

---

# TOWARDS THE CONTINUOUS AUTOMATED TESTING OF BUILDING DESIGN

by

SAŠA PEJIĆ

November 2021



Master thesis  
MSc. Building Engineering – Structural Design  
TU Delft

## Personal Details

**Saša Pejić**

5129702

Faculty of Civil Engineering and Geosciences (CiTG)

MSc. Building Engineering – Structural Design

Annotation: Integral Design and Management – IDM

[S.Pejić@student.tudelft.nl](mailto:S.Pejić@student.tudelft.nl)

+385919429248

## Graduation Committee

Chairman

**dr. ir. H.R. Schipper**

[H.R.Schipper@tudelft.nl](mailto:H.R.Schipper@tudelft.nl)

Faculty of Civil Engineering and Geosciences

1<sup>st</sup> supervisor

**ir. S. Pasterkamp**

[S.Pasterkamp@tudelft.nl](mailto:S.Pasterkamp@tudelft.nl)

Faculty of Civil Engineering and Geosciences

2<sup>nd</sup> supervisor

**dr. ir. S. van Nederveen**

[G.A.vanNederveen@tudelft.nl](mailto:G.A.vanNederveen@tudelft.nl)

Faculty of Civil Engineering and Geosciences

Company supervisor

**dr. ir. J.L. Coenders**

[jeroencoenders@white-lioness.com](mailto:jeroencoenders@white-lioness.com)

White Lioness technologies

## Acknowledgements

This report is the result of the work performed in the last seven months in order to obtain the Master of Science degree in Civil Engineering at the TU Delft. Overall, this project has been a great learning experience, but I would not have come as far as I have without the support of several people. I want to take this space for expressing my deepest gratitude to all these people.

First of all, I would like to thank my family for huge sacrifices in their lives and making it possible for me to study at this prestigious university. I will always be grateful for their never-ending love and infinite support.

I would like to express my warmest gratitude to my committee for the support and guidance throughout the graduation process. A sincere thank you goes to Roel Schipper for his help in starting my thesis, setting up the thesis committee and being a great chairman. Also, for his help during the two years of my master studies, during which he made himself available to help the students whenever needed. I could not wish for a better coordinator of our master track. I also want to thank Sander Pasterkamp and Sander van Nederveen for their interest in the research topic, their time and criticism which pushed me to always strive for new and better solutions. Afterwards, I would like to also thank Jeroen Coenders from White Lioness technologies for giving me the opportunity to conduct my research in the company and be part of a fantastic team which I had the pleasure working with. His feedback and supportive approach were extremely helpful. Lastly, I would like to thank Milou Klein from White Lioness technologies for her efforts during this project.

Next, I would like to thank all my friends who have been part of these incredible two years and were always here to support and motivate me. Finally, an immense thank you goes to my girlfriend Melisa for believing in me and making this journey easier and more beautiful.

*Saša Pejić*

*November 2021, Delft*

## Abstract

In recent years the AEC industry has started implementing more and more new technologies. Still, the adoption process is very slow. One of the crucial steps during the building designing phase is a justification of design according to previously defined requirements. That process is still manual to a large extent, therefore unnecessarily time-consuming and prone to errors. The need for automation of requirements compliance checking has been rising in recent years, and the science community presented various approaches for automation. Still, most of the proposed methods are overly complex and represent a black-box approach, which means that the designer can not understand how the software performs the check. Also, it requires that the designer is highly skilled in programming, which is not common. Another limitation of current methods is focus on only one type of requirements, most often geometry related. Finally, most of the concepts focus only on the first step of automated compliance checking, which is transferring requirements into computer-readable scripts. Therefore, there is a need for a new approach that can cover automation of compliance checking in general, use white box-approach so that designer can understand the process and cover different types of requirements. The rise in popularity of parametric design and tools like Grasshopper, Dynamo opens the possibility to overcome some of the limitations present in the current methods. Also, the possibility of using requirements management software to have the process fully organized supports the belief in automating compliance checking. The fundamental objective of this project is to explore the possibilities of automating the requirements verification for a building design by using requirements management software to systematically structure the requirements and Grasshopper to generate the rules, which afterwards can be verified. To achieve this objective, the project was divided into three main parts. First, the conceptual framework for automated code checking is developed, explaining each step in the process to the details. Afterwards, the system architecture of the prototype tool was explored, and instructions for scripting the tool were given. Finally, after the tool was scripted, it was tested on a real building model. During the testing, the tool showed some evident strengths, but also it has particular weaknesses.

From the literature review and analysis of Building Decree 2012, the two main types of requirements are explained, the functional and performance requirement. Since the functional requirements are qualitative and can not be quantified, that category is not suitable for checking with the approach proposed in this project. Therefore, the focus was on performance requirements. The central part of the project is the creation of a framework for automated compliance checking. The five steps are defined, and for each step is explained what it is and how to achieve it. The five steps are:

- 1.) Requirements defining and logical structuring into RMS
- 2.) Interpretation of requirements
- 3.) Building model preparation
- 4.) Checking phase
- 5.) Reporting phase

After the theoretical basis is set, the modelling of the tool is elaborated. Firstly, the requirements for the tool are set. Afterwards, the system architecture is explored, and finally, instructions for scripting the tool are developed. The tool must be fast, robust and easily extensible, which means that by following the instructions, new functions should be added easily. Furthermore, the system architecture of the tool consists of four main components: designer, computational engine, visualiser and requirements management software. Lastly, instructions for scripting the tool depends on the type of function, which can be: basic function, reporting function or method for determination.

After the prototype tool is scripted, it is tested on a real building model, and it shows clear advantages compared to other approaches or manual work, still, it also has some disadvantages. Test of the tool proved that the Visual programming language environment is a great platform for developing a white-box approach for automated compliance checking. Also, testing on the real-world building model shows that a five-step approach for automated testing of building design works and can be used. Lastly, the test shows that the proposed system architecture and instructions for scripting the tool can result in a well-operating tool.

The biggest strengths of the presented approach are speed, a wide variety of checks that can be covered and the fact that the designer does not have to be skilled in general programming. On the contrary, the most important weaknesses are the limited range of Grasshopper functions that complicate the scripting of some methods for determination. In addition, the tool is dependent only on Karamba3D models for the structural domain, and the tool requires big help from a designer to perform checks.

Finally, recommendations for future research are given, and these are:

- Continue with the research of automatically deconstructing requirements by using machine learning techniques
- Standardize the vocabulary and logic for defining requirements
- Explore the methods for the automation of the script building in Grasshopper
- Explore how to connect Grasshopper with BIM software
- Explore methods for semantic enrichment of building models
- Test the scalability of the tool with a large number of requirements
- Explore how to develop a visual detection of failed requirements in Rhino
- Explore the implementation of the automated compliance checking

## Table of contents

Acknowledgements.....	ii
Abstract.....	iii
1. Introduction.....	1
1.1. Motivation.....	1
1.2. Automated code checking – state of the art.....	1
1.3. Visual programming language.....	5
2. Problem definition.....	6
2.1. Research objective.....	6
2.2. Research question.....	6
2.3. Research Scope.....	7
2.4. Methodology.....	7
3. The structure of the building requirements.....	9
4. Framework for automated code checking.....	13
4.1. Requirements defining and logical structuring into RMS.....	13
4.2. Interpretation of requirements.....	17
4.3. Building model preparation.....	20
4.4. Checking phase.....	21
4.5. Reporting of the results.....	21
5. Modelling of the tool.....	23
5.1. Functional requirements for the tool.....	23
5.2. System Architecture.....	24
5.3. Instructions for developing Grasshopper functions.....	27
6. Developing of Grasshopper functions – examples.....	36
6.1. Distance between two objects.....	37
6.2. Flexural buckling.....	39
6.3. Functions for synchronization of Microsoft Excel and Grasshopper.....	42
6.4. Reporting function.....	44
7. Pilot study.....	46
7.1. Test-case.....	46
7.2. Results.....	48
7.3. Impressions from the testing.....	52
8. Discussion.....	55

8.1. Vision .....	55
8.2. What has been done?.....	55
8.3. Impressions – strengths and weaknesses.....	58
8.4. Assumptions and limitations .....	59
8.5. Contributions of this research .....	61
9. Conclusions.....	62
10. Recommendations.....	64
References.....	67
List of figures.....	70
List of tables.....	71
Appendices.....	72
Appendix A.....	72
A.1. Distance between objects .....	72
A.2. Flexural buckling .....	77

# 1. Introduction

Before diving into the project directly, it is necessary to explain the motivation for conducting this project and the problems of the existing solutions.

## 1.1. Motivation

The adoption of new technologies in the AEC industry is very slow. While society and other industries are taking advantage of information technology, the AEC industry struggles with the transformation (Coenders and Rolvink, 2014). For many years the two most important technologies were the Finite Element Method (FEM) software for structural analysis and Computer-Aided Design (CAD) software for communication of the design. Building Information Modeling (BIM) started replacing CAD in recent years, enabling better communication and new opportunities because models contain much more data. However, the complete process from initial designing to building the structure is not supported by appropriate technology and methodology. Although digital tools are used for a variety of tasks, the justification of the chosen design is still a very manual process. Due to the absence of integrity and digitisation of building design justification, it appears that the industry is unable to profit from the information that might be saved and explored if the process is automated and digitised. The automation of compliance checking could potentially deliver benefits in the form of financial gain, but also an increase in structural safety as automated procedures can be used to apply checks and codified experience to designs that transcend the knowledge of a single human designer (Coenders and Rolvink, 2014).

One of the emerging digital technologies in the AEC industry is parametric design using software like Grasshopper or Dynamo, visual programming languages. Programmes are created by dragging components onto the canvas instead of writing complex codes, making it more straightforward for architects and structural designers who are not skilled in classic programming. Using these technologies would be possible to digitise even the previously mentioned code checking, which is now a significant problem. Most previous researches in this field required the designer to be a highly skilled programmer or use a black-box approach without understanding what is happening during the verification process. The idea of using Grasshopper arose as a potential solution to these problems. Also, Grasshopper offers the possibility to check the requirements during the designing process continuously.

This project will explore the possibilities to automate the checking of the building design requirements by using Grasshopper.

## 1.2. Automated code checking – state of the art

The general requirements that every building must fulfil are given in documents released by the national bodies. Usually, these are building codes and decrees, which consist of many checks that have to be taken into account. Logically, the building design must be checked to



fulfil all these requirements. On top of that, there is a large number of requirements from all stakeholders. Nevertheless, even though we are in the 21<sup>st</sup> century, code compliance checking is still a very manual process.

Firstly, during the project's design stage, architects, structural designers, engineers, fire safety engineers etc., are doing manual checks of drawings and calculations. Later, after submitting all documents to authorities, the officers also do some of the checks manually. Of course, a process like that, with very limited automation involved, is error-prone and unnecessarily laborious (Preidel and Borrmann, 2016). Moreover, as the complexity of the projects is growing, the number of requirements follows that trend, resulting in a more demanding compliance checking process (Rao 2021).

Many researchers tried to find a more efficient way of building design justification through automated code checking. The software which performs automated compliance checking is not modifying the building design, it just assesses a building's design related to the requirements (Nawari 2019.). The main results of such software are a pass or a fail grade for the proposed design (Eastman *et al.*, 2009). Furthermore, a tool like that can be developed for three different platforms: a) as a plug-in for designing software, which allows the designers to do the check whenever they wish; b) as an independent application that runs parallel with a designing software; or c) as a web-based application that can accept design documents from a variety of sources, which is the most suitable for authorities (Eastman *et al.*, 2009). This project is focused on the first group, and the checking platform is imagined as a plug-in for designing software, in this case, Grasshopper.

In 2009, Eastman *et al.* wrote a paper about state of the art in automated rule-checking. In that paper, they explained the principles of automated code checking, and it has served as a starting point for almost all research projects in the following years. According to them, there are four main stages of the automated rule-checking process (Eastman *et al.*, 2009):

### **Stage 1: Requirement interpretation and logical structuring into rules**

This includes defining the requirements and afterwards transferring them into the computer-readable rules, which can be checked. Until now, that is the most complex and most researched part of the rule-checking process.

### **Stage 2: Building model preparation**

In this step, all necessary data from the building model are extracted. Until now, almost all researchers have been using Industry Foundation Classes (IFC) models. That is because it is independent and supported by all BIM design tools.

### **Stage 3: Rule execution phase**

In this step, the previously prepared data from the building model are compared to the rules from stage 1.

### **Stage 4: Reporting of results**

This step is essential because all checks must be documented. Also, this is the least complex part of the process.

These four steps are extended and serve as a basis for developing a framework in this project. In recent years many researchers have been trying to come up with a general solution for automation of the compliance checking. It is difficult to provide a complete solution to the problem due to the wide extent of the rules involved (Solihin, Dimiyadi and Lee, 2019). All proposed solutions are based on defining a Domain Specific Language (DSL), which would be able to express the domain requirements into computer-readable rules. The internal DSL and external DSL are the two main categories in which all projects could be separated (Solihin, Dimiyadi and Lee, 2019).

For the purpose of the internal DSL, some already existing language is used as a basis, and it is then extended and adapted to get a domain-specific language. The most important internal DSL are: Semantic Web Rule Language (SWRL), in which a series of semantic web triplets is used to define a rule (W3C, 2004); Lua script, which is being used in a commercial application FORNAX to program rules by using its APIs based on C++ (NovaCityNets, 2016); SPARQL Inferencing Nation (SPIN) is another internal DSL which combines query languages, rule-based systems and concepts from object-oriented languages to define rule language based on the Semantic Web and rule-based system such as DROOLS; in this category also fall mvdXML (Chipman, Liebich and Wiese, 2015), BPMN (Dimiyadi et al., 2016), LegalRuleML, RuleML (Solihin, Dimiyadi and Lee, 2019), which are all languages based on XML. The wide variety of resources and available tools is an advantage of being a part of the larger community and standardized languages, which encourages the use of internal DSL's. Contrary, those base languages are not developed specifically for the construction domain and BIM, and that imposes extra work to twist or extend the language to fit BIM needs. This results in many additional steps and very weird syntax in the end (Solihin, Dimiyadi and Lee, 2019).

On the other hand, the external DSL is designed explicitly for the purpose of BIM base code checking, and therefore it is usually more compact, concise and feels more natural in defining the rules. The main problem of external DSL is that the user has to learn new syntax, and it requires its own custom parser (Solihin, Dimiyadi and Lee, 2019). In the literature, several external DSL's can be found: BERA (Lee, 2011); KBIM that uses meta-programming concept (Park, Lee and Lee, 2016), QL4BIM (Daum and Borrmann, 2013); BIMRL (BIM Rule Language) that combines built-in support for spatial operators, SQL based query language and function extensions (Solihin, 2015); the approaches that use visual programming languages can also be classified as external DSL. On top of Domain-Specific Languages, there are other tools proposed for automating the rule checking, such as Natural Language Processing (NLP) that uses AI to analyze the requirements and transfer them directly into computer-executable rules (Zhang and El-Gohary, 2017); there is also manual markup method used in RASE, which analyzes the semantics of requirements and transforms the rules into IFC constraint model (Hjelseth and Nisbet, 2011). The comparison and specific limitations of all the methods mentioned above can be found in (Solihin, Dimiyadi and Lee, 2019).

As is already mentioned, the basic way of translating rules into a code checking system is manual implementation by software developers. Unfortunately, that results in inaccessibility to codes for third parties, which means it can not be verified. This is called a "black box"

approach (Nisbet *et al.*, 2008). The problem with that approach is a lack of trust from the domain experts and dependence on software developers, making it less responsive to changes in regulatory documents. Nevertheless, the majority of proposed automated code checking systems use the black-box approach. One of the first and most known is CORENET e-PlanCheck from Singapore, which is able to check compliance with fire-safety rules and barrier-free access (Solihin, 2004). The main component of the system is the FORNAX library which has been developed and maintained by a private company (Preidel and Borrmann, 2016). Therefore, a detailed look at the checking process is restricted, which is already described as the biggest setback. Another example is the Solibri Model Checker, which is also the most advanced and used application for code checking. Most of the proposed methods use an ontology-based approach for the representation and checking of the rules. Still, that approach has many constraints: primary, as it is previously mentioned, the translation of the rules into an ontological representation is too complex for domain experts; secondary, the ability of the resulting systems to represent semantically higher constructs is very restricted. So, some complex geometric and topological regulations, which usually take a big portion of codes, cannot be checked (Preidel and Borrmann, 2016).

Consequently, there is a need for the development of a code representation language which is:

- 1.) Easy to use for domain experts who are not software developers
- 2.) Capable for the processing of non-geometric but also complex geometric requirements

(Preidel and Borrmann, 2015) were among the first who tried to overcome these issues by introducing Visual Code Checking Language (VCCL). Instead of using the classic approach, they used graphic notation to represent the rules of a code in a human and machine-readable language. They presented the syntax and semantics of its major components. Of course, many issues must be solved in future work. The usual critics of Visual Programming Language are that more complex rules will result in an unclear and messy script, which can not be read by the user anymore. To face that, the authors applied few instruments like the nesting approach, the control flow elements and the embedded UI controls. A few years after, Korean authors (Kim *et al.*, 2017) developed the KBim Visual Language (KBVL) by using a similar approach. They first analyzed sentences from regulation documents and classified the components of each sentence into three categories: (1) building objects, (2) methods for checking and (3) reference and delegation information according to sentence relation (Kim *et al.*, 2017). The visual components of KBVL are systematized using that semantic structure. Also, the components pool of building-related items and methods is defined, which reduce errors that may happen if the user must define methods manually.

By analyzing the literature, it can be seen that the focus is on developing a platform that will be able to check geometry related requirements. The requirements about structural safety, energy efficiency and other types do not get a lot of attention.

### 1.3. Visual programming language

In recent years, Visual Programming Languages have been on the rise in the field of building design. The definition of a visual language would be “a formal language with a graphical notion”. In other words, instead of textual codes, the modular system of signs and rules is developed by using visual elements (Myers, 1990; Hils, 1992; Schiffer, 1998). The graphical notation makes the interpretation of the codes much easier and faster for humans, and proof for that can be found in cognitive psychology, which says that for the processing of the visual information brain can use two hemispheres instead of only one. Furthermore, the complex processing structures are presented as a flow of information, that is why it is usually said that Visual Programming Language has a flow-based nature (Preidel and Borrmann, 2016).

The most widely used Visual Programming Language software in the building design industry are the plug-ins Grasshopper for Rhinoceros3D and Dynamo for Revit. These were initially developed for 3D parametric modelling, but their capabilities extended significantly by the activity of a third-party community that is creating many additional plug-ins. The fact that Dynamo is operating only in the Revit environment gives a big advantage to the Grasshopper in the field of code checking. The openBIM approach is very desirable to allow all designers to work with the tool, regardless of the BIM software they use.

The Visual Programming Language environment can solve the problem of the black-box approach in automated code checking, which is necessary to get the trust of the designers. If the tool for automated code checking is developed correctly, the user should be able to understand and inspect every step of the process. Knowing that the designer is responsible for his design, it is particularly important to get a complete insight into the compliance checking process. In reality, that makes the process semi-automated instead of fully automated, but for now, it is inevitable to involve the user at some steps of the compliance process.

## 2. Problem definition

After detecting the problems in the automated compliance checking field, the plan for the project has to be set. This chapter discusses the research objective, research question, scope, and methodology used.

### 2.1. Research objective

As explained in previous sections, code compliance still depends to a large extent on manual checking by designers or authorities. Therefore, there is a strong need to find a solution for automated compliance checking. The rise of Visual Programming Languages popularity between architects and designers opens the opportunity of using it for revolution in requirements verification. In addition, the possibility to use the requirements management software supports the belief for the possibility of improvement.

Therefore, the fundamental objective of this project is to explore the possibilities of automating the requirements verification for a building design by using a requirements management software to systematically structure the requirements and Grasshopper to generate the rules, which afterwards can be verified.

### 2.2. Research question

The main research question of this research project is:

**“How can manual verification of design requirements be automated by using a requirements management software and Grasshopper?”**

To answer this question, a few sub-questions will be asked.

1.) Which requirements should building design fulfil?

Before focusing on specific requirements, it will be looked into all requirements that building design should fulfil. For that, it is necessary to look into the Building Decree 2012. After defining all requirements that building design has to fulfil, the project will focus on verifying only a few requirements to prove the concept.

2.) How to approach automated testing of building design?

This question is driving the development of a conceptual framework for automated compliance checking. The particular steps have to be defined, and then specific procedures and instructions inside each step must be examined.

3.) How can Grasshopper verify the requirements?

After structuring the requirements in requirements management software, it must be figured out what is the best way to generate the script in Grasshopper which can perform the checks.

#### 4.) How does the tool work in practice?

After the tool is finished, it should be tested to prove the concept. Sweco will provide the test case.

After answering these four sub-questions, it should be possible to answer the main question and make a prototype script that can check the codes for a given design.

### 2.3. Research Scope

Since the verification of requirements is an extremely broad topic, the project's scope and restrictions have to be set to stay on the path during the whole research process. The main goal of this thesis is to deliver a framework for automated compliance checking and proof of the concept tool, which is not a fully developed tool ready for wide use. Also, many possible features are discussed, which could be added in the future, but not all of these are implemented in this project. The implemented ones show the possibilities, but the fully developed tool would not be limited to only these functions. This project is not focused particularly on one of the four steps of automated compliance checking, which was the typical approach in other research projects. Most of these projects focused on the analysis of requirements and developing specific methods for transferring the requirements into computer-readable rules. Since many papers are already related to that specific step, this project scope is zoomed out and focused on a more general approach to the whole process. The specifics of this project is adding verification tracking and abilities of Grasshopper, as a Visual Programming Language, to the scope. Another unique detail of the project is not focusing on geometric requirements only but including other types such as structure related rules. Some scientists have focused already on structural or energy-related requirements, but in a very isolated manner and not combined with all other categories. Finally, the prototype tool has particular limits introduced due to time restrictions, but it does not affect the project's validity. The main limitations of the prototype are that it works only on steel structures and the ability to check seven specific requirements. At the end of the project, all these limitations are discussed to determine the impact of each one.

### 2.4. Methodology

This section presents the methodology used in the project to fulfil its objective and answer all research questions. The section will describe only the main steps. More details can be found in the previous Chapter 2.2. under the title Research question. The project is mainly divided into three parts:

- **Phase 1:** Defining the **framework** for automated code checking. First, it is important to understand the problems related to automating the compliance checking. In order to retrieve this data, it is necessary to do a literature study. After the issue is explored, then by brainstorming, the potential solution can be derived in the form of a conceptual framework for automated rule checking. Therefore exploration of the building requirements (sub-question 1) and development of the framework for automated compliance checking (sub-

question 2) are in focus during Phase 1. This part aims to standardize and semi-automate the requirements management part of the process, which precedes the verification. Also, the goal is to set the theoretical foundations for modelling the prototype tool.

- **Phase 2:** Developing the system architecture and instructions for scripting the **verification tool** in the Grasshopper and finally scripting it (sub-question 3). This is the central part of the project where automation will become a reality. Here, a mix of research-led and design-led approaches will be used. The first one is following the classic scientific methods, trying to analyse everything in the deep and then developing solutions, while the second approach is driven by proposing a solution and taking users as partners involved in the design process. The project will be used to determine what users require from the tool, and then a proposal for the framework can be deduced. Afterwards, during the scripting of a prototype tool, the focus will be on a design-led approach, which will be used to improve the tool iteratively in response to users feedback.
- **Phase 3: Testing** the proposed framework and scripted tool to examine its effectiveness in practice and propose recommendations for future applications. The tool will be tested on a real project provided by Sweco (sub-question 4).

The research strategy is summarized in the figure below:

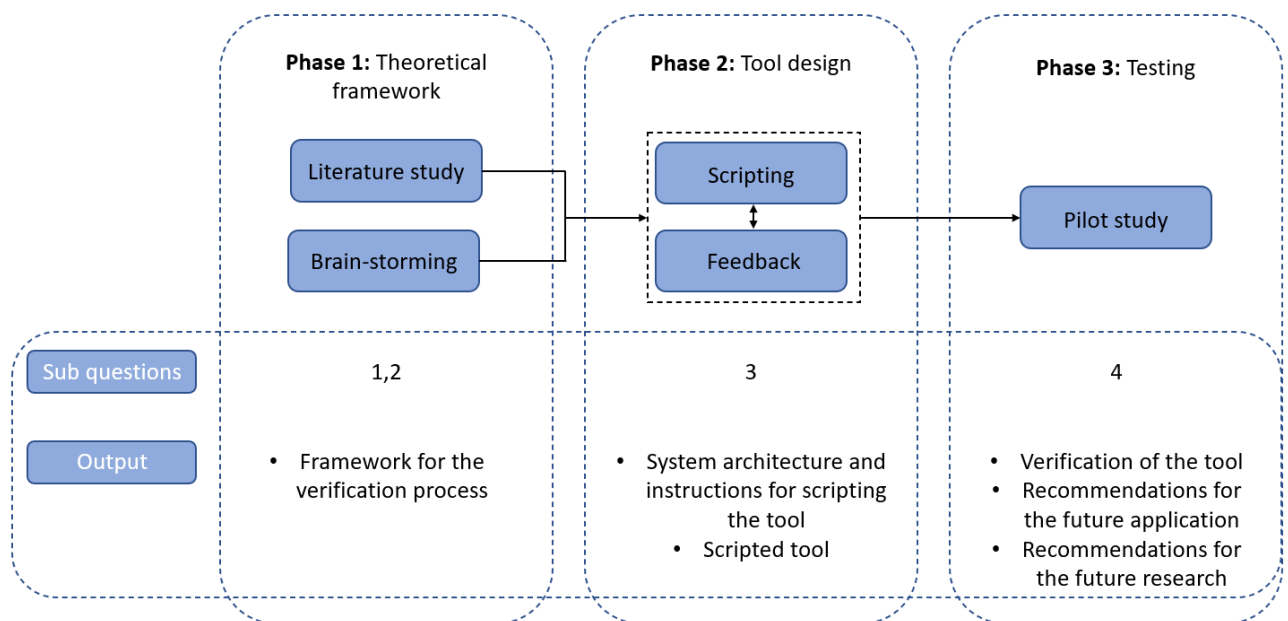


Figure 1 Research strategy and relation with sub-questions and output

### 3. The structure of the building requirements

Every building has to fulfil a large number of requirements given from various sides. During the project, there are two main groups of requirements according to the timing of a project. First, while the building is still only a concept on paper, it must fulfil design requirements. Later, when the construction starts, it must meet construction requirements. In Figure 2, this can be seen.

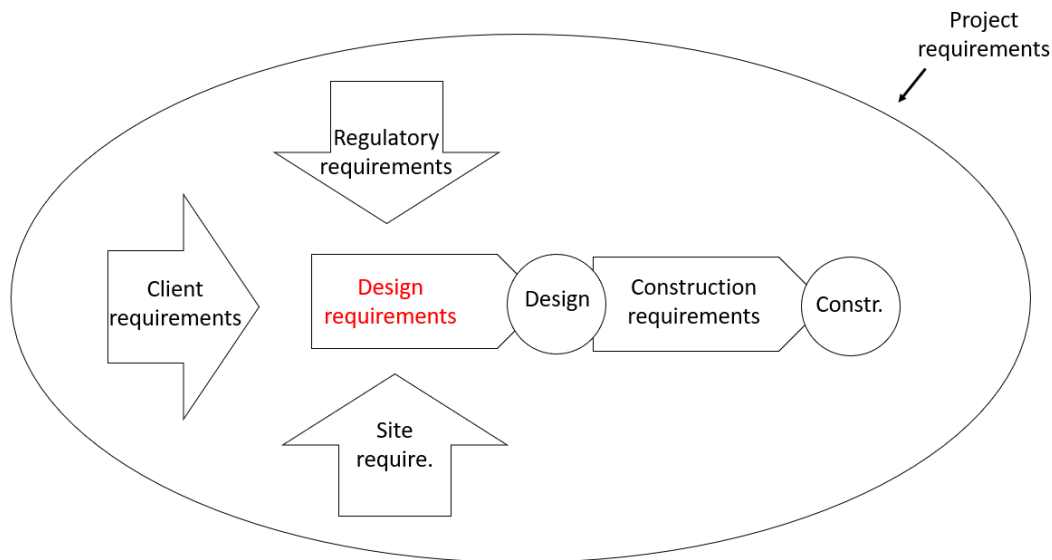


Figure 2. Project requirements (Kamara, Anumba and Evbuomwan, 2000)

The design requirements are in the interest of this project. These are the requirements for design which are a translation of the client needs, but also site and regulatory requirements. Table 1 summarizes all the requirements and their description.

Table 1. The types of requirements in a project (Kamara, Anumba and Evbuomwan, 2000)

Type of requirements	Meaning
Client	Requirements of the client that describe the facility that satisfies their business need. Incorporates user requirements and those of other interest groups.
Site	These describe the characteristics of the site on which the facility is to be built (e.g. ground conditions, existing services, etc.).
Environmental	These describe the immediate environment (e.g. climatic factors) surrounding the proposed site for the facility.
Regulatory	Building, planning, health and safety regulations, and other legal requirements that influence the acquisition, existence, operation, and demolition of the facility.
Design	These are the requirements for design which are a translation of the client needs, site and environmental requirements. They are expressed in a format that designers can understand and act upon.
Construction	These are the requirements for actual construction that follow from the design activity.



This project will zoom more deeply into design requirements, mainly from the regulatory and client group. The way in which building codes are given is one of the obstacles to the automation of compliance checking. For instance, regulatory documents often reference other sources, which means they are not self-contained. Also, codes often refer to knowledge that all experts should be familiar with, but such data is not always represented in a formal expression. Furthermore, a deep understanding of the field is required from the user of codes (Nawari and Alsaffar, 2015). On top of that, heuristics and experience are required to decide when to look into other referenced standard and when to proceed based on presumed compliance. To summarize, the negative attributes of the building provisions are (Nawari, 2018):

- Inconsistent usage of terminologies
- Subjectivity
- Exceptions, various interrelationships, and complexity of code structuring

Every country has its own law referring to a supreme regulatory document which prescribes the mandatory conditions that every building design must fulfil. In the Netherlands, that is the Building Decree 2012. Any work related to refurbishing, building, demolishing, or occupying a building, must comply with the Building Decree 2012. This decree contains the technical regulations that represent the minimum requirements for all structures in the Netherlands. Decree usually refers to specific codes in which are given methods to determine performance. For example, how to exactly determine strength, acoustic isolation, fire-safety etc.

The way in which the regulations are given in Building Decree 2012 can be divided into:

- The form of the regulations
- The object level of a rule

The regulations can be given in three different forms, related to their subjectivity and way of verifying. The first type is functional requirements, which are qualitative and indicate which goals the building has to fulfil, but without giving a concrete way to do it or how to measure the compliance. One example is: “*A proposed structure shall be such that fire and smoke cannot develop quickly*”. (Building Decree 2012, Article 2.68)

The second form is performance requirements, which are more detailed and quantifiable. It consists of a quantified limit value and unambiguous determination method, which often refers to some other codes. An example of a performance requirement is: “*The part of a side of a structural component which adjoins outdoor air and is located higher than 13 m shall comply with fire class B as determined in accordance with NEN-EN 13501-1.*” (Building Decree 2012, Article 2.70). The first part of the sentence gives the limit, while the second part refers to the code describing the determination method.

The third form is presence requirements, which are in fact performance requirements indicating that a certain “something” must be present for a particular use function.

Furthermore, the important aspect of the regulations is the object levels for which the conditions are given. These objects are 'bodies' that have to fulfil requirements. The following object levels can be distinguished:

- 1.) Parcel (consisting of buildings, open yard and grounds)
- 2.) Construction work
- 3.) Usage functions
- 4.) Space
- 5.) Construction element
- 6.) Installation
- 7.) Material
- 8.) Act of usage (such as placing furnishing elements and storing goods)

By analysing the Building Decree, seven main groups of requirements can be extracted, of which each has few subgroups. These are:

- 1.) **Safety**
  - 1.1) The strength of a building
  - 1.2) The fire safety
  - 1.3) The safety in use
  - 1.4) Burglar resistance
- 2.) **Health**
  - 2.1.) Soundproofing
  - 2.2.) Moisture resistance
  - 2.3.) Ventilation
  - 2.4.) Harmful conditions
  - 2.5.) Daylight
- 3.) **Usability**
  - 3.1.) The residential area and living space
  - 3.2.) Sanitary areas
  - 3.3.) Building accessibility
  - 3.4.) Outdoor storage and outdoor space
  - 3.5.) Installation location for the sink, cooking, heating, and hot water appliance
  - 3.6.) Parking space for bicycles
- 4.) **Energy efficiency and environment**
  - 4.1.) Energy performance
  - 4.2.) Thermal insulation
  - 4.3.) Airtightness
  - 4.4.) Environmental performance limit
- 5.) **Installations**
  - 5.1.) Presence of artificial lighting
  - 5.2.) Electricity, gas and heat supply
  - 5.3.) Water supply
  - 5.4.) Sewage system
  - 5.5.) Fire safety installations
  - 5.6.) Accessibility of building for disabled

- 5.7.) Preventing common crime
- 5.8.) Safe maintenance of the building
- 6.) **Use**
  - 6.1.) Fireproof use
  - 6.2.) Safe use of escape routes
  - 6.3.) Other provisions
- 7.) **Build and demolish**
  - 7.1.) Construction work procedure
  - 7.2.) Demolition work procedure
  - 7.3.) Safety and limitation of a nuisance for the environment

The first five groups are related to design requirements, while the sixth describes the proper use of the building, and the seventh is related to safety during construction works.

Another important group of requirements for this project are the ones provisioned by the client. Those usually follow some established methods to compose design brief, but which differ per company. Often, the semantics of the requirement do not have any particular logic that is followed while defining it. Usually, they are expressed in human language, based on experience and adaptability of the human mind, but that enhance the complexity of transforming requirements into computer-readable rules.

Another possible classification of requirements is given by Nawari (Nawari, 2018), and it has four main categories:

- 1.) Conditional clauses – easy to transfer into formal rules directly from the textual document. Requirements that are quantifiable and have all standard features.
- 2.) Contents clauses – requirements that cannot be transformed into False or True expressions. Those rules are normally utilized for descriptions and definitions, for example, the definition of high-rise building, firewall, smoke evacuation etc.
- 3.) Ambiguous clauses – subjective clauses, which usually have words such as *about*, *close to*, *relatively*, *maybe* etc.
- 4.) Dependant clauses – complex requirements that are consisted of more sub-requirements. One section or sub-requirement is reliant on compliance of other clauses.

As mentioned in the project's scope, the type of requirements and how to automatically transfer each clauses category into a computer-readable script is not explored in depth. There has been a significant amount of research on that topic, but here only the general idea of input requirements into requirements management software manually will be given. Furthermore, this project will cover performance requirements only because it would not be possible to verify the functional requirements by using a Visual Programming Language. The main reason for that is the inability to quantify the functional requirements.

## 4. Framework for automated code checking

In this chapter, the conceptual framework for automated code checking is explained. This framework describes the general idea of the proposed method. As a basis for developing a framework, four steps of automated code checking presented by Eastman et al. (Eastman et al., 2009) are used. The process is a little bit extended, and one extra step is introduced, therefore the five steps are:

- 1) **Requirements defining and logical structuring into RMS**
- 2) **Interpretation of requirements**
- 3) **Building model preparation**
- 4) **Checking phase**
- 5) **Reporting phase**

The difference between the framework proposed in this project and Eastman's is introducing the defining of requirements and concept of requirements management software in a story. The first step of Eastman's framework is a requirements interpretation and logical structuring into rules. This step is divided into two, and in the new first step is added requirements management software which enables verification tracking during the whole process. Also, requirements defining, collecting, and grouping is addressed in that first step. Approach from this project emphasises the entire process of compliance checking from defining the requirements to checking it and tracking the verification, which is addition on other projects that only focus on the pure checking part.

Five sub-chapters indicate five steps of the automated rule checking process.

### 4.1. Requirements defining and logical structuring into RMS

The first step of the process is divided into two sub-steps. Firstly, requirements have to be defined and collected, afterwards in the second sub-step, they are input in requirement management software in a logically structured way.

#### 4.1.1. Requirements defining

Requirements defining and logical structuring is the first step of automated code checking, but in fact it cannot be fully automated. A certain level of manual effort is required from the designer. Collecting all requirements has to be done by a designer in charge of that specific requirements group. Afterwards, all these requirements should be grouped and put in some requirements management software, and there are two main reasons why the specific separation is necessary.

The first reason is to be better organized and make the verification process easier to track. If only one infinite list is used, the designer cannot focus only on the part he is interested in, and some conditions can be skipped easily.

The second reason why it is necessary to have a few groups of requirements is to make the structure of the Grasshopper tool clearer and more user friendly. Different groups use different functions for the verification. For instance, some conditions need to check the distance between two objects, and some check whether the beam's bending strength is sufficient. The structure of the scripts that must be built in Grasshopper for checking these two are different, so the grouping of the requirements can be done in a way that similar verification functions are used.

Designing a building is a very complex process in which many roles are involved. In this project, four specific roles are used for explaining the concept of grouping the requirements. Those four roles are: the architects, the structural engineers, the building physics related designers and designers for services. Following that separation of work, it would be logical to group the requirements in the same way. So, the groups are: architectural requirements, structural requirements, building physics related requirements and services requirements. By using these four groups, each designer is only working with a list of conditions that he is interested in, and others are not disturbing him. Also, the Grasshopper tool can then have four independent parts, of which each can be adapted to the specifics of the group. Still, each part will be based on the same framework and use some basic functions shared by all of the groups. Moreover, each category will perform checks on different object levels, which require different building model preparation process per category, but that will be explained in the third step. It is important to mention that these groups can be separated differently in an actual project, maybe based on some other roles, but the concept of grouping to keep the process more organized is the same. Still, the proposed separation should be able to cover a large number of requirements because the chosen roles are the standard and main ones in every project. In this project it will be shown that different types of requirements can be covered, by using examples from architectural and structural domain. Examples of other requirement groups will not be explored in deep due to time restrictions.

The first group is architectural requirements, which are mainly focused on the spatial configuration of the building design. From the name, it is evident that the architects have the main interest in this group, but the functions defined in this group will be helpful for others as well. So, this serves as a base category, which everyone should be able to use. Some of the functions needed to check code compliance for this category are the minimum and maximum distance, length, height, area and similar geometric related functions. The one example for this category could be: *“A bathroom space as referred to in Article 4.18. shall have a floor area of at least 1,6 m<sup>2</sup> and a width of at least 0.8 m.”* (Building Decree 2012, Article 4.19(1)).

The second group is structural requirements which consist of all the conditions prescribed for the load-bearing structure. Consequently, all roles related to the load-bearing structure should use this category. These could be structural engineers, structural designers, fire-safety engineers and even façade designers when designing the load-bearing structure for the façade. The functions used here are specific to the content and are more in calculation forms like bending resistance, shear or any similar check depending on the chosen material. The main requisite for structures from the Building Decree is given in a functional form, and it is: *“A load-bearing structure shall not collapse, during the designed useful life referred to in NEN-EN 1990, under the fundamental combinations of loads as referred to in NEN-EN 1990.”*

(Building Decree 2012, Article 2.2.). But it refers to Eurocodes in which are given methods to quantify the requirements and assess the performance of the design.

The requirements related to building physics form the third category. Usually, there are five building physics fields: Heat, Air, Moisture, Light and Acoustics, and this requirement category is reserved for experts from these areas. Therefore, the functions needed to check code compliance are very diverse, and some of them are energy performance coefficient, lighting intensity, minimum noise protection and many more. One representative of this group can be: *“An exterior partition of a staying area, toilet space or bathroom space shall have a heat resistance as determined in accordance with NEN 1068 at least equal to the value given in Table 5.1.”* (Building Decree 2012, Article 5.3(1)).

In the last group are assigned all requirements related to services. The electricity, heating and water system designers are the target group for it. The functions required for code checking are very diverse to be able to cover those complex fields. Some of the functions needed for the Grasshopper tool are nominal pressure, presence of emergency lights, voltage requirements and many more. The example of provision from Building Decree is: *“A proposed gas supply shall comply with: NEN 1078 for a nominal operational pressure not exceeding 0.5 bar and NEN-EN 15001-1 for a nominal operational pressure exceeding 0.5 bar but less than 40 bar.”* (Building Decree 2012, Article 6.9(1)).

In the following Table 2 is shown the summary of analysis based on roles who are using them and the type of functions that are needed.

Table 2. The characteristics of different groups of requirements

	Architectural req.	Structural req.	Building physics req.	Services req.
Roles using it	Architect	Structural engineer Fire safety designer	Façade designer Climate designer Acoustics designer	Electricity designer Water supply designer Heating system designer
Type of functions needed for verification	Spatially related functions: Min. and max distance Length Height	Structural calculations: Bending resistance Shear resistance Torsion resistance	Energy performance coeff. Thermal resistance Lightning intensity Minimum noise protection	Presence of emergency lights Nominal pressure Voltage requirements

#### 4.1.2. Logical structuring into requirements management software

The process of putting the requirements into requirements management software is going parallel with grouping and collecting. Regardless of which requirements management software is used, the type of data that has to be input is always the same. So, there is a need for a standardized table or template for input to ensure that all the necessary data are collected. The standardized table should have seven main columns, which represents:

### 1.) Object

This is the “substance” that has some property that has to be checked. In Chapter 3 are listed all object levels or types of objects that can be distinguished from the Building Decree. For example, the object can be a whole building whose area should be checked or a concrete beam with sufficient bending strength. In the case of the presence requirements, the object is the thing that must contain the property.

### 2.) Property

The property is an attribute or quality of an object that is being checked. In other words, it has to match some predetermined value or range of values. For instance, if a building must have more than 100 m<sup>2</sup> of area, then the area is the property that has to fulfil the requirement of 100 m<sup>2</sup>. In case of presence requirements, the property is a thing that has to be contained in an object.

### 3.) Limiting operator

The limiting operator determines the required relation between the property and the limit value. There are five basic limiting operators: Greater than (>), less than (<), equal (=), greater than or equal to ( $\geq$ ), less than or equal to ( $\leq$ ).

### 4.) Limit value

The limit value is a target value that the property has to fulfil.

### 5.) Unit of measurement

The unit of measurements is fundamental to define to get correct results. A number without a unit of measure does not have any meaning. In the case of the presence requirement, the unit of measurement is “property”. So, for example, if the rule is that the kitchen must have two power outlets, then the power outlet is property but also the unit of measurement.

### 6.) Determination method

The determination method is an extra column that is not necessary for all requirements. For example, if the area has to be checked, there is no need to specify the determination method because that is something fundamental and understandable to everyone. But, if the shear strength of a concrete beam has to be checked, then some document in which the method is explained should be provided. In this case, the EN-1992-1-1. This is important for the building of a code in visual programming language environment later.

### 7.) Conditions

The column with the conditions is necessary to have in order to define more complex requirements, which are constructed from two or more requirements that have specific relations. It is essential to decompose the one big requirement into a set of more small ones and then, after putting them in requirements management software, connect it with AND, OR, IF and THEN operators. For instance, in a requirement: *”a floor which is more than 13 m above an adjacent floor, grounds or water, shall have a floor partition with*

a height of at least 1.2 m as measured from the floor., (Building Decree 2012, Article 2.8(2)) it is first necessary to check one condition before checking the main requirement. So, here the condition is that floor is more than 13 m above an adjacent floor, ground or water, which has to be defined as a requirement which IF is TRUE THEN the core requirement of the article should be checked. In this case, that would be is the height of the partition at least 1.2 m.

The example of a template is shown in Table 3 in which is analysed the Article 2.8(2) from Building Decree 2012.

Table 3. Template for the input of requirements into requirements management software

Rule nr.	Object	Property	Limiting operator	Limit value	Unit of measurement	Determination method	Conditions
1.1	Floor	Distance from ground	≥	13	m	The distance measured from the ground, water or adjacent floor	
1.2	Floor partition	Height	≥	1.2	m	Height measured from the floor	IF 1 TRUE, THEN check

## 4.2. Interpretation of requirements

The second step in an automated code checking process is the interpretation of the requirements. After collecting all of the requirements, it is necessary to build the computer-readable code in a visual programming language, which is then used to perform the checks. This step is very complex to automate, therefore it will be divided into two phases. The first phase is still manual to a large extent because the designer has to build the code by using previously defined functions in a Visual Programming Language environment. After the science develops more reliable methods based on natural language processing or some other of the proposed principles, the code building can be fully automated. So that would be the second phase.

### 4.2.1. Phase 1: Manual code building

In the first phase, the “scripting” part will be still done manually by a designer who wants to check specific requirements. Scripting, in this case, means dragging onto a canvas and connecting functions that a software developer has already defined. This means that the designer must understand the logic of the requirement, which functions to use and in what order to connect them. This approach can be classified as a white-box approach, but it becomes a black-box at some point. While a designer can see all functions and the flow of the checking process, which is the characteristic of the white-box approach, at the same time, he must use previously defined functions that he cannot manipulate. So, from that, it is visible that the



black-box approach is necessary for simplicity at some point. For instance, to check the maximum distance between two objects, the function maxdistance has to be developed, which is a very complex process, and the designer is not interested in it. He must have it ready for use. Of course, the designer must trust these previously defined functions. Figure 3 shows graphically how the deeper level of one predefined function looks like, and it represents the part inaccessible to the designer.

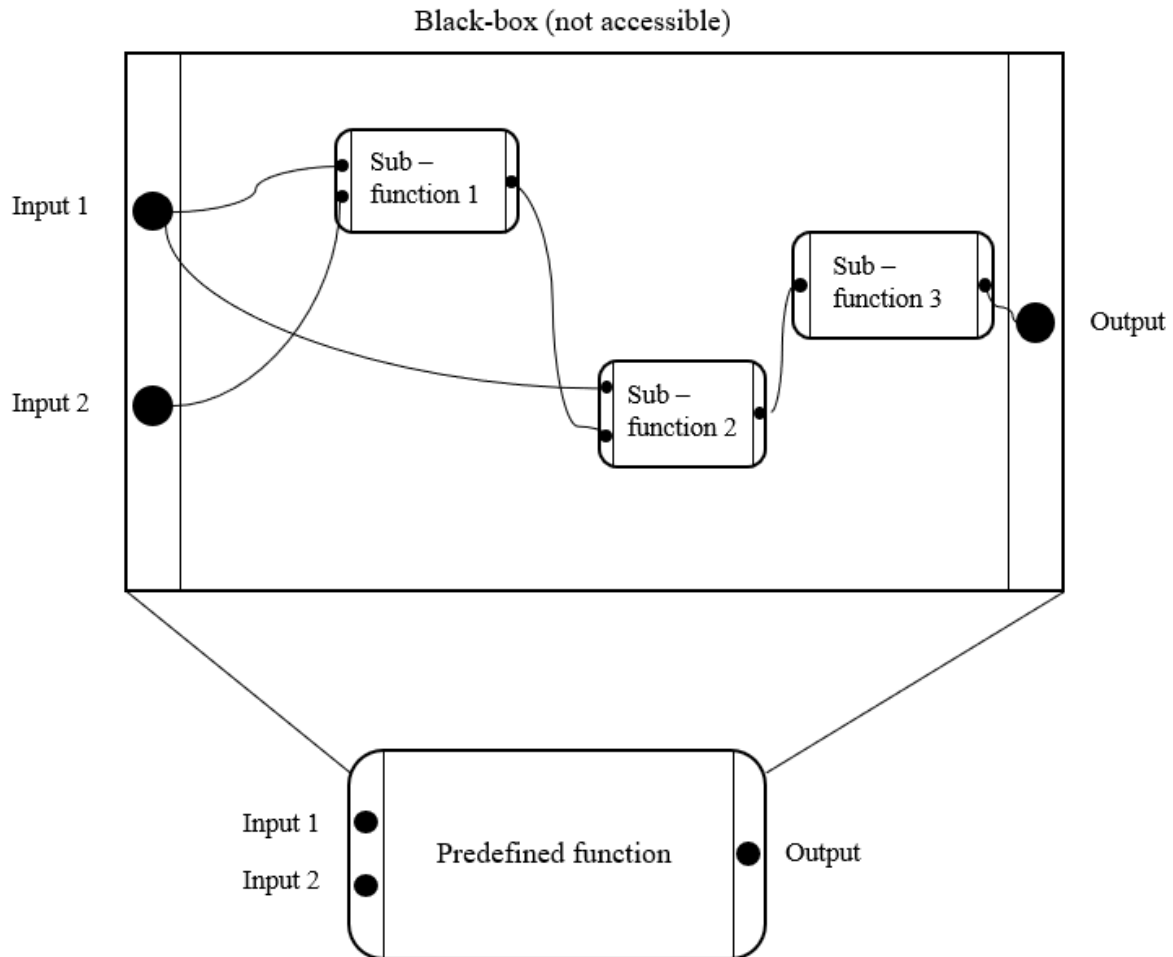


Figure 3 Internal structure of predefined function which is not accessible to designer

To have a well functioning tool, it is necessary to predefine a few categories of functions:

- **Methods for determination (role-specific):** This category covers the role-specific methods for determining compliance with the codes. The four presented groups of requirements use very diverse determination methods, so the functions must be developed in cooperation with a specialist from specific roles. Also, the structure of these functions and how they operate are very different, so it is desirable to have four divided parts of the Grasshopper tool, each for one requirement group. For example, the structural part needs functions that can calculate and check the bending strength of the beam, or deflection, while building physics related part needs a function for the thermal resistance of the wall. Furthermore, more geometric related functions are needed for the architectural group.

- **Reporting functions:** The functions that can be used for generating stand-alone reports; usually PDF outside of the Grasshopper environment
- **Basic functions:** It is necessary to have basic methods which enable the proper functioning of the tool. These functions are mainly already defined in Grasshopper and serve for calculations, defining objects, properties, limits and relations. These include:
  - Logical functions: operators such as *Boolean values, And, Or, If, Then*
  - Mathematical functions: basic calculation operators (*Plus, Minus, Product, Division*) and set operators (*Union, Intersection, Complement, Difference*)
  - Geometric-Topological functions: spatial predicates returning a Boolean value including topological predicates (*Equal, Disjoint, Touch, Overlap, Contains, Within*), geometric predicates (*CloserThan, FartherThan*) and directional predicates (*Above, Below*); geometric evaluation operators (*ShortestDistance, MaximalDistance...*) (Borrmann, Hyvärinen and Rank, 2009)
  - Objects: Functions for defining objects are important for the building model preparation stage. The designer must be able to extract and define an object whose properties have to be checked.
  - Property: Property functions are similar to object functions and have the same purpose. The only difference is that these are the subject of checks.
  - Limiting operators: The basic limiting operators are already defined in the Grasshopper environment.

If the designer has all of these functions already defined, he should just follow the logic of the requirement and connect the appropriate functions to perform the check. The specific instructions and examples are given in the second part of the research.

The logic and development process of all these functions is too broad for consideration in this research. Only a few functions required for the test case will be explained more deeply in the following chapters.

#### 4.2.2. Phase 2: Automated code building

The second phase of the script building would be automated, which means that software would be able to read the tables from requirements management software and then generate the script for code checking.

For automatic translation, many concepts are developed, and the most recent ones are based on semantics and syntax analysis of codes which are then implemented in natural language processing (NLP) scripts. That type of software is still not developed to a level when it can operate without problems. The main reason for that is the nature of the requirements, which are written in human language, and to a large extent, based on experience and adaptability of the human mind. Therefore, there is no standardized specific logic behind all requirements,

making it very hard to implement it in some natural language processing software. Furthermore, even when science succeeds in overcoming these obstacles, it is important that the designer can act and manipulate the script in Grasshopper, enabling the white-box approach. The functions used are the same as in the previously explained manual process.

### 4.3. Building model preparation

Before performing the checks, all required information must be extracted from the building model. In the past, while 2D drawings were still the main subject of communication, the most crucial property of the drawings was that they must be visually correct and to have included various information needed for rule checking. After the development and spread of BIM-based software, this has changed. The objects that are being checked now consist of more data, like type and properties. For instance, an object that looks like a staircase but is defined from small slabs will not be interpreted as a staircase in software. It has to be converted to a stair object by assigning some of the stair properties such as riser, tread, run etc. Therefore, the building models nowadays must fulfil much stricter prerequisites than earlier drafts (Eastman *et al.*, 2009).

The building model preparation process in a Grasshopper tool has to be developed in two phases. In the first phase, only Grasshopper building models can be used, while in the second phase, the tool should be able to handle inputs from a variety of BIM software which opens much more space for usage. In this project focus will be on phase 1.

#### 4.3.1. Phase 1: Checks on Grasshopper models

The first phase would be to work only on models that are being developed in a Grasshopper environment. That kind of tool is easier to make because it does not need to communicate with a variety of other software to get all the required data. It is possible to make a tool that only uses Grasshopper models due to a large number of plug-ins for Grasshopper, which allows a different type of analysis of the building design.

Depending on the requirement group, building model preparation differs significantly. Also, the separate model views should be used to derive the specific data and to extract required elements or objects, which is also proposed by Han et al. (Han, Kunz and Law, 1998). Almost all research until now followed this approach.

In general, there are three model preparation steps:

- 1.) Acquiring data from the already existing model – The first step in the building model preparation is acquiring data from an already existing model. These are the general data about objects and their properties related to geometric, materials, quantities etc.
- 2.) Assigning new data to existing models – In some cases, it is necessary to give extra input to the model to make it suitable for performing specific checks. For example, the use functions of the room are maybe not specified in the original model, and they are necessary for some checks. Furthermore, sometimes it is necessary to define borders between some objects, rooms, use functions etc.

- 3.) Performing additional analysis for role-specific checks – This step requires different approaches for a different group of rules. For instance, to check the design's structural integrity, it is necessary first to run some FEM software and get the data like stresses and deformations. Only after the collection of this information it is possible to perform the checks. A variety of plug-ins for Grasshopper have to be used for this part of the building model preparation. Of course, each role uses its own plug-ins, making this step hard to tackle without input from role experts.

#### 4.3.2. Phase 2: openBIM – checks on models from different BIM sources

In the second phase, the Grasshopper tool should be enriched with the possibility to communicate with various third-party software and check the models made outside of its environment. As in phase 1, these software differ per specific requirements group and roles. Some of the basic software that should be involved are BIM-based programs for architects, like Revit, Allplan; structural FEM software like Diana, Robot, Scia, etc. For the other two groups, software most used by the designers should be included as well. One of the leading problems is proper data transfer from software to software. All these uses different data languages and structures, which makes the connection difficult.

#### 4.4. Checking phase

The checking phase is a stage in which the Grasshopper script is run, and the computer is performing calculations to get the results. Manual work is not present here, and the designer only has to wait for the report.

The checks can be performed whenever the designer wishes to verify something, it is only important to go through the first three steps of the process to ensure that all required data are prepared. So, this could be used during the designing process to verify the chosen solutions immediately, but of course, it can be used only in the end as well, which is more suitable for regulatory bodies.

The designer or officer must have complete trust in this step, so the functions and the tool must be adequately tested. Just a tiny mistake in the checking phase could ruin the tool's reputation.

#### 4.5. Reporting of the results

The last step of the automated rule checking process is a reporting of the results. Satisfactory design conditions must be separated from those that failed to fulfil the limits.

There are some general rules that the reporting part of the tool must have. In the report, specific objects and properties must be related to the requirement, so if the condition is not fulfilled, it can be detected which object missed the goals. For example, there are many beams in the building, so the one that fails must be easily indicated. The tool should have the option to place the camera view on the object that did not satisfy the requirement, which improves the effectiveness of the communication.

Furthermore, the specific design and look of the report depend on the requirements group which is being reported. Also, it depends on reporting requirements from a regulatory body to a large extent. It may even be necessary to show the calculation steps and assumptions used in a process for some provisions. Therefore, the specific report design must be developed in collaboration with role-specific designers and according to regulatory requirements for the reports.

## 5. Modelling of the tool

Before the actual modelling of the tool, it is necessary to define a scripting procedure and structure of the functions. This chapter starts with describing the objectives and requirements which the tool for automated testing of building design must fulfil. Afterwards, all parts of system architecture are explained, how they work and communicate among themselves. The final chapter gives instructions for developing the functions of the tool in Grasshopper's environment.

### 5.1. Functional requirements for the tool

The first step in developing a tool is setting up the functional requirements that a well-functioning tool must have. Here are shown only general functional requirements. Many more detailed requirements are specific to a certain part of the tool, but these are not in this project's scope. For example, when the real tool will be developed, there will be conditions on how the report should look like, which checks the tool must be able to perform etc.

#### **Speed**

One of the crucial properties that the tool for automated testing of building design must have is a reasonable speed to perform the checks. The definition of reasonable time spent on checks changes depending on the purpose of the checks and who is performing them. Suppose the checks are performed only at the end of the designing process, like in a case of a regulatory body. In that case, the speed can be lower than if the designer uses the tool for continuous checking of building design. The speed of the checks is also important when a large number of checks has to be verified because considerable computational power is required for it. Many techniques can be used to increase the speed of the script, and that will be elaborated in one of the next chapters.

#### **Robustness for the input data**

The robustness of the tool is an important property that a developer must strive for while developing the tool. This means that the functions should be able to work with different types of data. More specifically, the tool should be able to perform checks on many types of buildings with different types of materials and geometric shapes. For instance, the function for verification of minimum distance between two objects must perform the check no matter are these objects defined in a Grasshopper as curves, surfaces or points. More precisely, the tool must not be dependent on a particular type of data that is being input.

## Extensibility

The tool must be structured in a way that the database of functions can be extended easily following the same logic and without making a fundamental modification in a tool's structure. If the designer understands the proposed framework and logic of the tool, he should be able to add new functions for verification and then, if necessary, develop even the functions specific for the design that is being verified.

## 5.2. System Architecture

The system architecture of the tool for automated testing of building design consists of four main parts. The central part is a computational engine operating in a visual programming language environment, which serves as a tool for building and running scripts for verification. The next important component is the designer, who is the user of the tool and serves as an intelligent manual force to support the work of the computational engine because the latter is not able to acquire and process all necessary data on its own. Another component of the system is the requirements management software which serves for collecting and structuring all requirements. Finally, the visualiser makes the last segment of the system. It serves as a support for communication between designer and computational engine. In Figure 4, the conceptual map of system architecture is shown.

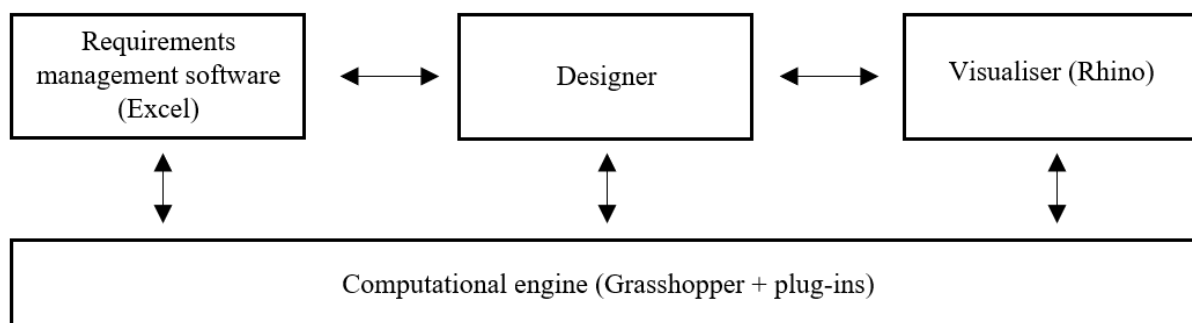


Figure 4 Four elements of a system architecture

All these four fundamental segments are repeatedly used during the verification process. To completely understand how the tool functions, it is necessary to zoom in on each of the five steps proposed in the framework and analyse which of the basic segments and software are used. Also, the data flow between steps must be examined.

### 1. Requirements defining and logical structuring into RMS

The two main components that operate in the first step of the process are designer and requirements management software. The designer's responsibility is to collect all the requirements and input them in requirements management software by using a standardized table. The data put in requirements management software are words, numbers, and relational operators, representing the meaning of the requirement. The specific categories used in the

table are explained in Chapter 4.1.1. For this project, Microsoft Excel served as an improvised requirements management software. The initial idea was to use Briefbuilder as requirements management software, but since the API between Grasshopper and Briefbuilder has not yet been developed, it was left for later research. The requirements management software is used for storing, organizing requirements and tracking the verification process. After the requirements are put in requirements management software, they are sent to the computational engine, which is, in this case, visual programming language-based software Grasshopper. In this project, data as rule number, limiting value, limiting operator and units are sent automatically from the Excel table to Grasshopper. If the Briefbuilder had been used, the table would have been converted into a json file that can be manually imported in the Grasshopper environment. Later, in phase 2, the API between requirements management software and the computational engine should be developed to ensure automated and continuous communication between system architecture components.

## **2. Interpretation of requirements**

In the second step, the communication between the designer and the computational engine takes place. The designer uses predefined functions to interpret data from the requirements management software table and make a logical script that is performing verification. This step relies on the designer's intelligence and ability to interpret the logic of the requirement correctly. Also, the computational engine reads the data obtained previously from requirements management software to get the limiting values. The limiting values are expressed in number format, both float and integer. For some checks, the limiting value is not expressed in quantified value immediately, therefore the function has to calculate it. In that case, the function has already a built-in method for the determination of the limiting value. An important remark is to ensure that units of measurements are the same in the requirements management software table and Grasshopper model.

The main functions that the designer has to connect in Grasshopper in this step are the method for determination, relational arguments between two requirements in case of the complex requirement and assigning the appropriate number to the requirement. The input of the requirement number in the method for determination must match a certain number in requirements management software. By assigning that number, the function can get the limiting value, limiting operator and units from the requirements management software table. After choosing an appropriate method for determination, the designer has to define which data must be prepared and acquired in the next step of the process

## **3. Building model preparation**

In the building model preparation step, the designer communicates with the computational engine, more precisely with Grasshopper and its plug-ins, to input all necessary data in the method for determination function. In this step, different types of data flow through the system. As previously explained in Chapter 4.3.1, the building model preparation consists of three steps. The first step is acquiring data from the already existing model when the designer selects the objects from the Grasshopper model and then connects it to the method for determination



function, which can read its properties. The second step is assigning new data to an already existing model, and here the type of data that is input depends on the check. It can be some geometric property, integer, float, string etc. For example, to verify the bending resistance of the beam, the designer must input the appropriate partial factor for materials. Lastly, the third step is performing additional analysis for role-specific checks. This step is necessary to perform to get the data that the tool is not able to calculate on its own, therefore it needs help from a designer. The type of data obtained from this step varies per check, but most often, it is a number that describes the checked property and is compared to the limiting value afterwards. For instance, to verify the already mentioned bending resistance, the tool must get data from structural analysis. The structural analysis should be performed during the designing phase, so it should already exist in the script. Therefore, the designer does not have to make it specifically for the verification process but only to connect appropriate data from it to the method for determination function.

After all necessary data are prepared by the designer and sent to the method for determination function, the tool is ready for performing verification.

#### **4. Checking phase**

The checking phase is a step in which is involved only the computational engine. It processes all acquired information and performing the checks. The most important property of this step is to have a reasonable operational speed. After the checks are completed, the Grasshopper script in this step produces a string with the value Pass or Fail, which indicates compliance with the requirement. Subsequently, the string with compliance is being sent to the last step of the process.

#### **5. Reporting phase**

The last step in the automated code checking process is a reporting of the verification process. As previously described, this step depends a lot on the purpose of the report and the body that issues it. In general, the computational engine in this step gets the string with a Pass or Fail value related to the requirement number. Then it has to process and structure the compliance in a way that suits the designer the best. In some requirements, additional data are sent from step 4 to step 5, such as calculations for structural checks, but the processing part is similar. The computational engine also has to communicate with the visualiser to make the indication of the failed objects easier.

Another aspect that is interesting for the future is communication between computational engine and requirements management software. After the Grasshopper performs the checks, it should also send the string with Pass or Fail value to the requirements management software to have the fully automated process. This is the main reason for choosing MS Excel as a requirements management software because it is simple to synchronize it with Grasshopper. On the contrary, if Briefbuilder had been used, then communication between Grasshopper and Briefbuilder would not have been possible due to missing API.

The visualisation of the software used and data flow between steps is shown in Figure 5.

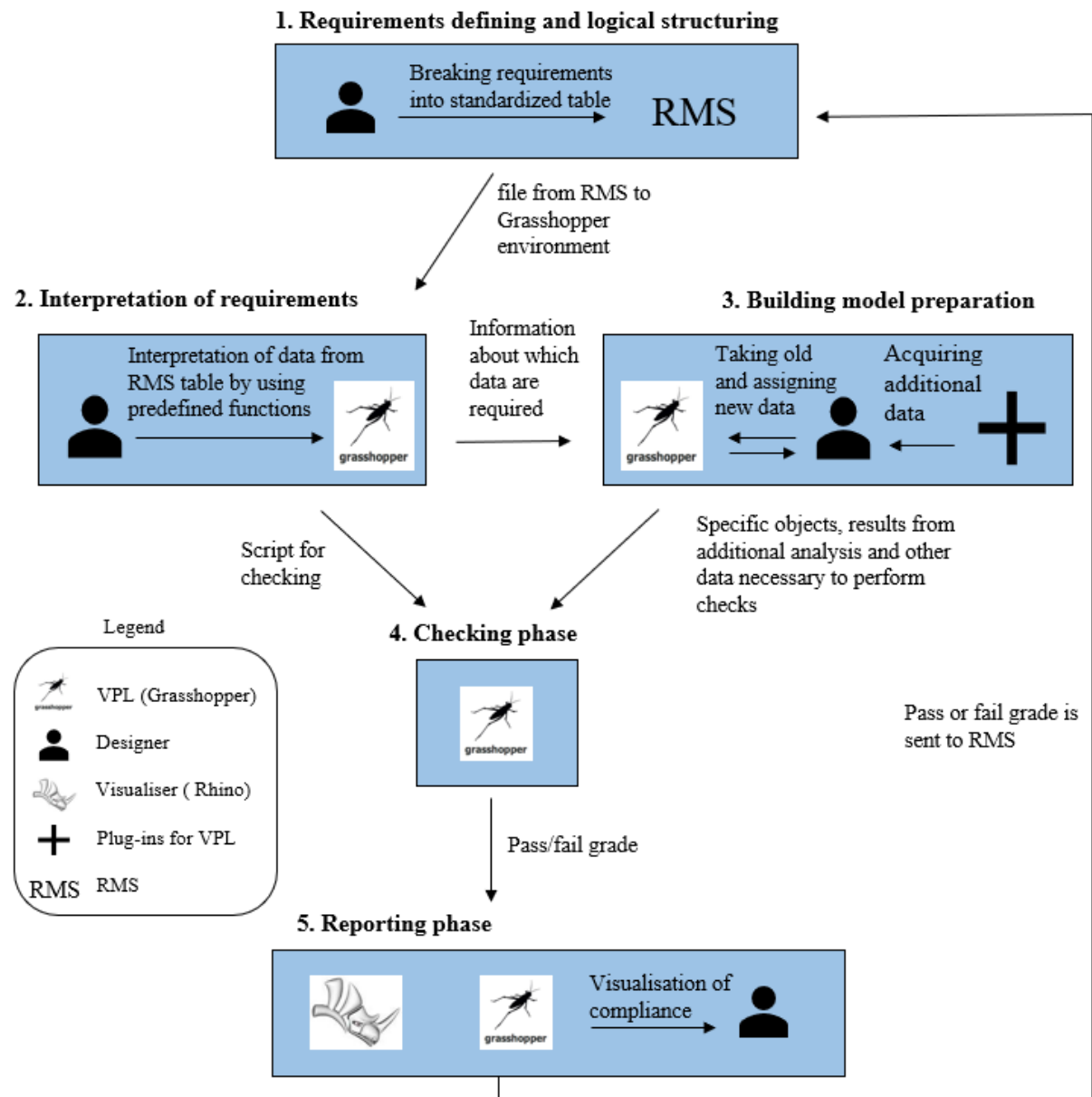


Figure 5 The components used and data-flow per each step of the process

### 5.3. Instructions for developing Grasshopper functions

In Chapter 4.2.1, the three groups of predefined functions that are necessary to be developed for the code checking process are given and explained. These are: Methods for determination, Reporting functions and Basic functions. As is already mentioned, these functions have to be developed by a software developer and ready for use by a designer.

The fact that requirements do not have any standardised logic and differ a lot per requirements group makes it very hard to define standardised instructions or template how to develop these methods for determination. Nevertheless, there are some general rules applicable to all

requirements that must be followed to make a tool with sufficient speed, robustness, and extensibility. The instructions for making predefined functions will be given per each function group.

### 5.3.1. Basic functions: Objects, Properties, Limiting operators etc.

The first group to be observed are basic functions that contain objects, properties, limiting operators and other logical, mathematical and geometrical functions. They will be explained together at once because these functions are mostly already available in the Grasshopper environment. Therefore, they do not have to be developed specifically for this project. Grasshopper has an extensive library of functions divided into few categories: Params, Maths, Sets, Vector, Curve, Surface, Mesh, Intersect, Transform. These categories contain all necessary functions to select objects, define relations between them, manipulate the lists, set limiting operators etc. In other words, all data preparation and selection can be made with functions from Grasshopper. The functions for defining and calculating the value of properties are also already in Grasshopper or in plug-ins for it. As an example, it can be used the standard check for bending resistance of the steel beam. In order to verify the check, bending action has to be compared to the bending resistance of the beam. In this case, bending action is a property that has a particular value. The value can be derived from the structural analysis made by some of the plug-ins for Grasshopper, like Karamba or Kiwi. This example gives an insight into how plug-ins for Grasshopper can be used to define properties and their values in the script.

The functions for synchronization of requirements management software and Grasshopper script can also be classified as basic functions. It is crucial to have two-way communication between requirements management software and Grasshopper to ensure continuous verification tracking. Those functions are highly dependent on the requirements management software used, and API with Grasshopper must be developed. Since this is not part of the compliance checking, further instructions will not be given for it, and it is left for programmers and developers of requirements management software to work on it.

Another essential function to have is the one for connection of complex requirements. The concept of complex requirements is explained in Chapter 4.1.2. That function can be scripted using GHPython. It just has to check if the output of the first sub-requirement is True or False and subsequently send the appropriate information to the next check, which depends on the compliance of the previous one. Also, this function must store the data about all sub-requirements to have it in one place, enabling generating the report fast. But, function for complex requirements is not in the focus of this project because no complex requirements are used, therefore it has not been scripted nor explained to the details.

Considering all of that, it can be concluded that the designer who is performing the checks must have a good understanding of the Visual programming language environment, especially its functions and logic of operating script.

### 5.3.2. Methods for determination

The central part of the verification script is a method for determination. That is a predefined function made separately and uniquely for each check. All other methods are used to collect data and feed the method for determination, which is then processing that data, performing calculations, and finally comparing the real and limiting value.

Figure 6 shows the concept of the method for determination component in the Grasshopper environment. On the left side are inlets that serve to input the data into the component, while on the right side are outlets that send out the processed information after the check is performed.

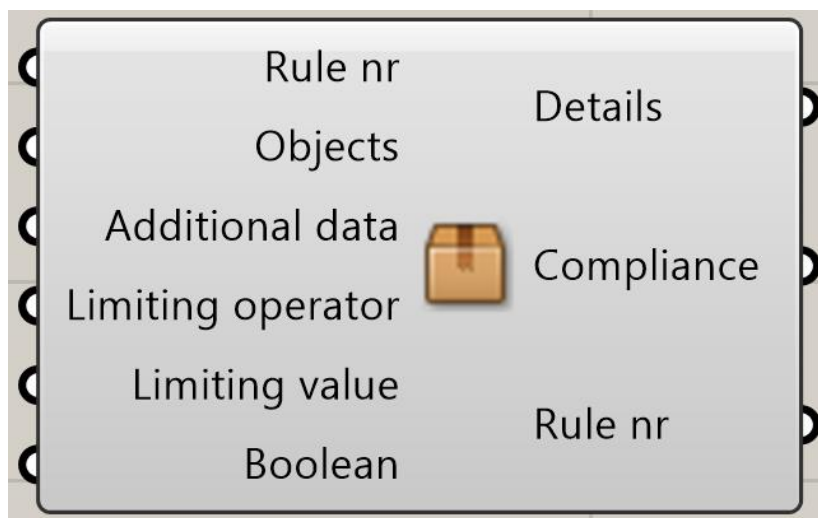


Figure 6. Concept of the method for determination function in Grasshopper

Figure 6 also shows which are standard inlets that every method for determination must contain. These are:

- **Rule nr:** each method for determination must have an inlet for rule number because all requirements have one, and it must flow through the whole process to track which requirements are already verified and do they pass or fail. Rule nr must be input as a float number. The reason why it is a float and not integer lies in the fact that many requirements are composed of a few sub-requirements. In that case, all sub-requirements have a number consisted of the main requirement number and, after the decimal point, the number of that sub-requirement, which always starts from 1 for each requirement. The example can be seen in Table 3.
- **Objects:** the inlet for one or more objects is reserved for a specific object whose property is checked. In this inlet can go a wide variety of data depending on the specific check, but most often that is an object in Grasshopper consisted of points, curves and surfaces.

- **Additional data:** another inlet on the input side is for the additional data required for performing the checks. The example shown in Figure 6. has only one inlet for additional data, but in reality, that depends on the specific check. Some methods for determination do not need any additional data, while some need a few of them. For example, the additional data could be results from structural analysis or specific partial factors that the designer must define.
- **Limiting value:** the inlet for limiting value is not mandatory for all methods for determination component. It will often be there, and then it takes a number in a float or integer form. In cases where the limiting value is unknown, it has to be calculated or determined inside the method for determination. Therefore, it needs additional data which are input previously, and the type of data varies per check. For instance, that situation is seen in most structural checks, where limiting value is a resistance that depends on many factors and has to be calculated inside the component. This is explained more in deep later in the text.
- **Limiting operator:** the input for the limiting operator is one of the fundamentals. The designer must input or from requirements management software can be acquired the limiting operator, which is then set between limiting value and value of the property. Afterwards, the script can determine is that correct or not.
- **Boolean:** the last inlet which every method for determination component must have is an input for Boolean value. The method for determination is run once when it receives the true value from the Boolean inlet. This is an important inlet to have in order to handle more complex requirements consisted of two or more sub-requirements. After the first sub-requirement is checked, depending on its result, the second sub-requirements will be run or not. In the case of complex requirements, the Boolean value is obtained from an “IF-THEN” function.

Furthermore, in the same Figure 6, standard outlets are shown that every method for determination function should contain. These are:

- **Details:** the first output is reserved for details that are specific for each method for determination. From that outlet, a component can send the various type of data that are calculated inside it. This largely depends on the type of requirement. For example, if the component for verifying maximal distance is used, the details outlet can give the value of distance that is being compared to the limiting value. Furthermore, if structural requirement is observed, then the outlet can provide some steps of calculation procedure for limiting value.
- **Compliance:** the most important data given by the method for determination component is compliance with a requirement. The second outlet serves for that purpose and it provides Pass or Fail string.

- **Rule nr:** the last outlet of the component gives a rule number, which is the same one as entered the component.

An important remark is that every inlet works with a specific type of data, and the designer must take care while using the component to input appropriate data. In Figure 7 is marked in red the description that software gives to the user when the mouse cursor is placed on top of the inlet's name.

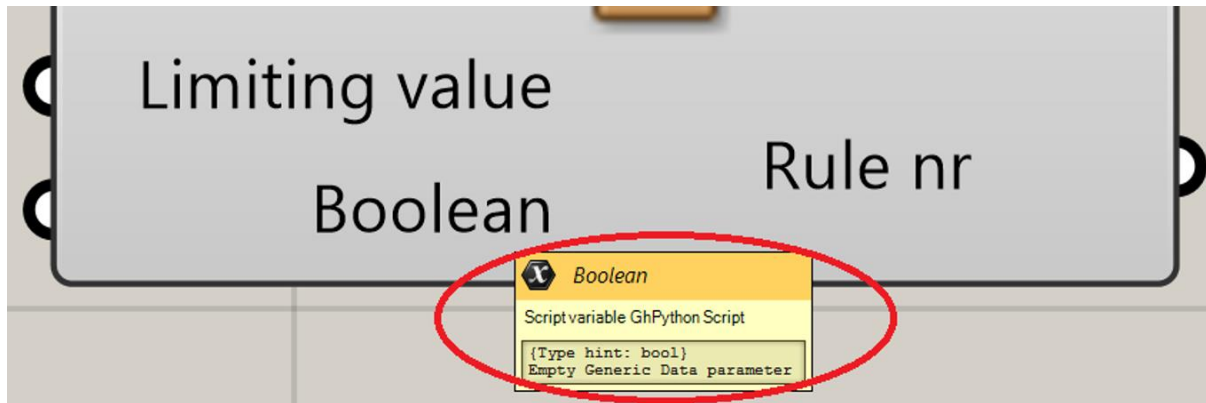


Figure 7. The specification of general description and data type used(*bool*) for a particular inlet

After defining standardised input and output parameters, the internal structure of the method for determination components must be examined. The method for determination functions can be separated into two main groups, for which will be given a general outlook of the internal structure. The two groups are: methods that calculate the value of the property or action; and the second methods that calculate the limiting value. All components from both groups compare the value of the property and limiting value after the calculation part.

The first group are methods that **calculate the value of the property or action**. The important attribute of this group is that the limiting value is input as a quantified number and then the method calculates or determine the value of the specified property. In this group would go most of the geometric related checks. In geometric checks, the limits are prescribed before, but the real value from the model has to be determined and then compared to the limit. There are three specific parts of the script for components from this group:

- **Data filter:** a large number of checks must have the data filter part of the script, in which the type of input data is being detected and depending on that, the further procedure is determined. To give an example, it can be used a check for determining the distance between two objects. The objects can be given as a point, curve or surface, and the method must determine the distance regardless of the object type that is input. The exact way of calculating the distance depends on which object type is being analysed, so the developer must think of all possible options and develop a mini script for each option inside the big script. In case that method for determination has to work with only one type of data or the procedure is the same for any type, then the data filter part of the script is not present.

- **Calculation/determination part:** the central part of the script is a part that calculates the value of the property. The developer must strive to come up with an optimal solution for a script. This is important in order to fulfil one of the most important requirements for the tool, speed. Also, the developer must come up with solutions for all realistic types of data that could enter the component.
- **Comparison part:** the final part of the script inside the method for determination components is a comparison part. Logically, in this final stage, the script compares the value of the property, calculated in the previous step, with a limiting value received by the designer or requirements management software table directly. Finally, the results of the comparison are handed to component outlets.

The second group are methods that **calculate the limiting value**. The important attribute of this group is that the value of the property is input as a quantified number, and then the method calculates the limiting value. In this group would go most of the structurally related checks. In structural checks, the action values are calculated by using some plug-ins prior to entering the method for determination, and then inside the method, the resistance of the element is being determined. There are three specific parts of the script for components from this group:

- **Data processing part:** the first part of the script inside the method for determination components is a data processing part. For performing a calculation, it is necessary to have a large amount of adequately structured data. While developing the component, the developer should strive for a solution in which the designer has to do the least amount of work possible. The perfect solution is the one in which the designer input the required data, and the component can structure the data itself properly for later usage. This can be achieved with no problems by restricting the type of input data. For instance, if the component is restricted to working only with the Karamba plug-in, then the designer can just connect the Karamba model, and the component can easily read and structure all required data. But, in that case, the designer is forced to use the Karamba plug-in for FEM analysis, which may not suit his preferences. So, there is a trade-off between the robustness of input data and level of automation or precisely the designer's involvement.
- **Calculation part:** the central part of the script is a part that calculates the limiting value. Most often, for this part developer must use a specific procedure prescribed in a particular code, for example, Eurocode. Therefore, the developer must analyse the required documents and find an appropriate determination procedure or get help from a specialist in the field. Formulas for the assessment should be input by the GHPython component, which is able to perform all calculations.
- **Comparison part:** the final part of the script inside the method for determination components is a comparison part. Logically, in this final stage, the script compares the value of the property received from a designer with a limiting value calculated in a previous step. Finally, the results of the comparison are handed to component outlets.

Regardless of which method for determination is being scripted, the developer must strive for the optimal solution. Also, for each check, it has to be decided what is the perfect ratio of manual work from the designer and the robustness of component with performing speed. If the designer derives all data manually before sending it to the component, then the computational engine does not have much work and can perform the check very fast. But in that case, the designer has made a lot of effort. Since the project's goal is to automate the process as much as possible, the large involvement of the designer is not desirable. Contrary, if the designer only connects the model to the method for determination and does not specify everything in detail, then it will take much more time for the computational engine to perform the check. Therefore, while developing the method for determination, it has to be tested which speed can be achieved with the least involvement of the designer, and then in few iterations, the optimal solution can be found. The speed of each check is very important in a situation where tens or hundreds of checks will be verified simultaneously.

Before scripting, the designer should make a plan and a structure for developing the script inside the component. For that reason, a template Table 4 is created. This table serves as a starting point for a designer to specify which additional input data are needed and for which data type each input must be scripted. After the input and output are described, then designer in the last row can explain how the script works and its purpose.



Table 4. The template for describing the method for determination

Name of the method for determination			
		Data type	Description
Input	Rule nr		
	Objects		
	Additional data		
	Limiting value		
	Boolean		
Output	Details		
	Compliance		
	Rule nr		
Description of script's operation			

After the method for determination is finished, the developer must give it a logical name and write a one-sentence description that explains what is doing that exact method for determination. This information must be specified in the Grasshopper environment, so that designer can simply understand every component. In Figure 8 is shown how that should look in general, and in Figure 9 is shown how that looks like for one already existing Grasshopper component.

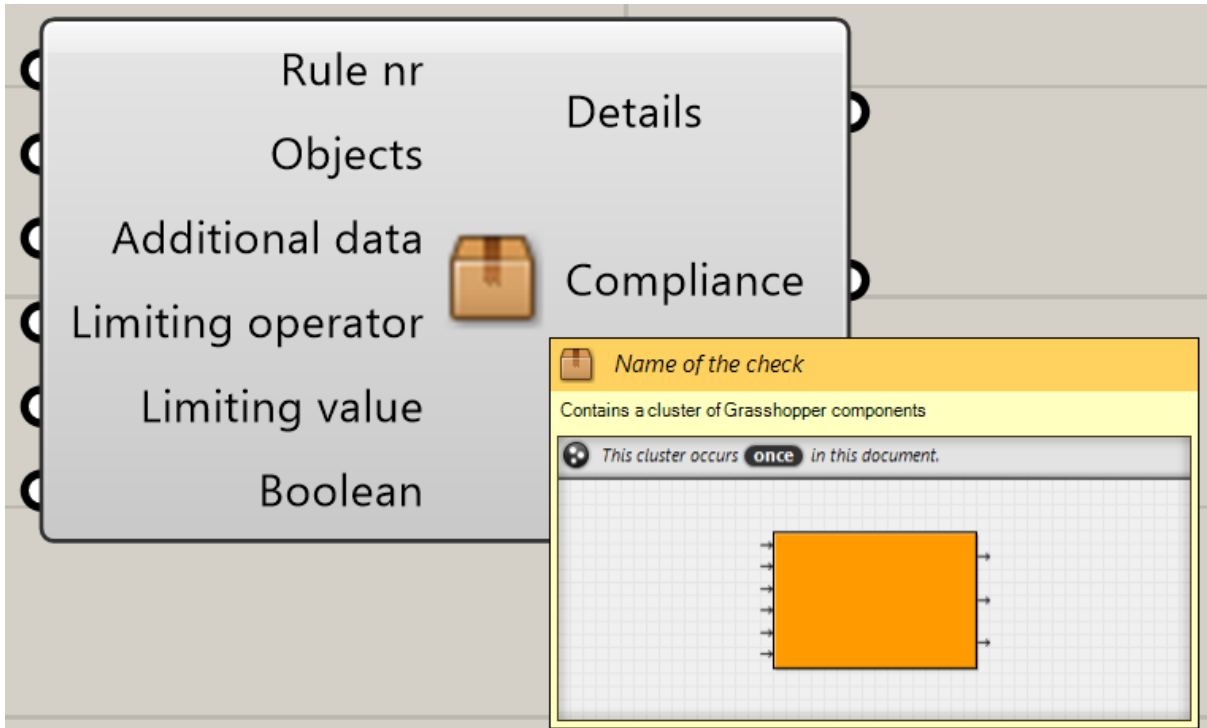


Figure 8. The general example of component's description

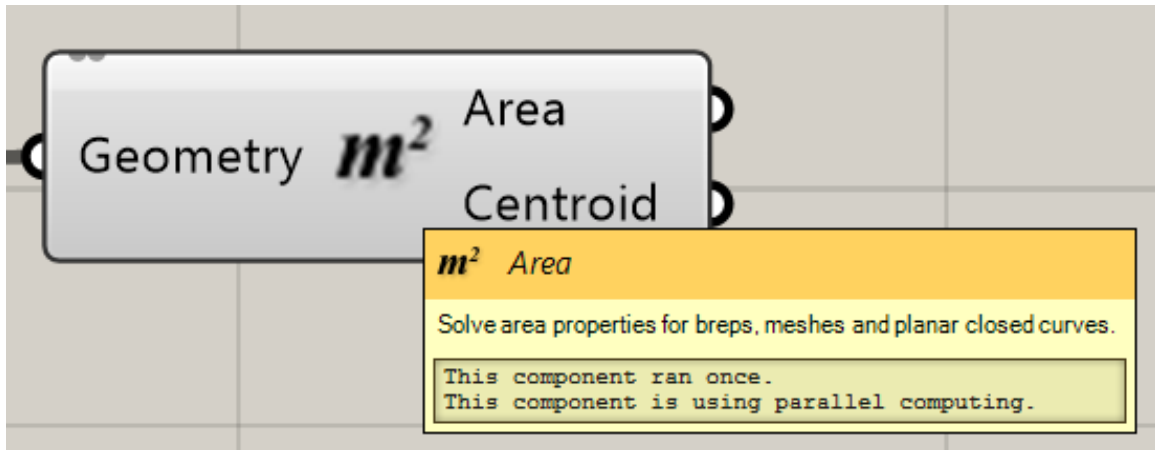


Figure 9. The component for determining Area and its description

### 5.3.3. Reporting functions

The last group of functions necessary for the tool's operation are the ones that enable generating the report of the check outside of the Grasshopper environment. The type of the report and details in it depend on the specific requirement and for what purpose it is used. Since reporting is specific for each situation, it is hard to give general instructions for scripting those functions. Therefore, the developer must find the best way of reporting 'on the spot' for each purpose, and further instructions will not be given inside this project.

## 6. Developing of Grasshopper functions – examples

This chapter explains the path of developing the two functions of the prototype tool and how the functions operate. The first function is geometry related, and it is one where the method for determination calculates the value of the property or action. The second function is a structural check for flexural buckling, where the method for determination calculates the limit value. To test the framework and prototype tool, the additional functions for verifying requirements are developed. These are:

The functions in which script is determining **the value of the property**:

- Distance between objects
- Intersection of objects
- Minimum height of the object

The functions in which script is determining **limiting value**:

- Bending check on steel profiles
- Axial force check on steel profiles
- Bending + axial check on cross-section level
- Flexural buckling check

During the scripting of the tool few plug-ins for Grasshopper were used, and these are:

### **Karamba3D**

Karamba3D is a parametric structural engineering tool that provides accurate analysis of spatial trusses, frames and shells (Karamba3D, 2021). It is finite element software fully embedded in the Grasshopper, which enables real-time analysis of the structures. Karamba3D enables the easy combination of parametrized geometric models, FEM calculations and optimization algorithms from other plug-ins. In this project, it is used for the verification of structural requirements.

### **GHPython**

GHPython is the Python interpreter component inside the Grasshopper environment that allows to execute dynamic scripts of any type (Food4Rhino, 2021a). It enables using many Python and .Net modules and libraries inside the Grasshopper. In this project, it is mainly used for working with data (structuring, extracting etc.).

### **Pterodactyl**

Pterodactyl is an open-source plug-in for Grasshopper created for the purpose of generating custom documents, reports, articles etc. (Food4Rhino, 2021b). It enables updating the documents in real time. In this project, it is used for generating structural reports.

## TT Toolbox

TT Toolbox is a plug-in for Grasshopper made by CORE Studio, and it has a wide range of functions (Food4Rhino, 2021c). The most important function for this project is the connection of Microsoft Excel with Grasshopper, which is used to read the requirements table and write the compliance in it after performing the checks.

### 6.1. Distance between two objects

The function for comparing the distance between two objects to a limiting value is a basic example of a function in which the script calculates the value of the property, in this case, the value of the distance. Figure 10 below shows the input and output parameters of the function, these are the standardized ones as explained in Chapter 5.3.2. Working with this function is very simple, and the designer has to input the main object and other objects from which the distance has to be checked, rule number and Boolean value to run the script. The limit value, limiting operator and units are taken from requirements management software immediately, in a way that the designer has to input requirement number into the method and then it automatically acquires the assigned values. In practice, inlets for Limitvalue and Limitingop are combined into one, together with units, and then the excel table is directly connected into it. In Figure 10. this was separated to show all input data.

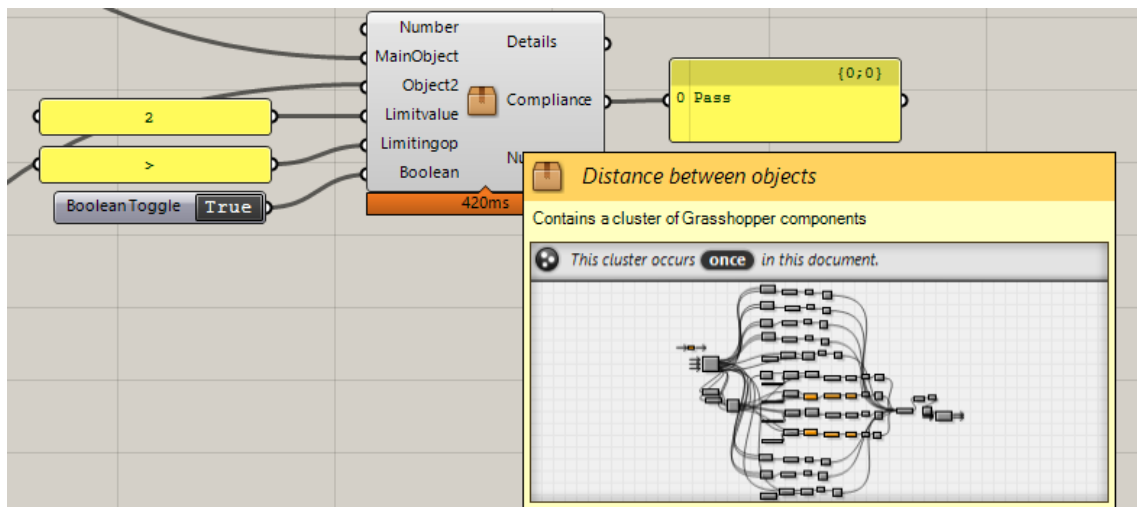


Figure 10. Distance between objects function

The internal structure of the method is shown in Figure 11. The structure is divided into three specific parts explained in the previous chapters. The first part is a Data filter that determines the object type and sends it to the appropriate part of the script in the calculation step. The object type can be point, curve, surface or brep and the procedure for calculating the distance between any of these two objects differs per type of object and its combinations. For the data filter and comparison part, the GHPython is used. The details of the script are shown in Appendix A. After a certain part of the calculation script is triggered, it calculates the distance and sends it to the comparison part, where the distance is compared to the limit value.

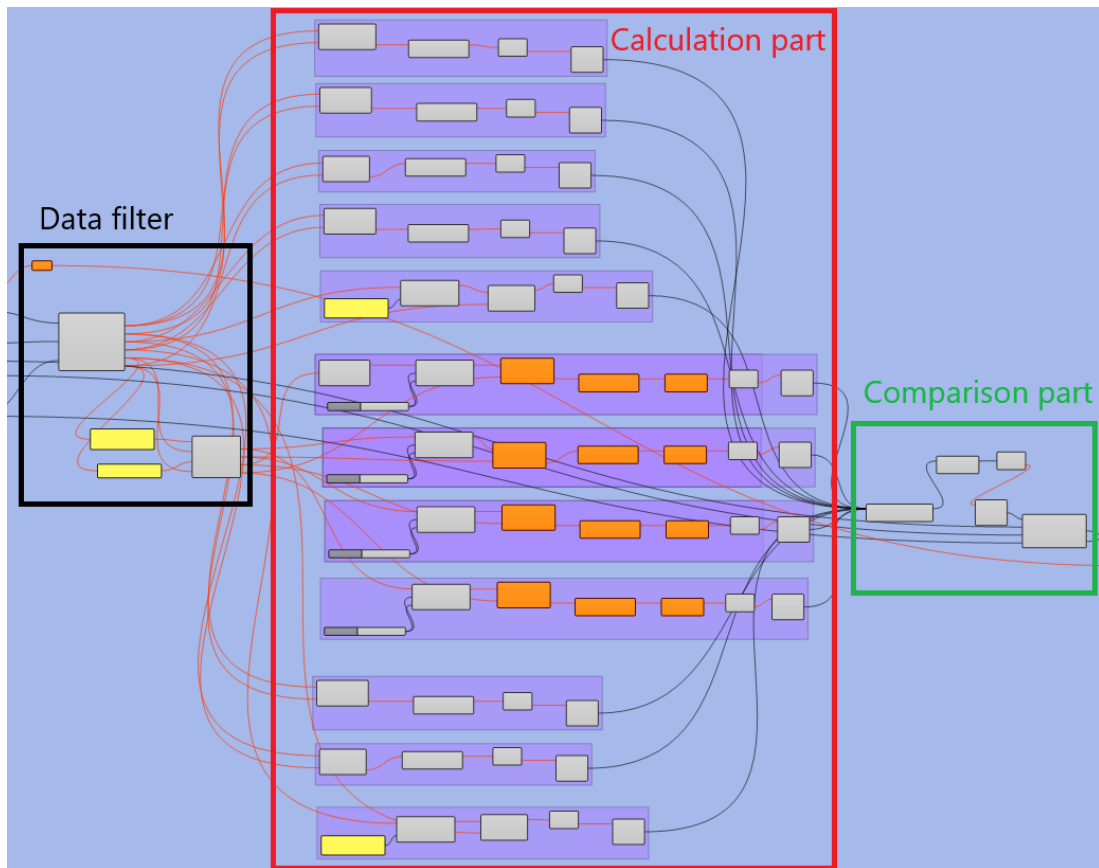


Figure 11. The internal structure of the function for determining a distance between objects

During the scripting, the main problem arose because Grasshopper does not have a function for measuring the distance between two surfaces. Nevertheless, there is a solution to find that distance, but it is not exact and relies on dividing the surfaces into many points and working with distances between them. Unfortunately, by following that procedure certain level of measurement error must be accepted. Furthermore, the error can be lowered by increasing the density of the points, but then the tool's speed falls radically because it has to determine millions of distances. That shows the tool depends on Grasshopper's abilities, and the developer must actively strive to find faster and more simple solutions while scripting.

In Table 5, a description of the function is shown by using the standard table defined in previous chapters.

Table 5. Description of the function for determining a distance between objects

Name of the method for determination			
		Data type	Description
Input	Rule nr	Float	The number which is assigned to the same rule in requirements management software
	MainObject	Point, Curve, Surface, Brep	The main object from which the distance is measured
	Object2	Points, Curves, Surfaces, Breps	Objects whose distance from the main object is measured
	Limitvalue	Float	The value of the border distance assigned to that rule in requirements management software
	Limitingop	String (<, >, ≤ or ≥)	The limiting operator assigned to that rule in requirements management software
	Boolean	True or False	The Boolean Toggle function from GRASSHOPPER. When True, then check is run.
Output	Details	Strings and floats	
	Compliance	String (Pass or fail)	It says if the requirements satisfied
	Rule nr	Float	The number which is assigned to the same rule in requirements management software
Description of script's operation	The script first determines the input combination of object types (Main object type + Object2 type). According to that, it makes appropriate lists with combinations and sends them to a part of the script which calculates the distance between exactly these two types of data. After the distances between MainObject and all Object2 are found, then the minimal is chosen and compared to Limitvalue.		

## 6.2. Flexural buckling

The function for verifying the system members on flexural buckling is a basic example of a function in which the script determines the limiting value, in this case, the maximal value of the axial force in the element. Figure 12 below shows the input and output parameters of the function, these are the standardized ones as explained in Chapter 5.3.2. Working with this function is very simple, and the designer has to input only rule number, analysed model from Karamba and Boolean value to run the script. As it is already defined, in this project the tool is operating only with Karamba models. For the simplicity of the prototype tool and due to time restrictions, the function is limited to only steel I and H profiles from Eurocode. That does not make a difference in the tool's structure or approach, more profiles would have only added

extra lines of the code in the data processing part of the script. Therefore, the addition would not have given any extra value for proving the framework.

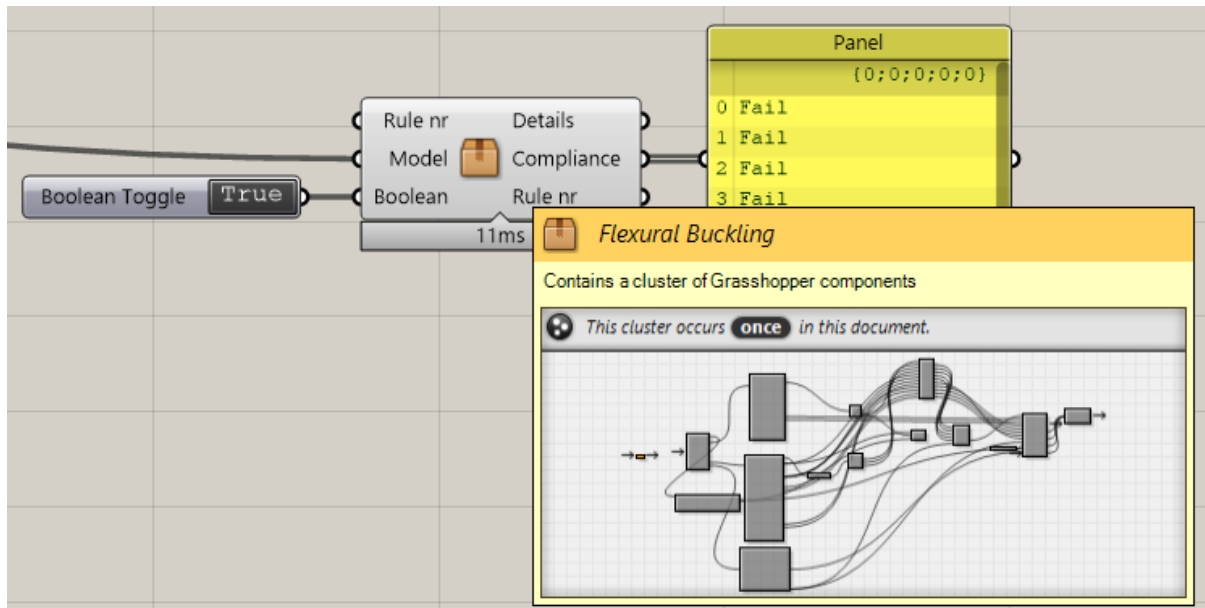


Figure 12. Method for checking flexural buckling

The internal structure of the method is shown in Figure 13. The structure is divided into three specific parts explained in the previous chapters. The first part is Data processing which is extracting the data from the Karamba model and structuring it in a usable way. Karamba by default, gives the bunch of data in the form of strings appended in a list, so the script must first locate and take out the required information. More on this is given in Appendix A. After the data are prepared, it is sent to the second and third part of the script, which are combined into one GHPython function. The second part is a calculation, and the third is the comparison part. The calculation part is scripted by using the appropriate formulas from Eurocode by transferring them in GHPython's lines of code. This check refers to Eurocode 1993-1-1 6.3 (EN-1993-1-1, 2005). After the maximal axial force in the element is calculated by following the formulas from the code, the determined value is compared to the previously calculated value of the force in the member. The calculation of real force values in the element is part of the building model preparation step. The details of the script are shown in Appendix A.

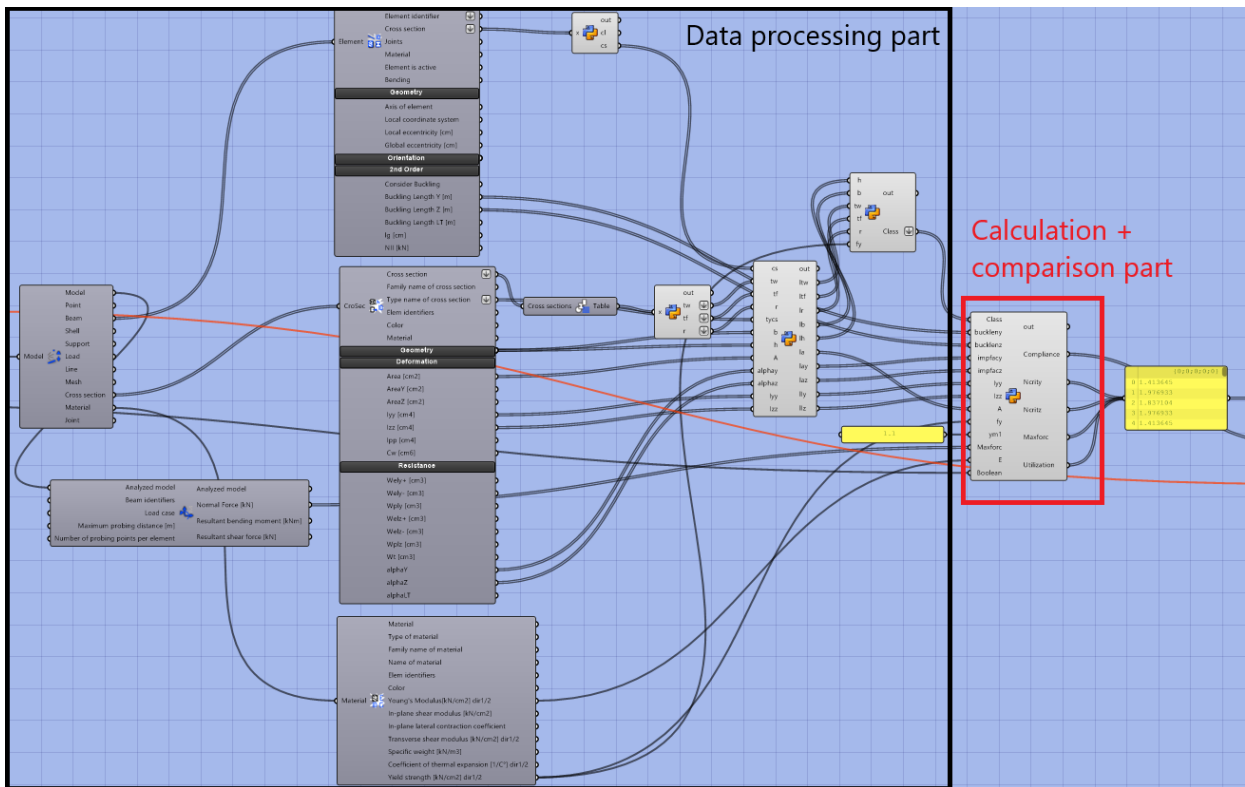


Figure 13. The internal structure of the function for verifying the flexural buckling

The structural checks like this one for flexural buckling are very similar in general and should not be too complex to script. The exception with steel are checks for lateral-torsional buckling and beam-column buckling in which the shape of the moment diagram between braces is important. In that case, the developer must come up with solutions to determine where braces are positioned and find out the shape of the moment diagram between them, which would probably require defining the segments of the moment diagram in a mathematical expression. Furthermore, in some more detailed checks, for example, on reinforcement level, new problems would arise due to limitations of the Grasshopper. But again developer would maybe be able to come up with an innovative solution.



In Table 6, a description of the function is shown by using the standard table defined in previous chapters.

*Table 6. Description of the function for verifying the flexural buckling*

Name of the method for determination			
		Data type	Description
Input	Rule nr	Float	The number which is assigned to the same rule in requirements management software
	Model	Previously analysed Karamba model	The model which is analysed in the building model preparation step
	Boolean	True or False	The Boolean Toggle function from Grasshopper. When True, then check is run.
Output	Details	Strings and floats	Details about the utilisation of the elements
	Compliance	String (Pass or fail)	If the axial force in the element is lower than maximal allowed according to the formulas, then the result is Pass. Contrary, the result is fail.
	Rule nr	Float	The number which is assigned to the same rule in requirements management software
Description of script's operation	The script first extracts the data about all steel profiles used in a model. Afterwards, it determines the class of the cross-section. Then the calculation according to EC3 formulas is performed for each element to get the maximal force in it. Finally, the actual force in the element is compared to maximal to verify the design.		

### 6.3. Functions for synchronization of Microsoft Excel and Grasshopper

The synchronization of MS Excel and Grasshopper is achieved by using the TT Toolbox plugin, which already has a built-in function that reads the Excel table and imports it in the Grasshopper environment. The example of the component is shown in Figure 14. The component imports the whole table, therefore it is necessary to have a GHPython component inside the method for determination which extracts specific data needed for the check. Those are rule number, limiting operator, limiting value and units.

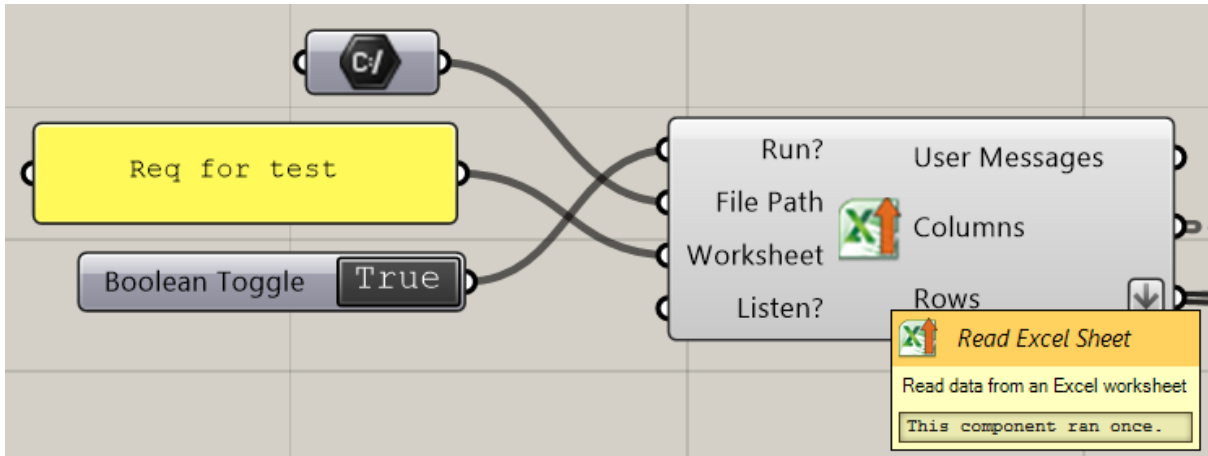


Figure 14. Component for importing Excel table into Grasshopper

The next component that had to be scripted is “Writing compliance in excel”. The input parameters are shown in Figure 15 and the inside structure of the component in Figure 16.

It is important to input the exact file path and worksheet in which compliance has to be written. The third inlet is ContentFromColumns, and there has to be input Columns output of the component for reading the Excel table. This is necessary in order to determine in which row to write the compliance for specific requirement. Other input parameters are very logical, and these are compliance, rule number and Boolean value for running the function. Inside the function, two components from TT Toolbox are used WriteOptions and Write to Excel. Also, two GHPython components are used to determine the exact row number in which compliance has to be written.

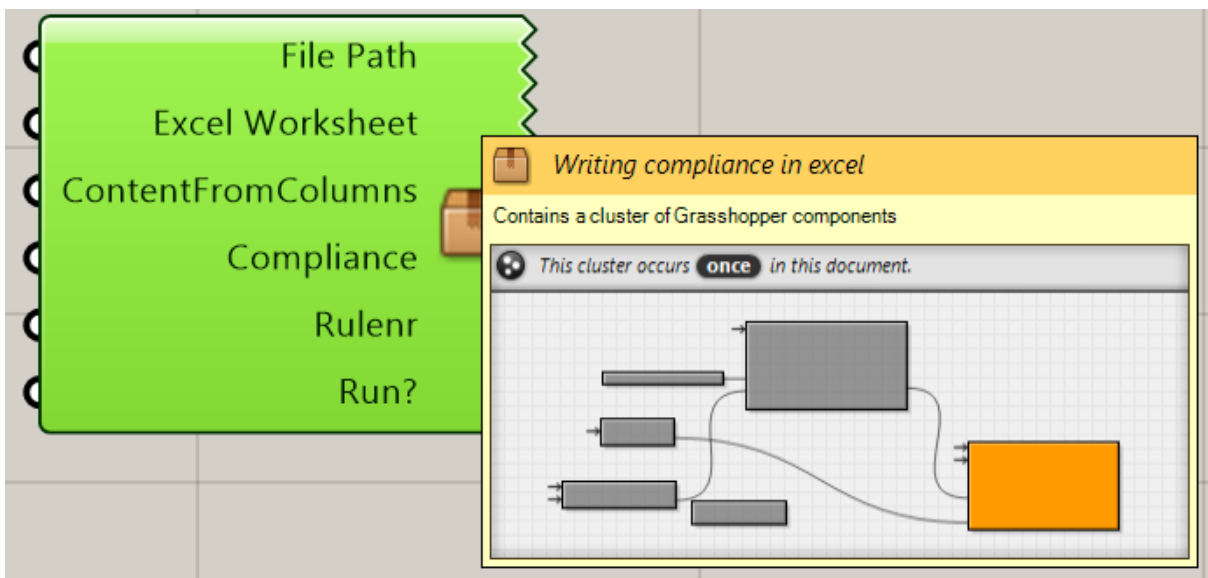


Figure 15. Component for writing compliance in excel

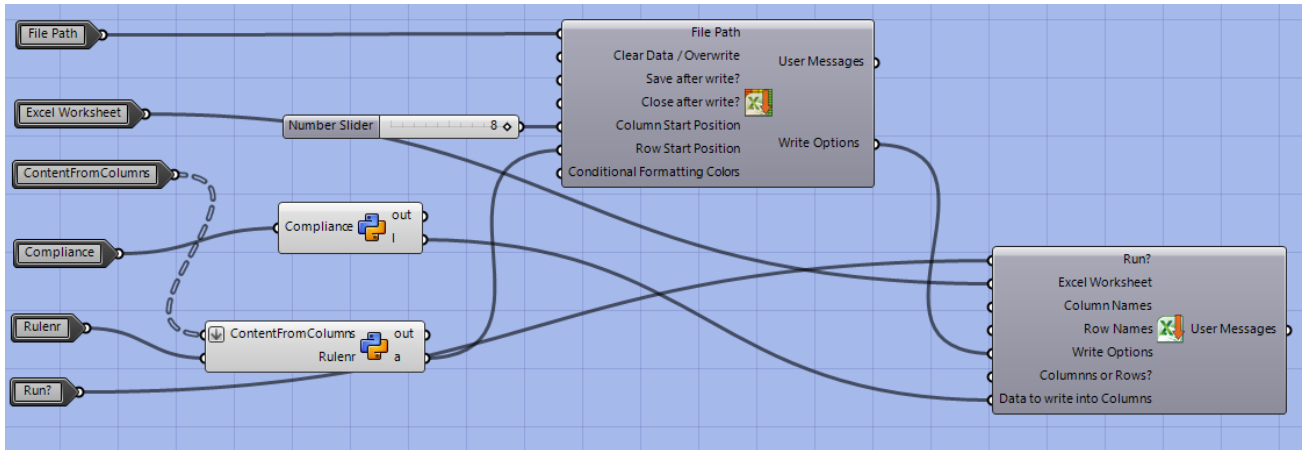


Figure 16. Inside structure of the component for writing compliance in excel

## 6.4. Reporting function

The final function that had to be developed is the one for generating the report. The input parameters of the function are shown in Figure 17 and the internal structure in Figure 18. The input parameters are Time in which time and date components from Grasshopper has to be input; Rule number; specific text to be written in the report and file path where to save it. In the Text inlet, already structured text has to be input, therefore inside the method for determination should be generated text for the report. The Details output in each method for determination is reserved for that, which has been already explained in chapter 5.3.2. The inside structure of the reporting function consists of three components from the Pterodactyl plug-in: Heading, Create Report and Save Report. Before that, one GHPython component is used to generate the Heading of the report.



Figure 17. Input parameters for the Reporting function

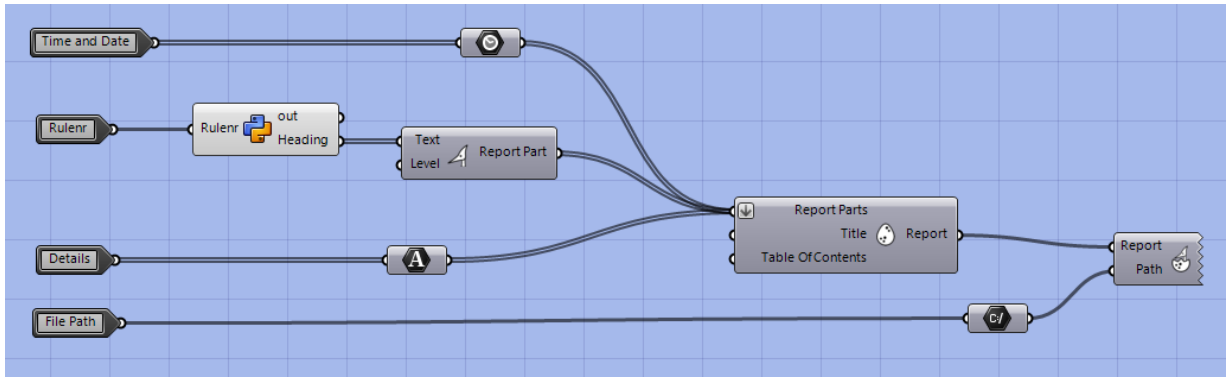


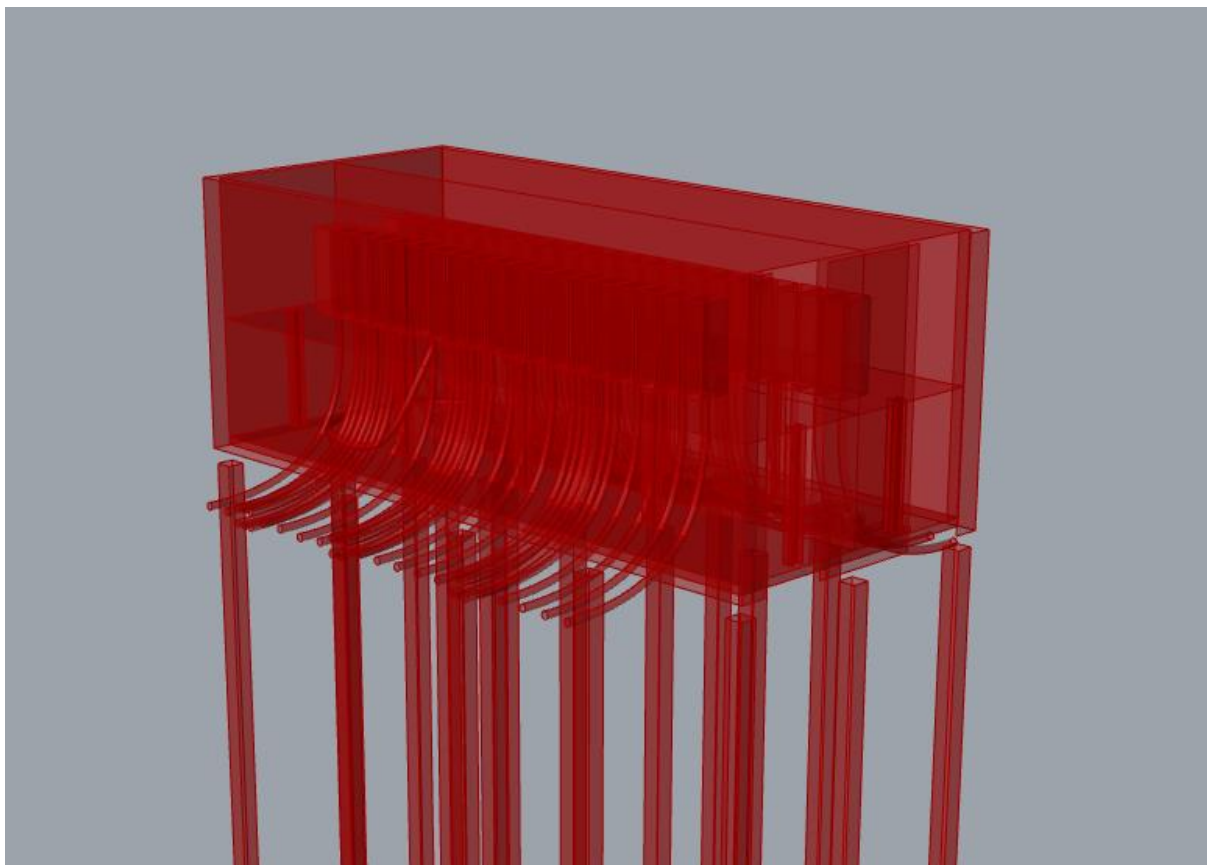
Figure 18. Inside structure of the function for generating the report

## 7. Pilot study

To validate the tool, it has to be tested on a real building model. Even though it has been tested continuously during the scripting phase on some mini models, the complete test is crucial for deriving a conclusion. The main validation strategy is to run the tool on different models, with emphasis on the main building model, which is explained in the following chapters. If the tool is able to verify different types of buildings, objects and requirements in a reasonable time, then it can be considered functional. Still, it is very hard to define reasonable time, so this has to stay a little bit qualitative and dependent on designer's impression. Nevertheless, to provide first approximation of time saved by using the tool, the tool is compared to manual requirements checking in a very basic way.

### 7.1. Test-case

The building model used for the final test is provided by SWECO and represents building related to the energy sector. The model is shown in Figure 19.



*Figure 19. The model of the building used for a test case*

The model of the building can be divided into a few categories of elements that are important for us. These are concrete walls and slabs, steel beams and columns, power boxes, cables, and

piles. All of these are involved in at least one check. The list of requirements for this building is given in Table 7.

Table 7. List of requirements input in a standardized table

Rule nr.	Object	Property	Limiting operator	Limit value	Unit of measurement	Determination method description
1.1	Room	Height	>	2.4	m	Height
2.1	First Box	Distance from TRANSVERSAL wall	>	1	m	Distance between two objects
3.1	Piles	Intersection with cables	=	0	Intersections	
4.1	Steel beams and columns	Utilization - bending	≤	1	-	EC 1993-1-1 6.2.5.
5.1	Steel beams and columns	Utilization - compression	≤	1	-	EC 1993-1-1 6.2.4.
6.1	Steel beams and columns	Utilization - bending + axial	≤	1	-	EC 1993-1-1 6.2.9.
7.1	Steel beams and columns	Utilization - flexural buckling	≤	1	-	EC 1993-1-1 6.3.1.

Seven requirements have to be checked, and these match seven methods for determination developed for the tool. The first three requirements are related to the model's geometry, and others are in a structural domain.

In order to get impression of the time saved by using the tool developed in a project, the time needed for checking these seven requirements manually has to be compared to time needed for verification by the tool. After these seven requirements are checked, few changes has been made in the building design and then the time spent for both option has to be checked again. Also, for the verification by the tool two version must be checked. One in which the script has to be built from scratch and second where part of the script has been already built, which could be in case of standard projects.

## 7.2. Results

It was important to go through each of the five steps of automated compliance checking to test the tool. Firstly, the requirements are input into requirements management software by using the standardized table. Microsoft Excel served as a replica of requirements management software because it is convenient for storing tables and communicating with Grasshopper. The standardized table from Excel is the same as the one previously shown in Table 7.

Afterwards, the script for code checking is generated manually. The script for checking requirement 2.1., which determines if the distance between the last box and transversal wall is larger than 1.0 m, is shown in Figure 20.

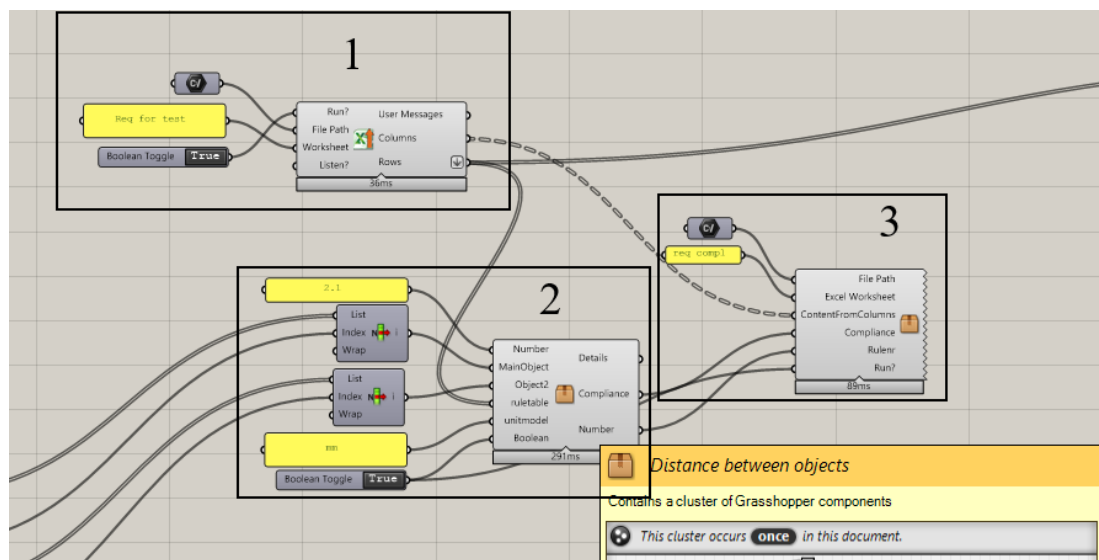


Figure 20. Script for code checking

The script consists of the three main parts indicated in Figure 20. The first part is a component that enables the communication between Excel and Grasshopper. The data about requirements number, limiting values, limiting operators and units of measurements are taken from the table and input in the second part of the script, which is the method for determination. The method is fed by all necessary data that is collected during the building model preparation step. Inside the method for determination, the calculation is performed, and compliance with a rule from the table is confirmed or not. Afterwards, the compliance is sent to the third part of the script, enabling communication between Grasshopper and Excel but in other direction than in step one. In the third step, compliance is written in the Excel table, and the example can be seen in Figure 21. The green field with a checkmark is used for the successfully fulfilled requirements, while the failed ones are presented by crossmark on the red field.

	A	B	C	D	E	F	G	H
1	Rule nr.	Object	Property	Limiting operator	Limit value	Unit of mesurement	Determination method description	Compliance
2	1.1	Room	Height	>	2.4	m	Height	
3	2.1	First Box	Distance from TRANSVERSAL wall	>	1	m	Distance between two objects	✓
4	3.1	Piles	Intersection with cables	=	0	Intersections	0	✓
5	4.1	Steel beams and columns	Utilization - bending	≤	1	-	EC 1993-1-1 6.2.5.	
6	5.1	Steel beams and columns	Utilization - compression	≤	1	-	EC 1993-1-1 6.2.4.	
7	6.1	Steel beams and columns	Utilization - bending + axial	≤	1	-	EC 1993-1-1 6.2.9.	
8	7.1	Steel beams and columns	Utilization - flexural buckling	≤	1	-	EC 1993-1-1 6.3.1.	✓

Figure 21. Table of requirements with updated compliance

The proof that requirement 2.1. is fulfilled can be checked by looking at the “Details” output of the method for determination. The information about actual distance is given there, and it is 1.06 m while the limit value is 1.0 m. Figure 22 shows how that looks in the Grasshopper environment.

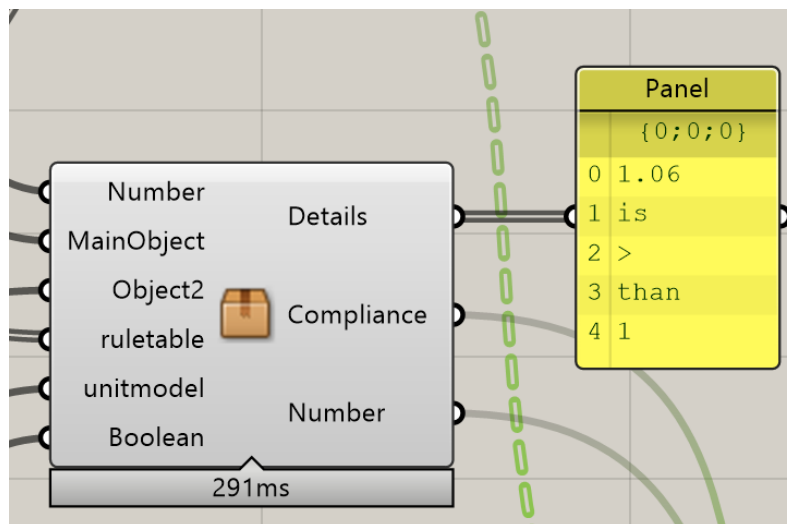


Figure 22. Details of the requirement 2.1.

Another aspect of the code which has to be checked is synchronization between Excel and Grasshopper. If the limit value in Excel changes, it must be automatically updated to Grasshopper, and a check must be performed again. Also, if the object's attributes in Grasshopper are changed, the check has to be performed automatically and update the new



compliance in a table. The test showed that synchronization is perfect, and Figure 23 shows changed compliance in an Excel table after the limit value is set to 1.5 m instead of 1.0 m.

	A	B	C	D	E	F	G	H
1	Rule nr.	Object	Property	Limiting operator	Limit value	Unit of mesurement	Determination method description	Compliance
2	1.1	Room	Height	>	2.4	m	Height	
3	2.1	First Box	Distance from TRANSVERSAL wall	>	1.5	m	Distance between two objects	✘
4	3.1	Piles	Intersection with cables	=	0	Intersections	0	✔
5	4.1	Steel beams and columns	Utilization - bending	≤	1	-	EC 1993-1-1 6.2.5.	
6	5.1	Steel beams and columns	Utilization - compression	≤	1	-	EC 1993-1-1 6.2.4.	
7	6.1	Steel beams and columns	Utilization - bending + axial	≤	1	-	EC 1993-1-1 6.2.9.	
8	7.1	Steel beams and columns	Utilization - flexural buckling	≤	1	-	EC 1993-1-1 6.3.1.	✔

Figure 23. Updated compliance after the limit value has been changed

The last part of the tool that must be checked is a reporting function. It was tested on a requirement 7.1. The „Details“ output from the method for determination is input in reporting function, then the report is generated and saved in a document outside of the Grasshopper environment. Figure 24 shows the report.

Structural report.md - Typora

File Edit Paragraph Format View Themes Help

Monday, September 6th 2021  
18:56:22

## Report for rule nr: 7.1

Cross-section	Maxforce	Relation	Ncrit	Utilization	Compliance
HEB180	182.895022	<	757.201135	0.241541	Pass
HEB180	482.806648	<	1118.74288	0.431562	Pass
HEB180	533.160218	<	1118.74288	0.476571	Pass
HEB180	741.581065	<	1118.74288	0.66287	Pass
HEB180	241.306317	<	563.127046	0.428511	Pass
HEB180	213.653985	<	650.435604	0.328478	Pass
HEB180	618.987264	<	1118.74288	0.553288	Pass
HEB180	509.796241	<	1118.74288	0.455687	Pass
HEB180	627.082142	<	1118.74288	0.560524	Pass
HEB180	213.462662	<	648.210133	0.329311	Pass
HEB300	0	<	2502.237766	0	Pass
HEB300	0	<	2699.803342	0	Pass
HEB300	0	<	2289.478227	0	Pass
HEB300	0	<	2183.481661	0	Pass
HEB300	0	<	2340.32342	0	Pass
HEB300	0	<	2520.279996	0	Pass
HEB300	0	<	2500.433481	0	Pass
HEB300	0	<	2336.640704	0	Pass

108 Words

Figure 24. Report for requirement 7.1.

After the tool's functionalities were tested, the comparison of the time spent on checks between manual verification and the tool was done. Table 8 shows the comparison of the time spent on checks. The manual verification of the requirements took 8 minutes and 45 seconds, while the first verification by the tool took 11 minutes and 30 seconds. So, in this case, manual verification was significantly faster. Then, the test was performed on the same example, but part of the script has already been defined, which is the case in some standardized projects. Then, the tool performed checks in 5 minutes and 30 seconds, which means it saved 3 minutes and 15 seconds on these few requirements.

Table 8. Comparison of time spent on checks in case of manual verification and verification by the tool

	Manual verification	Verification by tool (from scratch)	Verification by tool (partly built code)
Required time (min)	08:45	11:30	05:30

After the first test of the time saved, a few changes have been made on the building model, and then the time spent on checks has been tracked. The results are shown in Table 9. The manual verification took 4 minutes and 15 seconds, while verification by the tool took only 10 seconds. Here is shown the real strength of the tool. Once the script has been built, the tool can perform all checks continuously and instantly. Finally, when both tests are combined, the manual verification spent 13 minutes, while the tool needed 11 minutes and 45 seconds to perform the checks.

Table 9. Comparison of time spent on verification after a few changes have been made on the model

	Manual verification	Verification by tool
Required time (min)	4:15	0:10

### 7.3. Impressions from the testing

This section gives the impressions from the test of the previously explained model. Firstly, the operation of the tool is discussed based on the three functional requirements given for it. Afterwards, the impact of the designer and computational engine on the process were discussed. Finally, the difference between designer and developer is explained.

#### Speed

The tool is very fast in general. The time saved by using the tool instead of manual verification is significant if the designer is skilled in the Grasshopper environment. The important feature of the tool is that the designer can choose an acceptable ratio of speed and level of automation. An example is a check for requirement 3.1, which says that cables should not intersect with the piles. There are 96 cables and 17 piles in the model. If the designer chooses to check each cable with a pile close to it one by one, the tool can calculate one combination in 5 milliseconds, but the designer has to spend some time extracting the appropriate cable and pile; and repeat that for all piles or cables. A different approach would be to input all cables and piles in the method for determination at once and let the tool perform all 1632 (96 cables x 17 piles) iterations. The

tool needs around 35 seconds for that, which seems a lot, especially if the designer is sitting in front of the desktop and waiting. Nevertheless, the time spent on manual work for selecting and extracting all important cables and piles must be included in the first option. The test of time saved also showed that tool has a big advantage if it is used in standardized project, when script does not have to be built every time from scratch. Also, the tool is advantageous if some changes are made on the design, because it can perform checks instantly when the script is already built. This is showed in a small test, but in a real projects which are very complex and have a lot of iterations, then the tool shows its real strength and purpose.

### **Robustness**

The tool's robustness depends on the developer, more precisely on the limitations introduced by the developer. While scripting, it was specified what type of data could be used for each check, and the aim was to cover as many situations as possible. The testing was successful, and the tool was able to operate with all types of data from the model. Still, the model has to be prepared and adjusted for the tool. For example, the Karamba model had to be developed to perform structural checks because the methods for determination work only with the Karamba model, and other FEM software or plug-ins are not compatible. Also, it is advisable to test the tool on more models to ensure that the specific method for determination covers a wide variety of data types that can be used.

### **Extensibility**

The extensibility of the tool's structure is tested already during the scripting phase. All functions needed for the regular operation of the tool were successfully developed by following the proposed structure and instructions given in Chapter 5.3. It is expected that anyone can create new functions, especially methods for determination, by following the exact instructions and by that adjusting the tool to personal needs.

### **Designer**

The designer who serves as a manual intelligent force is a crucial component of the system's architecture. The test proved that the designer must be familiar with the Visual programming language environment and the building model. The tool's operation is very dependent on the designer's skills and ability to input appropriate data in the methods. For example, in requirement 2.1. (min distance between the first box and transversal wall must be 1.0 m) it is essential to extract and select the exact box and wall that are within the scope of this requirement. Therefore, the designer must know where these are stored in the script. That would make the use of the tool a little bit more complex for a third party, for instance, regulatory bodies, because the person who uses it should first get familiar with the model, which takes some extra hours. Moreover, that forces the designer to make a clearer script which could take more time initially, but in the long run, it results in a more organized process.

## **Computational engine**

During the scripting, it has become evident that the tool is very dependent on the Grasshopper's abilities. There are many checks for which developers must find a solution to check them because Grasshopper still does not have the functions to cover it. Also, Grasshopper dictates the speed of the verification process, which is already explained with an example of requirement 3.1. Moreover, in this test, only seven checks were performed simultaneously, and Grasshopper had minor lagging because of requirement 3.1. Therefore, the question of the tool's speed when more requirements are input is fundamental to investigate. There are hundreds of requirements for a building in real projects, and it is not clear how Grasshopper would cope with that.

## **Developer vs Designer**

The work is separated into two main roles in the automated code checking process, which can sometimes interfere. The first role is a developer of the functions. The developer is a highly skilled expert in Grasshopper and general programming (Python), who can script the functions for checking and make a tool by following the instructions given in this report. In practice, that person could be the company's employee who is designing the building or an employee of the company that owns the software for code checking. In the first scenario, every company has its own software or one open-source plug-in that can be adjusted or extended for the project's purpose. While in the second scenario, all tool development is performed by a software developing company, and the company that designs the building can only use the tool without adapting it. But, this could be classified as a black-box approach similar to what has already been on the market in software like Solibri, FORNAX etc.

The second role is a designer who is the user of the tool. He just needs to connect appropriate components to check the requirements. If the designer is skilled enough, he could also be a developer who adapts the tool to the project's specifics.

## 8. Discussion

This chapter discusses the motivation for starting this project, what has been done in it, and afterwards, the strengths and weaknesses of the tool are covered. Furthermore, assumptions and limitations are mentioned and what would have been different if some different decisions had been taken. Finally, contributions of this research are pointed.

### 8.1. Vision

The adoption of the new technologies in the AEC industry is very slow. While society and other industries are taking advantage of information technology, the AEC industry is struggling with the transformation (Coenders and Rolvink, 2014). Although digital tools are used for a variety of tasks, compliance checking is still a very manual process. Since there are many benefits related to code checking automation, many researches were conducted in the field. But, until now, no one succeeded to develop a general approach that could cover all steps of code checking. Also, all proposed methods have the problem that designers must be highly skilled in classic programming or use completely black-box solutions. The black-box approach is not preferable because the designer can not adjust the tool to his specific needs and can not understand how the check is performed. Therefore, there was a need to come up with a new solution. The rise of the popularity of visual programming languages and parametric design between designers opens the possibility to automate the compliance checking by using softwares like Grasshopper or Dynamo. Visual programming is substantially easier than classic programming, and a designer who works in one of these softwares is skilled enough to build a script for code checking. Also, it has a certain level of transparency which is necessary for a designer to understand the flow of the script. Furthermore, the Visual Programming Language environment is very responsive to changes in requirements because new checks can be added easily. Another advantage is the possibility of continuous checking during the designing phase. This project tries to set a general framework for automated testing of building design and develop a prototype tool which shows the capabilities of that approach today.

### 8.2. What has been done?

The main objective of this project was:

***“The fundamental objective of this project is to explore the possibilities of automating the requirements verification for a building design, by using requirements management software to systematically structure the requirements and Grasshopper to generate the rules which afterwards can be verified.”***

To achieve this objective, the project was divided into three main parts. First, the conceptual framework for automated code checking is developed, explaining each step in the process to the details. Afterwards, the system architecture of the prototype tool was explored, and

instructions for scripting the tool were given. Finally, after the tool was scripted, it was tested on a real building model. During the testing, the tool showed some evident strengths, but also that it has particular weaknesses.

At the beginning of the project, one research question was set, which was then divided into four sub-questions. The main research question was:

***“How can manual verification of design requirements be automated by using Requirements management software and Grasshopper?”***

The conclusions regarding the four sub-questions are discussed below:

### ***1.) Which requirements should building design fulfil?***

Before focusing on specific requirements, it was looked into all requirements that building design should fulfil. For that, it was necessary to look in Building Decree 2012. After careful literature review, the requirements were divided into two main categories based on the form in which they were given: functional and performance requirements. The functional requirements are qualitative and indicate which goals the building has to fulfil, but without providing a concrete way how to do it or measure. The second group consist of quantifiable performance requirements, and these were in the scope of this project. The functional requirements are not suitable for verification by using Visual programming language due to its qualitative nature, which can not be adequately quantified. Also, it was drawn from the literature that requirements are not standardized nor following some specific logic. They are instead based on the experience and adaptability of the human mind.

### ***2.) How to approach automated testing of building design?***

The four-step approach for automated compliance checking given by the Eastman was used as a basis for answering this question. The framework is expanded, and one extra step is added before the procedure proposed by Eastman. Also, each step is explored separately, and new ideas and approaches are proposed. The five steps are as follows.

Firstly, the requirements must be collected, grouped, and input in requirements management software. That step is called *Requirements defining and logical structuring into RMS*. The groups of the requirements are necessary to have a better-organized process. In this project proposed classification followed the separation of the roles in the project, and it was geometric, structural, building physics and services requirements. But, this is not strict and can be adjusted based on the preferences of the team. Furthermore, the requirements should be input in the requirements management software through the standardized table that has seven columns: Objects, property, limiting operator, limit value, unit of measurement, determination method and conditions.

The second step is the *Interpretation of the requirements*. In this step, the designer builds a Grasshopper script with previously defined functions. He is transferring the logic of requirement from requirements management software into a script. There are few categories of functions necessary to be predefined: basic functions, methods for determination and reporting

functions. Afterwards, when natural language processing or any of the other proposed methods comes to a higher level, this step can be fully automated.

Thirdly, all required data must be extracted from the model in a *Building model preparation* step. This project worked only with Grasshopper models. There are three steps in model preparation: acquiring data from an already existing model, assigning new data to existing models, and performing additional analysis for role-specific checks. In the future, communication with third-party software must be established, which would enable working with BIM models.

The fourth step is the *Checking phase*. This is the step in which Grasshopper is running a script, and it is crucial that this step is trustworthy.

The last step is the *Reporting phase*. The most important for this step is to adapt the report's design to the needs of the specific roles.

Following these steps should result in a well-functioning process of automated compliance checking, which is proven through answering the following questions.

### **3.) *How can the tool verify the requirements?***

The answer to this question is structured in three sections. Firstly, it was necessary to define the requirements which the tool has to fulfil. Afterwards, the system architecture had to be explored and finally, instructions for scripting the tool had to be developed.

The three main requirements that the tool has to fulfil are speed, robustness and extensibility. It is important to focus on achieving these requirements while scripting the tool.

Next, the system architecture consists of four main parts: requirements management software, designer, computational engine, and visualiser. Each of these has a specific role in one or more of the five steps of automated compliance checking.

The part of the answer on the third sub-question is answered by giving instructions on how to script all types of predefined functions that are needed for the operation of the tool. The three groups of predefined functions are: basic functions, methods for determination and reporting functions. The basic functions were briefly explored because these are ones already existing in the Grasshopper environment. The methods for determination are the central part of the script, and the focus was on instructions for developing this group. All methods for determination have a standardized structure of the script, and it consists of three parts: the data processing part, calculation part, and comparison. Finally, reporting functions were the last in scope, and these are straightforward. By following the proposed rules, the developer should be able to deliver a well functioning tool that can check many types of requirements.

### **4.) *How does the tool work in practice?***

To answer this question, it was necessary to test the tool on a real-world case. SWECO provided the building model for testing, and it was a building related to the energy sector. The seven requirements were defined, and the whole procedure proposed in the framework was applied to these requirements. The test was successful and proved that a well-functioning tool could be developed by following the proposed framework and instructions for scripting. Of course, the test brought up some of the advantages and disadvantages of the proposed framework and



prototype tool. The main benefits are the time saved on verification, the wide range of requirements that can be covered by this method, and finally, the designer does not have to be highly skilled in classic programming. Still, the process requires a lot of manual work because some steps can not be automated right now, also the tool is very dependant on the Grasshopper abilities, which do not have all functions necessary to perform the verification of all possible requirements. Finally, the tool can only work with Grasshopper models, which is not enough and the possibility to work on BIM models is necessary.

### 8.3. Impressions – strengths and weaknesses

After testing has been conducted, it can be concluded that the tool works properly, but it showed particular strengths and weaknesses.

#### Strengths

- **Time saving:** Significantly less time is spent on verification of requirements than in a situation when everything is manually checked and tracked. This becomes even more useful during designing a building while the designer is constantly making changes. With this tool, compliance with requirements can be checked almost instantly during the whole designing process. Of course, as already mentioned, to perform some checks, a little more time is needed (30 - 40sec), but that is negligible, taking into account a large amount of data that has to be checked. The test showed that tool can have big value in a standardized projects when the script is already built partly, and in projects which require a lot of iterations. On top of that, additional value from higher level of safety in the projects must be added plus time saved on rework. In both aspects automated tool outweighs the manual verification which is error prone.
- **Covers a wide variety of requirements:** Due to a large third-party community of Grasshopper users and many plug-ins for specific purposes, many unique requirements can be checked. For example, all previous researches were based only on geometry-related requirements and just a few on one or two specific requirements in the structural (Dhillon and Rai, 2017), building physics domain (Seghier, Ahmad and Lim 2019) or fire safety domain (Kinclova et al., 2020). But, by following the framework and instructions proposed in this project, it should be possible to cover a wide variety of requirements. There are many plug-ins for Grasshopper that are calculating features related to building physics or structural domain. Moreover, in the research is shown that both geometry related and structure related requirements could be covered. By using some other approaches proposed in previous projects, that would not be possible.
- **The designer does not have to be skilled in programming:** By following the proposed approach and using Grasshopper, the designer does not have to be highly skilled in programming. It is true that if he wants to take the role of the developer as well to adapt or extend the tool, he has to be highly skilled in Grasshopper, but that should not be a big

problem. With the rise of parametric design and the popularity of Grasshopper, the assumption is that designers will become more skilled in it with time. Also, for performing the checks, only basic knowledge of Grasshopper is necessary. In other researches, the designer must be highly skilled in programming or can use black-box solutions in which he can not adjust anything to a specific situation. Therefore, the approach proposed in this project solves both problems.

## Weaknesses

- **Still a lot of manual work:** The code checking as proposed in this project is still manual to a large extent. The designer has to build a script by using predefined functions, but also it has to input a lot of extra data that are not contained in the model. The problem here is a deficient level of details in Grasshopper models. Therefore the ability to import BIM models in the Grasshopper environment or perform the checks on some BIM platforms is crucial for broader tool usage.
- **Tool dependent on Karamba:** For checking the structural requirements current version of the tool works only with Karamba. That means it can only verify buildings that are analysed by using Karamba for FEM analysis. Luckily, Karamba is the most widely used FEM tool in the Grasshopper environment. Currently, it would be tough to enable checking models from FEM software outside of the Grasshopper environment.
- **At this moment, Grasshopper does not have all functions necessary to perform all checks:** Even though Grasshopper offers a variety in types of checks that could be performed, it still can not cover all checks. For example, Grasshopper does not have functions for determining the distance between two surfaces, and the designer has to come up with some unorthodox solution if he wants to check that. But, with the rise of its popularity, more functions should be added or scripted, specially for code checking. Still, that would require a very high knowledge of programming.

## 8.4. Assumptions and limitations

- **Grasshopper was used as a visual programming software:** At the beginning of the project, the preliminary decision was to choose a Grasshopper as a Visual Programming Language environment for scripting the tool. Since it is a crucial component of the system's architecture, it is understandable that the tool is very dependent on the Grasshopper's abilities. Firstly, the time spent on the checks depends on how fast can Grasshopper run the script and perform all calculations. This became clear in the check for finding an intersection between two objects when the time for one check surged over 30 seconds because 96 cables were checked against 17 piles. Also, the tool is restricted by the functions that are already developed in Grasshopper. For example, Grasshopper does not have an option for determining the distance between two surfaces, which narrows down the operating area of the tool.

Instead of Grasshopper, some other Visual Programming Language software could be used, for example, Dynamo. The advantage of Dynamo is its connection with Autodesk products, like Revit and Robot. Many designers use Revit and Robot, and for them, it would make sense to have a verification tool scripted in Dynamo. Also, objects taken from Revit have many properties written in themselves already, so the designer does not have to assign them only for the verification process. Still, the fact that Dynamo is operating only in the Revit environment gives a significant advantage to the Grasshopper in the field of code checking. The openBIM approach is very desirable to allow all designers to work with the tool, regardless of the BIM software they use. Furthermore, Grasshopper is more reliable than Dynamo, which still has a lot of bugs during operation.

- **Karamba 3D was used as a FEM software:** Karamba3D plug-in has been chosen as a FEM tool used in a building model preparation step for structure-related checks. There were other options, and one of them is Kiwi3D, but Karamba3D is the most widely used plug-in for structural FEM analysis in Grasshopper and has the most abilities. If some other plug-in had been chosen, the script inside the methods for determination would have been different, but the main structure and three standard parts would have been present again. Karamba3D has not yet been developed to the level of FEM software like Robot, Diana, RFEM, therefore the possibilities for checks are restricted due to its abilities. For example, Karamba3D is not a perfect plug-in for analyzing concrete structures, which makes the verification of concrete structures significantly more complex than steel. With the time and growth of Grasshopper's popularity, Karamba3D should evolve and improve its abilities, making more space for verification of different kinds of requirements.
- **The tool works with steel I and H sections only:** In the previous paragraph, it is mentioned that Karamba3D is significantly better suited for operating with steel than any other type of structure. Therefore, due to time restrictions, the tool is limited to working with steel I and H profiles that are Class 1,2 or 3. To prove the concept, this was detailed enough, but for everyday usage, the tool has to be extended. Adding other types of profiles or materials would not bring extra value to this project because the framework and instructions can be proven even with the chosen limitations. Following the instructions for scripting, the functions and framework for automating the verification process should be possible to include almost any other type of structure or profile in the tool. Of course, there are limitations imposed by the Grasshopper's and Karamba's abilities, but that has been covered already.
- **Microsoft Excel was used instead of any commercial requirements management software:** In this project, Microsoft Excel is used instead of BriefBuilder or any other requirements management software. The functions for synchronizing Grasshopper and Excel already exist, while API between Grasshopper and BriefBuilder or any other requirements management software has not been developed yet. That does not affect the project largely because it was possible to use the same standardized table in Excel as it would be in any requirements management software. Moreover, some project managers

even use Excel as an requirements management software. The programming of API between Grasshopper and requirements management software would have taken a lot of time and would not have given this project a lot of extra value.

- **The prototype tool can only perform seven checks:** For this project, only seven requirements were covered. The three requirements are geometry related, while four are verifying structural checks. This was enough to prove that concept works. It is shown that following the instructions for scripting and framework makes it possible to make a properly working tool and automate the verification process partly. Still, it is advisable to include even more checks in the prototype tool to test the scalability. It is still unknown how fast the tool would be with a large number of requirements running at the same time.
- **A standardized table for the breakdown of requirements is not suitable for all types of requirements:** It should be possible to deconstruct a large portion of building design requirements by using the standardised table. Still, it is expected that some requirements would not suit it. The main reason for that is a lack of standardized logic of requirements, which are written in human language and for the human mind, which is adaptable and learns from experience. Nevertheless, a large portion of requirements can be handled with this table, especially while the designer is the one who is building the code manually. After the tool becomes able to construct the script automatically, then it will be maybe necessary to find a better way of deconstructing the requirement. But also, if NLP methods become very advanced, perhaps the designer would not have even to deconstruct the requirement into pieces.

## 8.5. Contributions of this research

The major contributions from this research are highlighted below:

- Developed the general framework for automated compliance checking in five steps and explained how to approach each step in detail.
- Gave instructions for scripting the tool for automated code checking in a Grasshopper environment
- Scripted the prototype tool for checking three geometry related and four structure related requirements
- Tested current possibilities for automation of code checking
- Set a base for future research in the field
- Gave detailed recommendations for future research on the topic

## 9. Conclusions

This project has been conducted to develop an improved approach to automation of building design verification, focussing on creating a framework for automated compliance checking and modelling a prototype tool in the Visual Programming Language environment to prove the approach. This study has led to the following conclusions:

- By analysing the Building Decree 2012 and literature related to requirements management in the construction industry, it can be concluded that there are two main types of requirements: functional and performance. Also, it is shown in the project that performance requirements are suitable for verification with the proposed approach due to its quantitative nature. On the contrary, functional requirements are not quantifiable and therefore can not be checked with tools modelled in a visual programming language.
- Testing on the real-world building model shows that a five-step approach for automated testing of building design works and can be used. Still, a framework used in this project has a significant amount of manual work, and extra steps must be taken to reach the fully automated process.
- Testing the prototype tool on a building model shows that the proposed system architecture and instructions for scripting the tool can result in a well-operating tool. The system architecture must have four components: designer, computational engine, requirements management software and visualiser.
- The test showed that a prototype tool with clear advantages over manual compliance checking can be scripted by using Grasshopper. The biggest strengths of the presented approach are speed, a wide variety of checks that can be covered and the fact that the designer does not have to be skilled in general programming.
- The prototype tool also has some weaknesses that came up during the testing. The most important are the limited range of functions in Grasshopper that complicates scripting of some methods for determination, tool is dependent only on Karamba3D models for the structural domain, and the tool requires help from a designer to perform checks.
- Test of the tool proved that the Visual programming language environment is a great platform for developing a white-box approach for automated compliance checking.

To conclude, this project showed how to approach the automation of the compliance checking from the perspective of both designer and developer of the tool. Both the steps of automated compliance checking and instructions on how to achieve it were given. Due to a large number of requirements and different building designs, it would be tough to develop one general

solution that works for every building and scenario. Still, this project gave a general approach that can result in a higher level of automation for design justification. This project serves as a starting point for future researches in the field, and with further effort, the future of the automated testing of building design is bright.

## 10. Recommendations

This project presents the first study of a proposed framework and prototype tool for automated compliance checking. Concluding that it is a feasible approach with a lot of potential, areas for further investigation will be given. In the recommendations, the structure from the framework will be followed, therefore the next moves for each step of the framework will be presented.

### **Step 1: Requirements defining and logical structuring into RMS**

#### **Continue with the research of automatically deconstructing requirements by using machine learning techniques:**

Since there is already much research in this area, the recommendation is to continue until automation is reached. The most promising techniques are based on machine learning methods, and therefore focus should be on these. Especially because requirements are written in a non-standard logic based on human language, the tool must be adaptable, which rule-inferencing methods do not provide. Another essential aspect to explore is how to construct the requirements management software table. More precisely, which categories are needed to cover as many requirements as possible and that the tool can automatically disassemble the rule in those categories.

#### **Standardize the vocabulary and logic for defining requirements:**

Furthermore, a big move would be to standardize the project vocabulary and logic of requirements. Different companies and stakeholders use different terms for expressing their provisions. Standardization in that sense would make automation significantly easier. In the final stage of development, the designer would only have to write the requirement as it is given, and the tool would be able to break it down and put in requirements management software.

### **Step 2: Interpretation of requirements**

#### **Explore the methods for the automation of the script building in the Grasshopper:**

Currently, the transformation of requirements from the requirements management software table into Grasshopper script is a manual process. The next step would be to explore methods to automate script building.

The tool must use machine learning or rule-inferencing methods to read the requirements management software table and recognize which functions to use and how to connect them. Of course, to synchronize requirements management software and Grasshopper, the appropriate two-way API must be developed.

Two extra areas must be explored before automating this step: a sufficient level of details in the model and more standardized terms for the definition of requirements, as proposed in the previous step. A sufficient level of details is a part of the building model preparation step and will be explained there, but it is evident that for automated script building, the tool needs

identifications of every object and all data related to it. If that is missing, the designer must input the data manually and connect the appropriate object to the script.

### **Step 3: Building model preparation**

#### **Explore how to connect Grasshopper with BIM software:**

The first recommendation is to research the implementation of the openBIM approach into Grasshopper.

This is of crucial importance to have a tool for wide usage because the majority of designers prefer to use one of the BIM software for making a model and making an extra Grasshopper model would be unnecessarily time-consuming. Furthermore, Grasshopper models lack much data included in the BIM model, which makes the life of a designer harder. In the Grasshopper designer must input those missing data manually. Therefore, enabling the checking on BIM models from other software would give this idea potential commercializing value. The potential solution could be the use of cloud-based platforms such as Packhunt.io, which enables the communication between different software and Grasshopper.

#### **Explore methods for semantic enrichment of building models:**

The second recommendation in this step is exploring the methods for semantic enrichment of models. An example of such research is the work of Bloch and Sacks (Bloch and Sacks, 2018), who compared machine learning and rule-based inferencing for the semantic enrichment of BIM models. Since BIM models usually lack some data, it would be helpful to have a method to add those data automatically. That particular paper explores how the software can recognize the room's function on its own, without the designer's input of borders and functions for each room. For the rule checking, this could have immense value and save a lot of time that the designer has to give for providing all the details about the building. From that research, it seems that machine learning is a better option for this particular purpose, but there are other situations in which rule-based inferencing is more feasible. Therefore, for automation of code checking it has to be looked in all extra data that are usually needed to perform the verification, and then the best method for automatically assigning missing data can be explored.

### **Step 4: Checking phase**

#### **Test the scalability of the tool with a large number of requirements:**

The most important feature to investigate in the checking phase is the scalability of the tool. In this research, only seven requirements were run simultaneously, and one was making minor problems in operation. Hundreds of clauses and provisions are set for a building in the actual project, which means the required computational power will be much larger. Therefore, the research with more checks should be conducted to get an impression of the actual number of requirements that could be run at the same time. Furthermore, the idea of cloud computational power should be explored because it has a lot of potentials to increase the extent of the checks. Of course, with the constant development of hardware components and more optimized software, this remark will become less relevant.



## **Step 5: Reporting phase**

### **Explore how to develop a visual detection of failed requirements in Rhino:**

For the reporting phase, it could be explored how to visualize objects that are not fulfilling the requirements to make detection of those objects easier and straightforward. Since the eyes are one of the humans basic senses, it would be helpful to have the ability to visualize failed objects in the model. This should not be problematic for skilled software developers because this can be seen in many software.

### **Explore the implementation of the automated compliance checking:**

The last recommendation is not related to any of these five steps but to the implementation of the automated code checking process in practice. Once this concept is fully developed, the companies will have to make some operational changes to implement it, and this has to be investigated. For example, some of the questions that should be answered are:

What new roles will be needed in the designing process? (developer and its role)

What new obligations will the designer have? (tracking the verification; one leading designer for verification or everyone for themselves?)

Who takes responsibility if the code checking software made a mistake? ( depends if the software is open-source or developed by some company etc.)

Is it economically feasible to implement automated compliance checking? (cost of software, training of staff, benefits of time saved, increased safety and less rework etc.)

There are still many areas of this topic that must be further investigated, so the framework and ideas from this project should serve as a basis for further research.

## References

- Bloch T., Sacks R., (2018), *Comparing machine learning and rule-based inferencing for semantic enrichment of BIM models*, *Automation in Construction*, 91, p. 256-272
- Borrmann A., Hyvärinen, J., Rank, E. (2009), *Spatial constraints in collaborative design processes*, *International Conference on Intelligent Computing in Engineering*, Berlin, Germany
- Bouwbesluit 2012, (2012), *Staatsblad van het Koninkrijk der Nederlanden*
- Chipman T., Liebich T., Weise, M., (2015), *mvdXML - Specification of a standardized format to define and exchange Model View Definitions with Exchange Requirements and Validation Rules*
- Coenders J., Rolvink A., (2014). *Structured automated code checking through structural components and systems engineering*, *Proceedings of the IASS-SLTE 2014 Symposium "Shells, Membranes and Spatial Structures: Footprints"*, Brazil, p. 15-19
- Daum S., Borrmann A., (2013), *Checking Spatio-Semantic Consistency of Building Information Models by Means of a Query Language*, *Proc. Of the International Conference on Construction Applications of Virtual Reality*, London, p. 30-31
- Dhillon K.R., Rai S.H., (2017), *Automated Building Code Compliance for Structural Safety*, *International Journal of Computational Engineering Research*, 07(08), p. 2250-3005
- Dimyadi J., Solihin W., Eastman C., Amor R., (2016), *Integrating the BIM Rule Language into Compliant Design Audit Processes*, *33rd CIM International Conference on IT in Construction*
- Eastman C., Lee J.M., Jeong Y.S., Lee J.K., (2009), *Automatic rule-based checking of building designs*, *Automation in Construction*, 18(8), p. 1011–1033.
- EN 1993-1-1, (2005), *Eurocode 3: Design of steel structures- Part 1-1: General rules and rules for buildings*,
- Food4Rhino, *GHPython*, viewed 20 Sep 2021a,  
<https://www.food4rhino.com/en/app/ghpython>
- Food4Rhino, *Pterodactyl*, viewed 20 Sep 2021b,  
<https://www.food4rhino.com/en/app/pterodactyl>

- Food4Rhino, *TT Toolbox*, viewed 20 Sep 2021c, <https://www.food4rhino.com/en/app/tt-toolbox>
- Han C. S., Kunz J. C., Law K. H. (1998) *A Client/Server Framework for On-line Building Code Checking*, Stanford University
- Hils D.D., (1992), *Visual languages and computing survey. Data flow visual programming languages*, *Journal of Visual Languages and Computing*, 3(1), p. 69-101
- Hjelseth E., Nisbet N., (2011), *Capturing normative constraints by use of the semantic mark-up (RASE) methodology*, 28th International Conference- Applications of IT in the AEC Industry
- Kamara J. M., Anumba C. J., Evbuomwan N. F. O. (2000), *Establishing and processing client requirements—a key aspect of concurrent engineering in construction*, *Engineering, Construction and Architectural Management*, 7(1), p. 15–28
- Karamba3D, *Karamba*, viewed 20 Sep 2021, <https://www.karamba3d.com/>
- Kim H., Lee J.K., Shin J., Choi J., (2017), *BIM supported Visual Language to define building design regulations*, *Proceedings of the 22nd CAADRIA Conference*, Suyhou, China, p. 603-612
- Kinclova K., Boton C., Blanchet P., Dagenais C., (2020), *Fire Safety in Tall Timber Building: A BIM-Based Automated Code-Checking Approach*, *Buildings*, 10(7), p. 121
- Lee J.K., (2011), *Building environment rule and analysis (BERA) language and its application for evaluating building circulation and spatial program*, Georgia Institute of Technology
- LegalRuleML\_TC, (2012), *LegalRuleML*, OASIS, [https://lists.oasis-open.org/archives/legalruleml/201208/msg00040/LegalRuleML-palmirani2012\\_-RuleML2012v3.pdf](https://lists.oasis-open.org/archives/legalruleml/201208/msg00040/LegalRuleML-palmirani2012_-RuleML2012v3.pdf),
- Myers B.A., (1990), *Taxonomies of visual programming and program visualization*, *Journal of Visual Languages and Computing*, 1(1), p. 97-123
- Nawari, N.O. (2018) *Building Information Modeling: Automated Code Checking and Compliance Processes*; CRC Press: Cleveland, USA
- Nawari N.O., Alsaffar A. (2015), *Understanding computable building codes*, *Journal of Civil Engineering and Architecture*, 3(6), 163-172

- NovaCityNets, (2016), *FORNAX*, <http://www.novacitynets.com/fornax/index.htm>
- Park S., Lee Y.C., Lee J.K., (2016), *Definition of a domain-specific language for Korean building act sentences as an explicit computable form*, ITCON, 21(26), p. 422-433
- Preidel C., Borrmann A., (2016) *Towards code compliance checking on the basis of a visual programming language*, *Journal of Information Technology in Construction* ITCON, 21, p. 402-421
- Preidel C., Borrmann A., (2015) *Automated Code Compliance Checking Based on a Visual Language and Building Information Modeling*, *Proceedings of the 32nd ISARC*, p. 1-8
- Rao, S., (2021), *Semantic enrichment of design requirements using Object Type Libraries for automated verification*. MSc. Thesis, TU Delft.
- Schiffer, S. (1998), *Visuelle Programmierung: Grundlagen und Einsatzmöglichkeiten*, Addison-Wesley, Bonn.
- Seghier T.E., Ahmad M.H., Lim Y.W., (2019), *Automation of Building Envelope Thermal Performance Assessment Using Computational BIM*
- Solihin, W. (2004), *Lessons learned from experience of code-checking implementation in Singapore*, BuildingSMART Conference, Singapore.
- Solihin W. (2015), *A Simplified BIM Data Representation Using a Relational Database Schema for an Efficient Rule Checking System and Its Associated Rule Checking Language*, Georgia Institute of Technology
- Solihin W., Dimyadi J., Lee Y. (2019), *In search of open and practical language driven BIM based automated rule checking systems*. *Advances in Informatics and Computing in Civil and Construction Engineering* (January), p. 577–585.
- W3C, (2004), *SWRL: A Semantic Web Rule Language - Combining OWL and RuleML*, W3C, online at <http://www.w3.org/Submission/SWRL/>
- Zhang J., El-Gohary N.M., (2017), *Integrating semantic NLP and logic reasoning into a unified system for fully-automated code checking*, *Automation in Construction*, 73, pp. 45-57

## List of figures

Figure 1 Research strategy and relation with sub-questions and output.....	8
Figure 2. Project requirements (Kamara, Anumba and Evbuomwan, 2000).....	9
Figure 3 Internal structure of predefined function which is not accesible to designer .....	18
Figure 4 Four elements of a system architecture .....	24
Figure 5 The components used and data-flow per each step of the process .....	27
Figure 6. Concept of the method for determination function in Grasshopper .....	29
Figure 7. The specification of general description and data type used(bool) for a particular inlet .....	31
Figure 8.The general example of component's description .....	35
Figure 9. The component for determining Area and its description .....	35
Figure 10. Distance between objects function .....	37
Figure 11. The internal structure of the function for determining a distance between objects	38
Figure 12. Method for checking flexural buckling .....	40
Figure 13. The internal structure of the function for verifying the flexural buckling .....	41
Figure 14. Component for importing Excel table into Grasshopper.....	43
Figure 15. Component for writing compliance in excel .....	43
Figure 16. Inside structure of the component for writing compliance in excel .....	44
Figure 17. Input parameters for the Reporting function .....	44
Figure 18. Inside structure of the function for generating the report.....	45
Figure 19. The model of the building used for a test case .....	46
Figure 20. Script for code checking .....	48
Figure 21. Table of requirements with updated compliance.....	49
Figure 22. Details of the requirement 2.1. ....	49
Figure 23. Updated compliance after the limit value has been changed .....	50
Figure 24. Report for requirement 7.1. ....	51
Figure 25 Internal structure of the distance between objects component .....	72
Figure 26 Data filter of the distance between objects component .....	73
Figure 27 First component for determining object type .....	73
Figure 28 Second component for determining object type .....	74
Figure 29 First component for extracting the data from the requirements management software table .....	74
Figure 30 Second component for extracting the data from requirements management software table.....	75
Figure 31 Lines for determining the distance between the different combinations of object types .....	75
Figure 32 Lines for determining the distance between different combinations of object types (nr 2).....	76
Figure 33 Comparison part of the distance between objects script .....	77
Figure 34 Internal structure of the flexural buckling component .....	78
Figure 35 Data processing part of the script for flexural buckling check.....	79
Figure 36 First component of the data processing .....	79

Figure 37 Second component of the data processing.....	80
Figure 38 Third component of the data processing .....	80
Figure 39 Fourth component of the data processing.....	81
Figure 40 Calculation and comparison part of the flexural buckling check .....	82
Figure 41 Reporting part of the flexural buckling check.....	83
Figure 42 Component for structuring the data for reporting.....	83

## List of tables

Table 1. The types of requirements in a project (Kamara, Anumba and Evbuomwan, 2000) ..	9
Table 2. The characteristics of different groups of requirements .....	15
Table 3. Template for the input of requirements into requirements management software ....	17
Table 4. The template for describing the method for determination .....	34
Table 5. Description of the function for determining a distance between objects.....	39
Table 6. Description of the function for verifying the flexural buckling.....	42
Table 7. List of requirements input in a standardized table .....	47
Table 8. Comparison of time spent on checks in case of manual verification and verification by the tool .....	52
Table 9. Comparison of time spent on verification after a few changes have been made on the model.....	52

## Appendices

### Appendix A

The details about a script for two methods for determination will be explained in this chapter. First, the method for checking the distance between objects is covered. Afterwards, the check for flexural buckling is presented.

#### A.1. Distance between objects

This method is briefly explained in chapter 6.1, but here more details will be given. First, the internal structure of the method is shown in Figure 25. The script has three main parts: data filter, calculation part and comparison part, but that is already explained in the central part of the report.

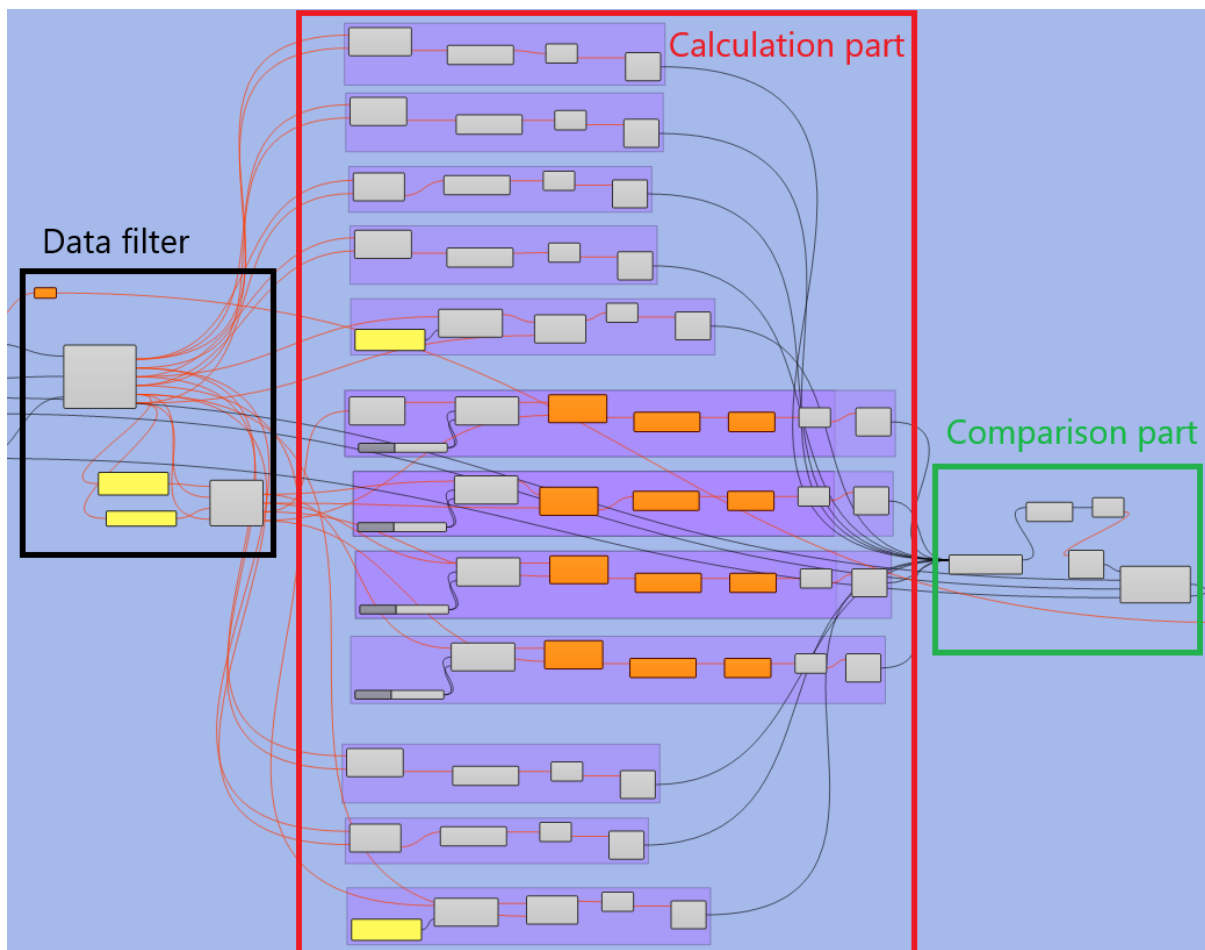


Figure 25 Internal structure of the distance between objects component

The data filter part has two sub-parts shown in Figure 26. The first part gets the objects between which distance has to be determined. Since objects could be points, curves, surfaces or breps, it is first necessary to determine the object's type. The second part gets the data from the

requirements management software table and extracts the limit value, limiting operator and rule number.

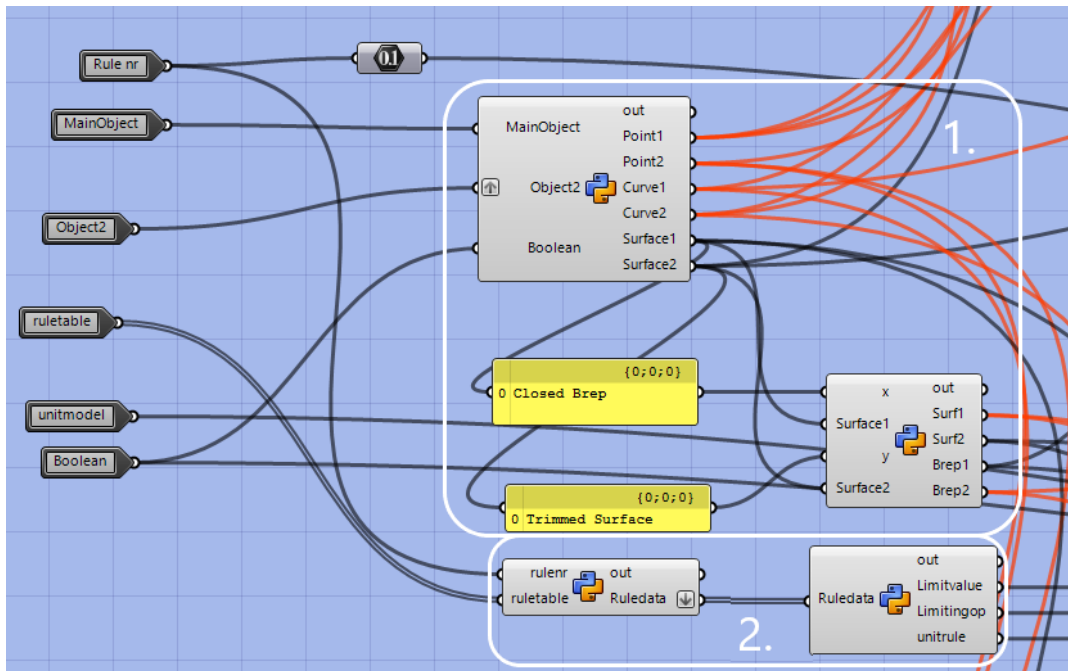


Figure 26 Data filter of the distance between objects component

The GHPython code of the first two components is shown in Figure 27 and Figure 28.

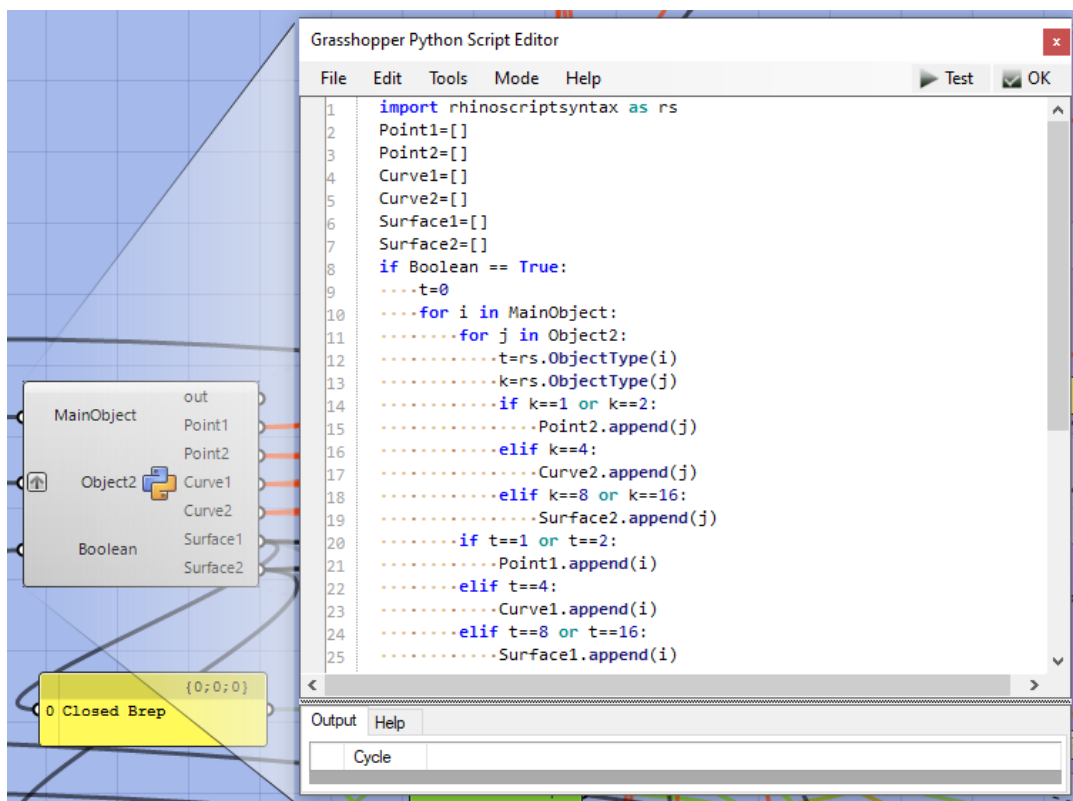


Figure 27 First component for determining object type



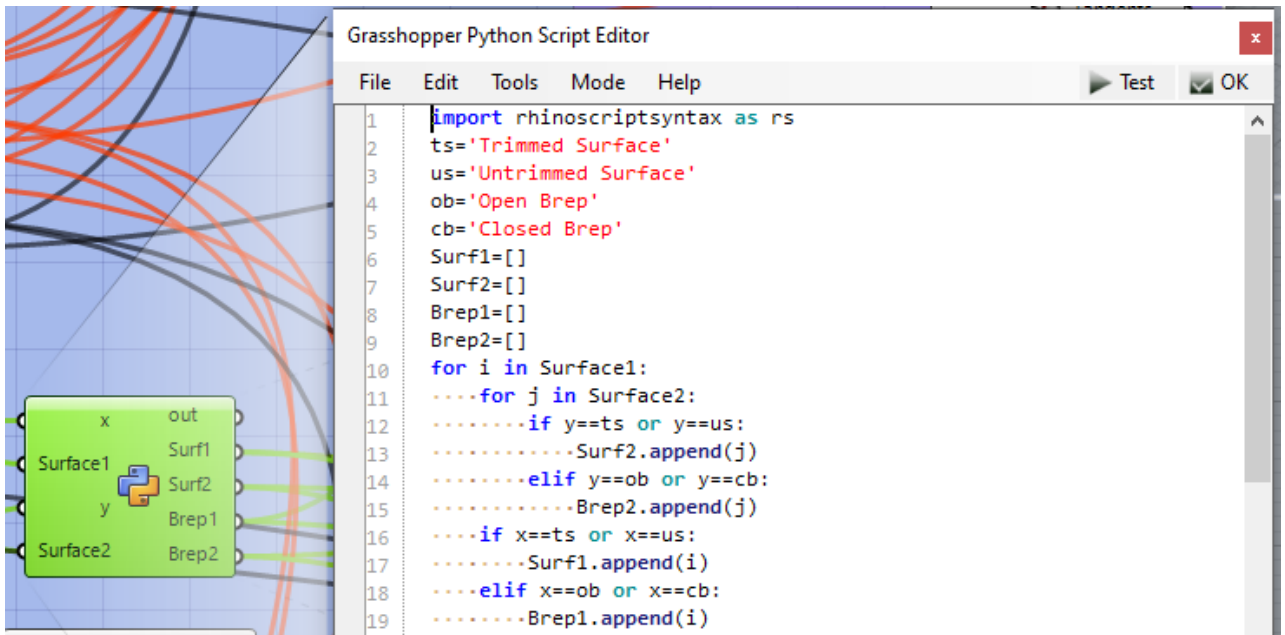


Figure 28 Second component for determining object type

The GHPython code of the components for extracting the data from the requirements management software table is shown in Figure 29 and Figure 30.

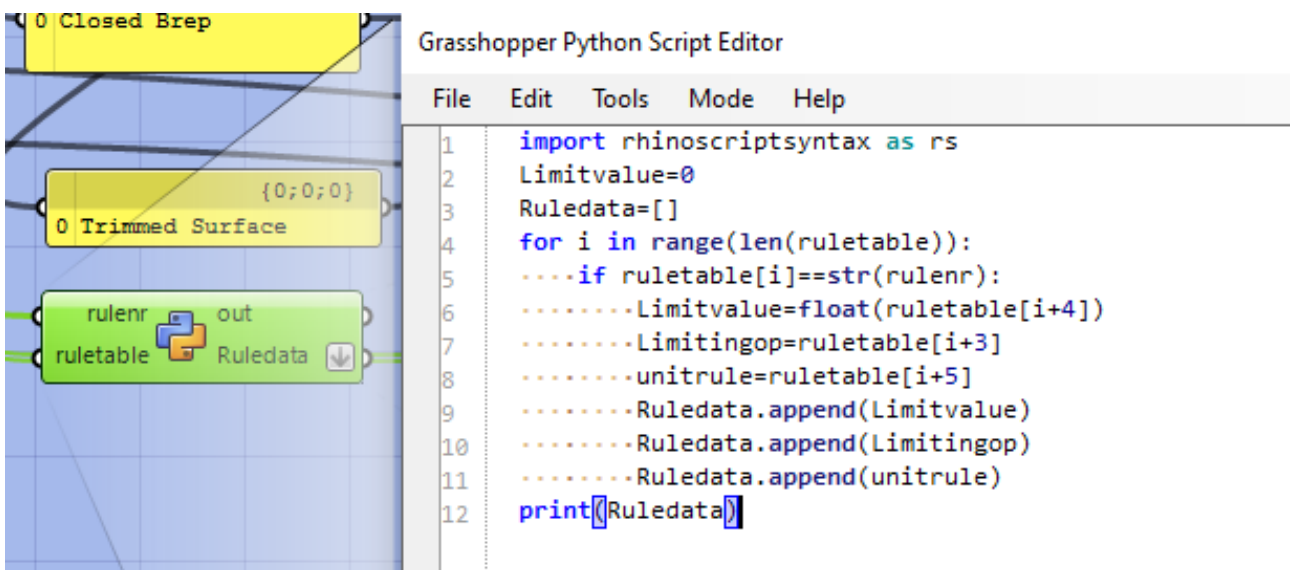


Figure 29 First component for extracting the data from the requirements management software table

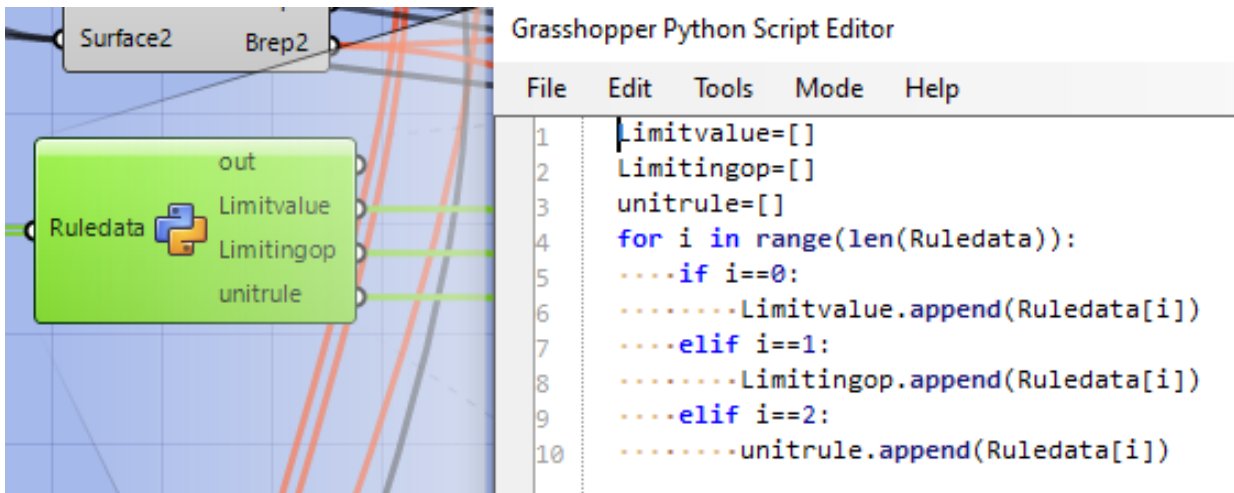


Figure 30 Second component for extracting the data from requirements management software table

The second part of the script consists of 12 different lines, each for one possible combination of object types. It is shown in Figure 31 and Figure 32.

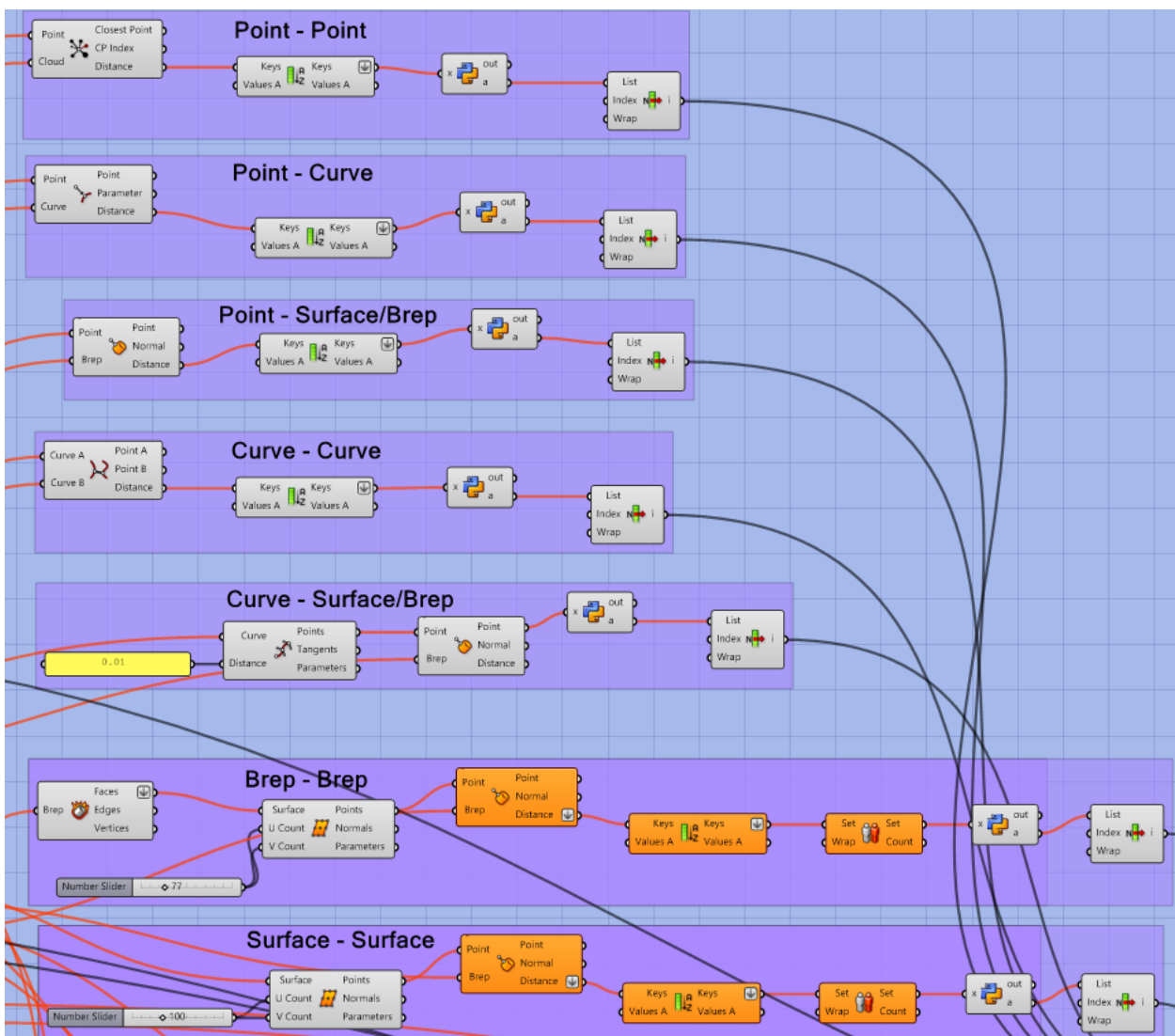


Figure 31 Lines for determining the distance between the different combinations of object types

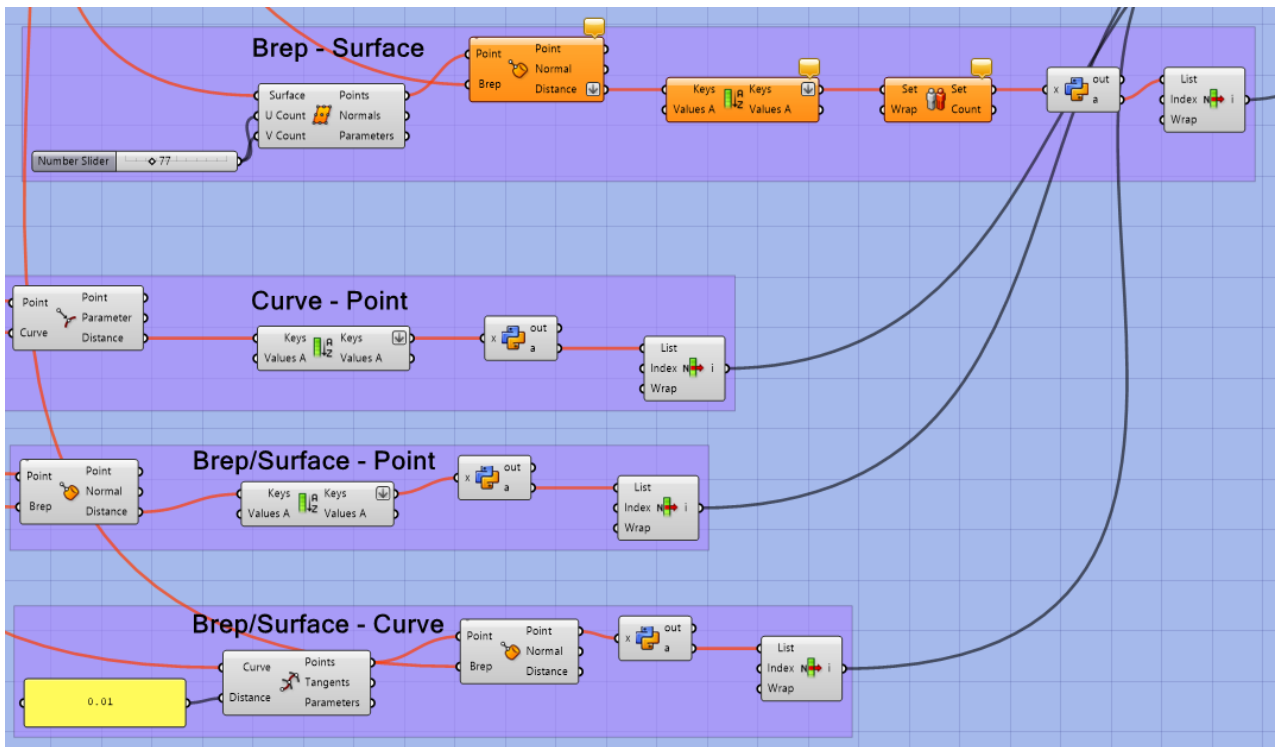


Figure 32 Lines for determining the distance between different combinations of object types (nr 2)

Finally, the third part of the script compares the calculated distance and limiting value obtained from the requirements management software table. The GHPython component first takes into account units used and then compares two values. The code is shown in Figure 33.

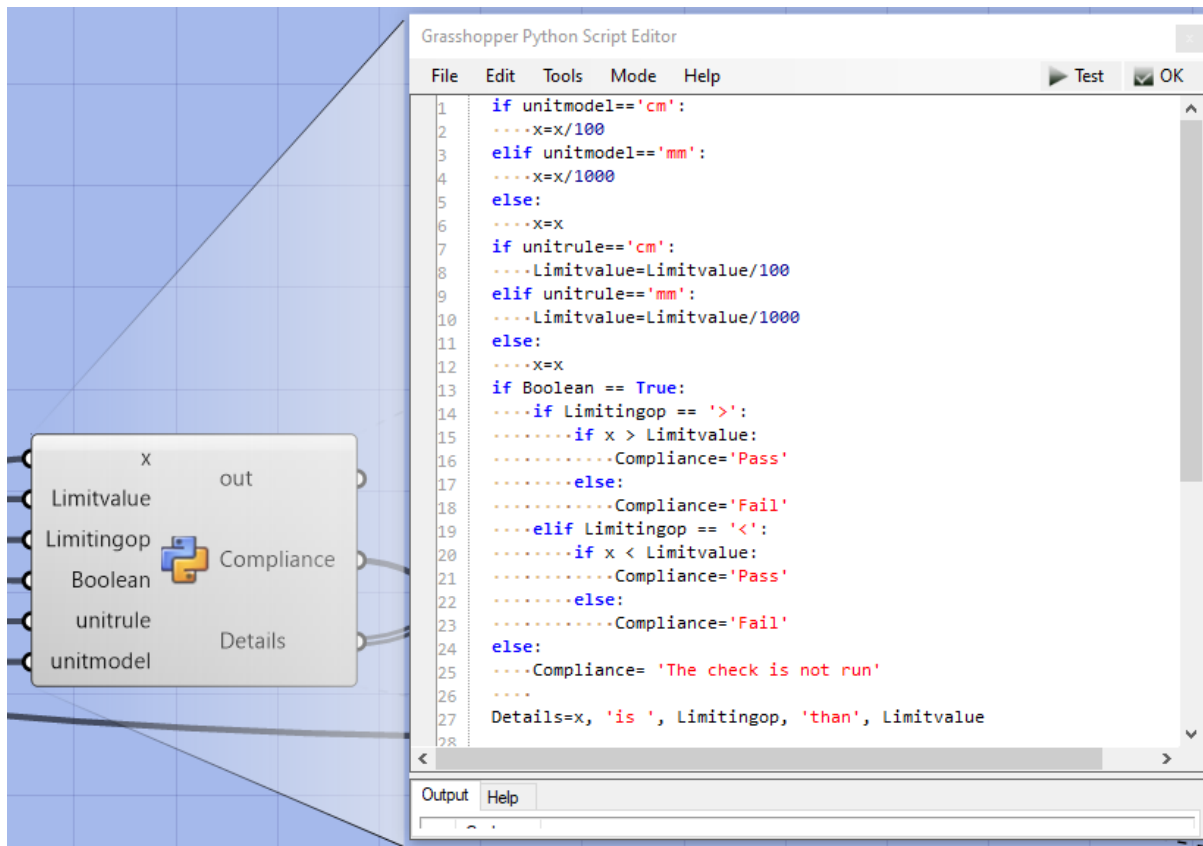


Figure 33 Comparison part of the distance between objects script

## A.2. Flexural buckling

This method is briefly explained in chapter 6.2, but here more details will be given.

First, the internal structure of the method is shown in Figure 34. The script has three main parts: data filter, calculation part and comparison part combined in one and finally reporting part; but that is already explained in the central part of the report. In chapter 6.2. the method is shown without reporting part because reporting is specific for each user and not standardized. Therefore, it is only shown here where specific code scripted for the purpose of this project is presented.

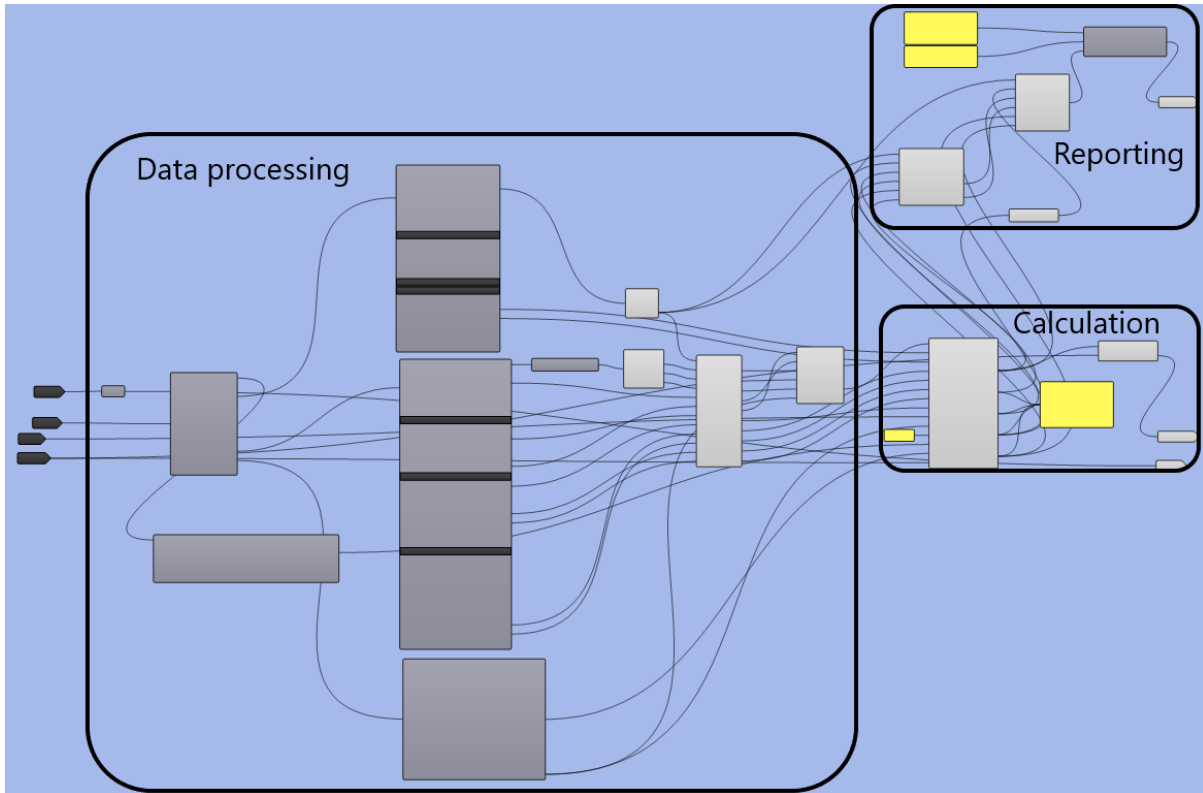


Figure 34 Internal structure of the flexural buckling component

Firstly, the data processing part takes the data from the Karamba model, extracts it, and structures it in a useful way for future usage. Beside components from Karamba, it also has four components scripted in GHPython just for this purpose. Four components are shown in Figure 35. The first function extracts the cross-section type for each element, and its code is shown in Figure 36. The second function extracts the geometric data about steel profiles, more precisely the width of flange, web width, and radius. The structure of the second component is shown in Figure 37. The third function structures all required data to classify steel profiles, and its code is shown in Figure 38. Lastly, the fourth function determines the class of the cross-section by following formulas given in Eurocode. The script for the classification of cross-sections is shown in Figure 39.

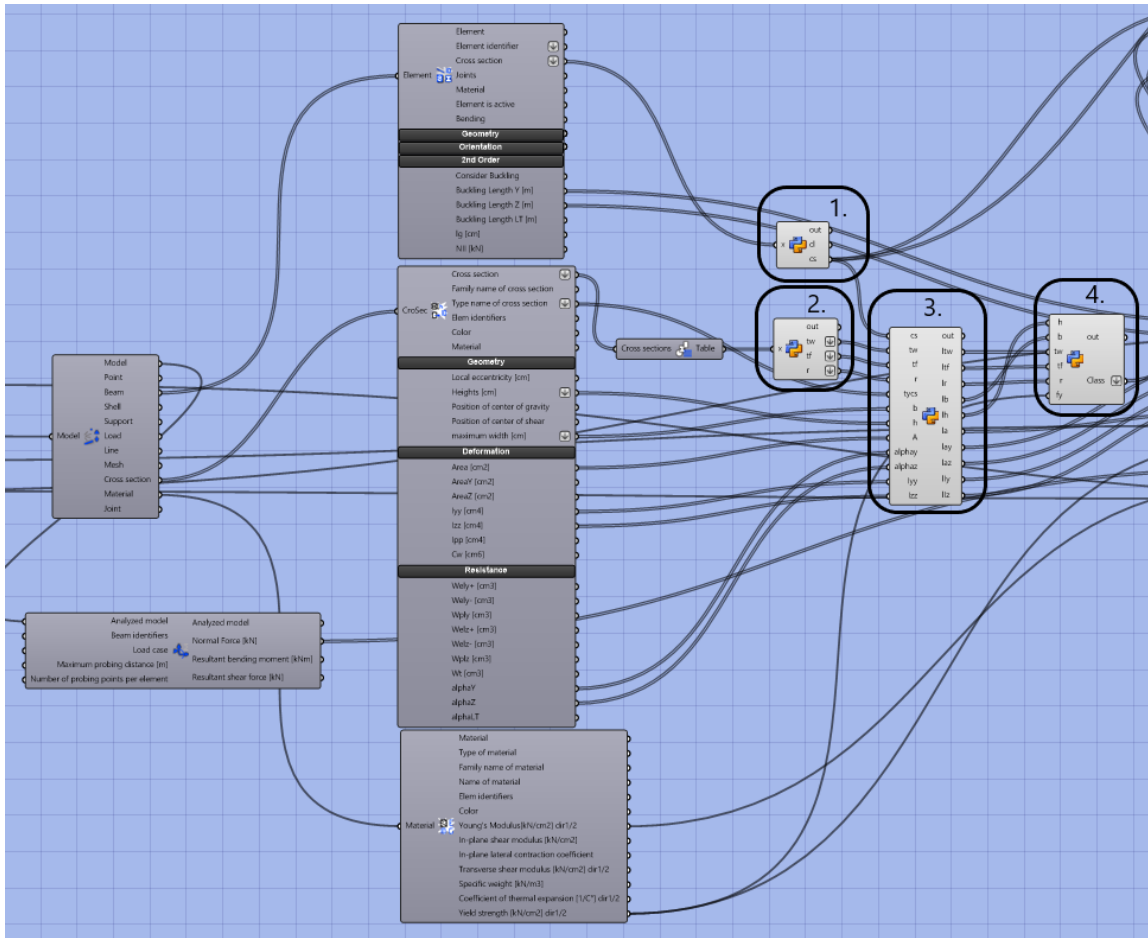


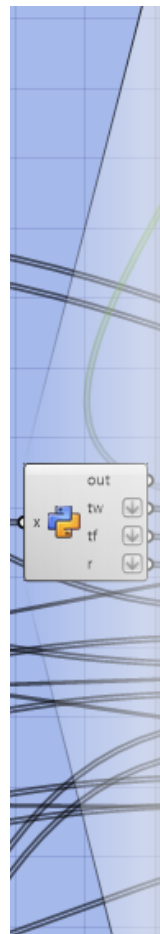
Figure 35 Data processing part of the script for flexural buckling check

```

File Edit Tools View Help
1 import rhinoscriptsyntax as rs
2 cl=[]
3 cs=[]
4 rcl=[]
5 rcs=[]
6 k=0
7 s=""
8 z=""
9 for i in range(len(x)):
10     for j in range(len(x[i])):
11         if x[i][j]=="":
12             k=k+1
13             if k==1:
14                 rcl.append(x[i][j])
15                 if k==3:
16                     rcs.append(x[i][j])
17             rcs.pop(0)
18             rcl.pop(0)
19             s=" ".join(rcl)
20             z=" ".join(rcs)
21             cl.append(s)
22             cs.append(z)
23             rcs=[]
24             rcl=[]
25             k=0
26 print(cl)
27 print(cs)

```

Figure 36 First component of the data processing



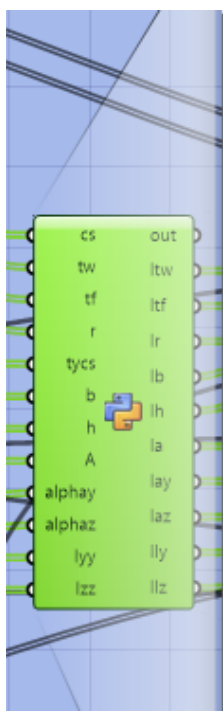
The component interface for Figure 37 shows an input port 'x' and an output port 'out'. The 'out' port is connected to five sub-ports: 'tw', 'tf', 'r', 'rtw', and 'rtf'. Each sub-port has a corresponding arrow icon pointing downwards.

```

1  import rhinoscriptsyntax as rs
2  tw=[]
3  tf=[]
4  r=[]
5  rtw=[]
6  rtf=[]
7  rr=[]
8  k=0
9  l=''
10 s=''
11 z=''
12 for i in range(len(x)):
13     if i>1:
14         for j in range(len(x[i])):
15             if x[i][j]==";":
16                 k=k+1
17                 if k==6:
18                     rtw.append(x[i][j])
19                     if k==8:
20                         rtf.append(x[i][j])
21                         if k==11:
22                             rr.append(x[i][j])
23                             rtw.pop(0)
24                             rtf.pop(0)
25                             rr.pop(0)
26                             s=''.join(rtw)
27                             z=''.join(rtf)
28                             l=''.join(rr)
29                             tw.append(float(s))
30                             tf.append(float(z))
31                             r.append(float(l))
32                             rtw=[]
33                             rtf=[]
34                             rr=[]
35                             k=0

```

Figure 37 Second component of the data processing



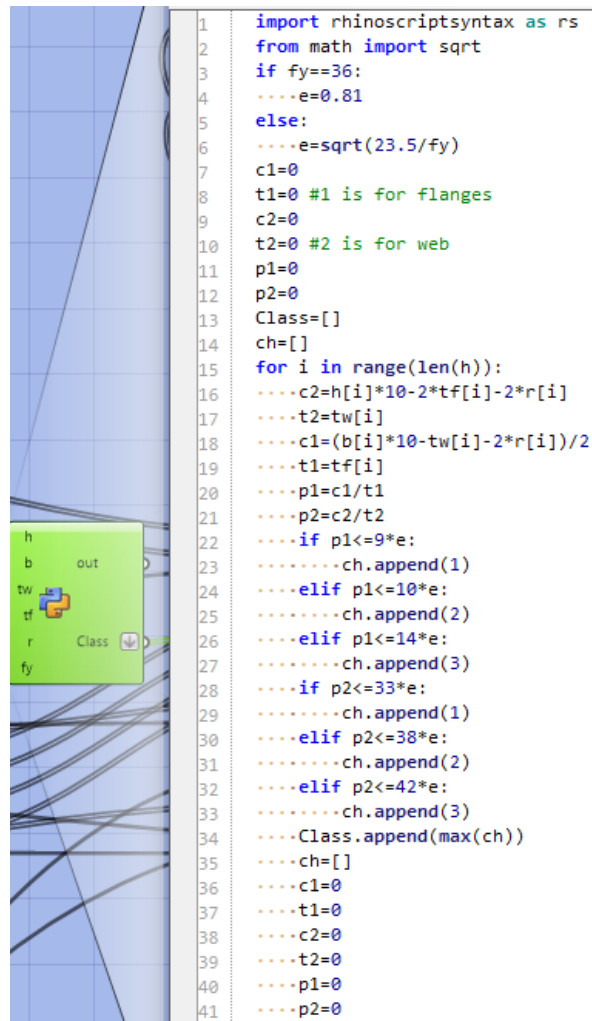
The component interface for Figure 38 shows an input port 'cs' and an output port 'out'. The 'out' port is connected to twelve sub-ports: 'ltw', 'ltf', 'lr', 'lh', 'lb', 'la', 'lay', 'laz', 'lly', and 'liz'. Each sub-port has a corresponding arrow icon pointing downwards.

```

1  ltw=[]
2  ltf=[]
3  lr=[]
4  lh=[]
5  lb=[]
6  la=[]
7  lay=[]
8  laz=[]
9  lly=[]
10 liz=[]
11 for i in range(len(cs)):
12     for j in range(len(tycs)):
13         if cs[i]==tycs[j]:
14             ltw.append(tw[j])
15             ltf.append(tf[j])
16             lr.append(r[j])
17             lh.append(h[j])
18             lb.append(b[j])
19             la.append(A[j]/10000)
20             lay.append(alphay[j])
21             laz.append(alphaz[j])
22             lly.append(Iyy[j]/(100**4))
23             liz.append(Izz[j]/(100**4))

```

Figure 38 Third component of the data processing



```

1  import rhinoscriptsyntax as rs
2  from math import sqrt
3  if fy==36:
4  ....e=0.81
5  else:
6  ....e=sqrt(23.5/fy)
7  c1=0
8  t1=0 #1 is for flanges
9  c2=0
10 t2=0 #2 is for web
11 p1=0
12 p2=0
13 Class=[]
14 ch=[]
15 for i in range(len(h)):
16 ....c2=h[i]*10-2*tf[i]-2*r[i]
17 ....t2=tw[i]
18 ....c1=(b[i]*10-tw[i]-2*r[i])/2
19 ....t1=tf[i]
20 ....p1=c1/t1
21 ....p2=c2/t2
22 ....if p1<=9*e:
23 .....ch.append(1)
24 ....elif p1<=10*e:
25 .....ch.append(2)
26 ....elif p1<=14*e:
27 .....ch.append(3)
28 ....if p2<=33*e:
29 .....ch.append(1)
30 ....elif p2<=38*e:
31 .....ch.append(2)
32 ....elif p2<=42*e:
33 .....ch.append(3)
34 ....Class.append(max(ch))
35 ....ch=[]
36 ....c1=0
37 ....t1=0
38 ....c2=0
39 ....t2=0
40 ....p1=0
41 ....p2=0

```

The image shows a Python script in a code editor. On the left side, there is a variable inspector window with a green background. It lists variables: 'h', 'b', 'out', 'tw', 'tf', 'r', 'fy', and 'Class'. The 'Class' variable is highlighted with a dropdown arrow. The script itself is a loop that iterates over the length of 'h'. It calculates 'c1', 't1', 'c2', and 't2' for each element. Then it calculates 'p1' and 'p2'. It uses conditional logic to append values to 'ch' based on 'p1' and 'p2' values compared to 'e'. Finally, it appends the maximum value of 'ch' to 'Class' and resets 'ch' for the next iteration.

Figure 39 Fourth component of the data processing

Inside the calculation part, the most important component calculates the resistance of the elements and then compares it to real value. The code of that component is shown in Figure 40.



Class	out
bucklency	
bucklencz	Compliance
impfacy	
impfacz	
lyy	Ncrity
lzz	
A	Ncritz
unit	
fy	
ym1	Maxforc
Maxforc	
E	Utilization
Boolean	

```

1  from math import pi
2  from math import sqrt
3  Utilization=[]
4  Compliance=[]
5  Ncrity=[]
6  Ncritz=[]
7  if fy==36.0:
8  ....fy=35.5
9  fy=fy*10000
10 E=E*10000
11 if unit=='m':
12 ....um=1
13 elif unit=='cm':
14 ....um=100
15 elif unit=='mm':
16 ....um=1000
17 if Boolean==True:
18 ....for i in range(len(Class)):
19 .....if Class[i]==1 or Class[i]==2 or Class[i]==3:
20 .....    lambda1=pi*(sqrt(E/fy))
21 .....    Ncry=((pi**2)*E*Iyy[i])/((bucklency[i]/um)**2)
22 .....    Ncrz=((pi**2)*E*Izz[i])/((bucklencz[i]/um)**2)
23 .....    lmbdy=sqrt((A[i]*fy)/Ncry)
24 .....    lmbdz=sqrt((A[i]*fy)/Ncrz)
25 .....    sigmy=0.5*(1+impfacy[i]*(lmbdy-0.2)+lmbdy**2)
26 .....    sigmz=0.5*(1+impfacz[i]*(lmbdz-0.2)+lmbdz**2)
27 .....    hiy=1/(sigmy+sqrt((sigmy**2)-(lmbdy**2)))
28 .....    hiz=1/(sigmz+sqrt((sigmz**2)-(lmbdz**2)))
29 .....    if hiy>1.0:
30 .....        hiy=1.0
31 .....    if hiz>1.0:
32 .....        hiz=1.0
33 .....    Nby=hiy*(A[i]*fy)/(ym1)
34 .....    Nbz=hiz*(A[i]*fy)/(ym1)
35 .....    elif Class[i]==4:
36 .....        Ncrd='The profile is Class 4'
37 .....        if abs(Maxforc[i])<=Nby and abs(Maxforc[i])<=Nbz:
38 .....            Compliance.append('Pass')
39 .....        elif abs(Maxforc[i])>Nby or abs(Maxforc[i])>Nbz:
40 .....            Compliance.append('Fail')
41 .....        Ncrity.append(Nby)
42 .....        Ncritz.append(Nbz)
43 .....        if Nby>Nbz:
44 .....            Utilization.append(abs(Maxforc[i])/Nbz)
45 .....        else:
46 .....            Utilization.append(abs(Maxforc[i])/Nby)
47 else:
48 ....Compliance='The check is not run'

```

Figure 40 Calculation and comparison part of the flexural buckling check

The third part of the script is the reporting part, which structures the data in an appropriate form that can then be sent through Details output to the reporting function. Three components, one GHPython, one Grasshopper and one from Pterodactyl, are making the script shown in Figure 41. The detailed code of the GHPython component is shown in Figure 42.

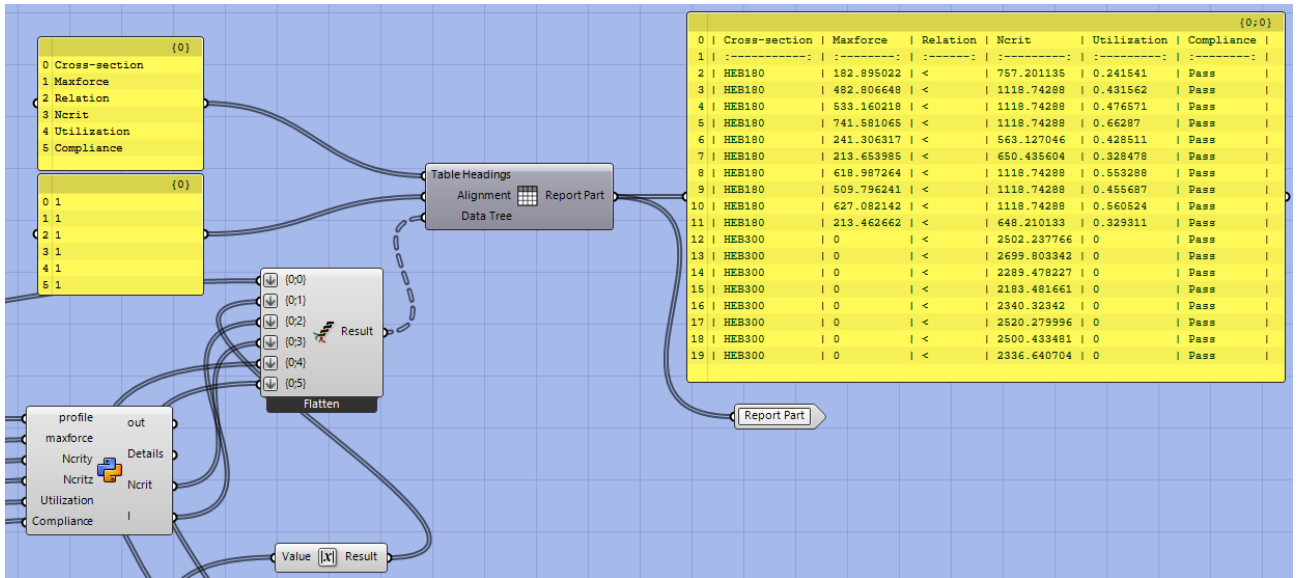


Figure 41 Reporting part of the flexural buckling check

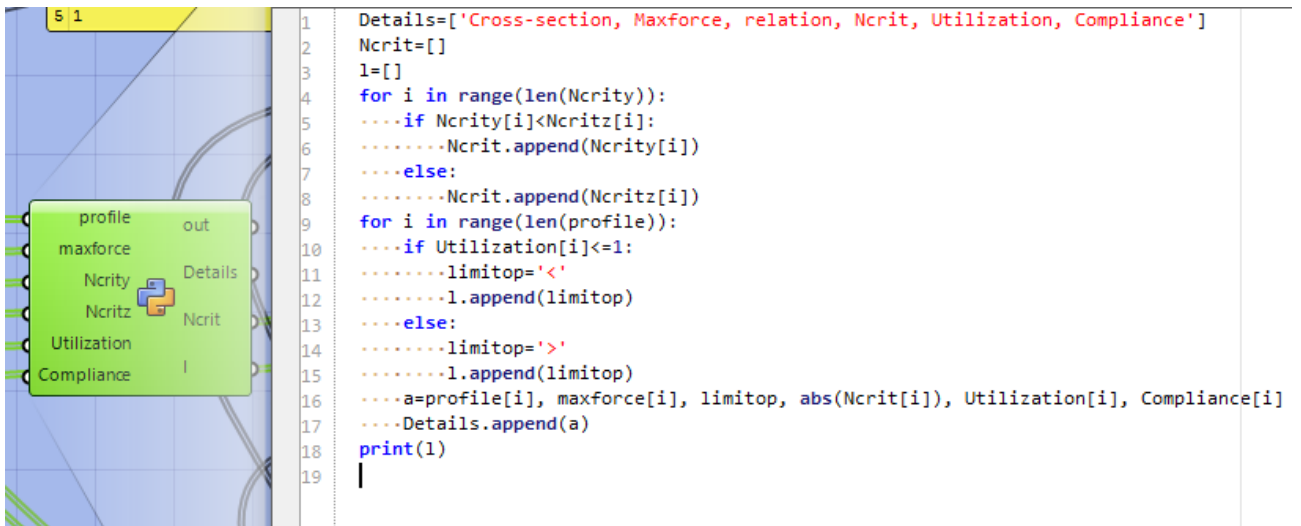


Figure 42 Component for structuring the data for reporting