-cpp. The possibility of implementing similar functionality in the prototype parser should be investigated to enhance the expressiveness of its error messages.

Expand the expressiveness of the converter module to accommodate the proposed format

As discussed in chapter 4.2.4, the converter module should be updated to match the current state of the proposed format. This would make the proposed format more available for use, as example input files written in Cfem could be automatically converted and the resulting YAML file could be parsed. The data types that must be implemented for the converter to match the data type library of the proposed format are:

- TSECTION
- ZVECTOR and YVECTOR
- TSECTION
- Material types
- PLCOMP (This type does not exist in UFO[9], but exists in Cfem)
- NODELOAD
- LOADCOMB

The NODELOAD and LOADCOMB records require that the Writer class is extended with functionality to write YAML sequences of arbitrary length in their attributes.

Automated format verification

As the amount of data to parse grows large, the current method for format verification using the prototype parser is insufficient. Currently, parsing errors are uncovered by manually comparing the output of the prototype parser to the content of the input file. This has proven effective in the infancy of the format, but it is not viable as the FEM data type library and size of the input files grow larger.

The emitter module should be used for format verification. By parsing a character stream, the resulting YAML representation could be emitted to a new file using a yaml-cpp `YAML::Emitter`. The process of comparing the two YAML files should then be automated with basic IO-functionality in C++. This is a more effective and secure method for format verification, as it is less laborious and reduces the possibility of human error.

These objectives are left for future projects.

Bibliography

[1] Niels Ottosen and Hans Petterson. *Introduction to the Finite Element Method*. Pearson Education Limited, 1992.

[2] Suohai Gu. Application of finite element method in mechanical design of automotive parts. *IOP Conference Series: Materials Science and Engineering*, 231:012180, sep 2017.

[3] Radu Crahmaliuc. 75 years of the finite element method (fem). https://www.simscale.com/blog/2015/11/75-years-of-the-finite-element-method-fem/. Accessed on 05.05.2019.

[4] G.P. Nikishkov. Introduction to the finite element method. http://homepages.cae.wisc.edu/~suresh/ME964Website/M964Notes/Notes/introfem.pdf, 2004. Lecture notes. Accessed on 06.06.19.

[5] Techopedia Inc. Serialization. https://www.techopedia.com/definition/867/serialization-net. Accessed on 05.05.2019.

[6] Kashyap Vyas. 23 Engineering Disasters of All Time. https://interestingengineering.com/23-engineering-disasters-of-all-time. Accessed on 15.06.2019.

[7] J. Michopoulus et al. FemML for Data Exchange between FEA Codes. *ANSYS Users Group Conference*, 2001.

[8] Dassault Systmes. Abaqus 6.14 Documentation. http://ivt-abaqusdoc.ivt.ntnu.no:2080/texis/search/?query=wetting&submit.x=0&submit.y=0&group=bk&CDB=v6.14. Accessed on 21.05.2019.

[9] USFOS A/S. Usfos user's manual input description usfos control parameters. http://www.usfos.no/manuals/usfos/users/documents/Usfos_UM_06.pdf. Accessed on 19.05.2019.

[10] Oren Ben-Kiki, Clark Evans, and Brian Ingerson. Yaml aint markup language (yaml) version 1.2. http://yaml.org/spec/1.2/spec.pdf, October 2009. Accessed on 11.05.2019.

[11] Bjørn Haugen. *Bucking And Stability Problems For Thin Shell Structures Using High Performance Finite Elements*. PhD thesis, University of Colorado, 1994.

[12] Kristoffer Markus Kopperud. File Based Input and Results Databse for "Focus Konstruksjon" Structural Analysis Software. Master's thesis, Norwegian University of Science and Technology (NTNU), 2017.

[13] Dimitri van Heesch. Doxygen (version 1.8.15). http://www.doxygen.nl/. Software licenced under GNU General Public Licence.

[14] Agile Alliance. Agile Alliance home page. https://www.agilealliance.org/. Accessed on 13.05.19.

[15] Agile Alliance. Agile 101. https://www.agilealliance.org/agile101/. Accessed on 13.05.19.

[16] Mark Beedle et al. Agile manifesto agile software development. https://agilemanifesto.org/. Accessed on 13.05.2019.

[17] cppreference.com. Fundamental types. https://en.cppreference.com/w/cpp/language/types. Accessed on 03.06.19.

[18] Python Software Foundation. Built-in types. https://docs.python.org/3/library/stdtypes.html. Accessed on 03.06.19.

[19] Laurence Tratt. Dynamically typed languages. *Advances in Computers*, 77:149–184, July 2009.

[20] Refsnes Data. w3schools - JavaScript Data Types. https://www.w3schools.com/js/js_datatypes.asp. Accessed on 03.06.2019.

[21] Oren Ben-Kiki, Clark Evans, and Brian Ingerson. Yaml homepage. https://yaml.org/. Accessed on 26.05.2019.

[22] Douglas Crockford. Introducing json. http://json.org/. Accessed on 29.11.2018.

[23] Technical Committee 39. Standard ecma-404 - the json data interchange syntax. http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf, December 2017. Accessed on 21.05.2019.

[24] Jesse Beder. yaml-cpp: A yaml parser and emitter in c++. https://github.com/jbeder/yaml-cpp. Accessed on 11.05.2019.

[25] Jesse Beder. yaml-cpp: Tutorial. https://github.com/jbeder/yaml-cpp/wiki/Tutorial. Accessed on 11.05.2019.

[26] Jesse Beder. yaml-cpp: How to emit yamll. https://github.com/jbeder/yaml-cpp/wiki/How-To-Emit-YAML. Accessed on 02.12.2018.

[27] Dassault Systmes. Abaqus/cae start page. https://www.3ds.com/products-services/simulia/products/abaqus/abaquscae/. Accessed on 21.05.2019.

[28] Dassault Systmes. Abaqus analysis user's manual - 1.3.1 defining a model in abaqus. https://www.sharcnet.ca/Software/Abaqus610/Documentation/docs/v6.10/books/usb/default.htm?startat=pt01ch01s03aus03.html. Accessed on 21.05.2019.

[29] Dassault Systmes. Abaqus analysis user's manual - 1.2.1 input syntax rules. https://www.sharcnet.ca/Software/Abaqus610/Documentation/docs/v6.10/books/usb/default.htm?startat=pt01ch01s02aus01.html. Accessed on 21.05.2019.

[30] Dassault Systmes. Abaqus user's manual - 2.9.1 defining an assembly. https://www.sharcnet.ca/Software/Abaqus610/Documentation/docs/v6.10/books/usb/default.htm?startat=pt01ch02s09aus26.html. Accessed on 21.05.2019.

[31] Dassault Systmes. Abaqus user's manual - 12.2.3 defining sections. http://dsk-016-1.fsid.cvut.cz:2080/v6.12/books/usi/default.htm?startat=pt03ch12s02s03.html. Accessed on 27.05.2019.

[32] Dassault Systmes. Abaqus user's manual - 12.13.7 creating composite shell sections. http://dsk-016-1.fsid.cvut.cz:2080/v6.12/books/usi/default.htm?startat=pt03ch12s13s02.html. Accessed on 27.05.2019.

[33] Dassault Systmes. Abaqus user's manual - 28.1.1 solid (continuum) elements. http://dsk-016-1.fsid.cvut.cz:2080/v6.12/books/usb/default.htm?startat=pt06ch28s01alm01.html#usb-elm-esolidcont-compshell. Accessed on 27.05.2019.

[34] Dassault Systmes. Abaqus user's manual - 29.3.3 choosing a beam element. https://www.sharcnet.ca/Software/Abaqus/6.14.2/v6.14/books/usb/default.htm?startat=pt06ch29s03alm08.html. Accessed on 02.06.2019.

[35] National Aeronautics and Space Administration (NASA). Early nasa dream computer program still optimizes designs. https://spinoff.nasa.gov/Spinoff2018/it_2.html. Accessed on 18.05.2019.

[36] Kevin Kilroy. Msc.nastran quick reference guide. *MSC.Software Corporation*, 2001.

[37] USFOS A/S. Usfos non-linear static and dynamic analysis of space frame structures. http://www.usfos.no/. Accessed on 19.05.2019.

[38] USFOS A/S. Usfos graphical user interface. userś manual. http://www.usfos.no/manuals/usfos/users/documents/Xact_UM.pdf. Accessed on 19.05.2019.

[39] Focus Software AS. Fokus konstruksjon. https://www.focus.no/focus-konstruksjon/. Accessed on 02.06.19.

[40] Oocfem. https://github.com/BjornArnesonHaugen/OOCfem. Private Github repository by Bjørn Haugen.

[41] Bjørn Haugen. Cfem input file format. https://github.com/BjornArnesonHaugen/OOCfem/blob/master/Documentation/cfem_input_file_main.pdf. Located at private Github repository by Bjørn Haugen.

[42] European Committee For Standardization. EN 1995-1-1: Eurocode 5: Design of timber structures - Part 1-1: General - Common rules and rules for buildings. *Brussels: BSi*, 2004.

[43] Leach et al. A universally unique identifier (uuid) urn namespace. https://tools.ietf.org/html/rfc4122#page-16. Accessed on 01.06.2019.

[44] King Chung Huang. Dont repeat yourself with anchors, aliases and extensions in docker compose files. https://medium.com/@kinghuang/docker-compose-anchors-aliases-extensions-a1e4105d70bd. Accessed on 26.05.2019.

[45] StarJiao and Jesse Beder. Enable items to be removed from a sequence (#582). https://github.com/jbeder/yaml-cpp/commit/4fb1c4b92bf8d94b32ebccdd890407d45b3bc794. Accessed on 04.06.19.

[46] cplusplus.com - std::find. http://www.cplusplus.com/reference/algorithm/find/. Accessed on 09.06.2019.

[47] cpluscplus.com - std::map::find. http://www.cplusplus.com/reference/map/map/find/. Accessed on 09.06.2019.

[48] Paul Tarjan. Online yaml parser. http://yaml-online-parser.appspot.com/. Last accessed on 05.06.19.

Appendix A

Documentation of the YAML based format for FEA

Introduction

The purpose of this document is to provide documentation for a YAML based input file format for FEA. The format is proposed as part of the master's thesis ”YAML Based Input File Format For Finite Element Analysis” by Oskar Hvalstad-Nilsen at the Norwegian University for Science and Technology (NTNU).

YAML

YAML is a human friendly data serialization standard for all programming languages[1]. The input file format is based on the YAML 1.2 specification, which can be found at:

https://yaml.org/spec/1.2/spec.html.

The YAML specification provides two syntax styles: YAML Block style and YAML Flow style. This document shows how the records are formulated in the YAML block style, but the input file format is also compatible with the YAML Flow style.

Use of YAML native data types

Each FEM data record is represented by a nested structure of two YAML *mappings*. The key of the outer mapping is the FEM keyword of the record (e.g. `NODE`, `BEAM`, or `PIPE`). The value

[1] Source: https://yaml.org/

of this mapping is a new mapping that holds the attribute data. Each attribute consists of an attribute name and a corresponding value. The attribute name and value are stored in the key and value fields (respectively) of the inner mapping. This template for this representation is presented in section A.

Definitions

bool	Boolean data type
double	Double floating point
int	Integer value
string	Text string

Documentation layout

Formulations

This documentation provides examples of how each data record is formulated in YAML. Every record has at least one formulation, but some have two.

- **Formulation 1** is the most rigid formulation with one expression for each attribute.
- **Formulation 2** shows how similar attributes can be assembled into a list containing each atomic attribute.

When Formulation two is possible, the sequential entries are expressed by the attribute names from Formulation 1 such that the order of the attributes is clarified. To see the data type of those attributes, review Formulation 1.

Required and optional attributes

Every data record has one or more required/mandatory attribute(s), and some have optional attributes as well. In order to create a valid data type, all the mandatory attributes must be defined, but one or more optional attribute(s) may be omitted. In this document, required and optional attributes are denoted as follows in the attribute name / description tables:

- Required attribute names are expressed in **bold face**.
- Optional attribute names are expressed with no text formatting.

Layout template

Formulation 1	**Formulation 2**

FEM_KEYWORD:
  Attr.1: data type
  Attr.2: data type
  Attr.3: data type
  ...
	FEM_KEYWORD:
	  Attr.1: data type
	  Attr.List: [attr, attr]
	  ...

Attribute name	**Description**
Attr.1	Description of attribute 1
Attr.2	Description of attribute 2
Attr.3	Description of attribute 3
Attr.List	Description of attribute list

A.1 Native Input File Commands

A.1.1 Nodal data

COORDSYS

COORDSYS defines a local coordinate used by nodes where boundary conditions are given wrt. local coordinates.

The dependency level of this data record is **null**.

Formulation 1	Formulation 2
COORDSYS: rotID: int Xx: double Xy: double Xz: double Yx: double Yy: double Yz: double Zx: double Zy: double Zz: double	COORDSYS: rotID: int X: [Xx, Xy, Xz] Y: [Yx, Yy, Yz] Z: [Zx, Zy, Zz]

Attribute name	Description
rotID	User defined external node number
Xx, Xy, Xz	Vector defining local X-direction
Yx, Yy, Zz	Vector defining local Y-direction
Zx, Zy, Zz	Vector defining local Z-direction

Table A.1: Name and description of the attributes that constitute a COORDSYS

Note: The formulations of the vectors defining local directions in Formulation 1 and Formulation 2 are mutually exclusive. For each data record, one of the formulations must be chosen.

NODTRANS

NODTRANS defines a local coordinate used by nodes where boundary conditions are given wrt. local coordinates. The YAML block style formulation of this data record is defined in the following.

The dependency level of this data record is **null**.

Formulation 1	Formulation 2

NODTRANS:
  rotID: int
  Xx: double
  Xy: double
  Xz: double
  Yx: double
  Yy: double
  Yz: double
  Zx: double
  Zy: double
  Zz: double
	NODTRANS:
	  rotID: int
	  X: [Xx, Xy, Xz]
	  Y: [Yx, Yy, Yz]
	  Z: [Zx, Zy, Zz]

Attribute name	Description
rotID	User defined external node number
Xx, Xy, Xz	Vector defining local X-direction
Yx, Yy, Zz	Vector defining local Y-direction
Zx, Zy, Zz	Vector defining local Z-direction

Table A.2: Name and description of the attributes that constitute a NODTRANS

Note: The formulations of the vectors defining local directions in Formulation 1 and Formulation 2 are mutually exclusive. For each data record, one of the formulations must be chosen.

PCOORDSYS

PCOORDSYS defines a local coordinate system used by nodes where boundary conditions are given wrt. local coordinates, analogous to the COORDSYS record. PCOORDSYS is defined by points for local origin, X vector and Z vector.

The dependency level of this data record is **null**.

Formulation 1	Formulation 2
PCOORDSYS: rotID: int Ox: double Oy: double Oz: double Xx: double Xy: double Xz: double Zx: double Zy: double Zz: double	PCOORDSYS: rotID: int O: [Ox, Oy, Oz] X: [Xx, Xy, Xz] Z: [Zx, Zy, Zz]

Attribute name	Description
rotID	User defined external node number
Ox, Oy, Oz	Points defining local origin
Xx, Xy, Xz	Points defining local X-direction
Zx, Zy, Zz	Points defining local Z-direction

Table A.3: Name and description of the attributes that constitute a PCOORDSYS

Note: The formulations of the vectors defining local directions in Formulation 1 and Formulation 2 are mutually exclusive. For each data record, one of the formulations must be chosen.

NODE

NODE defines a coordinate location (point) and its DoFs.

The dependency level of this data record is **one**.

<table>
<tr><th>Formulation 1</th><th>Formulation 2</th></tr>
<tr><td><pre>NODE:
  id: int
  x: double
  y: double
  z: double
  ix: bool
  iy: bool
  iz: bool
  irx: bool
  iry: bool
  irz: bool
  rotId: int</pre></td><td><pre>NODE:
  id: int
  xyz: [double, double, double]
  ix: bool
  iy: bool
  iz: bool
  irx: bool
  iry: bool
  irz: bool
  rotId: int</pre></td></tr>
</table>

Attribute name	Description
id	User defined external node number
x, y, z	X, Y, and Z coordinates for the node
xyz	X, Y, and Z coordinates for the node, given on list form. *This attribute is mutually exclusive with the x, y, and z attributes above.*
ix, iy, iz	Boundary condition code for the translational DOFs
irx, iry, irz	Boundary condition code for the rotational DOFs
rotID	Reference to local coordinate system

Table A.4: Name and description of the attributes that constitute a NODE

Note: Only the id, x, y, and z attributes are required. The other attributes (boundary codes and and coordinate system id) may be omitted. Omitting a boundary code implies that the node is free in the corresponding DOF. Omitting the local coordinate id implies that the node is defined with respect to the global coordinate system.

A.1.2 Element data

BEAM

The BEAM record defines a FEM beam element.

The dependency level of this data record is **two**.

Formulation 1	**Formulation 2**
BEAM: elemID: int node1: int node2: int material: int geoID: int vecID: int ecc1: int ecc2: int	BEAM: elemID: int nodes: [node1, node2] material: int geoID: int vecID: int eccentricities: [ecc1, ecc2]

Attribute name	Description
elemID	User defined external element number
node1, node2	External node definition numbers
nodes	External node definintion numbers given on list form. *This attribute is mutually exclusive with the attributes node1 and node2.*
material	Reference to external material number
geoID	Reference to external geometry identification number
vecID	Reference to direction vector. If omitted, a default direction is assigned.
ecc1, ecc2	Reference to eccentricity location vectors. If omitted, a default value is assigned.

Table A.5: Name and description of the attributes that constitute a BEAM

Note: elemID, node1, node2, material, and geoID are mandatory. vecID, ecc2, and ecc2 may be omitted.

TRISHELL

The TRISHELL record defines a triangular shell element.

The dependency level of this data record is **two**.

<table>
<tr><th>Formulation 1</th><th>Formulation 2</th></tr>
<tr><td>
<pre>
TRISHELL:
   elemID: int
   node1: int
   node2: int
   node3: int
   material: int
   secID: int
   vecID: int
   ecc1: int
   ecc2: int
   ecc3: int
</pre>
</td><td>
<pre>
TRISHELL:
   elemID: int
   nodes: [node1, node2, node3]
   material: int
   secID: int
   vecID: int
   eccentriciteies: [ecc1, ecc2, ecc3]
</pre>
</td></tr>
</table>

Attribute name	Description
elemID	User defined external element number
node1, node2, node3	External node definition numbers
nodes	External node defininton numbers given on sequential form. *This attribute is mutually exclusive with the attributes node1, node2, and node3.*
material	Reference to external material number
secID	Reference to external (composite) section identification number
vecID	Reference to direction vector defining local X direction. If omitted, a default direction is assigned.
ecc1, ecc2, ecc3	Reference to eccentricity location vectors. If omitted, a default value is assigned.
eccentricities	Reference to eccentricity location vectors on sequential form. *This attribute is mutually exclusive with the attributes ecc1, ecc2, and ecc3.*

Table A.6: Name and description of the attributes that constitute a TRISHELL

NOTE: If material is specified, secID **must** be equal to -1 and vice versa.

QUADSHELL

The QUADSHEL record defines a quadrilateral shell element. The dependency level of this data record is **two**.

Formulation 1	Formulation 2
QUADSHEL: elemID: int node1: int node2: int node3: int node4: int material: int secID: int vecID: int ecc1: int ecc2: int ecc3: int ecc4: int	QUADSHEL: elemID: int nodes: [node1, node2, node3, node4] material: int secID: int vecID: int eccentriciteies: [ecc1, ecc2, ecc3, ecc4]

Attribute name	Description
elemID	User defined external element number
node1, ..., node4	External node definition numbers
nodes	External node definintion numbers given on sequential form. *This attribute is mutually exclusive with the attributes node1, node2, node3, and node4.*
material	Reference to external material number
secID	Reference to external (composite) section identification number
vecID	Reference to direction vector defining local X direction. If omitted, a default direction is assigned.
ecc1, ..., ecc4	Reference to eccentricity location vectors. If omitted, a default value is assigned.
eccentricities	Reference to eccentricity location vectors on sequential form. *This attribute is mutually exclusive with the attributes ecc1, ecc2, ecc3, and ecc4.*

Table A.7: Name and description of the attributes that constitute a QUADSHEL

NOTE: If material is specified, secID **must** be equal to -1 and vice versa.

Eccentricity

The eccentricity record is defined with the ECCENT keyword.

The dependency level of this data record is **null**.

Formulation 1	**Formulation 2**

ECCENT:
  eccID: int
  eX: double
  eY: double
  eZ: double
	ECCENT:
	  eccID: int
	  eXYZ: [eX, eY, eZ]

Attribute name	Description
eccID	User defined external vector number
eX	X component of the eccentricity offset vector
eY	Y component of the eccentricity offset vector
eZ	Z component of the eccentricity offset vector
eXYZ	X, Y, and Z components of the eccentricity offset vector, on sequential form. *This attribute is mutually exclusive with the attributes eX, eY, and eZ.*

Table A.8: Name and description of the attributes that constitute an ECCENT record

Vector

A vector is defined by one of the following keywords: UNITVEC, ZVECTOR, or YVECTOR. Formulation 1 and Formulation 2 shows the syntax for defining a vector with the UNITVEC keyword, but the two alternative keywords have analogous definitions.

The dependency level of this data record is **null**.

Formulation 1	Formulation 2

`UNITVEC:`
`  vecID: int`
`  x: double`
`  y: double`
`  z: double`
	`UNITVEC:`
	`  vecID: int`
	`  xyz: [double, double, double]`

Attribute name	Description
vecID	User defined external vector number
x	X component of the unit vector
y	Y component of the unit vector
z	Z component of the unit vector
xyz	X, Y, and Z components of the unit vector, on sequential form. *This attribute is mutually exclusive with the attributes x, y, and z.*

Table A.9: Name and description of the attributes that constitute a vector record

These vectors are normalized upon parsing. Thus, they do not have to be unit vectors in the input data record.

A unit vector is used to denote element orientation. Elements with a reference to a vector (i.e. a vecID attribute) is oriented according to the vector direction.

UNITVEC and ZVECTOR defines local z-axis for beams, while YVECTOR defines local y-axis.

vecID must be unique across all vectors, i.e. across UNITVEC, ZVECTOR, and YVECTOR.

A.1.3 Material data

ISOMATERIAL

This record defines an isotropic material.

The dependency level of this data record is **null**.

Formulation 1	**Formulation 2**
ISOMATERIAL: matID: int type: string Emod: double poisson: double yield: double density: double thermX: double	No alternative formulation.

Attribute name	**Description**
matID	User defined external identification number
type	String literal identifying elasticity/plasticity. The value of this field must be either ”elastic” or ”plastic”.
Emod	Young’s modulus
poisson	Poisson’s ratio
yield	Yield stress. This attribute can only be specified for plastic materials. **This is a mandatory attribute for plastic materials.**
density	Material density
thermX	Thermal expansion coefficient

Table A.10: Name and description of the attributes that constitute an ISOMATERIAL record

A.1.4 Cross-sectional data

PIPE

The PIPE record defines a pipe-shaped cross-section.

The dependency level of this data record is **null**.

Formulation 1	**Formulation 2**

`PIPE:`
`  geoID: int`
`  D_o: double`
`  T: double`
`  shearY: double`
`  shearZ: double`
	No alternative formulation.

Attribute name	**Description**
geoID	User defined external identification number
D_o	Outer diameter of the pipe
T	Thickness of the pipe wall
shearY	Shear area faxtor of Y-axia. $Shear_area = shearY * A_y$
shearZ	Shear area faxtor of Z-axia. $Shear_area = shearZ * A_z$

Table A.11: Name and description of the attributes that constitute an PIPE record

TSECTION

The TSECTION record defines a T-profile cross-section used by for instance BEAM elements.

The dependency level of this data record is **null**.

Formulation 1	Formulation 2
TSECTION: geoID: int H: double T_web: double Width: double T_top: double	No alternative formulation.

Attribute name	Description
geoID	User defined external identification number
H	Height of profile
T_web	Thickness of the vertical part
T_top	Thickness of the top flange

Table A.12: Name and description of the attributes that constitute an TSECTION record

PLTHICK

This record is used to define the thickness of shell elements (plate thickness).

The dependency level of this data record is **null**.

Formulation 1	Formulation 2
PLTHICK: geoID: int T: double	No alternative formulation.

Attribute name	Description
geoID	User defined external identification number
T	Thickness of plate (shell element)

Table A.13: Name and description of the attributes that constitute an PLTHICK record

PLCOMP

This record is used to define a composite cross-section for shell (plate) elements.

The dependency level of this data record is **one**.

<table>
<tr><th>Formulation 1</th><th>Formulation 2</th></tr>
<tr><td><pre>
PLCOMP:
  secID: int
  z0: double
  plies:
   - matID: int
     T: double
     theta: double
   - ...
</pre></td><td>No alternative formulation.</td></tr>
</table>

Attribute name	Description
secID	User defined external identification number
z0	Starting coordinate for the first laminate layer
plies	Ordered list of plies, starting with the bottom ply. This list must have at least one entry, but can otherwise be of arbitrary length. Each entry (ply) must have matID, T, and theta defined.
matID	material ID for the ply
T	Thickness of the ply
theta	Orientation of the ply (in degrees)

Table A.14: Name and description of the attributes that constitute an PLCOMP record

A.1.5 Load commands

NODELOAD

The NODELOAD record defines a concentrated load on a NODE.

The dependency level of this data record is **one**.

Formulation 1	Formulation 2
NODELOAD: loadCaseID: int nodeID: int fx: double fy: double fz: double mx: double my: double mz: double eccID: int	No alternative formulation.

Attribute name	Description
loadCaseID	User defined external load case number
nodeID	Reference to external node on which the NODELOAD is acting
fx, fy, fz	Translational forces in X-, Y, and Z-directions. If omitted, the corresponding translational force is set to zero.
mx, my, mz	Concentrated moments about X-, Y-, and Z-directions. If omitted, the corresponding moment is set to zero.
eccID	Reference to external eccentricity location vector

Table A.15: Name and description of the attributes that constitute an NODELOAD record

LOADCOMB

The LOADCOMB record represents a combination of loads. It contains a collection of load cases, mapped to a scaling factor for each load case.

The dependency level of this data record is **two**.

Formulation 1	**Formulation 2**

<pre>NODELOAD:
  loadCombID: int
  factors:
    lci1: f1
    lci2: f2
    lci3: f3
    #and so on...</pre>
	No alternative formulation.

Attribute name	**Description**
loadCombID	User defined external load combination number
factors	Mapping from load case reference to scaling factor. *This attribute must contain at least one load case ID and one scaling factor*
lc1, lc2, ...	Load case identification numbers (integers).
f1, f2, ...	Scaling factors for all loads of load case lc (double floating points)

Table A.16: Name and description of the attributes that constitute an LOADCOMB record

Appendix B

YAMLParser - Developer's guide

Overview

YAMLParser is a software developed as part of the master's thesis ”YAML Based Input File Format For Finite Element Analysis” by Oskar Hvalstad-Nilsen at the Norwegian University of Science and Technology (NTNU). It contains a prototype program for parsing and emitting YAML based input files for Finite Element Analysis (FEA). Additionally, it contains a converter from Bjørn Haugen's input file format for FEA to the YAML based format.

The reader of this document is expected to be familiar with C++, YAML and yaml-cpp. This guide is meant to give a basic understanding of how to contribute to the development of YAMLParser, but does not contain detailed descriptions of the source code. For additional information the reader is advised to confer the Doxygen documentation, the guides to yaml-cpp (on Github), and the documentation of the YAML based input file format for FEA. For a more deep-going description of the YAMLParser software, the reader is reffered to chapter 3.2 of the masters thesis ”YAML Based Input File Format For Finite Element Analysis”.

Key resources

- Source code (see github repository)
- The documentation for the YAML based input file format for FEA of structural problems (Appendix A)
- Doxygen documentation for YAMLParser.

The github repository for YAMLParser is located at:

`https://github.com/OHvalstadNilsen/YAMLParser`

Getting started

YAMLParser is based on *yaml-cpp*. To build yaml-cpp, go to the yaml-cpp Github repository and follow the How to Build instructions. After the build, yaml-cpp must be linked to this project. In Visual Studio, this is done by adding the file path to the local yaml-cpp library both under C/C++ and Link in the project property pages. When the project is successfully linked to the yaml-cpp library, YAMLParser is ready to be built in Visual Studio.

Dependencies

YAMLParser is built with

- C++ (C++11 or newer)
- yaml-cpp (version 0.6.0)
- Doxygen (`http://www.doxygen.nl/`)

yaml-cpp is a YAML parser and emitter library for C++ matching the YAML 1.2 specification. It is an open-source software accessible on Github, licensed under the MIT License, and can be built on various platforms with CMake. The Github repository for yaml-cpp is located at `https://github.com/jbeder/yaml-cpp`.

Doxygen is used to autogenerate documentation from the source code, based on the comments.

Implementing support for new FEM data types in the parser module

Implementing support for a new FEM data type requires the following steps:

Create a class for the new data type

The class should follow the naming convention 'FEData_type', where Data_type is the name of the new FEM data type (FENode, FENodeLoad etc.). If the data type has an external user-defined identifier (ID), the class should be implemented as a child of Identifiable. The child may also be a child of one of the generic, intermediary classes such as GenericMaterial, GenericFE and so on. In this case, there is a set of functions inherited from this intermediary class.

The class must have data fields representing its attributes (e.g. FENode has fields for id, x-, y-, z-coordinates, and boundary conditions). These fields should be assigned data from the YAML::Node. If the data type has attributes with references to other data records (i.e. if its dependency level is >0), the corresponding data field should contain a pointer to the element. The pointer must be forwarded through the constructor's argument list.

Implement parse function in Parser class

The Parser class is responsible for traversing the YAML representation (graph of YAML::Nodes) and instantiate objects of the FEM data type classes based on the content of the YAML::Nodes. Logic for parsing the new data type must be implemented in Parser.cpp (and, of course, declared in Parser.h). The name of the parsing function should follow the naming convention parseData_type, where Data_type is the name of the new FEM data type. The return type of this function is `void` and the argument list should contain the `YAML::Node` representing the data record (passed by reference), and the data type name as a string, i.e. the FEM keyword. Pseuodocode for a generic parse function is presented in Code listing B.

Code listing: Pseudocode for a generic parse function

```
void Parser::parseData_type(YAML::Node& yamlNode, std::string type){
    try{
        bool exists = structure->checkData_typeExistence(yamlNode["id"].as<int>();
        //checkData_typeExistence represents the classes for checking data type existence in
    the Structure class.
        if(exists){
            throw runtime error alerting the user that the data record already exists in the
    structure object.
        }
        else{
            // Retrieve pointers to elements referenced in the attribute list.
            std::map<std::string, Identifiable*> ptr_map;
            ptr_map["data_type"] = structure->fetchDataType(yamlNode["id"].as<int>())
            //...

            //Instantiate object
            FEData_type *feDataType= new FEData_type(
            yamlNode,
            ptr_map["data_type1"],
            ptr_map["data_type2"],
            ...);

            //Add object to structure
            structure->addData_type(feDataType);
        }
    }
    catch (std::runtime_error &e){
        std::cout << e.what() << std::endl;
        logErrorMsg(e);
    }
}
```

Logic for calling the parseData_type function must be implemented on the correct dependency level (`parseDepLevelNull, parseDepLevelOne` etc.). Each function for parsing a dependency level has an iterator for traversing the YAML representation. The logic for calling the parseData_type function should be implemented inside a new `if` statement. An example is shown in Code Listing B.

Code listing: Call parse function

```
void Parser::parseDependencyLevelN(){
    for (int iterator = 0; iterator < structureNode.size(); ++iterator) {
            YAML::const_iterator it = structureNode[iterator].begin();
            std::string key = it->first.as<std::string>();

            if (key == "FEM_KEYWORD) {
                    nextNode = structureNode[iterator][key];
                    //Call parse function:
                    parseData_type(nextNode, key);
            }
```

```
        }
}
```

Doxygen documentation

The new data type class should be commented on the Doxygen format, allowing automatically generated documentation. The comment format is documented on the Doxygen home page.

Appendix C

YAMLParser - How to run

To run YAMLParser, yaml-cpp must be installed and its library must be linked to the YAML-Parser project in Visual Studio. When that yaml-cpp parameters are set correctly, the converter can be run in debug mode:

- Open the YAMLParser project in Visual Studio.
- Build the solution (in debug mode)
- Run *YAMLParser.sln*

UI for parser module

Figures C.1 to C.3 visualizes the command line UI with user interactions for an example run of the parser module:

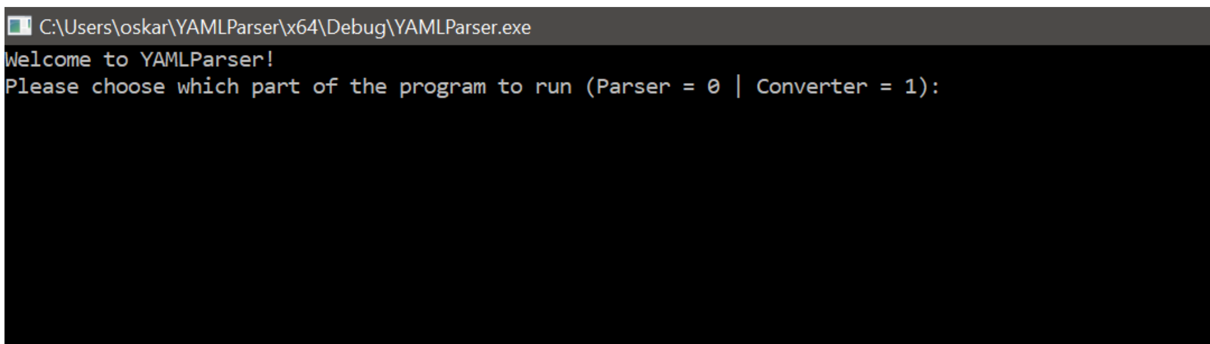


Figure C.1: Welcome screen in YAMLParser.

```
C:\Users\oskar\YAMLParser\x64\Debug\YAMLParser.exe
Welcome to YAMLParser!
Please choose which part of the program to run (Parser = 0 | Converter = 1): 0
Parser chosen.
Enter file path:
```

Figure C.2: Parser chosen. The user is asked to specify the path to the YAML input file.

When the user has specified the input file path, the parser is instantiated and the screen output is written:

```
Microsoft Visual Studio Debug Console
Welcome to YAMLParser!
Please choose which part of the program to run (Parser = 0 | Converter = 1): 0
Parser chosen.
Enter file path: short_test.yaml
CoordSys:    id: 0
1       0       0
0       1       0
0       0       1
CoordSys:    id: 1
1       0       0
0       1       0
0       0       1
CoordSys:    id: 2
1       0       0
0       1       0
0       0       1
CoordSys:    id: 3
0.267261        0.534522        0.801784
0.408248        -0.816497       0.408248
0.455842        0.569803        0.683763
CrossSection:           id: -1type: CROSS_SECTION_DUMMY
Pipe:    id: 1, D_o: 0.45, T: 0.025, shearY: 0.5, shearZ: 0.5
PLThick:    id: 2, T: 0.0025
PLComp: secID: 1, z0: 1.5, plies:
        - {matID: 1, T: 0.5, theta: 0}
        - {matID: 1, T: 0.5, theta: 90}
IsoMaterial:   id: 0, type: elastic, Emod: 1, poisson: 1, yield: 0, density: 1, thermX: 1
IsoMaterial:   id: 1, type:, Emod: 2e+11, poisson: 0.3, yield: -1, density: 8.05, thermX: 0
FENode:   id: 1, x: 0, y: 0, z: 0, ix: 1, iy: 1, iz: 1, irx: 1, iry: 1, irz: 1, rotID: 0
FENode:   id: 2, x: 0, y: 0.025, z: 0, ix: 1, iy: 1, iz: 1, irx: 1, iry: 1, irz: 1, rotID: 0
FENode:   id: 3, x: 0, y: 0, z: 1, ix: 1, iy: 1, iz: 1, irx: 1, iry: 1, irz: 1, rotID: 2
FENode:   id: 4, x: 1.5, y: 0, z: 0, ix: 0, iy: 0, iz: 0, irx: 0, iry: 0, irz: 0, rotID: 0
FENode:   id: 5, x: 1.5, y: 1.5, z: 1.5, ix: 0, iy: 0, iz: 0, irx: 0, iry: 0, irz: 0, rotID: 0
Eccentricity:    id: 0, eX: 0, eY: 0, eZ: 0
Vector: id: 0, components (xyz): 1      0       0
Eccentricity:    id: 1, eX: 1, eY: 0, eZ: 0
Eccentricity:    id: 2, eX: 0, eY: 0, eZ: 0
Vector: id: 1, components (xyz): 0.57735        0.57735 0.57735
Vector: id: 2, components (xyz): 0      1       0
FEBeam:    id: 1, node1: 1, node2: 3, material: 1, geoID: 1, ecc1: 1, ecc2: 2
FEBeam:    id: 2, node1: 1, node2: 3, material: 1, geoID: 1, ecc1: 0, ecc2: 0
FETrishell:   id: 1, node1: 1, node2: 2, node3: 3, material: 1, vecID: 1, ecc1: 0, ecc2: 0, ecc3: 0
FETrishell:   id: 2, node1: 1, node2: 2, node3: 3, secID: 1, vecID: 0, ecc1: 0, ecc2: 0, ecc3: 0
FEQuadshell:   id: 3, node1: 2, node2: 3, node3: 4, node4: 5, material: 1, vecID: 0, ecc1: 0, ecc2: 0, ecc3: 0, ecc4: 0
FEQuadshell:   id: 4, node1: 1, node2: 3, node3: 4, node4: 5, secID: 1, vecID: 1, ecc1: 0, ecc2: 0, ecc3: 0, ecc4: 0
FENodeLoad:    id: 1, loadCaseId: 1, node: 3, fx: 0, fy: 0, fz: 100, mx: 0, my: 0, mz: 0, ecc: 0
LoadComb:    id: 1 factors: 1: 2.500000,

Some numbers:
Size of coordSysList: 4
Size of materials list: 2
Size of cross sections list: 3
Size of compSectionList: 1
Size of nodes list: 5
Size of elements list: 12
Size of NodeLoad list: 1
Size of loadCombList: 1
Total number of objects: 29

C:\Users\oskar\YAMLParser\x64\Debug\YAMLParser.exe (process 4968) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

Figure C.3: Output after parsing *short_test.yaml*

UI for converter module

Figures C.4 to C.8 visualizes the command line UI with user interactions for an example run of the converter module:

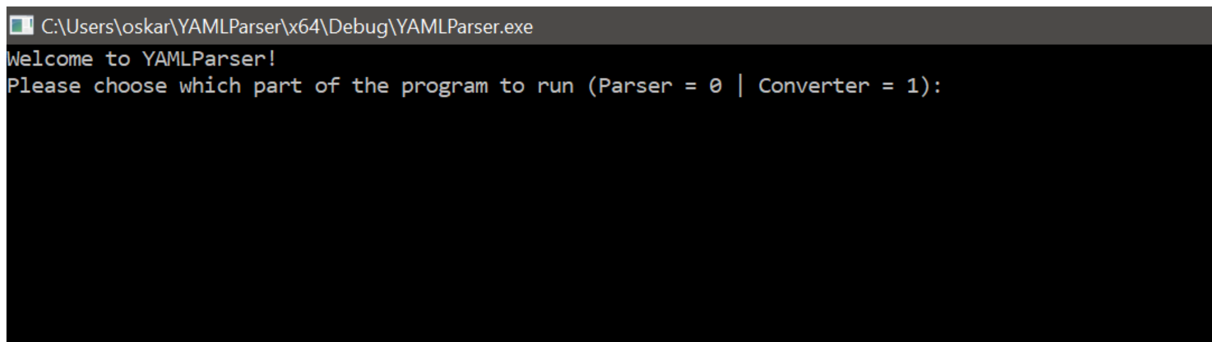


Figure C.4: Welcome screen in YAMLParser.

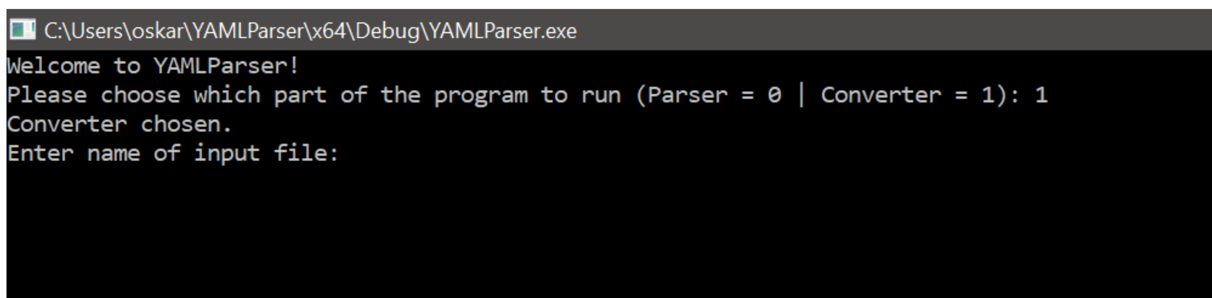


Figure C.5: Converter chosen. The user is asked to specify the path to the input file to convert.

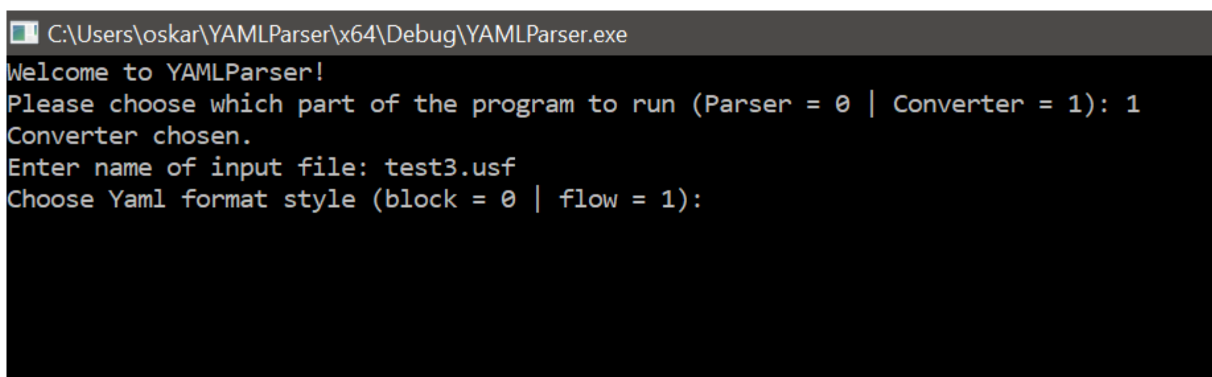


Figure C.6: When the input file path have been specified, the user must choose whether to write the output in YAML block style or YAML flow style

After choosing the YAML style, the user must specify the file name to which the YAML data will be written. If the file does not exist, YAMLParser will create it. NB: If you enter a path to an existing file, YAMLParser will overwrite the content!

```
C:\Users\oskar\YAMLParser\x64\Debug\YAMLParser.exe
Welcome to YAMLParser!
Please choose which part of the program to run (Parser = 0 | Converter = 1): 1
Converter chosen.
Enter name of input file: test3.usf
Choose Yaml format style (block = 0 | flow = 1): 0
Enter name of output file: example_output.yaml
```

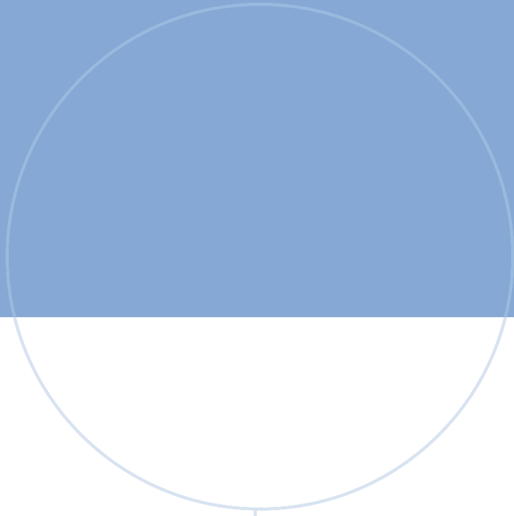
Figure C.7: Specify file path to the YAML file to which the output will be written.

Lastly, the converter module will convert the file and give indications on where it is in the process. The output after the conversion is finished is visualized in Figure C.8. The YAML file can now be found in the specified output file.

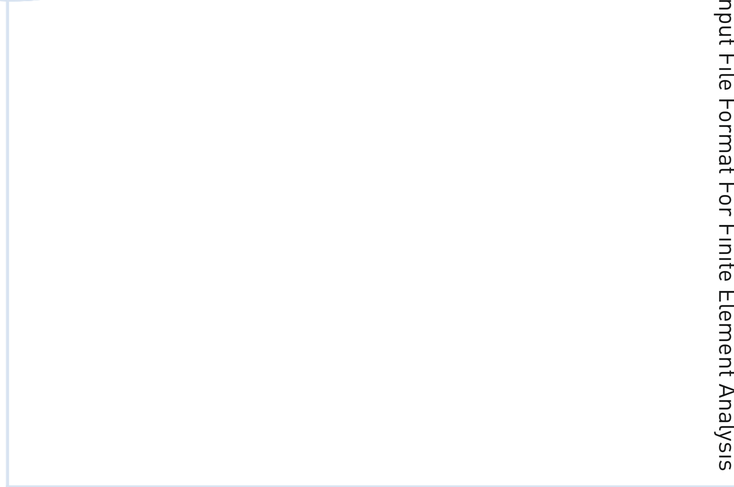
```
Microsoft Visual Studio Debug Console
Welcome to YAMLParser!
Please choose which part of the program to run (Parser = 0 | Converter = 1): 1
Converter chosen.
Enter name of input file: Test3.usf
Choose Yaml format style (block = 0 | flow = 1): 0
Enter name of output file: example_output.yaml
DataHolder instantiated.
DataHolder: Data read from 'Test3.usf'
Writer instantiated.
Writing data to 'example_output.yaml' ...

C:\Users\oskar\YAMLParser\x64\Debug\YAMLParser.exe (process 1388) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .
```

Figure C.8: Output indicating the conversion was successful.

Oskar Hvalstad-Nilsen



YAML Based Input File Format For Finite Element Analysis



NTNU
Norwegian University of
Science and Technology