Technische Universität München

Ingenieurfakultät Bau Geo Umwelt

Lehrstuhl für Computergestützte Modellierung und Simulation

Parametric model for automatic generation of fire-safety compliant building models

Bachelorthesis

for the Bachelor of Science Course Civil Engineering

Author:	Patrick Nordmann
Matriculation number:	
1. Supervisor:	M.Sc. Jimmy Abualdenien
2. Supervisor:	Prof. Dr.-Ing. André Borrmann

Begin Date:	December 1, 2020
Submission Date:	April 30, 2021

Abstract

In the process of designing a building, parameters are constantly changing. The reason for this can be adjustments in design, construction, budget, or other factors. Every change takes up a lot of time, because it must be checked for problems regarding fire safety and other issues. The motivation for this thesis is to make changes to the model easier and visualize their consequences immediately for the creator of a BIM-model. This is how the risk of additional expenses or damage to property and people due to mistakes regarding fire-safety in building design can be reduced. In this work a method is demonstrated that allows to create parametric building models that automatically obey certain Bavarian fire-safety regulations in Dynamo for Revit 2021. A program is created in the Dynamo workspace, where building parameters can be adjusted to instantly update the model in Revit. The thesis describes the process of the conceptualization by giving background information about BIM methods, the applicable building codes, and used software. Furthermore, the implementation of the program is explained by presenting its visual interface and source code. Moreover, decisions regarding the extent of the model and recommendations for the correct usage of the tool are explained.

Zusammenfassung

Im Verlauf der Planung eines Gebäudes ändern sich die Parameter ständig. Gründe für die Änderung können unter anderem Design, Konstruktion oder Budget sein. Jede Überarbeitung nimmt viel Zeit in Anspruch, da sie auf Probleme hinsichtlich des Brandschutzes und anderer Themen geprüft werden muss. Die Motivation für diese Arbeit ist, Änderungen am Modell zu erleichtern und deren Konsequenzen für den Ersteller eines BIM-Modells sofort sichtbar zu machen. So soll das Risiko von Mehraufwand oder Sach- und Personenschäden aufgrund von Fehlern im Brandschutz bei der Gebäudeplanung reduziert werden. Das Produkt der Bachelorarbeit ermöglicht es, parametrische Gebäudemodelle zur automatischen Einhaltung ausgewählter bayerischer Brandschutzvorschriften in Dynamo for Revit 2021 zu erstellen. Im Dynamo-Arbeitsbereich wird ein Programm erstellt, in dem Gebäudeparameter angepasst werden können, um das Modell in Revit sofort zu aktualisieren. Die Arbeit beschreibt den Prozess der Konzeptionierung, indem sie Hintergrundinformationen über BIM-Methoden, die geltenden Bauvorschriften und die verwendete Software gibt. Des Weiteren wird die Implementierung des Programms erläutert, indem die visuelle Schnittstelle und der Quellcode vorgestellt werden. Außerdem werden Entscheidungen über den Umfang des Modells und Empfehlungen für die richtige Verwendung des Tools erläutert.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

BIM	Building Information Modelling
API	Application Programming Interface
BayBO	Bayerische Bauordnung – Bavarian Building Code
MBO	Musterbauordnung – Model Building Code
AEC	Architecture, Engineering and Construction
MVD	Model View Definition
IFC	Industry Foundation Classes

1 Introduction

1.1 Motivation

Prior to the construction of any kind of building, the design process takes up a lot of time and manpower. The design of a building must fulfill individual and general specifications. Individual specifications such as budget, dimensions and quality come, for example, from clients or investors. Additionally, buildings must meet general requirements coming from several domains including fire safety, environmental impact, and accessibility by law. [3]

The design process has an iterative character, where experts of multiple disciplines exchange problems and information to develop solutions and increase the amount of detail throughout the design stages. A lot of information is missing in the early stages of design, because decisions have not been made yet or rely on not yet accessible information in advance. False assumptions that result from this uncertainty can cause critical delay along with higher costs and a loss in quality. [3, 5]

Nowadays Building Information Modelling (BIM) gains popularity as a set of tools for the involved parties to share information about a construction project and to help avoid making mistakes. A Building Information Model contains not only details about the geometry of a building and its components but further on material properties, costs, purpose and more. [25, 27]

Throughout the design process Building Information Models get more complex. Therefore, applying changes to a model is more expensive and time-consuming in late stages of design than in preceding ones. This makes it very valuable if problems, for instance, the model not complying with regulations can be identified in an early design stage. [27, 14]

In the event of a fire, for example, faults in fire protection lead to tremendous damage to property and often endanger human life. Improper constructional or technical protection arrangements can cause more damage including extinguishing water damage or smoke gas contamination. It is therefore mandatory to already pay attention to the fundamental requirements for fire prevention in the planning phase. Those requirements make sure that buildings have appropriate materials, escape routes, alarm systems and more. [27, 14]

1.2 Goal of the thesis

As addressed above, high uncertainty in early design stages of a project increases the probability of faulty assumptions and therefore mistakes. The resulting adjustments or the mistakes themselves can be obstructive, expensive, or even dangerous. The goal of this thesis is to create a BIM-tool to help prevent those early mistakes, while focusing on the aspect of fire safety. A parametric model is created, which automatically obeys local fire safety regulations. The relevant regulations in this case come from the Bavarian Building Code "Bayerische Bauordnung" (BayBO). The BayBO describes a set of laws that buildings must comply with. Building classes or "Gebäudeklassen" are introduced and matched to suiting regulations such as fire resistance classes for walls. The classification of a building is based on parameters including vertical and horizontal dimensions. By utilizing these parameters, the model should be customizable while automatically observing the limits for the specific building class. [14, 3]

The model is created in Dynamo for Revit 2021. Revit is a software by Autodesk containing multiple planning tools for architects, building technicians and structural engineers. Dynamo is a tool for visual programming and integrated in Revit. Even with minimal programming knowledge the user can access the functions in the Revit Application Programming Interface (API) via simple nodes. There is also an option for custom python nodes to further explore the Revit API. The interface feels intuitive and is therefore very well suited for the purpose of this work. [12, 9]

1.3 Structure of the thesis

This work is structured as follows: Section 1 introduces the reader to the topic, defines the motivation and goal of the thesis, and gives an overview over the following sections. Section 2 presents the utilized tools and gives background information. Section 3 describes the implementation of the model by explaining the programming procedure regarding code and interface and shows the end product by means of examples. Finally, Section 4 summarizes the thesis, explains boundaries, and provides an outlook on future work.

2 Background and related work

2.1 Building Information Modeling

In industry, digitalization is known for its potential to improve quality, productivity, and variety. The usage of these opportunities in the context of the Architecture, Engineering, Construction (AEC) industry seems to be lower than in other fields, which often leads to a loss of information when exchanging plans. BIM is intended to realize a meaningful and effective way to use computer technology in civil engineering. A Building Information Model is a complex visualization of a structure in digital form. In addition to the geometry of the building objects at a certain degree of detail, it includes semantic information like the materials, the technical properties or the costs and non-physical elements. These can be rooms, zones, a project organization, or timelines, as well as relationships between building elements. BIM includes creating such digital building models as well as their maintenance, use and exchange during the entire life span of the building. [15]

2.1.1 Parametric and generative design in BIM

Parametric and generative design or modeling are ways to make digital models more flexible and formalize discipline-specific know-how. In early design stages, parametric and generative models permit more iterations and allow for changes to be implemented more quickly compared to traditional models. [21]

Parametric modeling depicts objects through parameters and rules defining geometric and nongeometric properties or characteristics [20]. These rules can be linked to other entities, such that user input or altered context can automatically trigger updates [21, 20]. BIM models are often time consuming to build, human mistakes and inaccuracies can occur, and changes to finished models can be difficult to realize [23].

Parametric modeling is a way to solve these problems. It improves efficiency, lowers the risk of errors, and increases design flexibility through automation of tasks and the connection of components [6]. Additionally, an implemented parametric model can be re-used as a template for future projects or to compare different versions of a design in a short time [6, 24].

Generative design is a programming-centered methodology, which describes a process where the designer uses computer programs to autonomously produce possible solutions for a problem or a task. This is applicable for the AEC industry, because if often features a high variety in possible solutions for one problem, while there are a lot of factors that determine the optimal solution. [1]

Autodesk defines generative design as follows:

> “A goal-driven approach to design that uses automation to give designers and engineers better insight so they can make faster, more informed design decisions. Your specific design parameters are defined to generate many- even thousands -of potential solutions. You tell the software the results you want. With your guidance it arrives at the optimal design along with the data to prove which design performs best.” [8]

It enables the designer to define a set of parameters and create a mass of solutions, which would take a disproportionately long time to find manually [8].

2.2 Dynamo for Revit 2021

The BIM software Revit by Autodesk was first published in 2000 and is getting updated constantly. The concept of Revit is to allow two-dimensional and three-dimensional modelling of a component-oriented building model. It contains Revit Structure for structural design in civil engineering and building construction, Revit Architecture for building design and Revit MEP for building services engineering. [28]

Dynamo offers the opportunity to be used individually, but in this work the built-in plugin for Revit 2021 is chosen. Its advantage is that the software works with the API and the libraries of Revit [10]. Dynamos source code is open-source, enabling users and developers to create and share custom nodes and packages.

The package “Celery” which offers a custom dropdown menu node and the opportunity to automatically adapt the limits of an integer slider was employed for creating the model [7].

2.2.1 Visual Programming

Autodesk defines programming as follows:

> “Programming, frequently shortened from Computer Programming, is the act of formalizing the processing of a series of actions into an executable program.” [10]

The overall process of programming in Dynamo is categorized as Visual Programming, although in some places there is an option for text-based programming. In contrast to text-based programming, visual programming is characterized by multiple dimensions. These dimensions can appear either in the form of multi-dimensional objects, an additional time dimension, or spatial relationships as for instance in Dynamo. [10, 16]

The visual component of programming is supposed to address designers, make it more tangible and, consequently, more attractive to less experienced users. [10]

2.2.2 Features of Dynamo

The Dynamo interface can be divided into four main areas as highlighted in Figure 2.1.

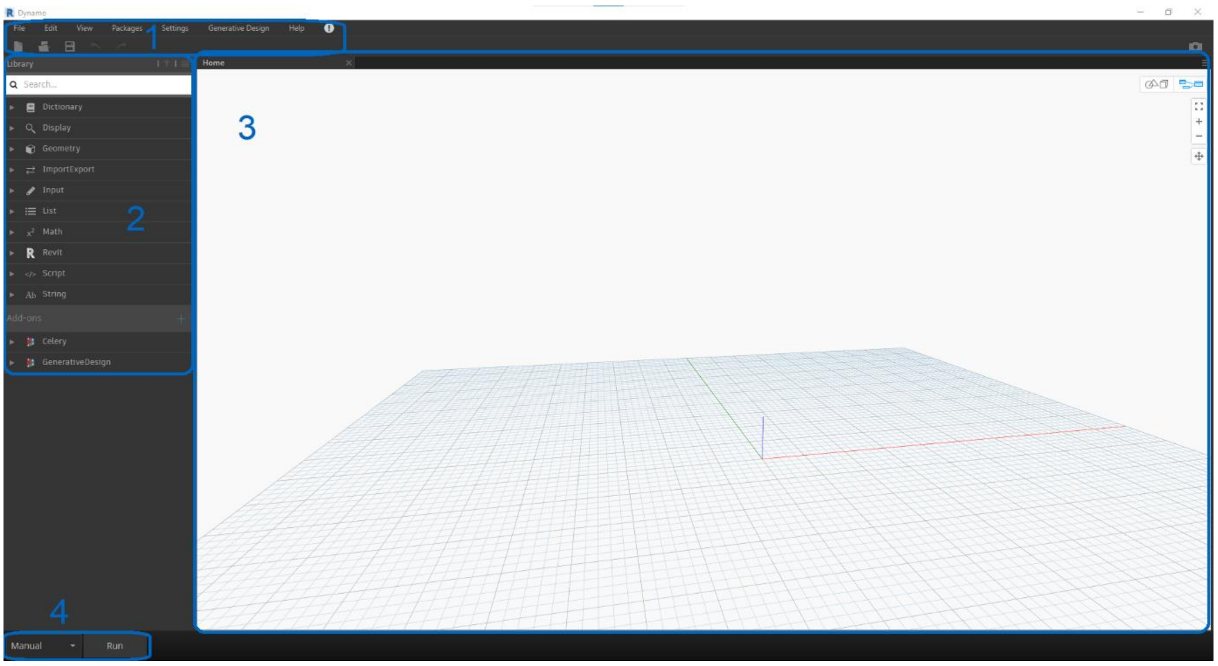


Figure 2.1 Dynamo interface areas

- Area 1 contains the basic functions for file-management, settings, help, and more [10].
- The library marked as area 2 is where the user can find the nodes of Dynamo and add-ons [10].

- Area 3, the workspace, takes up a major part of the window to provide a manageable overview of the developed program and the generated geometry [10].
- In the lower left corner, here area 4, the program can either be manually executed or the automatic execution mode can be activated [10].

Dynamo projects consist of nodes connected by wires in a two-dimensional layout. ports are the inputs and outputs of nodes. To create a logical flow of data, wires connect an output of one node with the input of another node. Instead of the input-port, some nodes offer a manual input through texts, dropdown menus or sliders. [10]

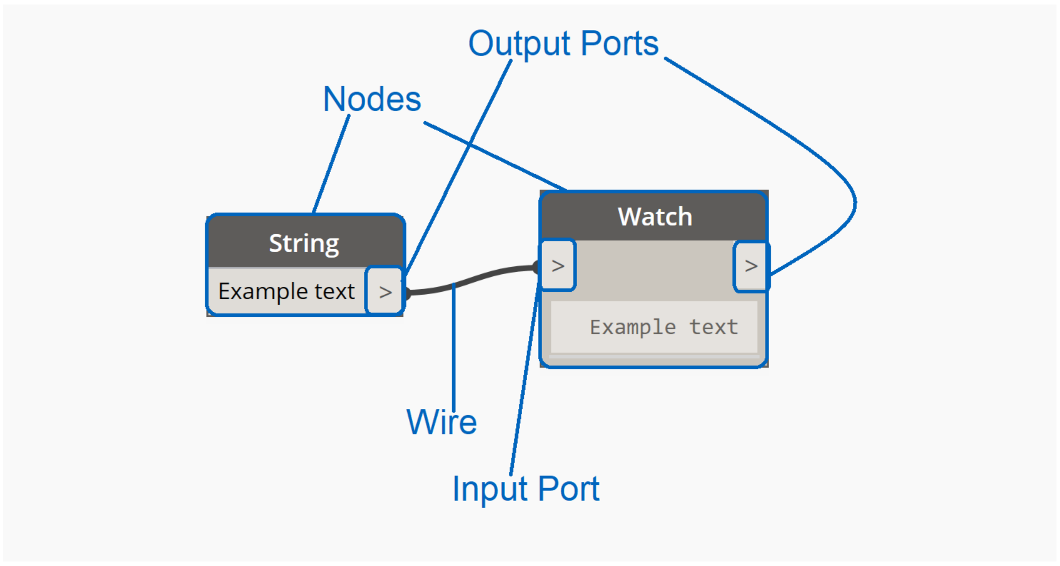


Figure 2.2 Example Dynamo program to display an input text

The example in Figure 2.2 shows a simple operation performed in Dynamo. The *String* node is given the manual input “Example text”, which it outputs to the *Watch* node. This then shows the text in a window below the ports. It also has an output port, which is not used in this case.

A variety of those nodes, which will later be explained in detail, were used for this work. Additionally, an add-on called “Celery” was implemented in order to be able to create parametric integer slider nodes and a node for a custom dropdown menu [7].

2.2.3 Python script nodes

Aside from the classic nodes mentioned above, Dynamo also offers python script nodes. These nodes have an arbitrary number of inputs and one output. Double clicking them opens a text window for python code. Python is a very common programming language, which is known for being comparatively easy to learn, supports modules and packages and can be integrated in existing systems. In Dynamo, this helps to maintain structure by combining the functionality of multiple nodes in one and extends the possibilities of programming in Dynamo, for example, through a greater selection of conditional statements and loops. [10]

2.3 BayBO Fire-safety regulations

In Germany, every state composes its own building code based on the “Musterbauordnung”, a federal guideline specifying minimum requirements for the individual state-laws [27]. Relevant for this work is the BayBO, which regulates the minimum standards for civil works in Bavaria. Particularly interesting are the included fire-safety regulations. There are five building classes in the BayBO dependent on scale and function of structures that govern certain fire safety measures [14]. The created Dynamo program is supposed to adjust the parameter limits according to the chosen building class.

For this thesis, parts of the work “Analysis of Exchange Requirements for BIM-based Fire Code Compliance Checking” by Pfuhl were consulted. The goal of the thesis was to create a Model View Definition (MVD) to check the compliance with fire protection regulations from the BayBO [27]. An MVD is a subset of the Industry Foundation Classes (IFC) scheme that specifies the requirements and specifications of the data shared between involved software tools [4]. In the process, the relevant contents of the BayBO were translated into english by Pfuhl and collected in graphics, like the following Figure 2.3. It shows the building classes introduced by the BayBO, which are also described below.

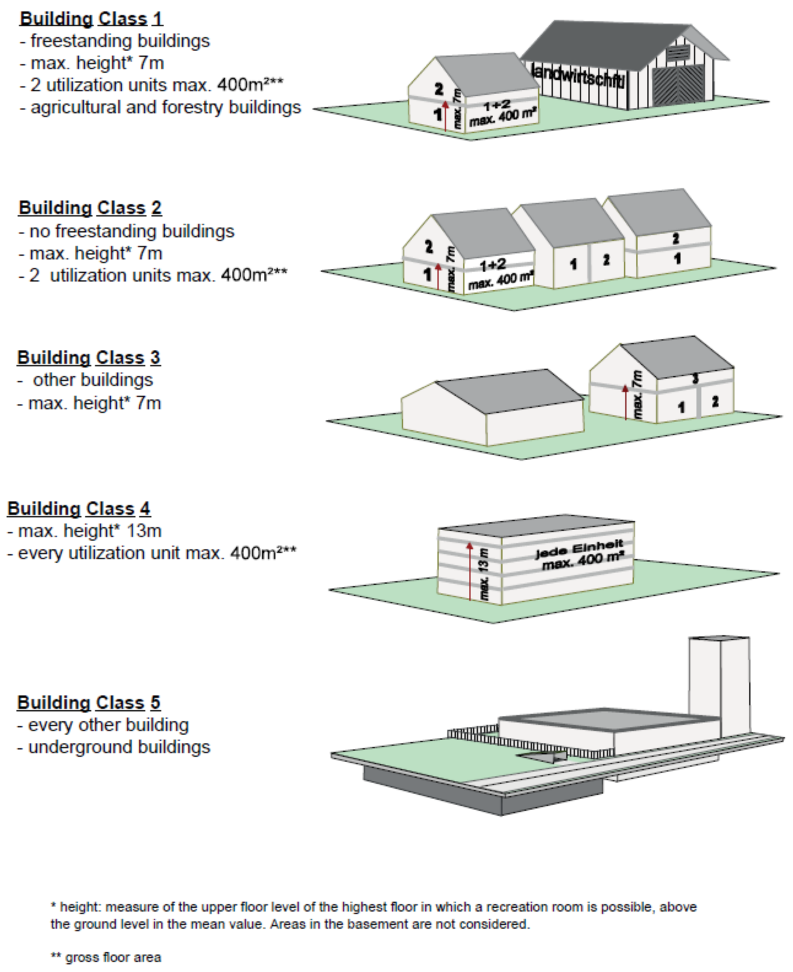
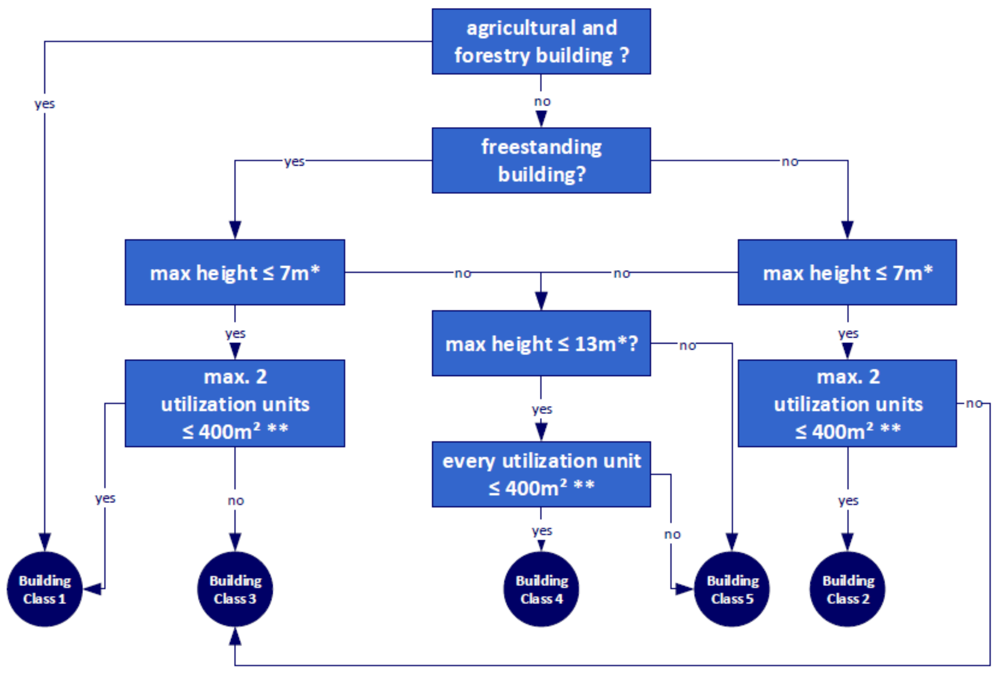

Building Class 1
- freestanding buildings
- max. height* 7m
- 2 utilization units max. 400m²**
- agricultural and forestry buildings
landwirtschaft
max. 7m
1+2 max. 400 m²
Building Class 2
- no freestanding buildings
- max. height* 7m
- 2 utilization units max. 400m²**
Building Class 3
- other buildings
- max. height* 7m
Building Class 4
- max. height* 13m
- every utilization unit max. 400m²**
max. 13 m
Jede Einheit max. 400 m²
Building Class 5
- every other building
- underground buildings
* height: measure of the upper floor level of the highest floor in which a recreation room is possible, above the ground level in the mean value. Areas in the basement are not considered.
** gross floor area

Figure 2.3 Building classes overview by Pfuhl [27]

- **Building class 1** includes freestanding buildings, which either are of agricultural or forestry usage, or meet the following requirements. They can have a height of 7 m or lower, a maximum of 2 utilization units and no more than 400 m² of gross floor area.
- **Building class 2** also permits a height of 7 m, gross floor area of 400 m² and 2 utilization units at most. In contrast to building class 1, these buildings are not freestanding.
- **Building class 3** involves all other buildings with a maximum height of 7 m. No further limitation for utilization units and gross floor area are given.
- **Building class 4** allows a maximum height of 13 m. The building can have an unlimited number of utilization units and there is no restriction regarding the gross floor area. However, each unit may have a maximum dimension of 400 m².
- **Building class 5** contains every other building, including underground structures.

In the process of developing the targeted MVD, Figure 2.4 was created by Pfuhl, which has proven useful for developing formulas for the project of this thesis. It depicts an algorithm for building class identification. It was used to determine the created models' building class, which is further addressed in Section 3.4. Furthermore, it was reverse engineered to connect the created input parameters and automatically adapt them to meet the chosen building class.



* * height: Measure of the upper floor level of the highest floor in which an abode room is possible, above the ground level in the mean value. Areas in the basement are not considered.

** gross floor area

Figure 2.4 Determination of building classes [27]

Besides the parameters shown in Figure 2.4, each building class requests minimum fire resistance classes for walls and floors [14]. Pfuhl translated the three relevant fire resistance classes F30, F60, and F90 to "fire retardant", "highly fire retardant", and "fire resistant".

For the implementation of the model, a certain degree of detail had to be chosen. On one hand, the program was expected to be uncluttered, which does not allow an excessive number of parameters. Furthermore, the program finds its application in early design stages, where details are secondary.

To help with the simplification and the choice of regulations that have to be applied, Abualdenien’s definition of Building Development Level (BDL) was utilized. BDL 2 defines a mass for the model by extending the outline by the building height [2]. BDL 3 includes information about the geometry of individual stories, their usage, and load-bearing components [2]. BDL 4 presents a room layout and wall openings, which are not considered for this work [2]. BDL 3 therefore best fits the degree of detail for this thesis.

The model created for this thesis filters materials of the Revit library, to ensure that only walls and floors with the proper fire resistance class can be chosen. The requirements resulting directly from the building class include minimum fire resistances for load-bearing walls, ceilings, and the basement ceiling [14]. Moreover, in the BayBO the wall to a neighboring building is handled differently than other walls. This is not depicted in the model, because no neighboring buildings are created.

To implement the mentioned collection of regulations, all walls in the model created in this thesis are assumed to be load bearing. Table 2.1 shows an overview over the requirements for fire resistance classes that were implemented. More information regarding the implementation will be given in Section 3.2.

Table 2.1 Overview of selected fire resistance class requirements

building class:	1	2	3	4	5
walls		fire retardant		highly fire retardant	fire resistant
floors					
basement ceiling	fire retardant		fire resistant		

2.4 Clarification of most used terms

Table 2.2 gives an overview of the most used terms in this thesis and their explanation.

Table 2.2 Clarification of most used terms

Term	Explanation
Building class	Buildings are categorized in building classes by the individual German state laws to assign them suitable requirements. The classification is dependent on height, area, usage, and location of the building. [14]
Gross floor area	Sum of the areas of all levels of the floor plan of the structure. [18]
Height (building class context)	The average height of the top edge of the highest floor above ground, in which an occupied room is possible. Areas in the basement are not considered. [14]
Utilization unit	An assembling of individual rooms with similar or other related rooms, for example, an apartment or a collection of classrooms. [27]
Fire resistance class	The fire resistance class refers to the German “Feuerwiderstandsklasse” and indicates how many minutes a building component – for example, a wall or ceiling – can withstand a fire. During this period, the load-bearing capacity must be ensured, and the thermal insulation must function so well that the surface temperature on the component side facing away from the fire does not rise. In addition, the so-called space closure must remain guaranteed. This is the case if the component prevents the penetration of flames and hot gases. [13]

Fire retardant	Fire resistance duration of 30 minutes means that the component will remain functional for a minimum of 30 minutes in a fire. This corresponds to fire resistance class F30 according to DIN 4102. [19]
Highly fire retardant	Fire resistance duration of 60 minutes means that the component will remain functional for a minimum of 60 minutes in a fire. This corresponds to fire resistance class F60 according to DIN 4102. [19]
Fire resistant	Fire resistance duration of 90 minutes means that the component will remain functional for a minimum of 90 minutes in a fire. This corresponds to fire resistance class F90 according to DIN 4102. [19]
API	An "Application Programming Interface" is a collection of commands, functions, protocols, and objects that developers can use to build software or work with an external system. [17]
Family	A component used in a building model, that has a set of attributes and a corresponding visualization. [26]

3 Implementation

In the following, the implementation procedure of the model will be discussed. To make the interface easier to understand, the workspace of the program is divided into four clusters as illustrated in Figure 3.1. A DIN A3 sized version of this overview can be found in Appendix A.3.

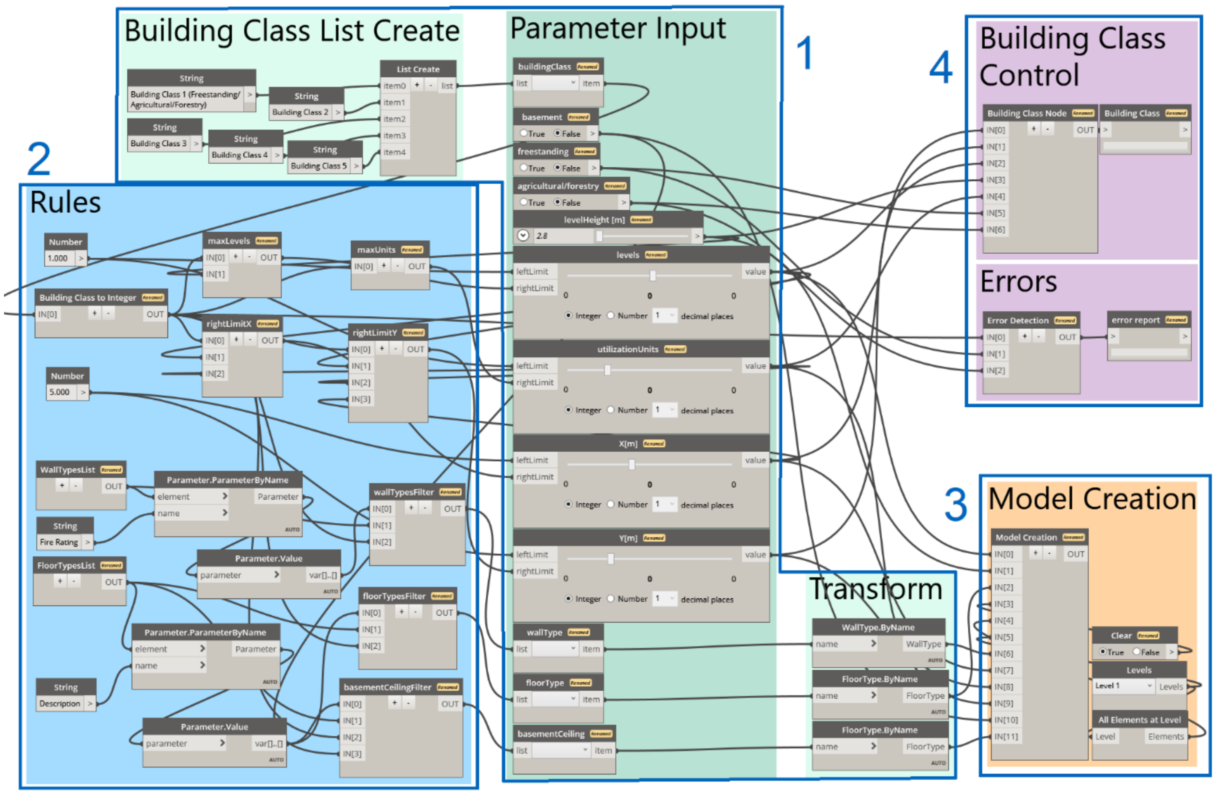


Figure 3.1 Workspace clusters: Parameters, Rules, Model Creation, Control

These are thematized in the following sections. The green “Parameters” cluster (1) consists of the node groups *Building Class List Create*, *Parameter Input* and *Transform*. The blue “Rules” cluster (2) involves nodes processing data from the *Parameter Input* group to send it back there or to the purple “Control” cluster (4). In the lower right corner, the orange “Model Creation” cluster (3) receives data from the “Parameters” cluster and generates the building model in Revit. The “Control” cluster receives data from the “Parameters” and “Rules” clusters and returns the actual building class and an error where appropriate.

3.1 Parameters cluster

As mentioned above, the “Parameters” cluster includes the node groups *Parameter Input*, *Building Class List Create*, and *Transform*. The *Parameter Input* group depicted in Figure 3.2 is the main interface for the user of the program and consists of all relevant nodes for the parameterization of the model.

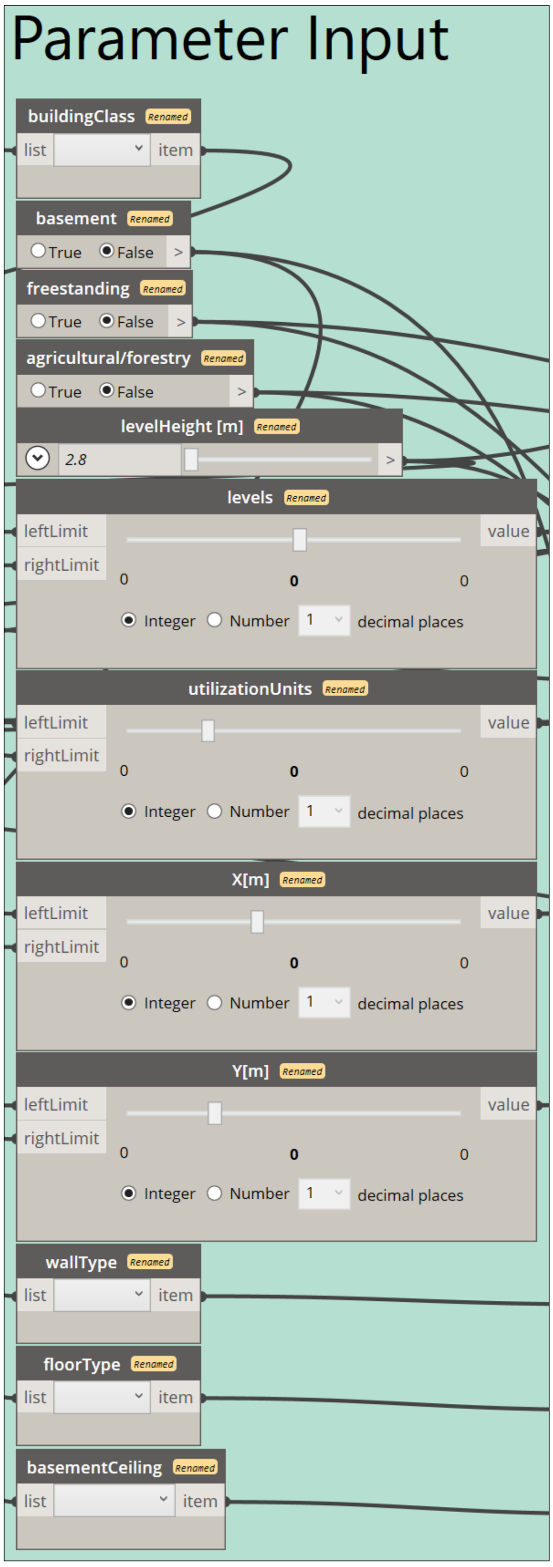


Figure 3.2 Overview of *Parameter Input* group

The nodes are arranged in such a way, that the user can work through them from top to bottom while executing the program regularly. Appendix A.1 contains a manual describing the optimal execution order for the parameters.

At first, the program is executed to create the list of building classes in the *Building Class List Create* group.

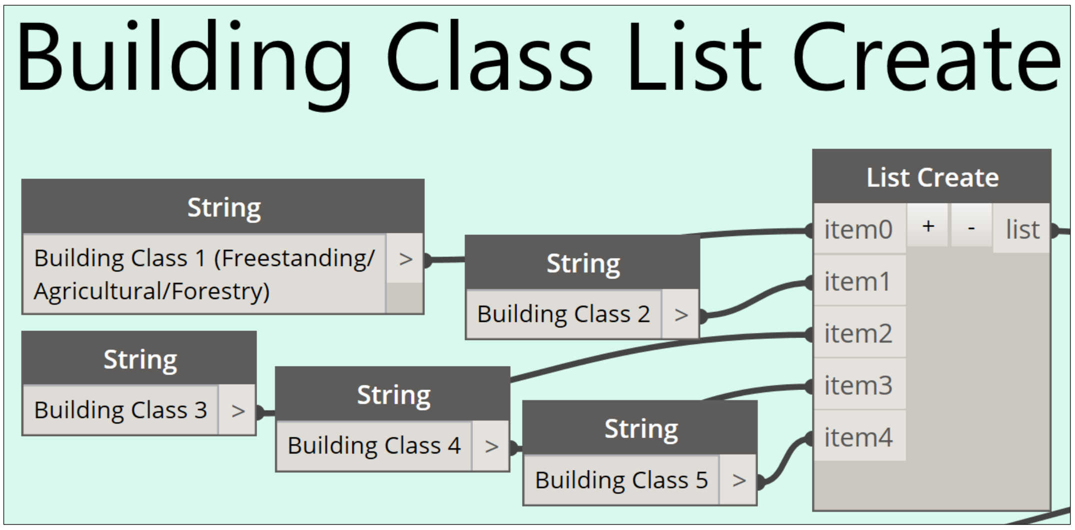


Figure 3.3 Overview of *Building Class List Create* group

Five *String* nodes get combined by a *List Create* node, which outputs a list to the first node in the *Parameter Input* group. This is a *ListItemSelector* from the “Celery” package, that lets the user select the building class, which the model should always comply with.

Beneath are three *Boolean* nodes where the user specifies wheter the modelled building will have a basement and whether it is freestanding or categorized as an agricultural or forestry building.

The following node is a *Number Slider* to set the level height in a range from 2.8 m to 4 m. The minimum of 2.8 m was chosen, because the BayBO defines a minimum of 2.4 m for room height and the floors in the models of this thesis are for the most part 0.4 m high [14]. The upper limit of 4 m was chosen because higher values are very uncommon.

The next four nodes are *InputBoundedNumberSlider* nodes from the “Celery” package. These get a minimum and maximum, called left and right limit from the *Rules* group, and output an integer. From top to bottom the user can adjust the number of levels, number of utilization units, and extent in X as well as Y direction.

Lastly, there are three *ListItemSelector* nodes to choose a wall type and two floor types. The *basementCeiling* node only displays a list if the *Boolean* node *basement* is set to “True”. The types get output to the *Transform* group in Figure 3.4, which contains one *WallType.ByName* and two *FloorType.ByName* nodes. These convert strings to the respective family type and then forward those to the *Model Creation* node.

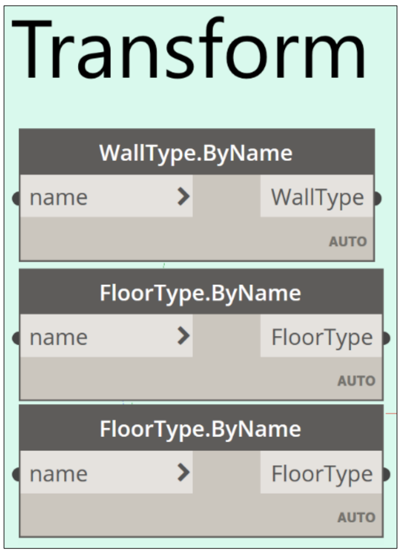


Figure 3.4 Overview of *Transform* group

When a parameter gets altered, the rest of the nodes immediately adapt to constantly obey the implemented regulations. This means that the limits of a node can automatically be altered and must be revisited to not produce unwanted or nonsensical values. The following Table 3.1 shows the dependencies between the parameters by listing the ones that the user must revisit after changing another.

Table 3.1 Parameter dependencies

alternated parameter	**dependent parameters to revisit**
buildingClass	all parameters
basement	*basementCeilingType*
freestanding	—
agricultural/forestry	—
levelHeight	*levels, utilizationUnits, lengthX, lengthY*
levels	*utilizationUnits, lengthX, lengthY*
utilizationUnits	*lengthX, lengthY*

X	*lengthY*
Y	—
wallType	—
floorType	—
basementCeilingType	—

3.2 Rules cluster

The first of two functions of the “Rules” cluster is to process data coming from the *Parameter Input* group, applying the regulations from the BayBO dependent on the given building class and giving it either back to the *Parameter Input* group or directly to the “Model Creation” or “Control” cluster. Figure 3.5 shows an overview of the relevant upper part of the “Rules” cluster.

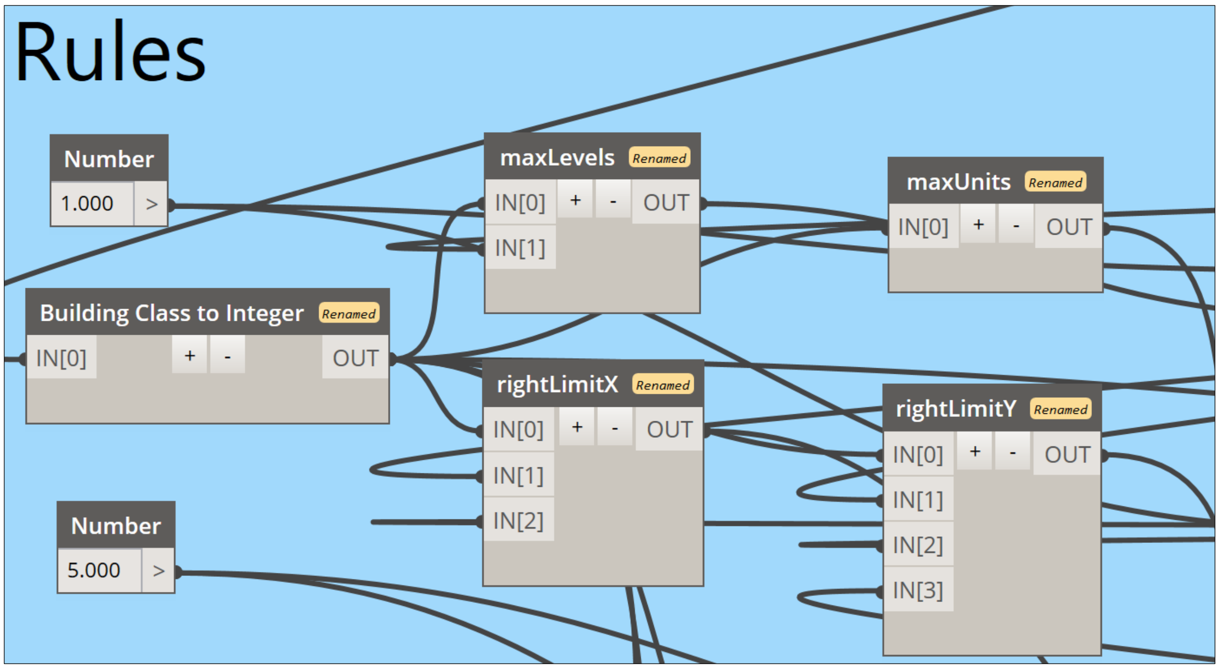


Figure 3.5 Overview of “Rules” cluster: parameter limitation section

After converting the *buildingClass* string to an integer in a python script depicted in Figure 3.6, it gets distributed to all nodes working with the parameter. The python script imports the relevant standard libraries, assigns the input building class to a variable, and creates an integer with the matching number dependent on the building class case.

```python
#Load the Python Standard and DesignScript Libraries
import sys
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *

#Assign input to variables
buildingClassString = IN[0]

#Convert building class string to integer
if buildingClassString == "Building Class 1 (Freestanding/Agricultural/Forestry)":
    buildingClass = 1
if buildingClassString == "Building Class 2":
    buildingClass = 2
if buildingClassString == "Building Class 3":
    buildingClass = 3
if buildingClassString == "Building Class 4":
    buildingClass = 4
if buildingClassString == "Building Class 5":
    buildingClass = 5

OUT = int(buildingClass)
```

Figure 3.6 Implementation of *Building Class to Integer* node

Two number nodes are utilized to set the left limits of the *levels* and *utilizationUnits* sliders to 1 as well as the left limits of the *X* and *Y* nodes to 5. This was done in such a way that every building has at least one level and unit and a minimum measure of five meters in each direction.

The rules to determine the building classes from the BayBO are integrated through the four python script nodes *maxLevels*, *maxUnits*, *rightLimitX* and *rightLimitY*. Each of these defines a maximum value for the corresponding parameter as shown in Table 3.2 below.

Table 3.2 Implementation of BayBO building class rules

Node/Output	Formula by Building Class				
	1	2	3	4	5
maxLevels	= floor(7/levelHeight)+1			= floor(13/levelHeight)+1	= 10

Node/Output	Formula by Building Class				
	1	2	3	4	5
maxUnits	= 2	= 2	= 100		

Node/Output	Formula by Building Class				
	1	2	3	4	5
rightLimitX	= floor(80/levels)		= 200	= floor(80/levels*units)	= 200

Node/Output	Formula by Building Class				
	1	2	3	4	5
rightLimitY	= floor(400/levels/x)		= 200	= floor(400/x/levels*units)	= 200

For each variable, the table presents the formula sorted by building classes. These were established with the help of Figure 2.4. For some parameters, the BayBO does not give an upper limit. As infinity would neither be realistic nor possible in Dynamo, custom limits were defined for this thesis.

For building class 5 a maximum of 10 levels was chosen. In general, the amount of utilization units was limited to 100, and the greatest selectable dimension in each direction is 200 m, if not limited otherwise. The four python script nodes to implement the addressed formulas are examined in detail in Sections 3.2.1 to 3.2.4.

The second purpose of the “Rules” cluster is to filter the wall types and floor types of the Revit family library to meet the requirements regarding fire resistance classes of the selected building class. The lower part of the “Rules” cluster visible in Figure 3.7 is responsible for this process.

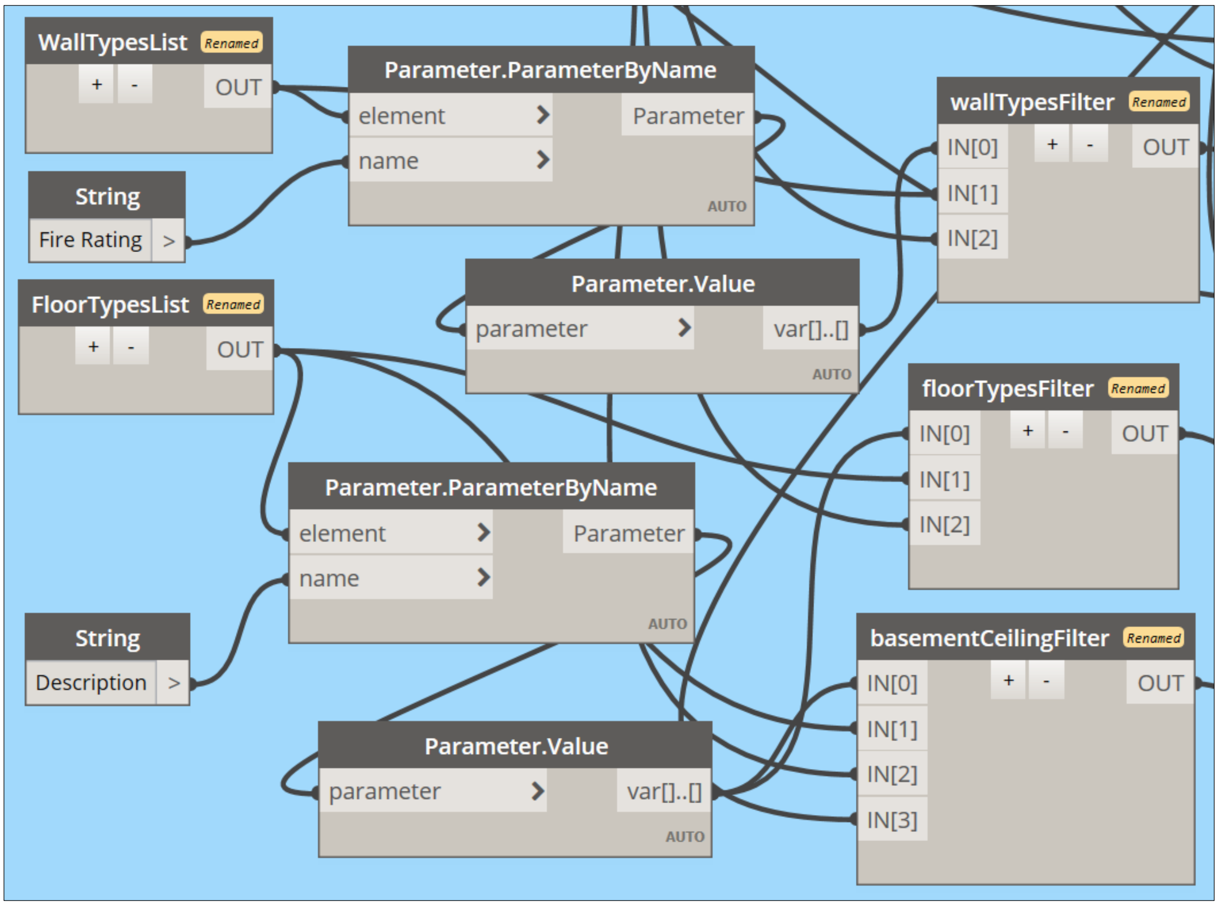
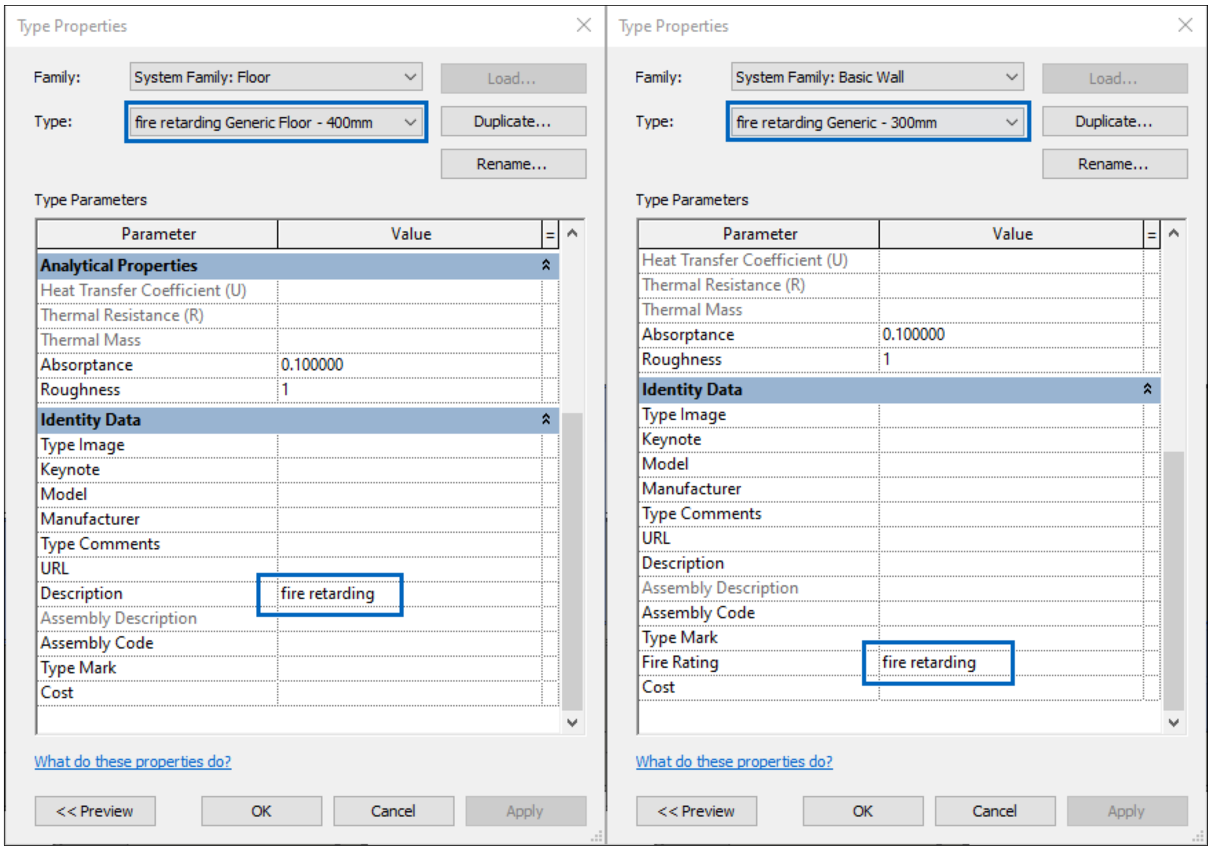


Figure 3.7 Overview of “Rules” cluster: fire resistance class filter section

The attribute “Fire Rating” is assumed to be an equivalent to the fire resistance class. It only exists for wall families and even there no values for it were given in the standard Revit library.

For this project, the family “Generic – 300mm” was copied three times. Each copy was given a fire resistance class, namely “fire retardant”, “highly fire retardant” and “fire resistant”. The attribute was assigned both as a value for the “Fire Rating” attribute and included in the name of the family.



Family: System Family: Floor
Type: fire retarding Generic Floor - 400mm

Type Parameters

Parameter	Value
Analytical Properties	
Heat Transfer Coefficient (U)	
Thermal Resistance (R)	
Thermal Mass	
Absorptance	0.100000
Roughness	1
Identity Data	
Type Image	
Keynote	
Model	
Manufacturer	
Type Comments	
URL	
Description	fire retarding
Assembly Description	
Assembly Code	
Type Mark	
Cost	

Family: System Family: Basic Wall
Type: fire retarding Generic - 300mm

Type Parameters

Parameter	Value
Heat Transfer Coefficient (U)	
Thermal Resistance (R)	
Thermal Mass	
Absorptance	0.100000
Roughness	1
Identity Data	
Type Image	
Keynote	
Model	
Manufacturer	
Type Comments	
URL	
Description	
Assembly Description	
Assembly Code	
Type Mark	
Fire Rating	fire retarding
Cost	

Figure 3.8 Properties of new wall and floor types

Analogously, the family “Generic Floor – 400mm” was copied for each fire resistance class. Since floor types do not have a suitable attribute, the fire resistance class was assigned to an attribute called “Description”.

The nodes responsible for the filtered lists of floor types are examined in detail in Sections 3.2.5 and 3.2.6.

3.2.1 *maxLevels* node

The *maxLevels* node provides a maximum value for the number of levels that can be selected with the *levels* slider. After importing the necessary standard and math libraries and assigning the variables for inputs *buildingClass* and *levelHeight*, a distinction is made between the five building classes. For building class 1, 2 and 3 the quotient of 7 m and the input *levelHeight* is rounded off and increased by 1 level, because the maximum height of 7 m refers to the top floor level.

The allowed levels for class 4 are calculated with a similar formula but with a maximum height of the top floor level of 13 m.

As described in Section 3.2, in the case of building class 5 the utmost levels are 10.

```python
#Load the Python Standard and DesignScript Libraries and Math
import sys
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *
import math

#Assign input to variables
buildingClass = IN[0]
levelHeight = IN[1]

#Determine and output maxLevels
if buildingClass == 1 or buildingClass == 2 or buildingClass == 3:
    maxLevels = math.floor(7 / levelHeight) + 1
if buildingClass == 4:
    maxLevels = math.floor(13 / levelHeight) + 1
if buildingClass == 5:
    maxLevels = 10

OUT = maxLevels
```

Figure 3.9 Implementation of *maxLevels* node

3.2.2 *maxUnits* node

To determine the maximum utilization units, the only input needed is the building class. This is assigned to the suitable variable after the standard libraries are imported. The output *maxUtilizationUnit* is set to a value of 2 for building classes 1 and 2 and to a

```python
#Load the Python Standard and DesignScript Libraries
import sys
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *

#Assign input to variables
buildingClass = IN[0]

#Determine and output maxUtilizationUnits
if buildingClass == 1 or buildingClass == 2:
    maxUtilizationUnits = 2
if buildingClass == 3 or buildingClass == 4 or buildingClass == 5:
    maxUtilizationUnits = 100

OUT = maxUtilizationUnits
```

Figure 3.10 Implementation of *maxUnits* node

value of 100 for building classes 3, 4 and 5 as discussed in Section 3.2. The result is transferred to the *utilizationUnits* parameter node.

3.2.3 *rightLimitX* node

For the *rightLimitX* node in Figure 3.11, again the first steps are to import libraries and to assign the input to variables. In this case, standard and math libraries are necessary, and the inputs are *buildingClass*, *levels*, and *utilizationUnits.* For building classes 1 and 2 the greatest allowed extent in the *X* direction is determined by dividing 80 m by the number of levels in the building. The value for the maximum gross floor area is 400 m². Considering the minimum extent of 5 m in *Y* direction, the *X* extent per level has a maximum of 80 m. The result is rounded off to be usable as an integer input for the *X* node.

In the case of building class 4 the maximum value for *X* applies for each utilization unit. Therefore, the formula differs only in that it is also multiplied by *utilizationUnits*.

As addressed in Section 3.2, the general maximum value for *X* is 200 m, which in this case applies for building classes 3 and 5.

```python
#Load the Python Standard and DesignScript Libraries and Math
import sys
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *
import math

#Assign input to variables
buildingClass = IN[0]
levels = IN[1]
utilizationUnits = IN[2]

#Determine and output rightLimitX
if buildingClass == 1 or buildingClass == 2:
    rightLimitX = math.floor(80 / levels)
if buildingClass == 3 or buildingClass == 5:
    rightLimitX = 200
if buildingClass == 4:
    rightLimitX = math.floor(80 / levels * utilizationUnits)

OUT = rightLimitX
```

Figure 3.11 Implementation of *rightLimitX* node

3.2.4 *rightLimitY* node

As for the *X* limitation, the standard as well as math libraries are imported and *buildingClass*, *levels,* and *utilizationUnits* are used. Additionally, the defined value for

X is assigned to a variable. The maximum extent in Y direction per level is determined by dividing 400 m² by *x*. Dividing this value by *levels* gives the maximum *Y* value for building classes 1 and 2. For building class 4 it needs to be multiplied by the amount of utilization units and for building classes 3 and 5 the fixed limit of 200 m applies. In order to output a usable integer for the parameter node *Y*, the variable must be rounded off. The exact implementation of *rightLimitY* is shown in Figure 3.12.

```python
#Load the Python Standard and DesignScript Libraries and Math
import sys
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *
import math

#Assign input to variables
buildingClass = IN[0]
levels = IN[1]
utilizationUnits = IN[2]
x = IN[3]

#Determine and output rightLimitY
if buildingClass == 1 or buildingClass == 2:
    rightLimitY = math.floor(400 / levels / x)
if buildingClass == 3 or buildingClass == 5:
    rightLimitY = 200
if buildingClass == 4:
    rightLimitY = math.floor(400 / x / levels * utilizationUnits)

OUT = rightLimitY
```

Figure 3.12 Implementation of *rightLimitY* node

3.2.5 Wall type filter algorithm

In the developed Dynamo program, the wall types in the Revit library are filtered to match a certain building class and the user can select the desired wall type with the aid of a parameter node. The first step to create the filtered list is to collect all available wall types from the Revit library. For this purpose the python script node *WallTypesList* depicted in Figure 3.13 is implemented.

```python
#import relevant libraries
import clr
clr.AddReference('RevitNodes')
import Revit
clr.ImportExtensions(Revit.Elements)
clr.AddReference('RevitServices')
import RevitServices
from RevitServices.Persistence import DocumentManager
clr.AddReference('RevitAPI')
import Autodesk
from Autodesk.Revit.DB import *
import System
from System.Collections.Generic import *

#Collect and output all wall types
doc = DocumentManager.Instance.CurrentDBDocument

collectedWallTypes = FilteredElementCollector(doc).OfCategory(
BuiltInCategory.OST_Walls.WhereElementIsElementType().ToElements()

OUT = collectedWallTypes
```

Figure 3.13 Implementation of *WallTypesList* node

With the help of the imported libraries, all wall types in Revit are collected to produce the list *collectedWallTypes*. The next node used is a *Parameter.ParameterByName*. This requires the list of wall types and the name of a parameter, which it gets from a string node with the content “Fire Rating”. The output of this node is a list of parameters with the given name for each wall type family. By forwarding this list to the *Parameter.Value* node beneath, each item in the list gets changed to the value of the parameter “Fire Rating” of the corresponding wall type.

As a last step, a python script node with the algorithm in Figure 3.14 matches the lists of fire resistance classes and wall types to filter out the appropriate items depending on the building class.

```python
#Load the Python Standard and DesignScript Libraries
import sys
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *

#Assign input to variables
fireRatings = IN[0]
collectedWallTypes = IN[1]
buildingClass = IN[2]

#Filter wall types dependent on building class
wallTypeNames = []

if buildingClass == 1:
    for i in range(0, len(fireRatings)):
        wallTypeNames.append(collectedWallTypes[i].Name)

if buildingClass == 2 or buildingClass == 3:
    for i in range(0, len(fireRatings)):
        if (fireRatings[i] == "fire retarding"
            or fireRatings[i] == "highly fire retarding"
            or fireRatings[i] == "fire resisting"):
            wallTypeNames.append(collectedWallTypes[i].Name)

if buildingClass == 4:
    for i in range(0, len(fireRatings)):
        if (fireRatings[i] == "highly fire retarding"
            or fireRatings[i] == "fire resisting"):
            wallTypeNames.append(collectedWallTypes[i].Name)

if buildingClass == 5:
    for i in range(0, len(fireRatings)):
        if fireRatings[i] == "fire resisting":
            wallTypeNames.append(collectedWallTypes[i].Name)

OUT = wallTypeNames
```

Figure 3.14 Implementation of *wallTypesFilter* node

Firstly, the algorithm distinguishes between the building classes. Secondly, every index of *fireRatings* is examined. If the item matches one of the listed fire resistance classes, the element of *collectedWallTypes* with the same index is appended to *wallTypeNames*.

For building class 1, all available wall types can be used, whereas for building classes 2 and 3 only those with actual fire resistance classes are appended. Building class 4 only allows the fire resistance classes “highly fire retardant” and “fire resistant” and for building class 5 nothing but “fire resistant” wall types are added to the *wallTypesNames* list. This list then gets output to the parameter node *wallType*.

3.2.6 Floor type filter algorithms

Analogous to the wall types above, the floor types need to be filtered for the parameter nodes. Since the floor families do not have an attribute called “Fire Rating”, “Description” was used inside the string node to create the list of fire resistance classes.

Similar to the wall types above, the *FloorTypesList* node collects the names of the floor families and forwards them to the *Parameter.ParameterByName* and *Parameter.Value* nodes. The resulting lists of floor type names and corresponding fire resistance classes are used further by the filtering python script nodes for floors and basement ceiling.

```python
#import relevant libraries
import clr
clr.AddReference('RevitNodes')
import Revit
clr.ImportExtensions(Revit.Elements)
clr.AddReference('RevitServices')
import RevitServices
from RevitServices.Persistence import DocumentManager
clr.AddReference('RevitAPI')
import Autodesk
from Autodesk.Revit.DB import *
import System
from System.Collections.Generic import *

#Collect and output all wall types
doc = DocumentManager.Instance.CurrentDBDocument

collectedFloorTypes = FilteredElementCollector(doc).OfCategory(
BuiltInCategory.OST_Floors).WhereElementIsElementType().ToElements()

OUT = collectedFloorTypes
```

Figure 3.15 Implementation of *FloorTypesList* node

Figure 3.16 describes the Implementation of the *floorTypesFilter* node. After loading the standard libraries, the lists *descriptions*, *collectedFloorTypes* and the building class are imported. The required fire resistance classes for the building classes are the same as for wall types. The output *floorTypes* list is utilized by the parameter node *floorType*.

```python
#Load the Python Standard and DesignScript Libraries
import sys
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *

#Assign input to variables
descriptions = IN[0]
collectedFloorTypes = IN[1]
buildingClass = IN[2]

#Filter floor types dependent on building class
floorTypes = []

if buildingClass == 1:
    for i in range(0, len(descriptions)):
        floorTypes.append(collectedFloorTypes[i].Name)

if buildingClass == 2 or buildingClass == 3:
    for i in range(0, len(descriptions)):
        if (descriptions[i] == "fire retarding"
            or descriptions[i] == "highly fire retarding"
            or descriptions[i] == "fire resisting"):
            floorTypes.append(collectedFloorTypes[i].Name)

if buildingClass == 4:
    for i in range(0, len(descriptions)):
        if (descriptions[i] == "highly fire retarding"
            or descriptions[i] == "fire resisting"):
            floorTypes.append(collectedFloorTypes[i].Name)

if buildingClass == 5:
    for i in range(0, len(descriptions)):
        if descriptions[i] == "fire resisting":
            floorTypes.append(collectedFloorTypes[i].Name)

OUT = floorTypes
```

Figure 3.16 Implementation of *floorTypesFilter* node

The implementation for the basement ceiling depicted in Figure 3.17 works analogously, except that the additional input *basement* is needed. If *basement* is “True”, *descriptions* is searched again and the corresponding items of *collectedFloorTypes* are appended to the output *basementCeilings*.

Building class 1 and 2 allow every floor type available in the Revit library with a fire resistance class better than “fire retardant”. The basement ceiling for building class 3, 4 or 5 must be “fire resistant”. If no basement is created, the output list has no entry. Consequently, no wall type can be chosen in the corresponding parameter node, which causes the following *FloorType.ByName* node to return an error. However, this is not relevant for the generation of the model and can be ignored.

```python
#Load the Python Standard and DesignScript Libraries
import sys
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *

#Assign input to variables
descriptions = IN[0]
collectedFloorTypes = IN[1]
buildingClass = IN[2]
basement = IN[3]

#Filter floor types dependent on building class
basementCeilings = []

if basement:
    if buildingClass == 1 or buildingClass == 2:
        for i in range(0, len(descriptions)):
            if (descriptions[i] == "fire retarding"
            or descriptions[i] == "highly fire retarding"
            or descriptions[i] == "fire resisting"):
                basementCeilings.append(collectedFloorTypes[i].Name)

    if buildingClass == 3 or buildingClass == 4 or buildingClass == 5
        for i in range(0, len(descriptions)):
            if descriptions[i] == "fire resisting":
                basementCeilings.append(collectedFloorTypes[i].Name)

OUT = basementCeilings
```

Figure 3.17 Implementation of *basementCeilingFilter* node

3.3 Model Creation cluster

The “Model Creation” cluster presented in Figure 3.18 includes four nodes and is responsible for implementing the collected data as a model in Revit. The three nodes on the right make sure that previously generated models are removed from the project if the contained *Boolean* node *Clear* is set to “True”. For this purpose, the *Level* node and the *All Elements at Level* node collect all elements at the default "Level 1" and output them to the *Model Creation* node.

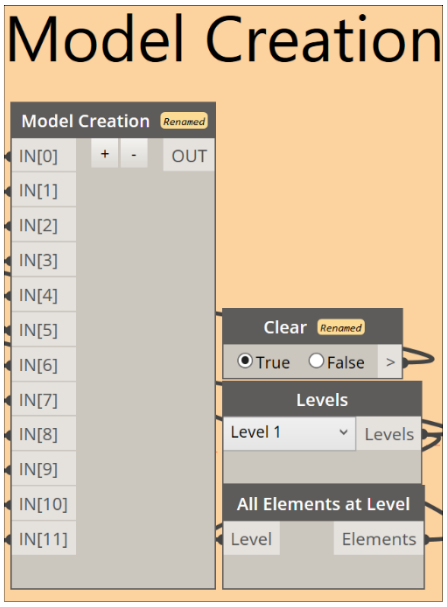


Figure 3.18 Overview of “Model Creation” cluster

The python script node *Model Creation* will be analyzed in the following section. The appurtenant source code in full length can be found in the Appendix A.2. Additional to the standard ones, the libraries loaded for this python script are math, “RevitNodes”, “RevitAPI”, “DocumentManager” and “TransactionManager”.

Thereafter, all the relevant inputs are assigned to variables and if necessary, converted to either an integer or a dynamo specific data type. This is done with the “UnwrapElement()” function, because Revit classes differ from the Dynamo equivalent [22]. In other Dynamo nodes this is done automatically, but if elements are input to a python script node, it is essential to unwrap them. Afterwards, an output list and three lists for levels, walls, and floors are created and the units of the used measurements are converted to meters.

After these preparations, a transaction is started in the Revit API, which enables Dynamo to make changes to the current Revit document. The first action to be done is clearing the current project of existing elements, so the process can be repeated iteratively. The algorithm collects all levels and clears all elements separately from the first level and those above, given that they exist. Additionally, the first level gets appended to *levelList*.

```
#Clear previous generated elements
levelArray = (FilteredElementCollector(doc)
    .OfCategory(BuiltInCategory.OST_Levels)
    .WhereElementIsNotElementType()
    .ToElements())

if levelArray.Count > 1:
    for levelElement in levelArray:
        if levelElement.Elevation != 0:
            doc.Delete(levelElement.Id)

if clear:
    for element in allElements:
        doc.Delete(element.Id)

levelList.append(level)
```

Figure 3.19 Clearance of previous generated elements

With the input *X*- and *Y*-dimensions, a list of corner points for the building model is created.

```
#Create corner points
x = [0, revitApiX, revitApiX, 0]
y = [0, 0, revitApiY, revitApiY]

points = []

point0 = XYZ(x[0], y[0], 0)
points.append(point0)
point1 = XYZ(x[1], y[1], 0)
points.append(point1)
point2 = XYZ(x[2], y[2], 0)
points.append(point2)
point3 = XYZ(x[3], y[3], 0)
points.append(point3)
```

Figure 3.20 Creation of corner points from x- and y-coordinates

Lines connecting the corner points are produced and utilized to build up the walls in the model. To create a floor, the corresponding level is needed as well as the outline as a curve array. If *basement* is “True”, the source code in Figure 3.21 generates a new level with an elevation of one negative level height, followed by the creation of lines, walls, a curve array and finally the basement floor.

```
#Create basement level, walls, and floor
if basement:
    basementLevel = Autodesk.Revit.DB.Level.Create(doc, (-1) * H)
    basementLevel.Name = "Level -1"

    lines = []

    line1 = Autodesk.Revit.DB.Line.CreateBound(
        XYZ(points[0].X, points[0].Y, (-1) * H),
        XYZ(points[1].X, points[1].Y, (-1) * H))
    line2 = Autodesk.Revit.DB.Line.CreateBound(
        XYZ(points[1].X, points[1].Y, (-1) * H),
        XYZ(points[2].X, points[2].Y, (-1) * H))
    line3 = Autodesk.Revit.DB.Line.CreateBound(
        XYZ(points[2].X, points[2].Y, (-1) * H),
        XYZ(points[3].X, points[3].Y, (-1) * H))
    line4 = Autodesk.Revit.DB.Line.CreateBound(
        XYZ(points[3].X, points[3].Y, (-1) * H),
        XYZ(points[0].X, points[0].Y, (-1) * H))

    lines.append(line1)
    lines.append(line2)
    lines.append(line3)
    lines.append(line4)

    curveArray = CurveArray()

    for line in lines:
        wall = Wall.Create(doc, line, wallType.Id, basementLevel.Id, H
        0, False, True)
        wallList.append(wall.ToDSType(False))
        curveArray.Append(line)

    floor = doc.Create.NewFloor(curveArray, floorType, basementLevel,
    True)
    floorList.append(floor.ToDSType(False))
```

Figure 3.21 Basement creation

Afterwards, the lines for the outside walls above ground are created on the ground level. These are used in the function for creating the outer walls of the model, which extend over all levels. This process is shown in Figure 3.22.

```
for line in lines:
    wall = Wall.Create(doc, line, wallType.Id, level.Id, levels * H, 0
    False, True)
    wallList.append(wall.ToDSType(False))
```

Figure 3.22 Above ground walls creation

The lines are then hitched together in a curve array. The distinction between whether a basement was created or not decides the floor type used in the ground level floor. The implementation of this is visible in Figure 3.23

```
if basement:
    floor = doc.Create.NewFloor(curveArray, basementCeiling, level, True)
else:
    floor = doc.Create.NewFloor(curveArray, floorType, level, True)

floorList.append(floor.ToDSType(False))
```

Figure 3.23 Ground level floor creation

Next, levels additional to the ground level are created in a for-loop shown in Figure 3.24. Once more, the corner points are connected by lines on each level, which then are connected in curve arrays. Furthermore, during the loop, the floors are produced, and the levels are named properly.

```
#Create levels and floors above ground
if levels >= 1:
    for levelNumber in range(1, levels + 1):
        newLevel = Autodesk.Revit.DB.Level.Create(doc, levelNumber * H)
        levelList.append(newLevel)

        line1 = Autodesk.Revit.DB.Line.CreateBound(
            XYZ(points[0].X, points[1].Y,  levelNumber * H),
            XYZ(points[1].X, points[1].Y, levelNumber * H))
        line2 = Autodesk.Revit.DB.Line.CreateBound(
            XYZ(points[1].X, points[1].Y, levelNumber * H),
            XYZ(points[2].X, points[2].Y, levelNumber * H))
        line3 = Autodesk.Revit.DB.Line.CreateBound(
            XYZ(points[2].X, points[2].Y, levelNumber * H),
            XYZ(points[3].X, points[3].Y, levelNumber * H))
        line4 = Autodesk.Revit.DB.Line.CreateBound(
            XYZ(points[3].X, points[3].Y, levelNumber * H),
            XYZ(points[0].X, points[0].Y, levelNumber * H))

        curveArray = CurveArray()
        curveArray.Append(line1)
        curveArray.Append(line2)
        curveArray.Append(line3)
        curveArray.Append(line4)

        floor = doc.Create.NewFloor(curveArray, floorType, newLevel, True)
        floorList.append(floor.ToDSType(False))

        newLevel.Name = "Level " + str(levelNumber + 1)
```

Figure 3.24 Creation of levels and floors above ground level

Concluding, the utilization units are created as shown in Figure 3.25. They are distributed as even as possible between the levels. If the number of units is not divisible by the number of levels, the lower levels will have more units. For example, 8 units and 3 levels would mean 3 units on the first and second level and 2 units on the third level.

To implement this, a list for the number of units per level is created. At first, each entry of the list is set to the quotient of units and levels rounded off. If this division leaves a remainder, the units that have not yet been assigned to a level are added from the bottom up with the help of a while loop. The following nested for-loops create lines and walls between the utilization units on the different levels.

```python
#Create inner walls to seperate utilization units
utilizationUnitsPerLevel = list(range(levels))

for i in range(levels):
    utilizationUnitsPerLevel[i] = int(math.floor(utilizationUnits / levels))

remainder = utilizationUnits - utilizationUnitsPerLevel[0] * levels

j = 0

while j < remainder:
    utilizationUnitsPerLevel[j] = utilizationUnitsPerLevel[j] + 1
    j = j + 1

for m in range(levels):
    for n in range(1, utilizationUnitsPerLevel[m]):
        utilizationUnitLine = Autodesk.Revit.DB.Line.CreateBound(
            XYZ((revitApiX / utilizationUnitsPerLevel[m]) * n, revitApiY, H * m),
            XYZ((revitApiX / utilizationUnitsPerLevel[m]) * n, 0, H * m))
        wall = Wall.Create(doc, utilizationUnitLine, wallType.Id, levelList[m].Id, H, 0, False, True)
        wallList.append(wall.ToDSType(False))
```

Figure 3.25 Creation of inner walls separating utilization units

To complete the process, the transaction is ended, and the lists of elements are appended to the output.

3.3.1 Sample models

The result of the *Model Creation* node is a building model in Revit consisting of outer walls, unit-separating walls, levels, floors, and a basement if applicable. The model gets updated with each execution of the program. This section will demonstrate the functionality by means of three sample models. For the first sample, the values selected for each parameter are displayed below along with the model in Revit.

Model A:

- ***buildingClass*:** Building Class 1 (Freestanding/Agricultural/Forestry)
- ***basement*:** False
- ***freestanding:*** True
- ***agricultural/forestry:*** False
- ***levelHeight [m]:*** 2.8
- ***levels:*** 2
- ***utilizationUnits:*** 2
- ***X [m]:*** 10
- ***Y [m]:*** 14
- ***wallType:*** fire retardant Generic – 300mm
- ***floorType:*** fire retardant Generic Floor – 400mm
- ***basementCeiling:*** -

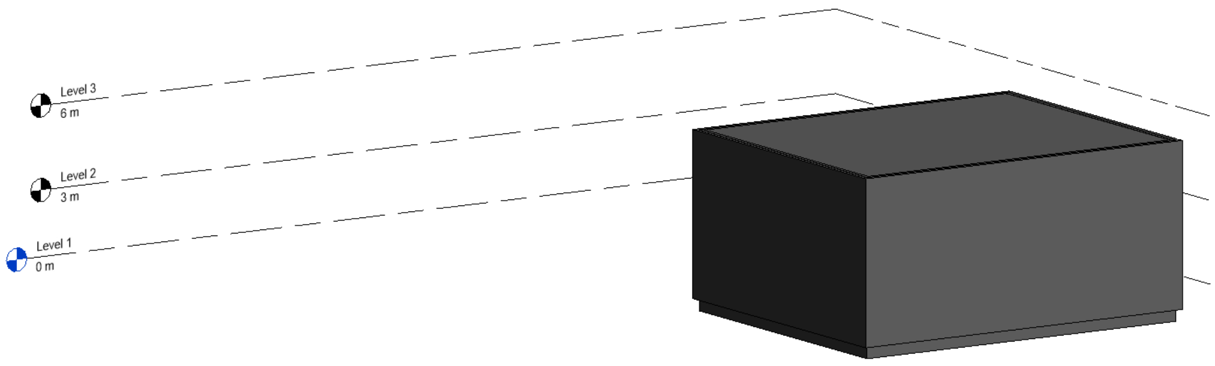


Figure 3.26 Model A in the Revit workspace

Model A is a freestanding building of class 3 with two levels of 2.8 m and a footprint of 140 m² or 240 m² of gross floor area. No basement was selected, and the building is neither agricultural nor forestry. Two utilization units mean one per level, so no partition walls are created. The wall and floor types are rated fire retardant.

Model B:

- ***buildingClass***: Building Class 3
- ***basement***: True
- ***freestanding:*** False
- ***agricultural/forestry:*** False
- ***levelHeight [m]:*** 3
- ***levels:*** 3
- ***utilizationUnits:*** 5
- ***X [m]:*** 14
- ***Y [m]:*** 18
- ***wallType:*** fire retardant Generic – 300mm
- ***floorType:*** fire retardant Generic Floor – 400mm
- ***basementCeiling:*** fire resistant Generic Floor – 400mm

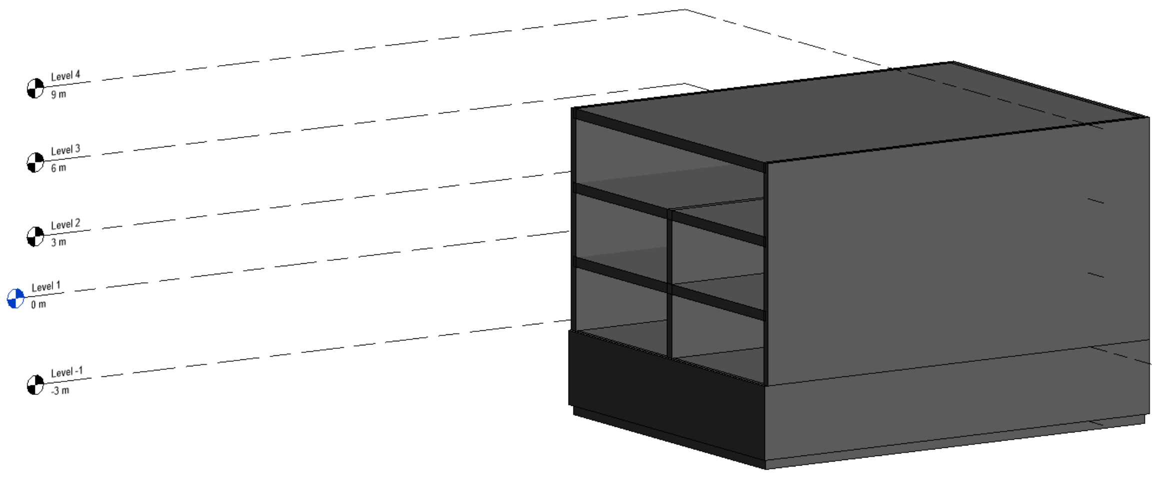


Figure 3.27 Model B in the Revit workspace

Model B is categorized as building class 3. It includes a basement and is neither freestanding nor agricultural/forestry. It has 3 levels of 3 m height, which means that the relevant height for the building class determination is 6 m. The gross floor area is not limited for building class 3 and amounts to 252 m². One outside wall was hidden manually, so the distribution of the five utilization units is visible in Figure 3.27. Walls and floors are rated fire retardant apart from the basement ceiling, which is fire resistant.

<u>Model C:</u>

- ***buildingClass***: Building Class 5
- ***basement***: True
- ***freestanding:*** False
- ***agricultural/forestry:*** False
- ***levelHeight [m]:*** 3
- ***levels:*** 5
- ***utilizationUnits:*** 15
- ***X [m]:*** 40
- ***Y [m]:*** 25
- ***wallType:*** fire resistant Generic – 300mm
- ***floorType:*** fire resistant Generic Floor – 400mm
- ***basementCeiling:*** fire resistant Generic Floor – 400mm

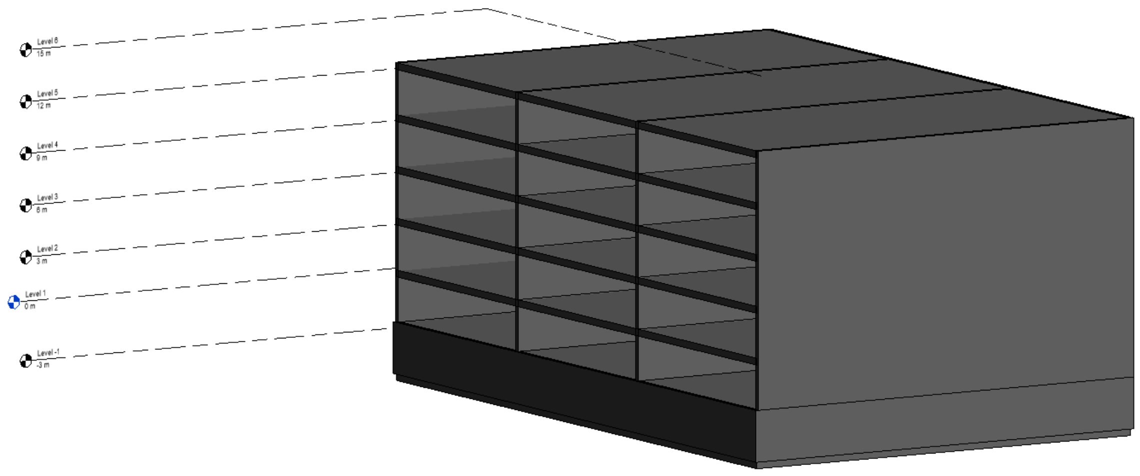


Figure 3.28 Model C in the Revit workspace

For Model C building class 5 was selected. The building with 15 utilization units that are evenly spread on five levels of 3 m has a basement is not freestanding or agricultural/forestry. The gross floor area equals 1000 m², which means that each utilization unit has an area of approximately 67 m². All walls and floors are rated fire resistant. Due to the choice of parameters, the Building Class Control group returns “Building Class 4”. This means, that although class 5 was selected the building could also be classified as building class 4 and therefore use a bigger variety of materials for example.

3.4 Control cluster

The “Control” cluster in Figure 3.29 contains an *Errors* group and a *Building Class Control* group.

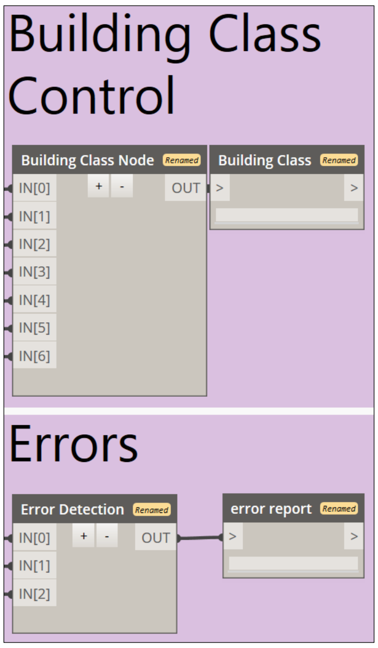


Figure 3.29 Overview of “Control” cluster

The python script node *Error Detection* reports an error if building class 1 is selected, but neither *freestanding* nor *agricultural/forestry* is set to “True”. This error message is then displayed on a *Watch* node. The error has no impact on the generation of the model.

```
#Load the Python Standard and DesignScript Libraries
import sys
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *

#Assign input to variables
buildingClass = IN[0]
freestanding = IN[1]
agricultureforestry = IN[2]

#Check for error with building class 1 parameters and output report
if buildingClass == 1 and not freestanding and not agricultureforestry:
        OUT = "Building must be freestanding\nor agricultural/forestry"
else:
        OUT = "no error"
```

Figure 3.30 Implementation of *Error Detection* node

The inputs to the *Building Class Control* group are all the relevant parameters for the building class determination. A python script node then identifies the actual building class of the generated model. This can show the user that a lower building class would also be applicable for the inserted parameters, which is supposed to motivate the user to adapt the design of the model. Alternatively, a higher building class indicates wrong usage of the program.

The python algorithm of this node is shown in Figure 3.31. The standard libraries are loaded, and the building parameters imported and assigned to variables. As a next step, the gross floor area and height of the model are calculated. The subsequent determination of the building class was implemented with the help of Figure 2.4. The area of the largest unit generated with this program is used for the calculation. The building class is forwarded to a *Watch* node to display it to the user.

```python
#Load the Python Standard and DesignScript Libraries and Math
import sys
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *
import math

#Assign input to variables
x = IN[0]
y = IN[1]
levels = IN[2]
levelHeight = IN[3]
units = IN[4]
freestanding = IN[5]
agricultureforestry = IN[6]

#Calculate building height and gross floor area
area = int(x * y * levels)
height = int((levels - 1) * levelHeight)

#Determine and output building class
if not agricultureforestry:
        if height <= 7:
                if area <= 400 and units <= 2:
                        if freestanding:
                                buildingClass = "Building Class 1"
                        else:
                                buildingClass = "Building Class 2"
                else:
                        buildingClass = "Building Class 3"
        else:
                if (height <= 13
                and (x * y / math.floor(units / levels)) <= 400):
                        buildingClass = "Building Class 4"
                else:
                        buildingClass = "Building Class 5"
else:
        buildingClass = "Building Class 1"

OUT = buildingClass
```

Figure 3.31 Implementation of *Building Class Node*

4 Conclusion

This section rounds up the thesis by summarizing and evaluating the project, addressing boundaries that were met and giving an outlook for possible improvement or expanding work for the future.

The goal of the work was to create parametric building models that automatically comply with fire-safety regulations. Dynamo for Revit 2021 was chosen as the platform for the implementation because it offers a visual programming interface and is therefore relatively easy to operate while using the well-established Revit database.

The program to generate the parametric model was built successfully. It includes eleven parameters to modify the building, three of which are material selections. All five building classes from the BayBO may be represented in a building model, which can be specified in terms of floor area, height, and division into floors as well as utilization units. Furthermore, the option for a basement was implemented.

The program checks the entries and informs about the possibility to classify the building in a lower class. Resulting from the selected building class, the program filters object types for walls and floors. The compliance with fire-safety regulations was implemented with the help of python script nodes, which automatically update the limits of parameters dependent on inputs. Python was also used to create the model with functions from Dynamo and the Revit API.

4.1 Boundaries

A boundary encountered in the course of the implementation was the limitation of parameters and their values. One reason for some compromises was the goal to keep the program clearly structured, in the sense of not being cluttered with parameters. As it would be very complicated or even impossible to enable any desired shape of a building to be generated, some kind of compromise is necessary to not exceed the scope suitable for this thesis. The author decided to settle with a simple rectangular building. A good alternative would have been to allow some selected shapes like L-formed or U-formed buildings. This would have needed several more nodes for parameterization and rule compliance. Allowing angles other than right angles would result in an additional parameter for each angle and a much more complicated calculation of the rule compliance.

Parameters that are not limited by any regulation needed to be assigned a custom maximum for this thesis because infinity was not found to be implementable with Dynamo in the parametric environment. Furthermore, to keep the program input clear and understandable the distribution of the utilization units is automated and units over multiple floors can be selected and are used for building class determination but are not represented in the model itself. The implementation of these functions would have led to a disproportionate number of additional parameters compared to the benefit gained from them.

Another boundary was found with Revit and Dynamo themselves. During the process of the implementation no practicable way to create a non-flat roof parametrically was found in Dynamo. Although this is not seen as a drastic limitation, as the model does not claim to be complete but serves as a basis for further modeling. Additionally, the Revit family libraries had to be prepared by adding suitable wall and floor types for the filtering because the existing types had no fire resistance class assigned. However, it is assumed that a user of the software, such as a design or construction firm has or can create a well-stocked library.

4.2 Outlook

Possible improvements for the developed program could be made in the form of a more detailed model and consequently more parameters. For example, the exact position and area of each utilization unit including multi-story units could be implemented. Furthermore, walls to neighboring buildings could be chosen and thus be handled differently because of fire-safety restrictions. Other fire-safety relevant details in a building that could be implementable are elevators, stairs, and hallways. In order to not clutter the parameter input section of the interface, some kind of graphic input with multiple points for positions, dimensions, or forms would be beneficial.

Dynamo was chosen since it offers an easy-to-understand interface that can be used even with little experience, and it also works together with Revit, which is widely known in AEC. The visual programming environment is relatively easy to use, but also brings limited possibilities with it. Alternative software with more flexible input options such as a graphical input or a separation of front-end and back-end could improve the functionality of the program.

Autodesk introduced generative design for Revit and Dynamo through internal functions and the new software Project Refinery Beta. Both are supposed to give members of the AEC industry the opportunity to explore and optimize Revit designs. The user can create and visualize sample studies for his project where a lot of solutions for the given task are generated and ranked based on input parameters. Autodesk mentions workspace layouts, optimized window views and other possible applications. Furthermore, Dynamo users can create their own custom studies, which would be very interesting to explore further with this work. [11]

5 Bibliography

[1] S. Abrishami, J. Goulding, F. Pour Rahimian, A. Ganah, Integration of BIM and generative design to exploit AEC conceptual design innovation, Information Technology in Construction 19 (2014), 350-359.

[2] J. Abualdenien, A. Borrmann, A meta-model approach for formal specication and consistent management of multi-LOD building models, Advanced Engineering Informatics 40 (2019), 135-153.

[3] J. Abualdenien, A. Borrmann, Vagueness visualization in building models across different design stages, Advanced Engineering Informatics 45, 2020.

[4] J. Abualdenien, S. Pfuhl, A. Braun, Development of an MVD for checking fire-safety and pedestrian simulation requirements, 2019.

[5] J. Abualdenien, P. Schneider-Marin, A. Zahedi, H. Harter, H. Exner, D. Steiner, M. Singh, A. Borrmann, W. Lang, F. Petzold, M. König, P. Geyer, M. Schnellenbach-Held, Consistent management and evaluation of building models in the early design stages, Electronic Journal of Information Technology in Construction 25 (2020), 212-232.

[6] ALLPLAN Blog, Parametric BIM Modeling - Efficiency And Flexibility In Planning Processes, 2020. https://blog.allplan.com/en/parametric-bim-modeling (visited: 2021-04-27)

[7] H. Anave (uma.px.anave.0901), Celery for Dynamo 2.5 Version 20.6.7, 2020. https://dynamopackages.com/# (visited: 2021-04-21)

[8] Autodesk, Demystifying Generative Design for Architecture, Engineering and Construction, 2018. https://damassets.autodesk.net/content/dam/autodesk/www/solutions/generative-design/autodesk-aec-generative-design-ebook.pdf (visited: 2021-04-27)

[9] Autodesk, Dynamo 2.6.1.8850, 2020. https://dynamobim.org/download/ (visited: 2021-04-14)

[10] Autodesk, Dynamo Primer, 2019. https://primer.dynamobim.org/en/index.html (visited: 2021-04-14)

[11] Autodesk, Generative Design in Revit now available, 2020. https://blogs.autodesk.com/revit/2020/04/08/generative-design-in-revit/ (visited: 2021-04-28)

[12] Autodesk, Revit 2021, 2020. https://www.autodesk.de/products/revit/overview?term=1-YEAR (visited: 2021-04-14)

[13] BaustoffWissen, Erklärt: Feuerwiderstandsklassen, 2013. https://www.baustoffwissen.de/baustoffe/baustoffknowhow/baurecht/erklaert-feuerwiderstandsklassen/ (visited: 2021-04-22)

[14] BayBO, Bayerische Bauordnung in the version of the announcement of August 14, 2007 (GVBl. S. 588, BayRS 2132-1-B), last modified pursuant to §1 of the Act of December 23, 2020 (GVBl. S. 663). https://www.gesetze-bayern.de/Content/Document/BayBO (visited: 2021-04-11)

[15] A. Borrmann et al., Building Information Modeling : Technology Foundations and Industry Practice, Springer International Publishing AG, 2018.

[16] M. M. Burnett, Visual Programming, in: Wiley Encyclopedia of Electrical and Electronics Engineering, J.G. Webster (Ed.), 1999. pp. 1-1.

[17] P. Christensen, API Definition, 2016. https://techterms.com/definition/api (visited: 2021-04-22)

[18] DIN 277-1, Grundflächen und Rauminhalte (Hochbau), 2016.

[19] DIN 4102-2, Brandverhalten von Baustoffen und Bauteilen; Bauteile, Begriffe, Anforderungen und Prüfungen, 1977.

[20] C. Eastman, P. Teichholz, R. Sacks, K. Liston, BIM Handbook: A Guide to Building Information Modeling for Owners, Managers, Designers, Engineers, and Contractors, John Wiley & Sons, 2011.

[21] R. Fernando, R. Drogemuller, A. Burden, Parametric and generative methods with building information modeling, Proceedings of the 17th International CAADRIA (2012), 537-546.

[22] O. Green, Dynamo Python Primer, 2020. https://dynamopythonprimer.gitbook.io/dynamo-python-primer/4-revit-specific-topics/unwrapping-revit-elements (visited: 2021-04-21)

[23] P. Janssen, Parametric BIM Workflows, Proceedings of the 20th International CAADRIA (2015), 437-446.

[24] E. Kalkan, F.Y. Okur, A.C. Altunışık, Applications and usability of parametric modeling, Journal of Construction Engineering, Management & Innovation Volume 1 Issue 3 (2018), 139-146.

[25] NBS, What is Building Information Modelling (BIM)?, 2016. https://www.thenbs.com/knowledge/what-is-building-information-modelling-bim (visited: 2021-04-14)

[26] N. Panchal, How Important is Revit Family Modelling for the AEC Industry?, 2018. https://www.xscad.com/blog/how-important-is-revit-family-modelling-for-the-aec-industry (visited: 2021-04-22)

[27] S. Pfuhl, Analysis of Exchange Requirements for BIM- based Fire Code Compliance Checking, Bachelor's Thesis, Technische Universität München, 2018.

[28] Wikipedia, Revit, 2020. https://de.wikipedia.org/wiki/Revit (visited: 2021-04-11)

Appendix A

A.1 Parameter Manual

→ **Run [F5]** to create building class list

- Select ❶ ***buildingClass***

→ **Run [F5]**

- Select ❷ ***basement,***

  ❸ ***freestanding,***

  ❹ ***agricultural/forestry***

- Select ❺ ***levelHeight***

→ **Run [F5]**

- Select ❻ ***levels***

→ **Run [F5]**

- Select ❼ ***utilizationUnits***

→ **Run [F5]**

- Select ❽ ***X***

→ **Run [F5]**

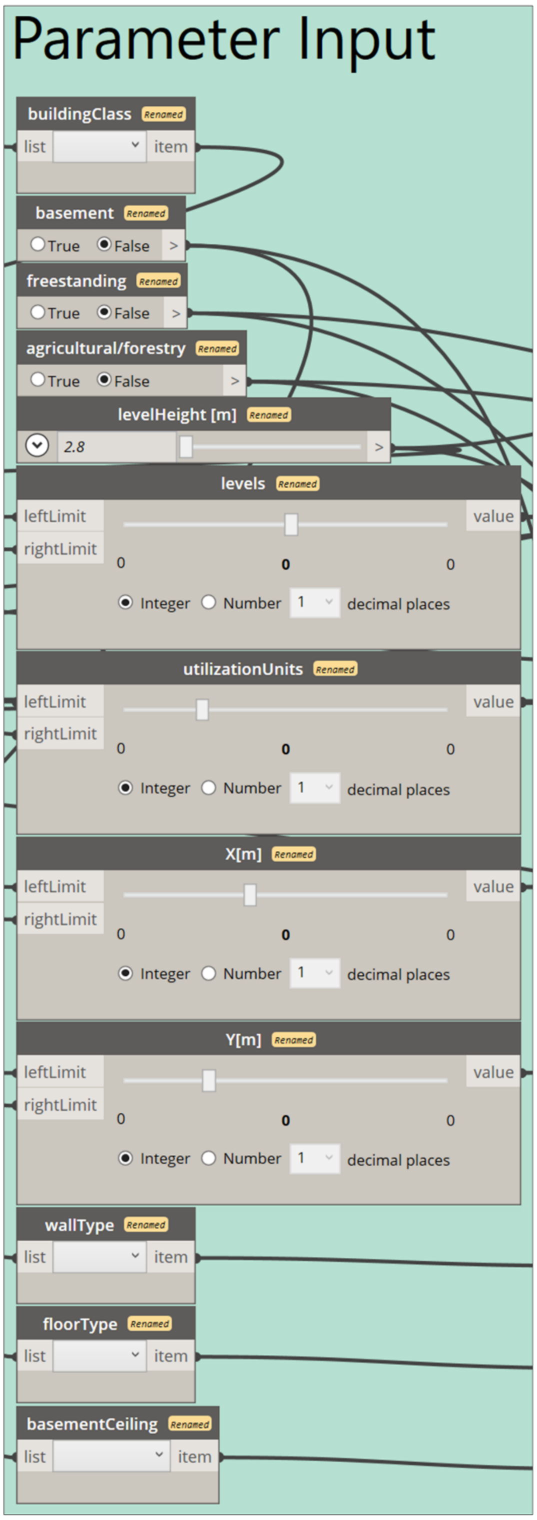
- Select ❾ ***Y***

- Select ❿ ***wallType,***

  ⓫ ***floorType***

- Select ⓬ ***basementCeiling***

→ **Run [F5]**



Parameter Input

buildingClass Renamed
list item

basement Renamed
True [✓ False]

freestanding Renamed
True [✓ False]

agricultural/forestry Renamed
True [✓ False]

levelHeight [m] Renamed
2.8

levels Renamed
leftLimit rightLimit value
0 0 0
[✓ Integer] Number 1 decimal places

utilizationUnits Renamed
leftLimit rightLimit value
0 0 0
[✓ Integer] Number 1 decimal places

X[m] Renamed
leftLimit rightLimit value
0 0 0
[✓ Integer] Number 1 decimal places

Y[m] Renamed
leftLimit rightLimit value
0 0 0
[✓ Integer] Number 1 decimal places

wallType Renamed
list item

floorType Renamed
list item

basementCeiling Renamed
list item

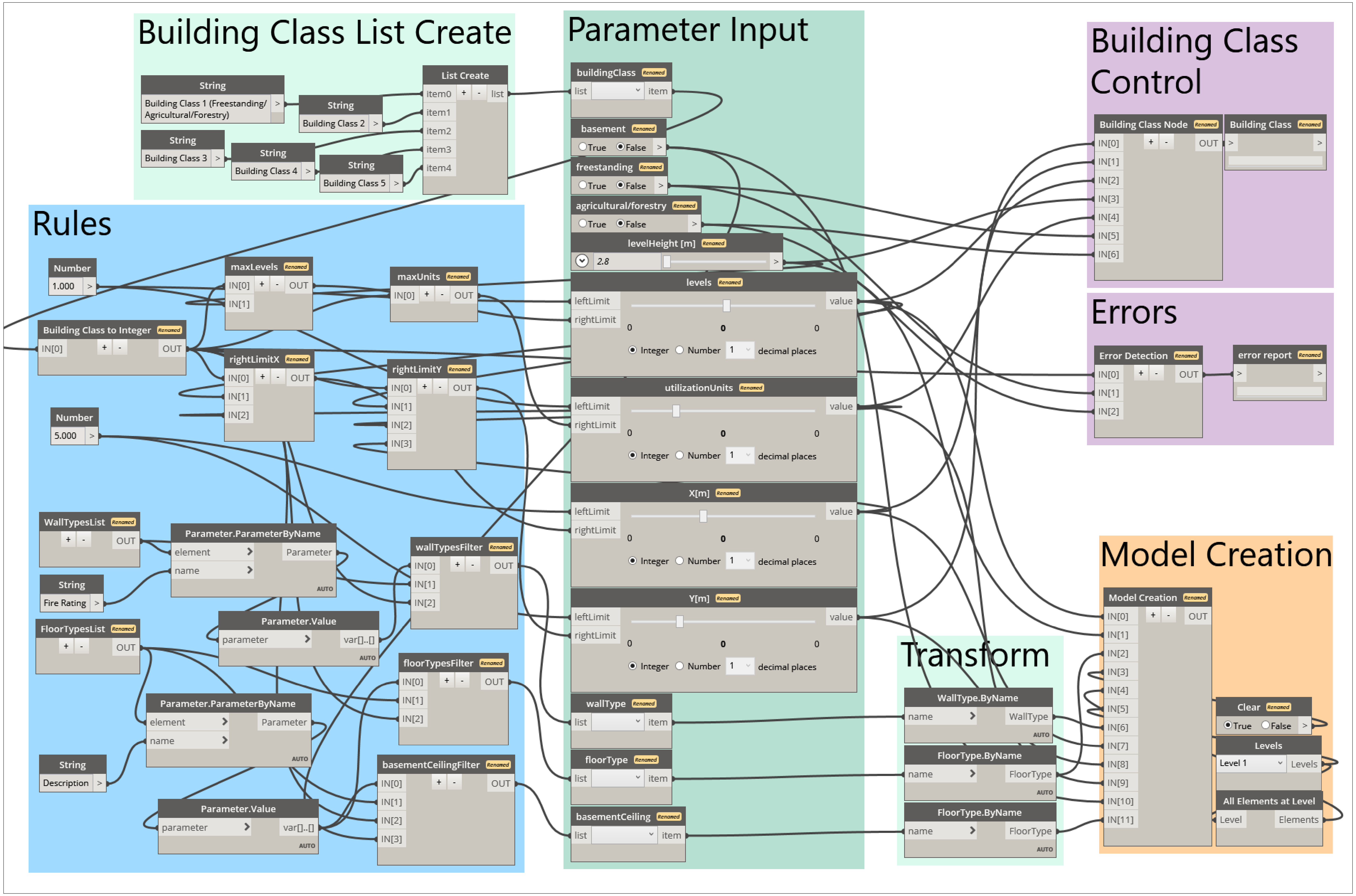
A.2 Model Creation Python Script

```python
#Load the Python Standard and DesignScript Libraries and Math
import sys
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *
import math

#Import Revit Nodes
clr.AddReference("RevitNodes")
from Revit.Elements import *
import Revit
clr.ImportExtensions(Revit.Elements)
clr.ImportExtensions(Revit.GeometryConversion)

#Import RevitAPI
clr.AddReference("RevitAPI")
import Autodesk
from Autodesk.Revit.DB import *
from Autodesk.Revit.DB import StairsEditScope
from Autodesk.Revit.DB import Parameter
from Autodesk.Revit.DB.Architecture import StairsRun
from Autodesk.Revit.DB.Architecture import *
from Autodesk.Revit.DB import IFailuresPreprocessor

#Import DocumentManager and transactionmanager
clr.AddReference("RevitServices")
from RevitServices.Persistence import DocumentManager
from RevitServices.Transactions import TransactionManager

#Assign input to variables
levels = int(IN[0])
height = IN[1]
floorType = UnwrapElement(IN[2])
clear = IN[3]
allElements = UnwrapElement(IN[4])
level = UnwrapElement(IN[5])
wallType = UnwrapElement(IN[6])
lengthX = IN[7]
lengthY = IN[8]
utilizationUnits = int(IN[9])
basement = IN[10]
basementCeiling = UnwrapElement(IN[11])

#Create output lists
output = []
wallList = []
levelList = []
floorList = []

#Convert units to meters
H = UnitUtils.ConvertToInternalUnits(height,
DisplayUnitType.DUT_METERS)
revitApiX = UnitUtils.ConvertToInternalUnits(lengthX,
DisplayUnitType.DUT_METERS)
```

```python
revitApiY = UnitUtils.ConvertToInternalUnits(lengthY,
DisplayUnitType.DUT_METERS)

#Open document and start transaction
doc = DocumentManager.Instance.CurrentDBDocument
TransactionManager.Instance.EnsureInTransaction(doc)

#Clear previous generated elements
levelArray = (FilteredElementCollector(doc)
    .OfCategory(BuiltInCategory.OST_Levels)
    .WhereElementIsNotElementType()
    .ToElements())

if levelArray.Count > 1:
    for levelElement in levelArray:
        if levelElement.Elevation != 0:
            doc.Delete(levelElement.Id)

if clear:
    for element in allElements:
        doc.Delete(element.Id)

levelList.append(level)

#Create corner points
x = [0, revitApiX, revitApiX, 0]
y = [0, 0, revitApiY, revitApiY]

points = []

point0 = XYZ(x[0], y[0], 0)
points.append(point0)
point1 = XYZ(x[1], y[1], 0)
points.append(point1)
point2 = XYZ(x[2], y[2], 0)
points.append(point2)
point3 = XYZ(x[3], y[3], 0)
points.append(point3)

#Create basement level, walls, and floor
if basement:
    basementLevel = Autodesk.Revit.DB.Level.Create(doc, (-1) * H)
    basementLevel.Name = "Level -1"

    lines = []

    line1 = Autodesk.Revit.DB.Line.CreateBound(
        XYZ(points[0].X, points[0].Y, (-1) * H),
        XYZ(points[1].X, points[1].Y, (-1) * H))
    line2 = Autodesk.Revit.DB.Line.CreateBound(
        XYZ(points[1].X, points[1].Y, (-1) * H),
        XYZ(points[2].X, points[2].Y, (-1) * H))
    line3 = Autodesk.Revit.DB.Line.CreateBound(
        XYZ(points[2].X, points[2].Y, (-1) * H),
        XYZ(points[3].X, points[3].Y, (-1) * H))
    line4 = Autodesk.Revit.DB.Line.CreateBound(
        XYZ(points[3].X, points[3].Y, (-1) * H),
        XYZ(points[0].X, points[0].Y, (-1) * H))

    lines.append(line1)
    lines.append(line2)
    lines.append(line3)
    lines.append(line4)

    curveArray = CurveArray()

    for line in lines:
        wall = Wall.Create(doc, line, wallType.Id, basementLevel.Id, H,
        0, False, True)
        wallList.append(wall.ToDSType(False))
        curveArray.Append(line)

    floor = doc.Create.NewFloor(curveArray, floorType, basementLevel,
    True)
    floorList.append(floor.ToDSType(False))

#Create outer walls
lines = []

line1 = Autodesk.Revit.DB.Line.CreateBound(points[0], points[1])
line2 = Autodesk.Revit.DB.Line.CreateBound(points[1], points[2])
line3 = Autodesk.Revit.DB.Line.CreateBound(points[2], points[3])
line4 = Autodesk.Revit.DB.Line.CreateBound(points[3], points[0])

lines.append(line1)
lines.append(line2)
lines.append(line3)
lines.append(line4)

for line in lines:
    wall = Wall.Create(doc, line, wallType.Id, level.Id, levels * H, 0,
    False, True)
    wallList.append(wall.ToDSType(False))

#Create ground level floor or basement ceiling
lines = []

line1 = Autodesk.Revit.DB.Line.CreateBound(points[0], points[1])
line2 = Autodesk.Revit.DB.Line.CreateBound(points[1], points[2])
line3 = Autodesk.Revit.DB.Line.CreateBound(points[2], points[3])
line4 = Autodesk.Revit.DB.Line.CreateBound(points[3], points[0])

lines.append(line1)
lines.append(line2)
lines.append(line3)
lines.append(line4)

curveArray = CurveArray()

curveArray.Append(line1)
curveArray.Append(line2)
curveArray.Append(line3)
curveArray.Append(line4)

if basement:
    floor = doc.Create.NewFloor(curveArray, basementCeiling, level,
    True)
else:
    floor = doc.Create.NewFloor(curveArray, floorType, level, True)

floorList.append(floor.ToDSType(False))

#Create levels and floors above ground
if levels >= 1:
```

```python
    for levelNumber in range(1, levels + 1):
        newLevel = Autodesk.Revit.DB.Level.Create(doc, levelNumber * H)
        levelList.append(newLevel)

        line1 = Autodesk.Revit.DB.Line.CreateBound(
            XYZ(points[0].X, points[1].Y,  levelNumber * H),
            XYZ(points[1].X, points[1].Y, levelNumber * H))
        line2 = Autodesk.Revit.DB.Line.CreateBound(
            XYZ(points[1].X, points[1].Y, levelNumber * H),
            XYZ(points[2].X, points[2].Y, levelNumber * H))
        line3 = Autodesk.Revit.DB.Line.CreateBound(
            XYZ(points[2].X, points[2].Y, levelNumber * H),
            XYZ(points[3].X, points[3].Y, levelNumber * H))
        line4 = Autodesk.Revit.DB.Line.CreateBound(
            XYZ(points[3].X, points[3].Y, levelNumber * H),
            XYZ(points[0].X, points[0].Y, levelNumber * H))

        curveArray = CurveArray()
        curveArray.Append(line1)
        curveArray.Append(line2)
        curveArray.Append(line3)
        curveArray.Append(line4)

        floor = doc.Create.NewFloor(curveArray, floorType, newLevel, True)
        floorList.append(floor.ToDSType(False))

        newLevel.Name = "Level " + str(levelNumber + 1)

#Create inner walls to seperate utilization units
utilizationUnitsPerLevel = list(range(levels))

for i in range(levels):
    utilizationUnitsPerLevel[i] = int(math.floor(utilizationUnits / levels))

remainder = utilizationUnits - utilizationUnitsPerLevel[0] * levels

j = 0

while j < remainder:
    utilizationUnitsPerLevel[j] = utilizationUnitsPerLevel[j] + 1
    j = j + 1

for m in range(levels):
    for n in range(1, utilizationUnitsPerLevel[m]):
        utilizationUnitLine = Autodesk.Revit.DB.Line.CreateBound(
            XYZ((revitApiX / utilizationUnitsPerLevel[m]) * n, revitApiY, H * m),
            XYZ((revitApiX / utilizationUnitsPerLevel[m]) * n, 0, H * m))
        wall = Wall.Create(doc, utilizationUnitLine, wallType.Id, levelList[m].Id, H, 0, False, True)
        wallList.append(wall.ToDSType(False))

#End transaction and output levels, floors, walls
TransactionManager.Instance.TransactionTaskDone()

output.append(levelList)
output.append(floorList)
output.append(wallList)
OUT = output
```

A.3 Dynamo Workspace Overview A3


Building Class List Create
String
Building Class 1 (Freestanding/ Agricultural/Forestry)
String
Building Class 2
String
Building Class 3
String
Building Class 4
String
Building Class 5
List Create
item0
item1
item2
item3
item4
list
Parameter Input
buildingClass
basement
True False
freestanding
True False
agricultural/forestry
True False
levelHeight [m]
2.8
levels
utilizationUnits
X[m]
Y[m]
leftLimit
rightLimit
value
Integer
Number
decimal places
wallType
floorType
basementCeiling
list
item
Building Class Control
Building Class Node
Building Class
Errors
Error Detection
error report
Rules
Number
1.000
maxLevels
maxUnits
Building Class to Integer
rightLimitX
rightLimitY
Number
5.000
WallTypesList
Parameter.ParameterByName
element
name
Parameter
String
Fire Rating
FloorTypesList
Parameter.Value
parameter
var[]..[]
wallTypesFilter
floorTypesFilter
basementCeilingFilter
String
Description
Transform
WallType.ByName
WallType
FloorType.ByName
FloorType
Model Creation
Model Creation
Clear
True False
Levels
Level 1
All Elements at Level
Level
Elements
IN[0]
OUT
Renamed
AUTO

Declaration of Originality

With this statement I declare that I have independently completed this Bachelor's thesis. The thoughts taken directly or indirectly from external sources are properly marked as such. This thesis was not previously submitted to another academic institution and has also not yet been published.

Munich, April 30, 2021

Patrick Nordmann