



A formal approach for algorithmic design of modular precast structures

Scientific work to obtain the degree

Master of Science (M.Sc.)

at the TUM School of Engineering and Design of the Technical University of Munich.

| | |
|----------------------|--|
| Supervised by | Prof. Dr. André Borrmann Sebastian Esser Chair of Computational Modelling and Simulation |
| Submitted by | Benedict Riley Harder |
| Submitted on | March 1st 2024 |

Abstract

This thesis focuses on the use of graph rewriting systems within the context of precast constructions to achieve algorithmic design. By employing a process model to encapsulate the algorithm logic, rewriting rules can be used to incrementally apply changes to the model represented in a graph. The process model serves the purpose of handling the sequence of necessary rule application as well as the geometric parameters of the basic modules. A method was proposed to handle the geometric conditions mandated by the structure by pre-processing the modules with the help of the process model, as well as post-processing the resulting graph. An implementation exploring the viability of this approach was developed using Rhino and Grasshopper, as well as an internally developed rule engine and algorithm. The implementation showed the usefulness and flexibility of graph rewriting as a method for algorithmic design but also pointed out the areas of improvement the approach has.

Contents

- 1 Introduction** **1**
 - 1.1 Problem Statement 1
 - 1.2 Scope and Structure 1

- 2 State Of The Art** **3**
 - 2.1 Fundamentals 3
 - 2.1.1 Geometric and Parametric Modeling 3
 - 2.1.2 Building Information Modeling 5
 - 2.1.3 Graph Theory 6
 - 2.1.4 Formal Languages 8
 - 2.1.5 Graph Rewriting 9
 - 2.1.6 Process Modeling 10
 - 2.1.7 Precast Structures 10
 - 2.2 Specific Related Engineering Science Papers 11
 - 2.2.1 Aggregations with Interlocking Parts 11
 - 2.2.2 Semi-Automated Generation of Infrastructure Models 12
 - 2.2.3 Graph-based mass customization of modular precast bridge systems . . . 12
 - 2.2.4 Graph-Based Version Control 13
 - 2.2.5 Parametric Building Graph Capturing 13
 - 2.2.6 More related works 14
 - 2.3 Research Gap 14

- 3 Methodology** **16**
 - 3.1 Parts and Components 16
 - 3.2 Rule Engine 17
 - 3.2.1 Main Graph 17
 - 3.2.2 Rule Definitions 17
 - 3.2.3 Rule Application 18
 - 3.2.4 Topological and Geometrical Conditions 18
 - 3.3 Process Model 21
 - 3.4 States 22
 - 3.5 Execution 22

- 4 Implementation** **25**
 - 4.1 Research Design 25
 - 4.1.1 Case Studies 25
 - 4.2 External 28
 - 4.2.1 Rhino 3D 28
 - 4.2.2 Rhino Compute 28
 - 4.2.3 Grasshopper 29

| | | |
|----------|--|-----------|
| 4.3 | Internal | 30 |
| 4.3.1 | Grammar Meta Model | 30 |
| 4.3.2 | Algorithm Meta Model | 34 |
| 4.3.3 | Case Study | 35 |
| 4.4 | Algorithm Overview | 36 |
| 4.4.1 | Details of Select Parts of the Program | 37 |
| 4.5 | Results | 41 |
| 4.5.1 | One Field Skeleton With Multiple Stories | 41 |
| 4.5.2 | Two Field Skeleton With One Story | 42 |
| 4.5.3 | Multi-Field, Multi-Story Skeleton | 43 |
| 4.6 | Implementation Issues | 45 |
| 4.6.1 | Part Parameters in Grasshopper | 45 |
| 4.6.2 | Grammar Definition and Debugging | 46 |
| 5 | Discussion | 47 |
| 5.1 | Research Questions | 47 |
| 5.1.1 | How easily can algorithmic design be achieved with the help of graph rewriting rules within the context of modular precast structures? | 47 |
| 5.1.2 | How detailed does the surrounding framework for rule application need to be to create usable structures? | 47 |
| 5.1.3 | How well can this approach be adapted to other fields of civil engineering? | 49 |
| 5.2 | Drawbacks and Improvements | 50 |
| 5.2.1 | Rule Definitions | 50 |
| 5.2.2 | Start State Specification | 53 |
| 5.2.3 | Planning State Specification | 53 |
| 6 | Conclusion | 54 |
| A | Files | 55 |
| | References | 56 |

List of Figures

| | | |
|------|---|----|
| 2.1 | Examples of a geometry using <i>BREPs</i> (a) and <i>tessellation</i> (b). Source: VILGERTSHOFER, 2022 | 4 |
| 2.2 | Examples of a geometry using <i>Constructive Solid Geometry</i> . Both (a) and (b) lead to the same result using cuboids and cylinders as primitives. Source: VILGERTSHOFER, 2022 | 4 |
| 2.3 | Example of various <i>sweeps</i> . (a): Extrusion (b): Rotation (c): Interpolation. Source: VILGERTSHOFER, 2022 | 5 |
| 2.4 | The dimension <code>column_distance</code> is set to a specific value. Meanwhile, the dimension <code>beam_length</code> is set to always equal the distance between the columns. | 6 |
| 2.5 | An example graph with directional edges. Vertices are labeled with letters while edges are labeled with numbers. | 7 |
| 2.6 | An example graph representing a door contained in a wall. | 7 |
| 2.7 | A basic example of a graph rewriting rule: The LHS sub-graph, consisting of one vertex connected to another one, which is connected to another one, would be replaced with the same graph with one more vertex and edge added. | 9 |
| 2.8 | Two examples of graph rewriting rules being applied: in the upper example, a sub-graph consisting of three vertices connected to each other is matched, and said three vertices are extended by another. In the second example, any vertex matches the LHS pattern. Yet, since the rule is only applied once, the matching LHS is only replaced once. | 10 |
| 2.9 | Topological interlocking assemblies: The main part (magenta) is constrained by its neighboring parts. Source: TESSMANN and ROSSI, 2019 | 11 |
| 2.10 | Aggregation of basic parts into an assembly using basic parts and a scalar field. The images show the progression of the assembly as more instructions are continuously executed along a scalar field. Source: TESSMANN and ROSSI, 2019 | 12 |
| 2.11 | Transformation of the initial graph state to a state including more details. Source: KOLBECK et al., 2023. | 13 |
| 2.12 | Transformation of the initial graph state to a state including more details. Source: ABUALDENIEN and BORRMANN, 2021. | 14 |
| 3.1 | Parts used as basic building blocks including their geometry, parameters and interfaces. The parameters can modify the geometry while the interfaces indicate surfaces other parts can connect to. | 16 |
| 3.2 | A set of two rules specifying how columns and beams are added to the graph. | 18 |
| 3.3 | A column standing on a foundation. | 18 |
| 3.4 | A simple aggregation consisting of two foundations, two columns and a beam. | 19 |
| 3.5 | Assembling the model from Figure 3.4 by using two foundations as our start symbol, then applying the rules defined in Figure 3.2. The column rule twice followed by the beam rule once. | 19 |

| | | |
|------|---|----|
| 3.6 | Example aggregation of parts using the WASP plugin for Rhino. Source: ROSSI, 2024 | 21 |
| 3.7 | The sequence of different states inside the process model. After the <code>StartState</code> , an arbitrary number of <code>PlanningState-AssemblingState</code> pairs can follow. | 23 |
| 3.8 | An example of how the process model controls the algorithm through its states. Each subfigure represents the result after applying the rules defined in either the start state (a), column planning state (b), beam planning state (c), or deck planning state (d). | 24 |
| 4.1 | Desired result for the first case study. | 26 |
| 4.2 | Desired result for the second case study. | 26 |
| 4.3 | Desired result for the third case study. | 27 |
| 4.4 | A simple Grasshopper script which creates a cube. | 29 |
| 4.5 | A cube in Rhino created with grasshopper. | 30 |
| 4.6 | General overview of the projects software architecture. | 31 |
| 4.7 | The architecture of the grammar meta model. | 32 |
| 4.8 | The architecture of the algorithm meta model. | 34 |
| 4.9 | Example structure of a case study. | 35 |
| 4.10 | General program flow of the algorithm during runtime. | 37 |
| 4.11 | Iterative calling of the rule engine. | 38 |
| 4.12 | Appropriate start state definitions for (a) 1x1 fields, (b) 1x2 fields, (c) 2x2 fields. | 39 |
| 4.13 | Different types of columns may be necessary even when placing them on the same type of foundation. In this case, the second column was placed using a completely different rule. | 40 |
| 4.14 | Mode resulting from defining one story and unequal side lengths. The red lines between the parts indicate that they share a connection (e.g. a column is connected to a beam). | 41 |
| 4.15 | Resulting model using a square plot and four stories. | 42 |
| 4.16 | Detail view showcasing that beams, decks and joints are correctly dimensioned. | 42 |
| 4.17 | The orientation of the foundations and their interfaces indicated by red lines. While the <code>StartState</code> itself may not change significantly (square geometries), the orientation is paramount for the subsequent placement of the columns. | 43 |
| 4.18 | Resulting model for two fields using the above mentioned rule catalogue. | 44 |
| 4.19 | Images detailing the columns with three or four consoles. | 44 |
| 4.20 | Example result using two fields and three stories. | 45 |
| 4.21 | Error in a grasshopper script: An input parameter is set in such a way that certain components do not receive their expected input. While Rhino may still be able to visualize the geometry without any problem, Rhino compute cannot. | 46 |
| 5.1 | Example of a resulting model using no process model setup. | 48 |
| 5.2 | Basic examples of what rule definitions could look like in infrastructure planning. | 49 |
| 5.3 | How a connection between the beam and its two neighboring columns is established according to the method used in the implementation. After the rule application, a connection check is issued and the relationship is established. | 51 |

| | | |
|-----|---|----|
| 5.4 | This new proposed rule definition for the placement of a beam resembles engineering knowledge more closely. | 51 |
| 5.5 | The node properties could be dependant on their neighboring nodes. Here, the length of the beam is conditional to the coordinates of the two columns. | 52 |
| 5.6 | Example application of a hierarchical rule. First, a set of parts is aggregated into one. Then the rule is applied which specifies that this aggregated part can be placed on top if it self. | 52 |

Acronyms

| | |
|-------------|-----------------------------------|
| API | application programming interface |
| CAD | computer aided design |
| GUI | graphical user interface |
| LHS | left-hand-side |
| OOP | object oriented programming |
| REST | representational state transfer |

Chapter 1

Introduction

Construction projects of any size, small or large, showcase a high complexity due to their multi-dimensional, collaborative, and long-term nature. This complexity in and of itself poses a great challenge when it comes to completing such a project within the bounds of the local building code, supply chains, availability of skilled workers, tight time-frames, and finances. The industrialization of the construction sector that lead to manual labor being replaced by machinery and materials becoming cheaper and more easily available has already greatly reduced overhead and costs and has helped overcome the complexity. Yet one crucial aspect of the construction process has seemingly become more intricate despite these advantages: the planning process.

Larger demand for buildings and infrastructure (BAUMANN et al., 2016) in conjunction with an ever-increasing amount of regulations, guidelines, and building codes (PREIDEL, 2020) have made the planning process a progressively more crucial as well as elaborate part of construction. This leads to a larger necessity of workers with the appropriate expertise and skill to perform the required tasks.

1.1 Problem Statement

So far, the detailed design of buildings has largely been the domain of adequately trained and educated engineers, architects and other specialists. The execution of their responsibilities has been aided in over recent decades by the development and adoption of digital tools and methods such as computer aided design (CAD) and model-based design and thus has reduced costs and overhead. But most of the design work still is conceived by the specialists themselves. Automating the design and planning process can further help to reduce the workload. On the one hand, even an only partly-implemented automation such as PREIDEL, 2020 or VILGERTSHOFER, 2022 proposed can yield great results and improvement. A full-scale automation of the entire design process from start to finish on the other hand has yet to be achieved. The plethora of factors that come into play for such an attempt at automation has been a major inhibiting aspect.

1.2 Scope and Structure

This thesis aims to develop an approach with which automatic algorithmic design can be achieved. Using graph rewriting and graph grammars as the basis of the approach, it should be able to automatically generate usable 3D-models of structures that adhere to the basic

structural concepts of construction. The approach will be tested within the context of building construction using precast modules with the help of an implementation that is designed to answer the following research questions:

- How easily can algorithmic design be achieved with the help of graph rewriting rules, within the context of modular precast structures?
- How detailed does the surrounding framework for rule application need to be to create usable structures?
- How well can this approach be adapted to other fields of civil engineering?

The structure of this thesis is as follows:

- Chapter 2 details the current state of the art concerning 3D-modeling, graph systems and graph rewriting, and process modeling.
- Chapter 3 explains the specific graph rewriting method used for the implementation.
- Chapter 4 describes the implementation and its technicalities in detail.
- Chapter 5 discusses the capabilities, drawbacks and limitations of the implementation.
- Chapter 6 concludes the topic and gives an outlook regarding future research.

Chapter 2

State Of The Art

2.1 Fundamentals

2.1.1 Geometric and Parametric Modeling

Core component of designing and planning products, whether these are buildings or otherwise, is a model of the product's geometry. While historically hand-made 2D-drawings were the norm, CAD has succeeded such drawings as of the late 80's (BJÖRK & LAAKSO, 2010). Capable of both 2D and 3D modeling, CAD software provides multiple ways to represent, modify, and visualize geometries (SARCAR et al., 2008). The choice of representation is crucial in order to enable changes in the geometry and the visualization of it in the desired manner. In the following, a selection of three-dimensional geometric representations will be briefly explained to form a basis of understanding of what the requirements for our geometry are.

Tessellation is a way of *explicitly* describing a geometry, meaning that the resulting geometry is stored as-is. Tessellation does this by discretising the object's surface with the help of polygons, though usually triangles are used (PHILLIPS, 2014). This is done by simply connecting points on the surface to triangles, with the accuracy of the geometry increasing with the decreasing size of triangles. This method is simple to understand, implement, and visualize and is used for visualizing as well as simulating purposes (BORRMANN et al., 2018). It also offers methods of computing its volume surface area, as well as being able to detect collisions with other geometries with a triangle-triangle intersection test such as the one proposed by MÖLLER, 1997. *Boundary representations* or *BREPs* for short describe the outside surface (or boundary) of a 3D volume model by connecting each surface to its neighbors. A (sur)face in this case is defined by its surrounding edges, which themselves are defined by their end vertices (STROUD, 2006). While modification is possible by directly changing the vertices, both tessellation and boundary representations suffer the disadvantage of being difficult to modify (BORRMANN et al., 2018). The major advantage of explicit geometries though is the ease of implementation and resulting cross-platform support they offer (VILGERTSHOFER, 2022).

In contrast to explicitly describing geometry we can also describe it *implicitly* by instead outlining the construction steps instead of the final geometry (OBERGRIESSER, 2016). This happens when using methods such as *Constructive Solid Geometry (CSG)*, where geometric primitives such as cuboids, cylinders, and more are combined to result in complex shapes. The primitives are either united (\cup), intersected (\cap), or subtracted (\setminus) from one another (FOLEY, 1996). Chaining these operations together eventually leads to the desired more complex shape, as seen in the example in [Figure 2.2](#).

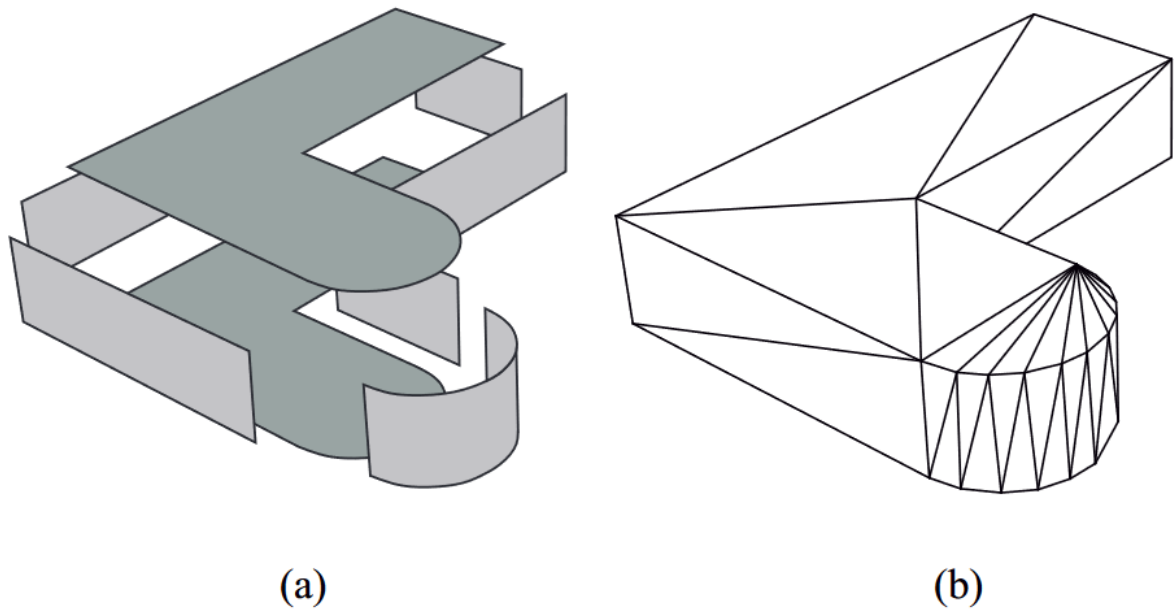


Figure 2.1: Examples of a geometry using *BREPs* (a) and *tessellation* (b). Source: VILGERTSHOFER, 2022

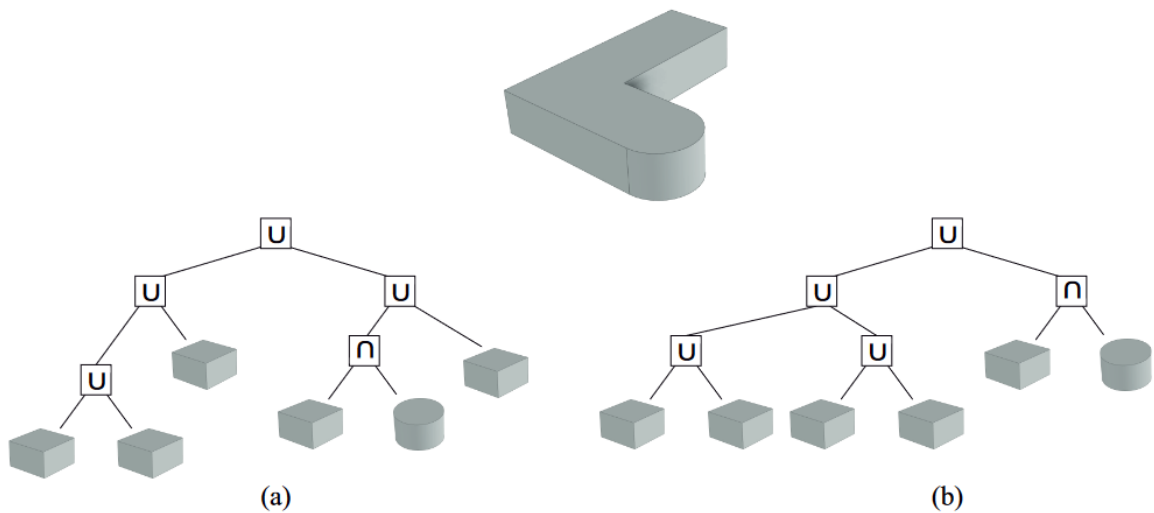


Figure 2.2: Examples of a geometry using *Constructive Solid Geometry*. Both (a) and (b) lead to the same result using cuboids and cylinders as primitives. Source: VILGERTSHOFER, 2022

In addition to CSG, *extrusion* or *sweeping* methods work by extending a 2D-surface along a path. As seen in [Figure 2.3](#), more complex paths such as rotation as well as interpolation between start- and endpoints are possible (VAJNA et al., 2018). Implicit representations of geometry offer an intuitive approach to geometry than their explicit counterparts since they describe *how* the geometry is created instead of describe *what* the geometry is. Implicit geometries can also be desired as a precursor of parametric modeling, making use of the ease of the modifiability (OBERGRIESSER, 2016).

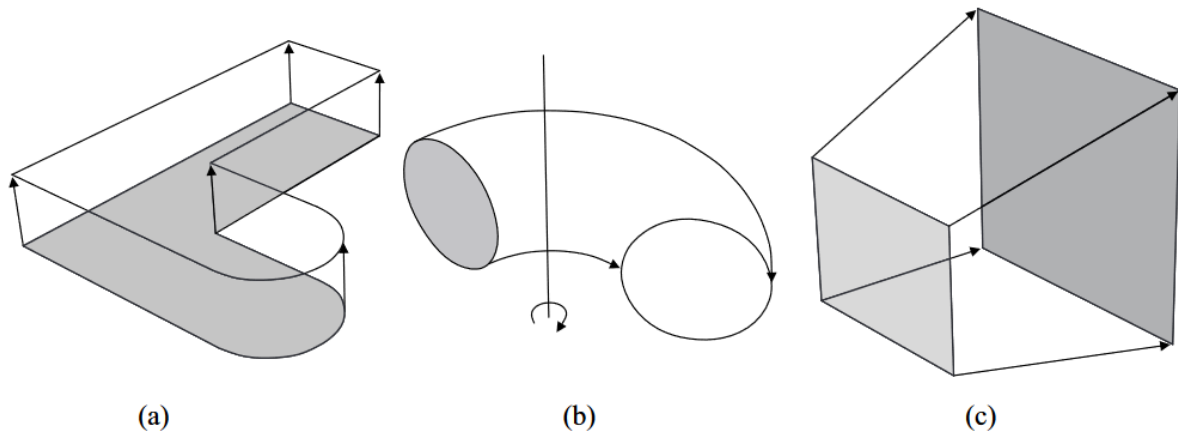


Figure 2.3: Example of various *sweeps*. (a): Extrusion (b): Rotation (c): Interpolation. Source: VILGERTSHOFER, 2022

Parametric Modeling

Above mentioned representations primarily focus on creating static geometries. In contrast, adaptable parametric models offer significant advantages. This kind of modeling is achieved using parametric constraints, that, as the name suggests, constrain parts of the geometry to adhere to certain conditions. These constraints include coincident, colinear, tangential, horizontal, vertical, parallel, perpendicular and fixated (SCHULTZ et al., 2017). Using these constraints can make the modeling process substantially faster and easier than even implicit geometries allow, by being able to propagate a change in one geometry to others (SHAH & MÄNTYLÄ, 1995). Making the length of one part be dependent on the distance between other parts for example, would not only make the modeling process faster but also more robust since certain conditions will always have to be met. In the case of a beam being placed between two columns, we would want the length of the beam to be equal to the distance between the columns (see [Figure 2.4](#)). While the Using parametric modeling is imperative to our approach since it offers us the tools necessary to dynamically adapt our model.

2.1.2 Building Information Modeling

Building Information Modeling (BIM) is a method of planning, constructing and operating buildings using digital tools. It aims to increase productivity, reduce costs, and improve the planning process of building over its entire life-cycle. The life-cycle specifically meaning every stage of the building from the conception, to planning, construction, operation and demolition (BORRMANN et al., 2018). BIM achieves this by aggregating all information concerning the building, specifically tying the geometric information, which is usually represented by a 3D-model, and the semantic information together. Semantic information is all the information apart from the geometry and can specify details including, but not limited to connections between components, materials used, attributes of the materials, and time information for the construction phase. BIM as a method has seen an increasing amount of use within the industry, especially within the last few

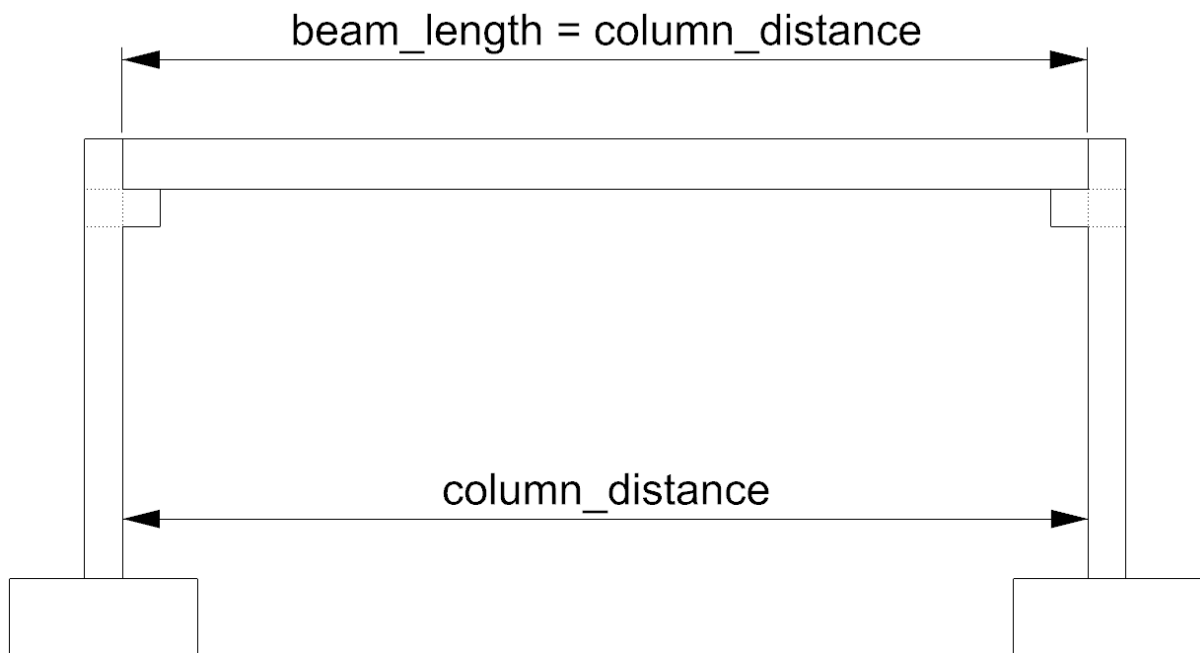


Figure 2.4: The dimension `column_distance` is set to a specific value. Meanwhile, the dimension `beam_length` is set to always equal the distance between the columns.

years (BORRMANN et al., 2018). Since the future of the construction sector will most certainly incorporate BIM concepts on a large scale, this approach must follow these concepts or at the very least be compatible with them.

2.1.3 Graph Theory

Graphs are mathematical structures that can depict relationships between entities. A graph G always consists of a set of vertices V (also called nodes, as well as a set of edges E (also called relationships), which itself simply are pairs of vertices: $e = x, y | e \in E | x, y \in V$. An edge can be either directional, meaning that the pair of vertices would be ordered and points *from* one vertex *to* the other ($x \rightarrow y | x, y \in V$) or non-directional, in which case the vertices are unordered. Each subset of edges and vertices $G_s = (V_s, E_s) | G_s \subset G | V_s \subset V | E_s \subset E$ where the endpoints of E_s only consists of vertices included in V_s is called a sub-graph of the original graph G (DIESTEL, 1996). See Figure 2.5 for an example.

Edges can describe the topological relationship between objects (represented by the vertices). To stay within the construction context, on such example would be a wall and a door each being represented by one vertex, and a directional edge going from the door to the wall detailing the relationship of the door being *contained in* the wall (see Figure 2.6). The vertices as well as the edges can of course itself contain more information such as a label. In our example, the labels of the vertices would be *Wall* and *Door* respectively, and the label of the directional edge would be *contained_in*.

Figure 2.6 merely illustrates as simple an example as possible. A graph can have an infinite amount of vertices and edges, with all vertices being able to be connected with each other.

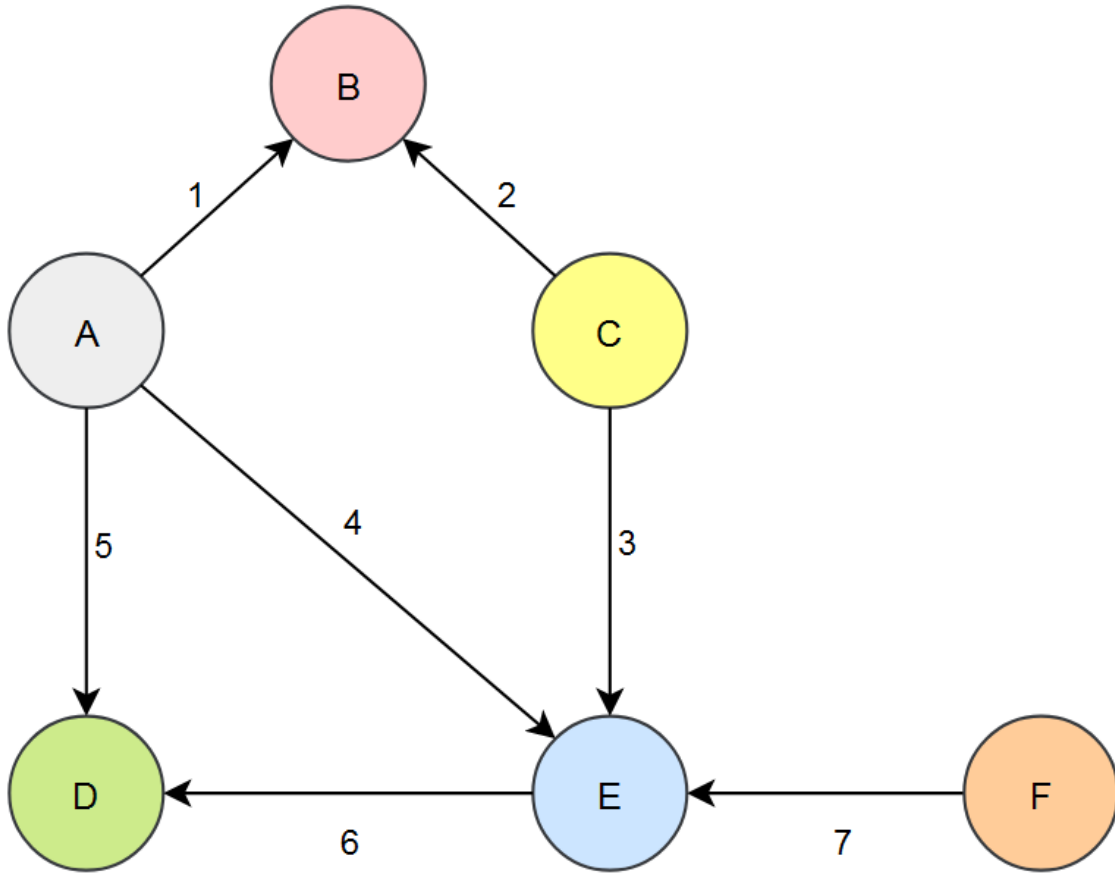


Figure 2.5: An example graph with directional edges. Vertices are labeled with letters while edges are labeled with numbers.

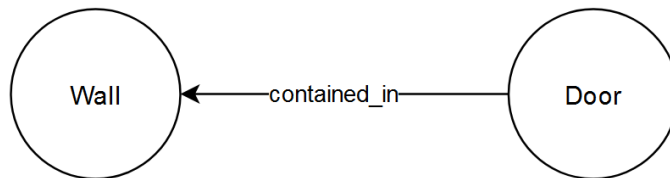


Figure 2.6: An example graph representing a door contained in a wall.

This structure provides a method of representing information of a complex nature by being able to associate objects with each other in a non-hierarchical way. This is because the above mentioned fact that each vertex can be connected to any other vertex, as opposed to there being a strict hierarchy as to what the relationship between nodes must look like. Of course one could impose a schema on such a graph structure and enforce certain conditions that nodes and relationships must follow. A binary tree, for example, is by its nature also a graph with a certain schema applied. In this case, there must be exactly one root node, with every node having a maximum of two outgoing relationships and itself only ever having one incoming relationship. Here we have enforced a certain hierarchy with the help of a schema which can help with representing information which closely follows certain rules.

The flexibility ability to encapsulate complex multi-layered data structures enable graphs to be used within many fields of computer science, for example in the form of databases (NEO4J, 2024). Keeping to the example, all information concerning a building (geometric as well as semantic) may be encapsulated within such a graph similar to how our door and wall were connected to each other (ISAAC et al., 2013). Of course the patterns this kind of graph must follow are far more intricate than what we have described so far. For example, a third node could be introduced with a label called *opening*. The *opening* would then *contained in* the *wall* and the *door* would be *contained in* the *opening*. Furthermore, the geometry of each of these objects could be represented inside separate nodes each connecting to their respective object, with of course the geometry itself also potentially forming a sub-graph. One example of a data schema for BIM that can be depicted using a graph are the Industry Foundation Classes (IFC) (ESSER et al., 2022; ZHU et al., 2023).

2.1.4 Formal Languages

Before continuing on with graph rewriting, let us briefly visit the concept of formal languages and formal grammar as a historic background. Originally formalized by Noam Chomsky in the late 50s, formal grammar is a concept in linguistics which describe whether a string of characters is valid within the context of its formal language. The basis of this concept is formed by rewriting rules which can transform a set of characters to another one (CHOMSKY, 1956).

$$Rule_1 \quad (R1) : A \rightarrow Ab$$

$$Rule_2 \quad (R2) : Ab \rightarrow Abc$$

$$Start \ Symbol : A$$

$$Rule_1 \rightarrow Rule_2 \rightarrow Rule_3 : A \Rightarrow Ab \Rightarrow Abb \Rightarrow Abcb$$

In this example, the string starts out with a single character *A* which forms our *start symbol*. A start symbol is the set of characters from which the rewriting process can be started. Along with our start symbol we have defined a set of rewriting rules, which each define the kind of transformation a symbol would undergo once the rule is applied. One of these *rewriting rules* is defined by its *left-hand-side* (to the left of the arrow), or the initial state before rewriting, and its *right-hand-side* (to the right of the arrow), or the following state after rewriting. A rule that is applied always replaces the given *left-hand-side* with the *right-hand-side*, provided the LHS exists within the symbol. In the context of formal grammars this system would be used to determine whether a string of characters can be created using nothing but the rewriting rules and a designated start symbol (CHOMSKY, 1956). If it can, the string is a valid within that formal language. The last part has little to no relevance for our use case, but the system of rewriting that has been established here cannot only be used inside the field of linguistics, but has also found wide adoption in mathematics and computer science, with graph rewriting being one example (HELMS, 2013).

2.1.5 Graph Rewriting

As explained earlier, graphs can be used to depict and represent information, by representing objects as vertices and their relationships between them as edges. We have also explored the concept of rewriting rules that can transform a string into another one. The combination of these two concepts is what is called graph rewriting. Similar to the aforementioned formal grammar, graph rewriting uses rewriting rules to transform one graph into another one (HELMS, 2013). These rules, also called *transformation rules*, behave in a similar fashion to their grammar counterparts as their general structure is identical. Both have a LHS as well as a RHS, with the RHS replacing the LHS once it has been matched. The difference being that the LHS of graph rewriting rules consists of a sub-graph that gets pattern-matched to the existing main graph G . Meanwhile the RHS consists of a replacing graph that, as the name implies, replaces said sub-graph in G . G in this case would also function as our *start symbol*.

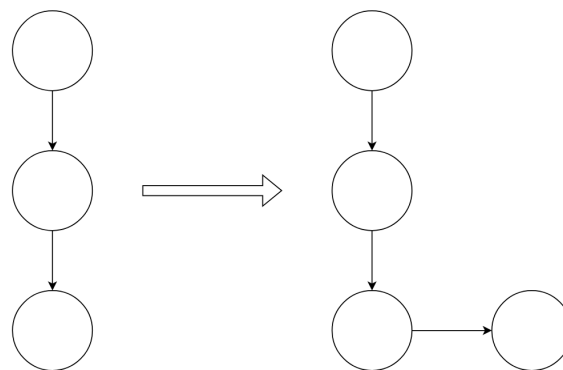


Figure 2.7: A basic example of a graph rewriting rule: The LHS sub-graph, consisting of one vertex connected to another one, which is connected to another one, would be replaced with the same graph with one more vertex and edge added.

In the case, that G may have multiple sub-graphs that match the LHS, still only one is chosen as seen in the latter example in [Figure 2.8](#). Which one of the matches is chosen to be replaced depends on the implementation. One could choose the first available match (which would in turn depend on the graph structure and the matching algorithm) or a random one. In either case, the rule is only applied once.

While the purpose of these rules in the context of formal grammars was to validate a string of characters, graph rewriting uses rules to purposefully transform a graph from one state to another. This way, a graph can be modified or changed without manually needing to do so by continuously applying transformation rules to the graph, all while adhering to the conditions set by the rules. By using graphs as a database for the designs, graph transformation can serve as an approach for creating designs algorithmically. Of course, accurate and appropriate rules first need to be defined for the approach to be successful. Yet, they can serve as a suitable way of adapting engineering knowledge into a computer-readable format, and are thus suited for a wide range of knowledge-intensive design tasks (HELMS, 2013).

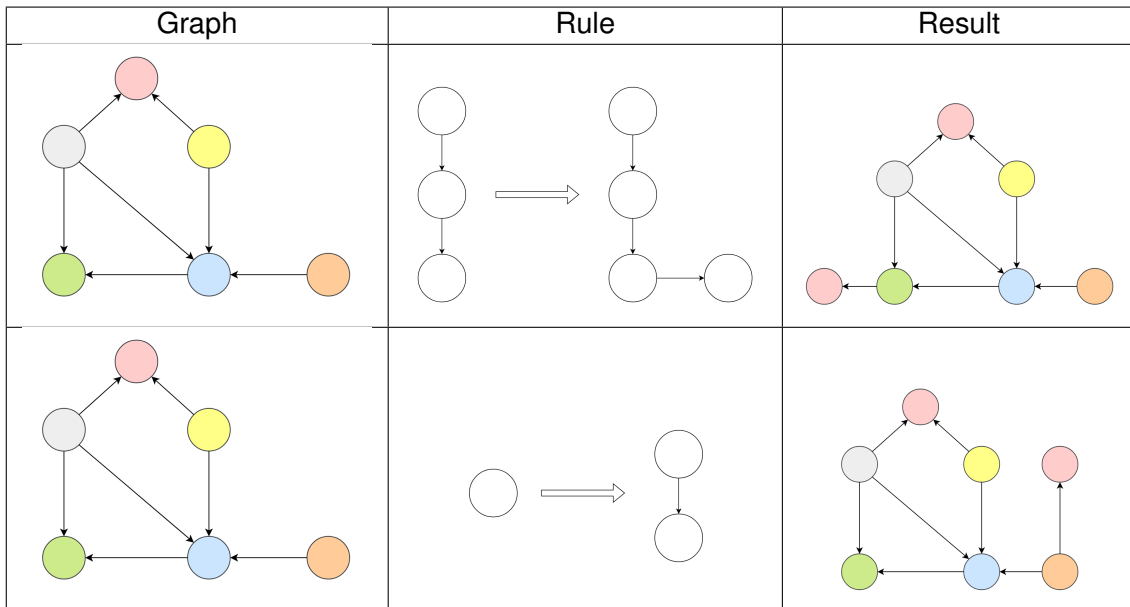


Figure 2.8: Two examples of graph rewriting rules being applied: in the upper example, a sub-graph consisting of three vertices connected to each other is matched, and said three vertices are extended by another. In the second example, any vertex matches the LHS pattern. Yet, since the rule is only applied once, the matching LHS is only replaced once.

2.1.6 Process Modeling

Process modeling, as the name suggests, describes a process through the means of a model. Such a model is either used to describe an existing process in the real world, i.e. a fabrication process, or serving as an assistance during the development of a process to formally illustrate the inner workings of it, similar to how geometric models can help visualize and understand certain aspects of designs. In both cases the model can help further optimize the process with regards to the elimination of bottlenecks, the usage of resources, and its viability. This works for both real-world processes such a fabrication and business processes, but also for software-internal processes ROLLAND, 1998. For our case, a process model will be highly useful to describe, optimize and control the flow of the program during runtime, especially in regards to rule application. There are many ways to illustrate process models. Including, but not limited to petri nets (ABEL, 2013), sequence diagrams (AMBLER, 2024) and business process model notation (ORGANIZATION, 2024).

2.1.7 Precast Structures

In contrast to in-situ concrete, where the concrete mixture is poured on-site, precast concrete is manufactured in advance off-site. Precast concrete is characterized by a superior control over resulting material quality as well as a lower overall cost of manufacturing (ALLEN & IANO, 2019). The process of designing and manufacturing precast modules that are only assembled on-site can be embedded in the broader concept of design for manufacturing and assembly (DfMA) (NGUYEN et al., 2024), which aims to design the product/building in such a way to ease the fabrication as well as assembly process (WEISHENG et al., 2021). KIM et al., 2016 have

found that precast modules are suitable for bridge construction in the UK while ANTONIOU and MARINELLI, 2020 propose standardizing the use of precast modules for high bridge building. The modular characteristics of precast concrete as well as their adaptability make it a promising material for the use of algorithmic design using graph rewriting systems.

2.2 Specific Related Engineering Science Papers

Since we have now established our core concepts that underpin our approach, let us explore related papers that have either already used graph rewriting in a similar fashion, or have researched algorithmic design in a tangential way.

2.2.1 Aggregations with Interlocking Parts

TESSMANN and ROSSI, 2019 developed a method for algorithmic design specifically for creating structures using modular units. Specifically, they use a concept called *topological interlocking* (see Figure 2.9) to aggregate parts so that simply by aligning them in a certain way, all degrees of freedom are constrained. This way, an aggregation of parts can also function for load-bearing purposes.

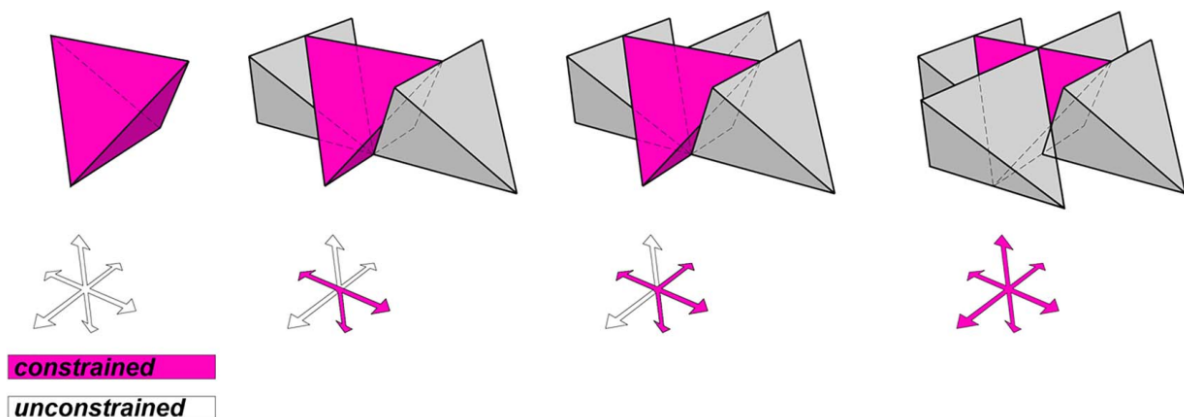


Figure 2.9: Topological interlocking assemblies: The main part (magenta) is constrained by its neighboring parts. Source: TESSMANN and ROSSI, 2019

TESSMANN and ROSSI, 2019 create these aggregation using a self-developed framework called *WASP*, which is a plugin for *Grasshopper for Rhino*. *WASP* uses interface-based rewriting rules to assemble parts into an aggregation using a topological connectivity graph. TESSMANN and ROSSI, 2019 call this *combinatorial design* since these basic parts are *combined* sequentially into a single discrete assembly. Objects are geometrically transformed so that the interface-planes face each other. This is possible by either using explicit sequence description, stochastic procedures, or gradient field-driven aggregations. The latter referring to using scalar fields as a frame for the aggregation (Figure 2.10), by continuously adding parts along the field depending on the scalar value of neighboring areas inside the field. With user-defined parts, a catalogue of rewriting rules and a scalar field this approach produces modular assemblies with reversible joints. TESSMANN and ROSSI, 2019 say that their approach challenges conventional parametric

design where the final form is defined a priori. In contrast, *WASP* follows a more sequential workflow where instructions are continuously executed until outputting the desired form.

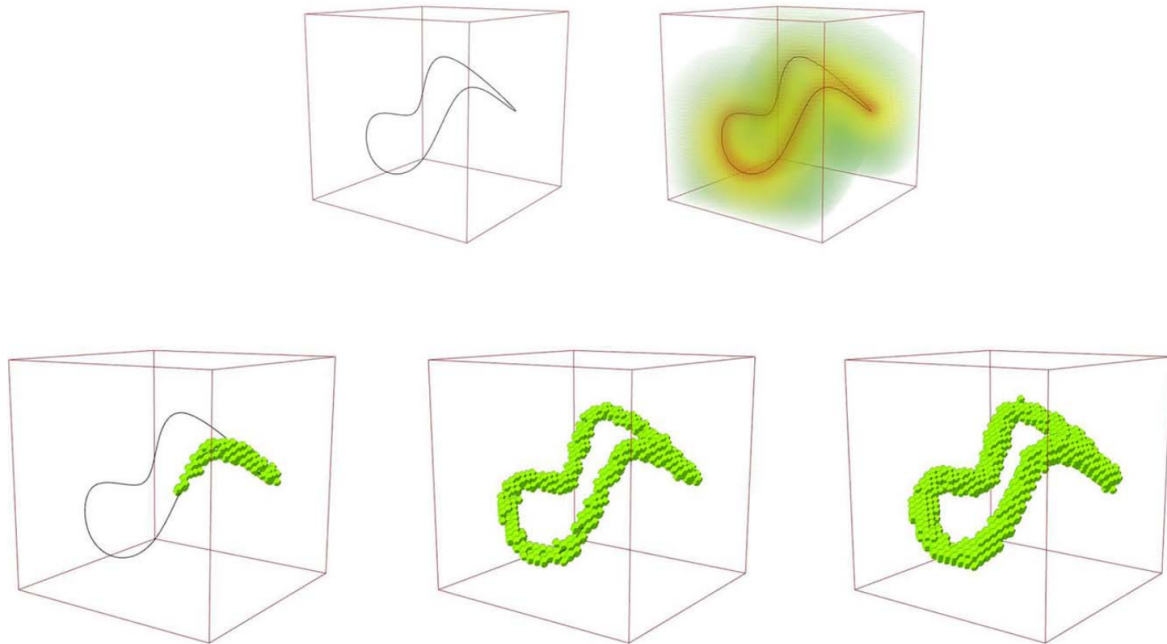


Figure 2.10: Aggregation of basic parts into an assembly using basic parts and a scalar field. The images show the progression of the assembly as more instructions are continuously executed along a scalar field. Source: TESSMANN and ROSSI, 2019

2.2.2 Semi-Automated Generation of Infrastructure Models

VILGERTSHOFER and BORRMANN, 2017 use graph rewriting rules to automate the planning process of infrastructure projects across multiple levels of detail (LoD). Different LoDs of a model are used during various stages and domains of the planning process. To expedite the propagation of changes in the model of one LoD to all other LoDs, VILGERTSHOFER and BORRMANN, 2017 employ graph rewriting rules using the *GrGen.NET* framework to achieve a method of consistently and accurately applying changes in one model to others. The challenges faced here are the fact that the dependencies necessary to correctly link each model to each other are "complex, time-consuming, and error-prone" (VILGERTSHOFER & BORRMANN, 2017) to define manually inside a more conventional parametric modeling environment (BORRMANN et al., 2014; VILGERTSHOFER & BORRMANN, 2017). Furthermore, graph systems allow the representation of engineering knowledge independently from any CAD-systems.

2.2.3 Graph-based mass customization of modular precast bridge systems

One of the most closely related research papers is by KOLBECK et al., 2023 which explores the viability of graph systems for modular bridge structures. Similar to this thesis, it focuses on adaptable precast modules as a basic unit for construction to produce scale effects and mass customization which can in turn be used to optimize the production and planning processes.

Instead of a graph rewriting approach, KOLBECK et al., 2023 only use graph transformation without the use of rewriting rules by directly converting a change in a certain parameter into a transformation of the graph, skipping the rule application part. This is done by translating the changes inside a steering sketch to the graph system. This resembles conventional parametric modeling more closely, albeit within a graph system context. KOLBECK et al., 2023 mention that in the future, an adaption of graph grammar for their approach would be feasible.

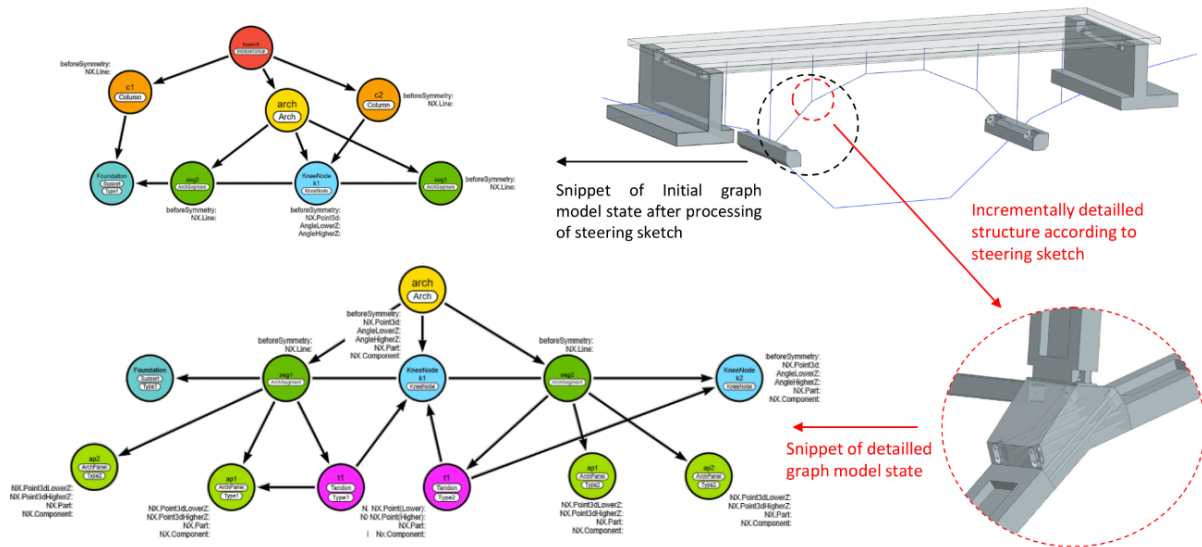


Figure 2.11: Transformation of the initial graph state to a state including more details. Source: KOLBECK et al., 2023.

2.2.4 Graph-Based Version Control

ESSER et al., 2022 propose the use of graph-based systems and graph rewriting for BIM version control. By representing BIM-data with a graph using the IFC schema, changes made to a model can also be expressed through changes made to a graph. By translating the contents of a .ifc-file into a graph structure before and after a change was made, an analysis is conducted to determine the differences between the two versions. This difference can then be packaged into a patch. This patch contains, among other things, a LHS and a RHS, as do graph rewriting rules.

This approach does not use graph rewriting as a means to design a building, but rather enables an asynchronous cooperative workflow. This becomes especially crucial for large scale buildings and design where multiple contractors are involved in the planning process.

2.2.5 Parametric Building Graph Capturing

ABUALDENIEN and BORRMANN, 2021 use a parametric building graph (PBG) to capture patterns within a BIM model that can then be matched to other projects or other parts of the same project. By capturing the objects, their relationships, and their context within a BIM authoring tool this information can then be used to create a rewriting/transformation rule. A different project can then be transformed into a graph representation. With both a graph representation and

a rewriting rule the pattern can be matched in the project and then brought back to the BIM authoring tool. This serves as a way to encapsulate and deploy architectural and engineering detail knowledge between projects. ABUALDENIEN and BORRMANN, 2021 find that being able to bring -successful- detailing changes from one model to another greatly influences the cost and performance of the building.

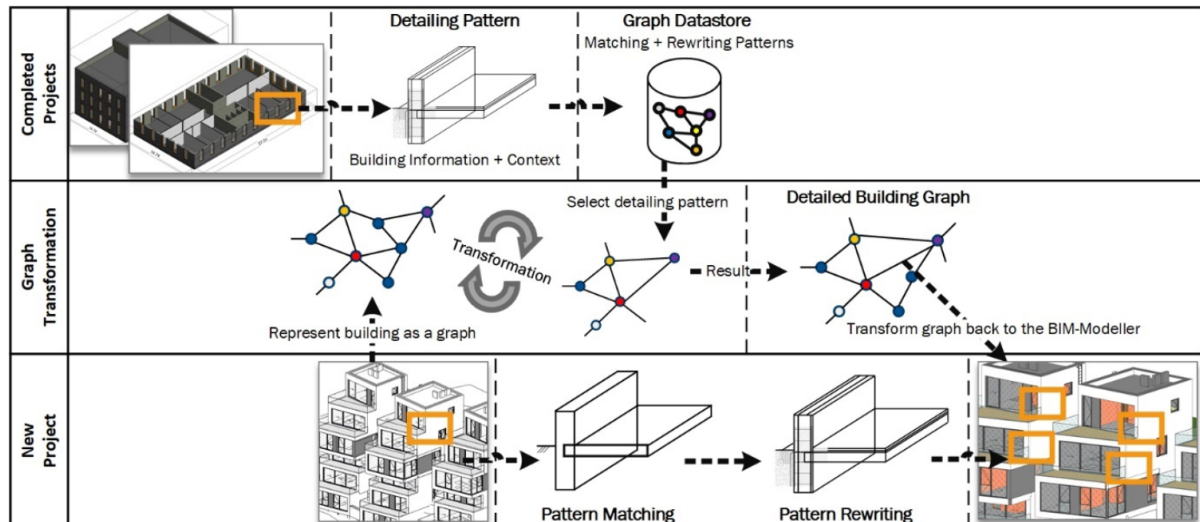


Figure 2.12: Transformation of the initial graph state to a state including more details. Source: ABUALDENIEN and BORRMANN, 2021.

2.2.6 More related works

KOLBECK et al., 2022 have shown that graph rewriting has had numerous appearances for automated and algorithmic design. Additionally, the works of HELMS et al., 2009, HELMS and SHEA, 2012 use graph systems to represent and synthesize design concepts in the field of mechanical engineering. MÜNZER et al., 2013, MÜNZER, 2016 and MUENZER and SHEA, 2017 also uses graph based representations to not only create but also automatically evaluate the resulting designs and filter valid designs accordingly. And STRUG et al., 2017, ŚLUSARCZYK et al., 2017 propose using graphs and graph rewriting as a method of capturing and representing engineering knowledge.

2.3 Research Gap

While the above mentioned research examples all employ graph rewriting and graph representations in differing yet useful ways, they share common ground as to why graph rewriting is used in a modeling and design context: the dimension of complexity of large construction projects can prove difficult to navigate and manage even with parametric modeling, while graph rewriting rules allow the formal definition and application of changes in a persistent and consistent manner. VILGERTSHOFER and BORRMANN, 2017 and ESSER et al., 2022 use graph systems to expedite the planning process by either proposing graph-based version control or change propagation, but TESSMANN and ROSSI, 2019 and KOLBECK et al., 2023 specifically use graph systems to

algorithmically create designs, as is the goal of this thesis. This approach stands in contrast to imperative approaches such as ABUALDENIEN et al., 2021 employ for their generative designs. Yet so far there has not been an attempt made at using graph rewriting and process modeling to design a full-scale model of a building from the ground up.

Chapter 3

Methodology

This chapter outlines the various design patterns and methods that in combination should lead to a functional approach. Specifically, we would like to discover a way to leverage graph rewriting rules in such a way that they can be used to create a 3D-model of a desired building type. These rules will be applied sequentially with the help of a process model framework that manages when and where these rules are applied in the main graph.

3.1 Parts and Components

We will be using parametric parts and components as our core building block for creating aggregations. Essentially, a part provides a blueprint for creating a component, similar to classes and objects in object oriented programming. Rule definitions use this abstraction to generally describe their LHS. Each part has a geometry which can be modified by changing parameters, as well as one or more interfaces which can connect to the interfaces of other parts. In the following, the basic part catalogue is outlined:

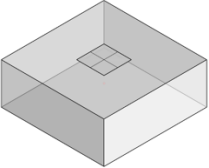
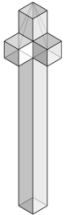
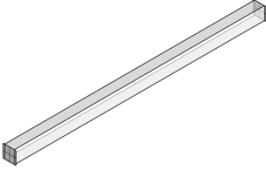
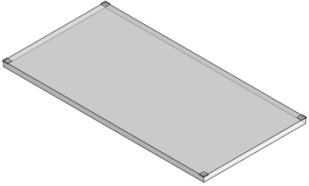
| | Foundation | Column | Beam | Deck |
|------------|--|--|--|---|
| Geometry |  |  |  |  |
| Parameters | <ul style="list-style-type: none"> • Side length • Thickness | <ul style="list-style-type: none"> • Height • Thickness | <ul style="list-style-type: none"> • Length • Thickness | <ul style="list-style-type: none"> • Side Lengths • Thickness |
| Interfaces | <ul style="list-style-type: none"> • Top Interface (Column) | <ul style="list-style-type: none"> • Bottom Interface (Foundation, Deck) • Console Interfaces (Beam) • Top Interface (Deck) | <ul style="list-style-type: none"> • End Interfaces (Column) | <ul style="list-style-type: none"> • Bottom Interfaces (Columns) • Top Interfaces (Columns) |

Figure 3.1: Parts used as basic building blocks including their geometry, parameters and interfaces. The parameters can modify the geometry while the interfaces indicate surfaces other parts can connect to.

Based on these parts, components can be created that are given the necessary parameters for the geometry and interfaces to be initialized and be added to the main graph.

3.2 Rule Engine

The foundation of this approach consists of the use of graph rewriting rules applied inside a process model to create parametric 3D models. Thus, having a solid rule engine which suits the needs of the approach is crucial. There are many graph system frameworks already developed which not only support graph systems but also graph rewriting, for example *GrGen.NET* GRGEN, 2024. Existing graph libraries can offer powerful and robust solutions when it comes to rewriting functionality. Additionally, not needing to implement the rule engine by oneself can be a major advantage leading to time savings. But, for this approach, the decision was made to implement a custom-made rule engine suited to the requirements of the approach. Two reasons were the leading factors for this decision: Firstly, to reduce overhead by only implementing the necessary components of the rule engine. Secondly, to facilitate the implementation of the necessary custom functionality enabling the approach. In the following, key details of the rule engine are listed.

3.2.1 Main Graph

The main graph is the central data structure containing all necessary information to represent the resulting model. How graphs are structured was illustrated in [chapter 2](#). For our casey the same rules apply with the additional requirements that the main graph be able to be easily converted into a visualizable format such as `.3dm`.

3.2.2 Rule Definitions

As described in the previous chapter, each rewriting rule has a *left-hand-side (LHS)* which specifies the sub-graph inside the main graph that is to be replaced by the *right-hand-side (RHS)*. In general, the LHS can be as complex as one wants it to be. While the ability to define such complex LHSs can be incredibly useful at times, being able to keep the rule definitions as simple as possible proves very valuable later on when wanting to apply them. To formally describe what the rule definitions look like, each rule specifies a single interface of a specific part (not a component) as the LHS, and another single interface of a specific part as the RHS. The interface of the RHS will then be connected to the interface of the LHS, as seen in some examples in [Figure 3.2](#).

Each and every rule defined must follow the schema below. This gives us the advantage of keeping the code which handles the rule application simple and manageable. Large LHSs are more difficult to handle when applying rules to a graph, thus, a simple definition is key. Of course this inhibits the ability to make large modifications to the graph (and the model) in one go. But for this approach, a rule confining to the schema in [Figure 3.2](#) is perfectly adequate since we are aiming to procedurally add to the model in small, incremental steps instead of making large modifications all at once. An excellent example for this is a column being placed on top of a foundation (see [Figure 3.3](#)). Other kinds of modifications to the model that need to be made mostly follow the same principle and can be expressed using straightforward rules.

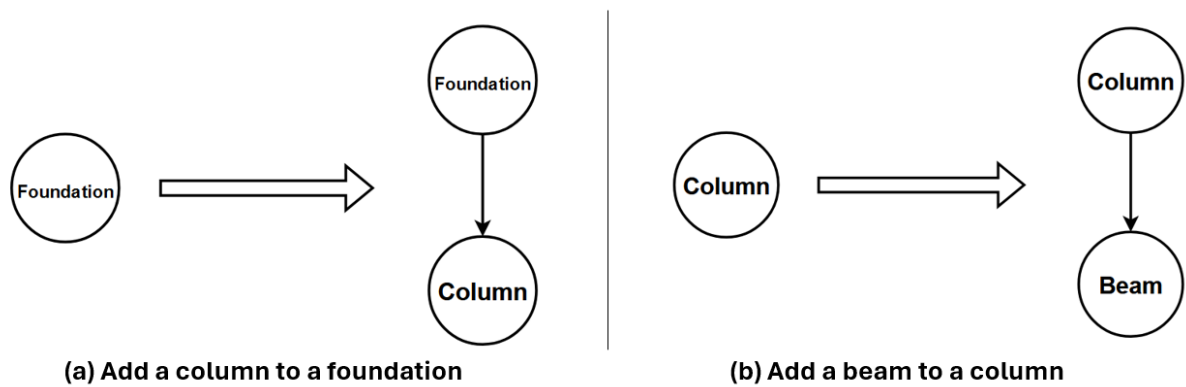


Figure 3.2: A set of two rules specifying how columns and beams are added to the graph.

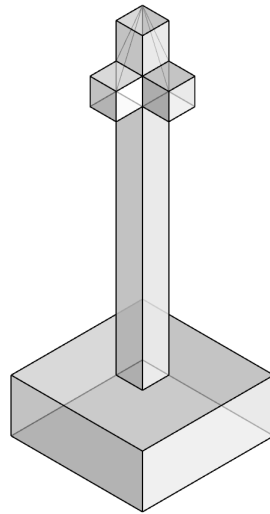


Figure 3.3: A column standing on a foundation.

3.2.3 Rule Application

We have now established our rule definitions and the main graph to which they should be applied to. But for the application to happen a software component is necessary that handles the logic of graph rewriting. This includes the following:

1. Match the LHS of the rule to a sub-graph in the assembly. Since the LHS definitions will be extremely simple, this is an easy feat.
2. Create the component included in the RHS. Same conditions as in step 1 apply.
3. Transform the component so that the two interfaces are geometrically identical.
4. Check whether other connections in the model have been closed after the transformation.

3.2.4 Topological and Geometrical Conditions

As described in [chapter 2](#), each rewriting rule has a *left-hand-side (LHS)* which specifies the sub-graph inside the main graph that is to be replaced by the *right-hand-side (RHS)*. At the most

basic level, the LHS is a pure topological construct meaning that the only relevant things are the nodes themselves as well as the relationships between them. As long as the 'condition' that the LHS exists within the main graph is fulfilled, the rule can be applied. Yet for this approach, there are more aspects to be considered apart from the topology. Let us examine the following situation:

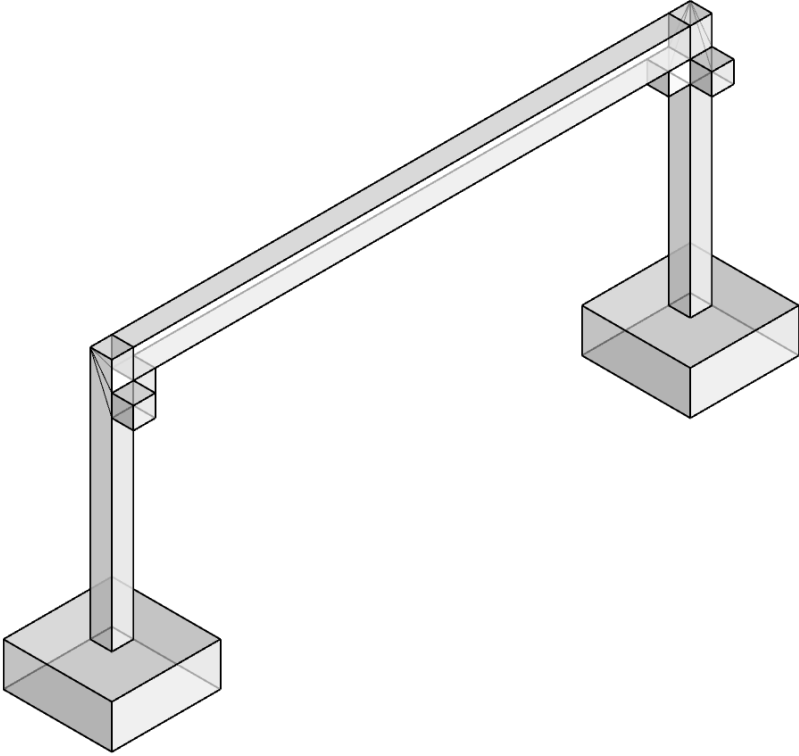


Figure 3.4: A simple aggregation consisting of two foundations, two columns and a beam.

In this example, two columns are placed on one foundation each, with a beam connecting the two columns. Assuming that the two foundations are our start symbol, one may be inclined to think that this aggregation can be created by defining and applying some simple rules, as seen in Figure 3.2 and Figure 3.5. By doing this, we arrive at a resulting graph that represents model as seen in Figure 3.4.

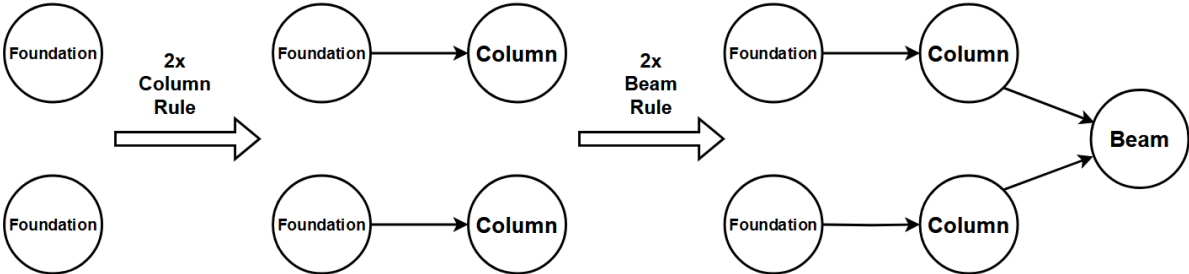


Figure 3.5: Assembling the model from Figure 3.4 by using two foundations as our start symbol, then applying the rules defined in Figure 3.2. The column rule twice followed by the beam rule once.

The rules applied here specify that certain components and their respective connections are added. But what the rules do not explicitly specify are the geometrical conditions that need to be fulfilled. For example, the bottom surface of the columns as well as the top surface of the foundations need to be correctly aligned for there to be a connection. Of course this can be easily achieved by simply transforming the newly added component immediately so that the two surfaces (or interfaces) align correctly. Since our rule definitions follow the above mentioned schema, we must simply move every added component after they have been added to the model/graph. This corresponds to step 2 in [subsection 3.2.3](#). The same logic is applied inside the *WASP* framework for *Grasshopper* (TESSMANN & ROSSI, 2019). This works well for our column-foundation example, but we encounter a problem when trying the same strategy for the beam. Visible in [Figure 3.5](#), the beam is not connected topologically to both columns, even though it is actually supported by both (in the model). Our rule definition specifies that the beam will only be connected to one column. But what about the other? This is where an algorithm triggers which examines whether any other interfaces have been connected/closed after a new component has been added and transformed. Keeping the example of the beam in mind, this means that initially it is only connected to one console after it has been created and transformed. Once that has happened, an algorithm that checks whether the other interface of the beam can be connected is invoked:

Connection Check

```

1 // loop through all open interfaces in the main graph
2 foreach (OpenInterface i in mainGraph)
3 {
4     // loop through all open interfaces in the new component
5     foreach (OpenInterface j in newComponent)
6     {
7         // loop through all available rules in the catalogue
8         foreach (RuleDefinition rule in ruleCatalogue)
9         {
10            // if the open interface in the graph equals the lhs
11            // and the open interface of the component equals the rhs of
12            // the rule
13            // and the geometry is identical
14            if (i==rule.lhs and j==rule.rhs and i.geometry==j.geometry)
15            {
16                // close the connections
17                i.connectedInterface = j
18                j.connectedInterface = i
19            }
20        }
21    }

```

The algorithm loops through all open interfaces of the assembly, then loops through all available rules inside the catalogue of rules, proceeds with checking if the open interfaces of the new component have a rule match with any of the open interfaces of the assembly, and then checks

whether the geometry of both interfaces are identical. If all conditions are met, the semantic information that there is a connection between these two interfaces is established. This algorithm is always part of the rewriting process. Whenever a rule is applied, this algorithm will be executed.

The algorithm mentioned above of course only established the connection should the geometries of the interfaces be identical. Depending on the circumstances this may not be the case, i.e. if the beam has the wrong length. If the beam is too short, it cannot be supported by two columns and only the original connection will be made. This means that we need to ensure that these geometrical conditions are fulfilled before the rule is applied.

3.3 Process Model

While the rule engine handles the application of a rule, the process model handles what rules are applied and when they are applied. When looking at some examples of what *WASP* can achieve when applying certain rules over and over (see [Figure 3.6](#)), the resulting aggregations are far from what one would call familiar. While *WASP* provides powerful tools to quickly and reversibly create such aggregations, this approach needs a way to ensure that certain rules are applied at a certain stage in the process. This is the purpose of the process model.

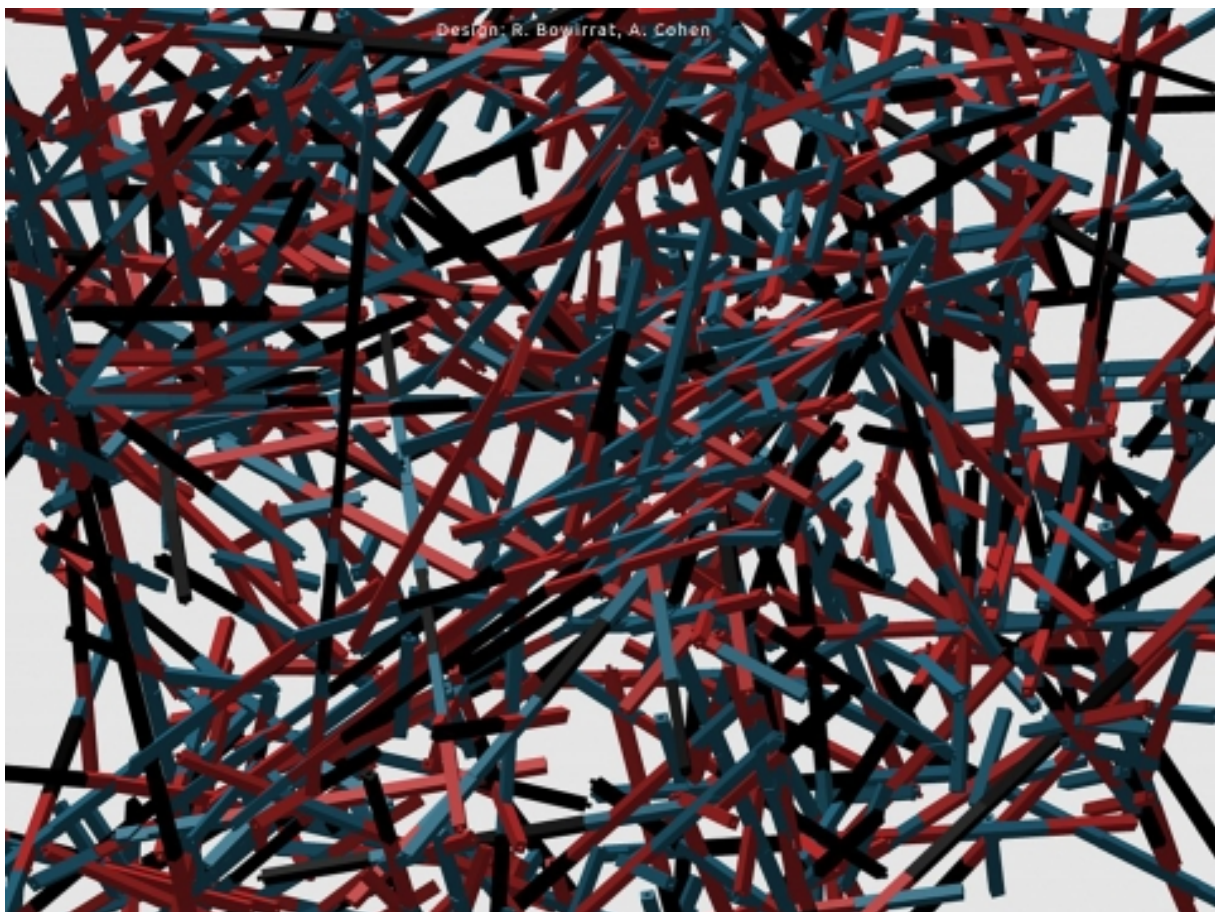


Figure 3.6: Example aggregation of parts using the WASP plugin for Rhino. Source: ROSSI, 2024

3.4 States

State definitions represent the various states that a process model can be in at a given point during the runtime. Each state has an assembly that it is tied to, and can also possibly have a subsequent state. In general, three types of states can be distinguished:

- `StartState`: This state contains all necessary information and methods to create a valid start symbol which serves as the base for the rewriting process.
- `PlanningState`: During this state rules that are to be applied and parameters that are to be set are defined and prepared.
- `AssemblingState`: Finally, during assembly all previously defined rules and parameters are applied and the assembly is modified.

The process model encapsulates instances of these states to compute them sequentially. The process model always has exactly one `StartState`. After the start symbol has been created, a series of `PlanningState-AssemblingState` pairs follow. In a planning state, not only are there rules that are defined, but also parameters of the parts that need to be changed. With rules and parameters defined, these instructions are executed upon in the following `AssemblingState`, where the rule engine will be invoked.

The start- and planning states here are the most crucial parts. A correct start symbol is the metaphorical (and literal) foundation for the graph rewriting system to actually work, while the planning states define what changes are supposed to happen to the model. If we recall the geometrical conditions explained in the previous sections, this is the step where the parameters must be defined in such a way so that these conditions are fulfilled. Going back to our foundation-column-beam example, we can set the `length` parameter of the beam to be equal to the distance between the columns (subtracting the thickness of the columns first, of course). Once this is defined correctly, the connection checking algorithm can correctly establish the appropriate connections.

3.5 Execution

At this point the following parts of the approach have been established:

- Part definitions offer the basic building blocks including geometry and connection interfaces.
- Rule definition give us a simple method of defining rewriting rules.
- The rule engine can apply such rules transform a graph.
- The process model provides the capability of start symbol definition, defining when and where rules should be applied, as well as parameter input.

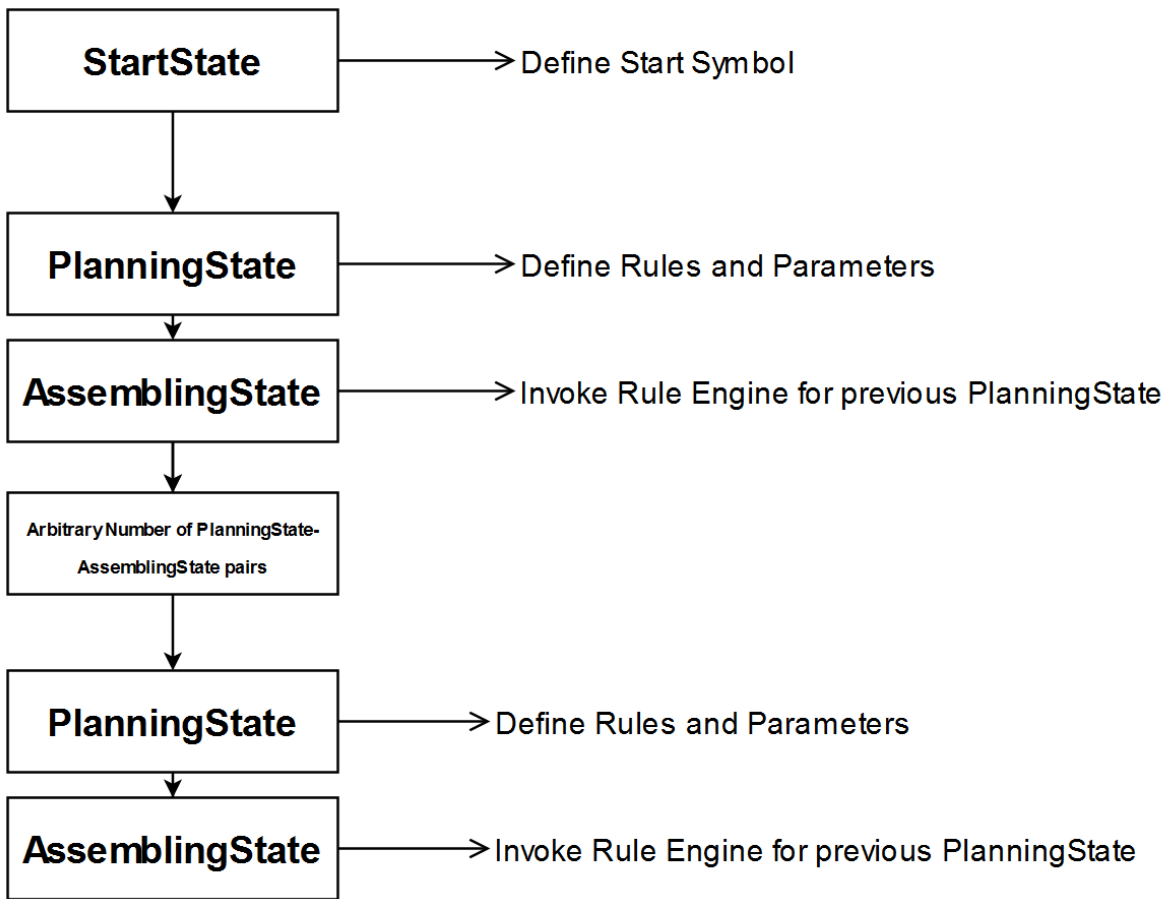


Figure 3.7: The sequence of different states inside the process model. After the StartState, an arbitrary number of PlanningState-AssemblingState pairs can follow.

What remains to be done is simply to create a graph grammar, or a catalogue of rewriting rules in order to create the structure of the respective case study. Within this case study, the user should be able to define their own set of rules, part definitions, and project parameters such as story height or field length. Furthermore, the rule sequence and part parameters must be defined for the process model to work. [Figure 3.8](#) illustrates this process.

Defining the graph grammar is use case dependant and must be carefully configured in advance. Since our rule definitions are as simple as they can be, this usually comes down to making sure that the rule catalogue covers any single connection that is valid and possible within the case study. For example, a column can be placed on top of a foundation or deck.

Once the grammar is defined, a process model is defined that contains the start symbol definition, part parameters and the respective rule sequences. The process model can then be executed upon.

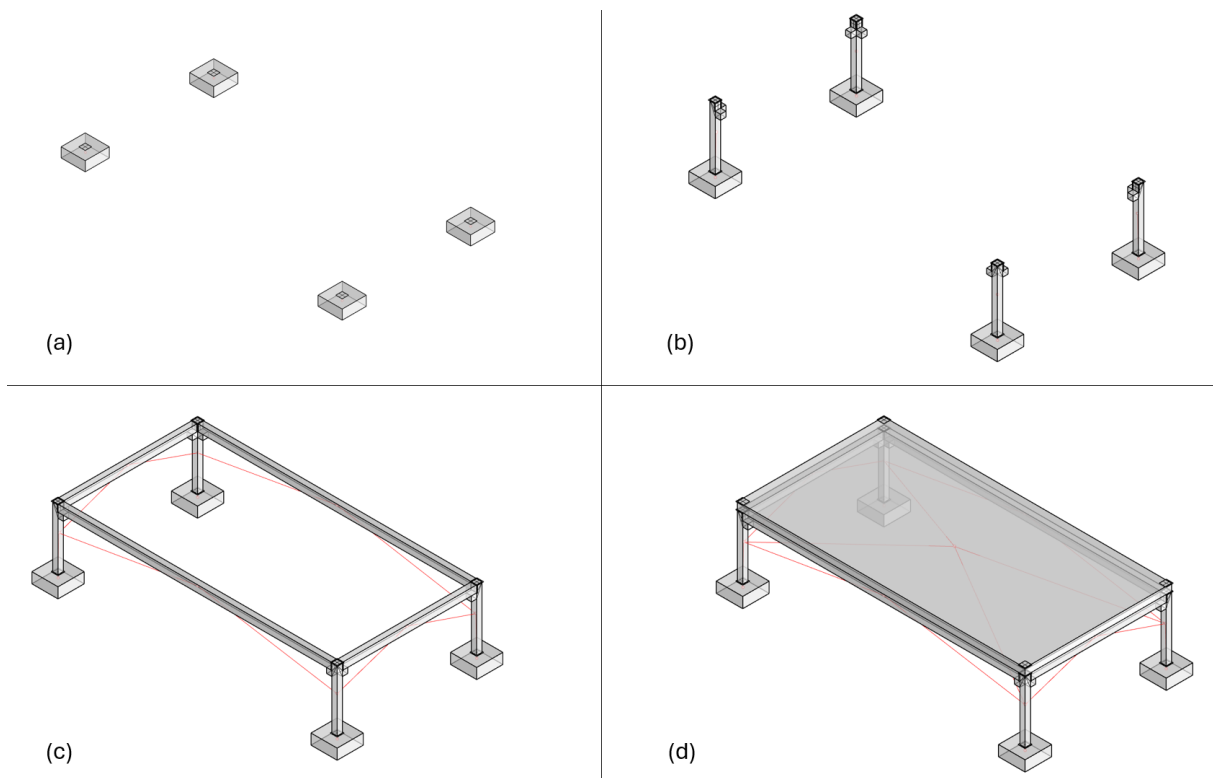


Figure 3.8: An example of how the process model controls the algorithm through its states. Each subfigure represents the result after applying the rules defined in either the start state (a), column planning state (b), beam planning state (c), or deck planning state (d).

Chapter 4

Implementation

This chapter details the structure, inner workings as well as the capabilities of the implementation which was used to explore the viability of the approach detailed in the last chapter. It is built upon the previous work of Lothar Kolbeck, who laid most of the ground work regarding the rule engine and the process model. Based on his work, further functionality was added in incremental steps.

4.1 Research Design

The objective of the implementation is primarily answering research question in regards to the viability of a graph rewriting approach for algorithmic design in construction and to explore the creation and viability of a framework that can facilitate graph rewriting for parametric modeling. This framework should be able to control and promote a geometrically sound and correct placement of objects within the model, as well as proper semantic information. In order to develop this 'rule application framework' we can utilize design science.

Design science in contrast to natural sciences aims to achieve a desired end result (in our case, a formal approach) by first developing one iteration of the design, followed by an evaluation based on its performance. By looping through this cycle of development and evaluation, we achieve incrementally better results each time, getting closer to the final desired result (PEFFERS et al., 2007).

To properly evaluate the different iterations of the implementation, we can define a number of case studies. The specifics of each case can be used to assess the overall advancement of the program. If the iteration can deliver the desired result, it can be considered successful. If the result was unsuccessful, it can give us more insight into the feasibility of the approach. In the following section, the individual case studies will be specified in ascending order of complexity. By starting out with case studies of lower complexity, problems of a higher magnitude can be dealt with in later case studies, when the essential work has already been done.

4.1.1 Case Studies

Case Study 1: One Field Skeleton With Multiple Stories

Adding on top of the previous case, we would like the skeleton to be able to scale to multiple stories. Meaning, a parameter can be defined in advance specifying the number of stories the resulting model should have. This case study is somewhat more complex than the previous one, mainly concerning parametrization aspects instead of graph rewriting.

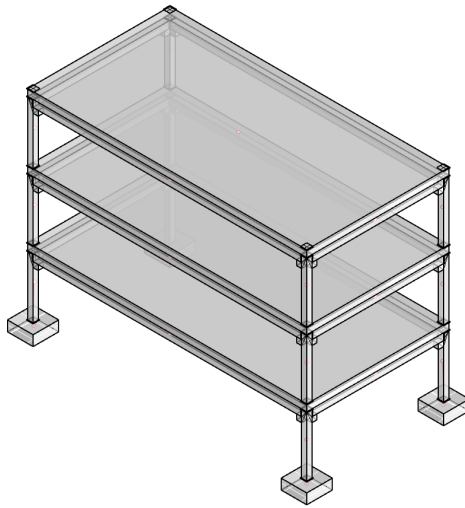


Figure 4.1: Desired result for the first case study.

Case Study 2: Two Field Skeleton With One Story

Ignoring the multiple stories of case study 1.1 for now, we will attempt to define the number of fields that will be used when creating the model. While previously we have placed foundation on each corner of the plot, we must now potentially place more foundations in between the corners, for example to span larger distances. This adds more complexity especially in regards to the alteration of the start symbol, which now must be parametrically changed depending on even more parameters. Once the start symbol is set and correctly configured, standard procedures can apply.

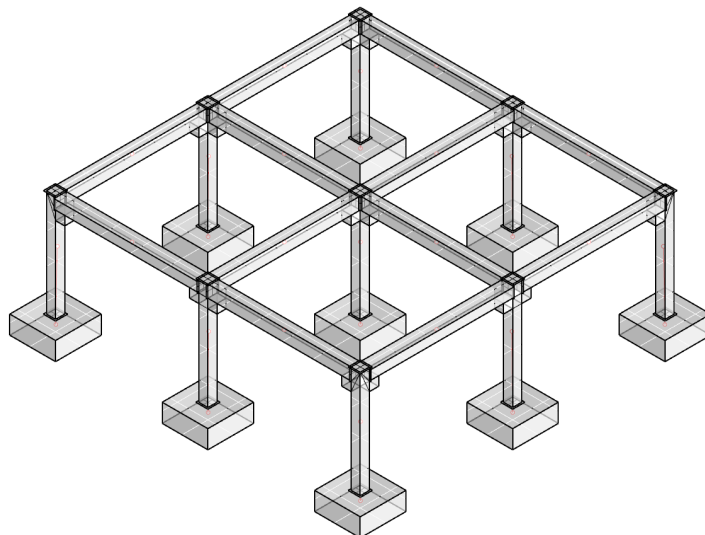


Figure 4.2: Desired result for the second case study.

Case Study 3: Multi-Field, Multi-Story Skeleton

Combining the "features" of the two previous studies, this case study attempts to determine the scalability of the approach.

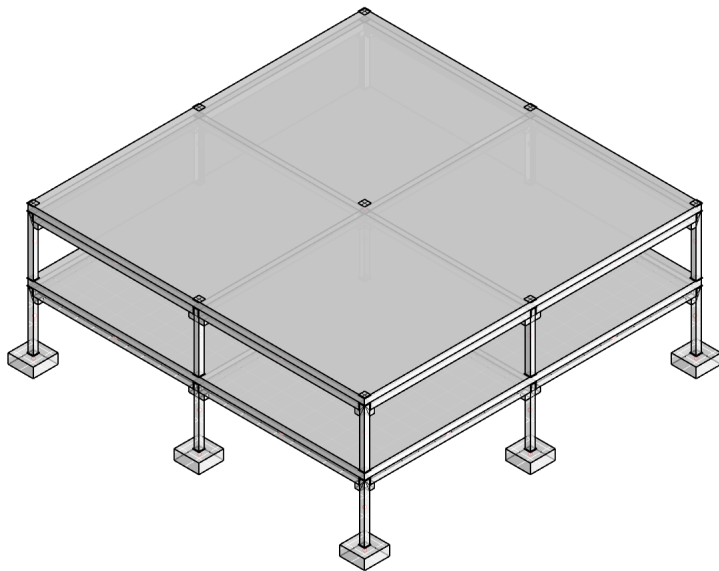


Figure 4.3: Desired result for the third case study.

To begin, let us detail the software components that make up the implementation. We can subdivide the components into external and internal ones:

- External Software/Libraries:
 - Rhino: Parametric 3D modeling tool, used for visualization.
 - Grasshopper: Algorithmic modeling & visual programming for Rhino.
 - Rhino Compute: Headless geometric computation for Rhino.
- Internal:
 - Grammar Meta Model: Contains the rule engine as well as all definitions necessary for running it.
 - Algorithm Meta Model: Contains the process model which encapsulates the algorithmic design process.
 - Case Study: Respective entry point where the process model is set up.

4.2 External

4.2.1 Rhino 3D

Rhino 3D is a 3D-CAD Software that enables geometric and parametric modeling as well as visualization. This piece of software serves as the basis for our implementation, providing the necessary framework for developing and testing the program. While Rhino has extensive capabilities regarding 3D-modeling in the program itself with the help of a GUI (what it is usually used for), we will disregard this functionality and solely use it as a platform to compute and visualize our resulting model by leveraging its compute API.

4.2.2 Rhino Compute

Rhino Compute is a headless geometrical computation library which uses the server-client pattern. Running in the background in the development environment, REST calls can be issued to create and manipulate common geometries such as points, planes, surfaces, meshes, and more. This is what the implementation will use to interface .NET with Rhino. This way, we can develop all graph transformation and process modeling in a separate piece of software, while 'outsourcing' geometric computation by making use of Rhinos extensive and solid geometric kernel. Rhino and its computation library were chosen for their easy integration into the overall software as well as their performance. But, in theory, any geometry library compatible with .NET could be used instead. One such example would be the Revit API. Since the geometric computation only serves as a means to modify and manipulate geometries but is only an external component, it will not be detailed any further.

4.2.3 Grasshopper

Grasshopper is a plugin for Rhino adding visual programming functionality. Visual programming languages provide a way of creating programmable scripts with the help of a GUI. Core building block of a Grasshopper script as a visual programming language is the so-called 'component': A small sub-script which usually takes one or more inputs, processes these inputs, and then returns one or more outputs. These outputs can then serve as an input for other components. Grasshopper offers many built-in components including, but not limited to, several types of input components, basic and advanced math capabilities, and geometry creation and manipulation. To give a brief example, consider this grasshopper script in [Figure 4.4](#) which creates a simple cube:

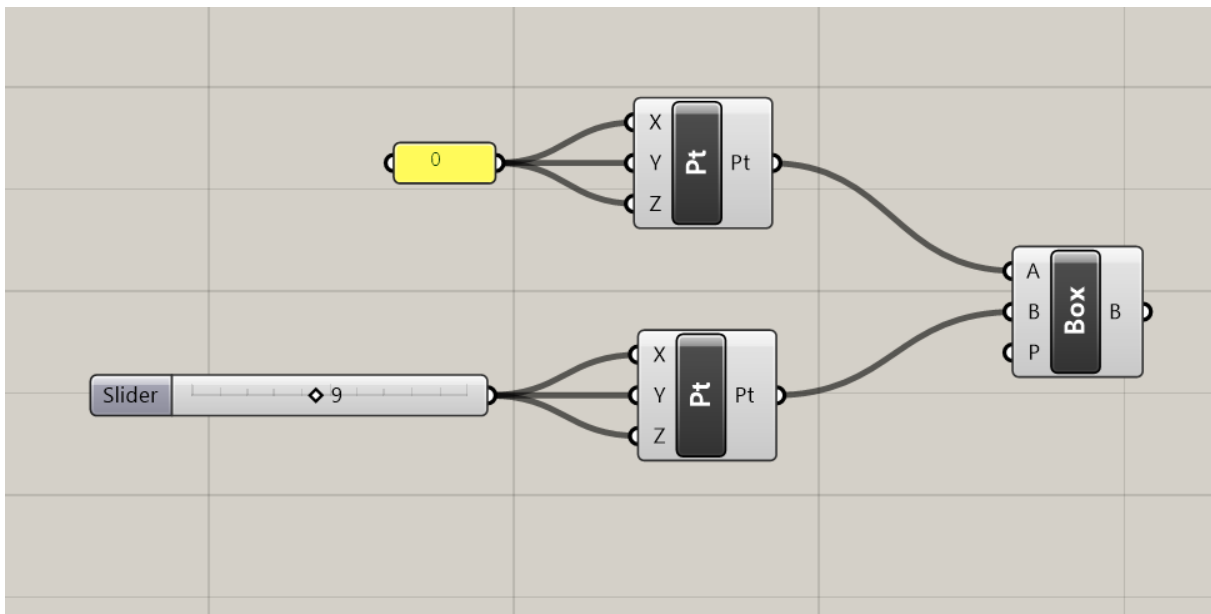


Figure 4.4: A simple Grasshopper script which creates a cube.

We define a number '0' with the help of a `panel`-component and route the number to each input of a `Construct Point` component. In parallel, we define another number with the help of a `Number Slider` component, and route that integer to another point construction. Feeding these two points to a `Box-2-point` component, the geometry of a cuboid with each edge length being the same (also known as a cube) is created and then visualized in Rhino as seen in [Figure 4.5](#)

By changing the value of the number slider, we can change the edge length and thus the volume of the cube. This way, Grasshopper provides a programmable and scriptable method of parametrization of 3D-geometry. Furthermore, grasshopper script files can be read and modified by the Rhino compute library mentioned earlier. By creating the geometries and interface surfaces with the help of grasshopper scripts, we can quickly achieve the necessary building blocks for our algorithmic design. By also making use of the rhino compute API, we can leverage the inherent parametrization of grasshopper scripts by changing the input parameters inside our algorithm. This implementation will be using grasshopper scripts to model the core building blocks including including their parametrization, to achieve a simple way of dynamic adaption. In

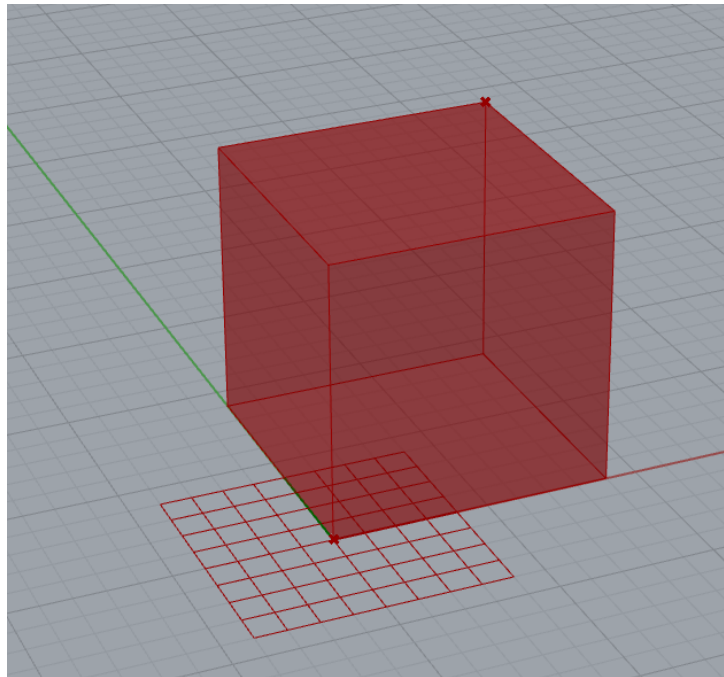


Figure 4.5: A cube in Rhino created with grasshopper.

summary, Rhino, Rhino Compute as well as Grasshopper offer us a great basis for creating our resulting model algorithmically.

4.3 Internal

We are using the Rhino compute API to interface a development platform such as .NET with the Rhino geometric kernel. This lets us write our own rule engine and process model for our algorithmic design with the help of standard programming tools such as .NET. In this section, we will detail the different internal components within this program, which make up the main part of enabling the graph transformation approach.

4.3.1 Grammar Meta Model

The grammar meta model contains all necessary logic and functionality for creating a graph representation of the model and modifying it through the means of graph rewriting. It is important to note that as is this a meta model, meaning that there are no use case specific details. Instead it contains the core rule engine with which we can apply transformation rules to a given graph as well as anything that is necessary to apply them. The 'meta' in this case meaning that it is not system-specific but instead should be generally applicable to a wide variety of use cases.

Assembly

This is the single central component which provides the information necessary to represent the model as a graph. It contains a list of components which were aggregated into the assembly

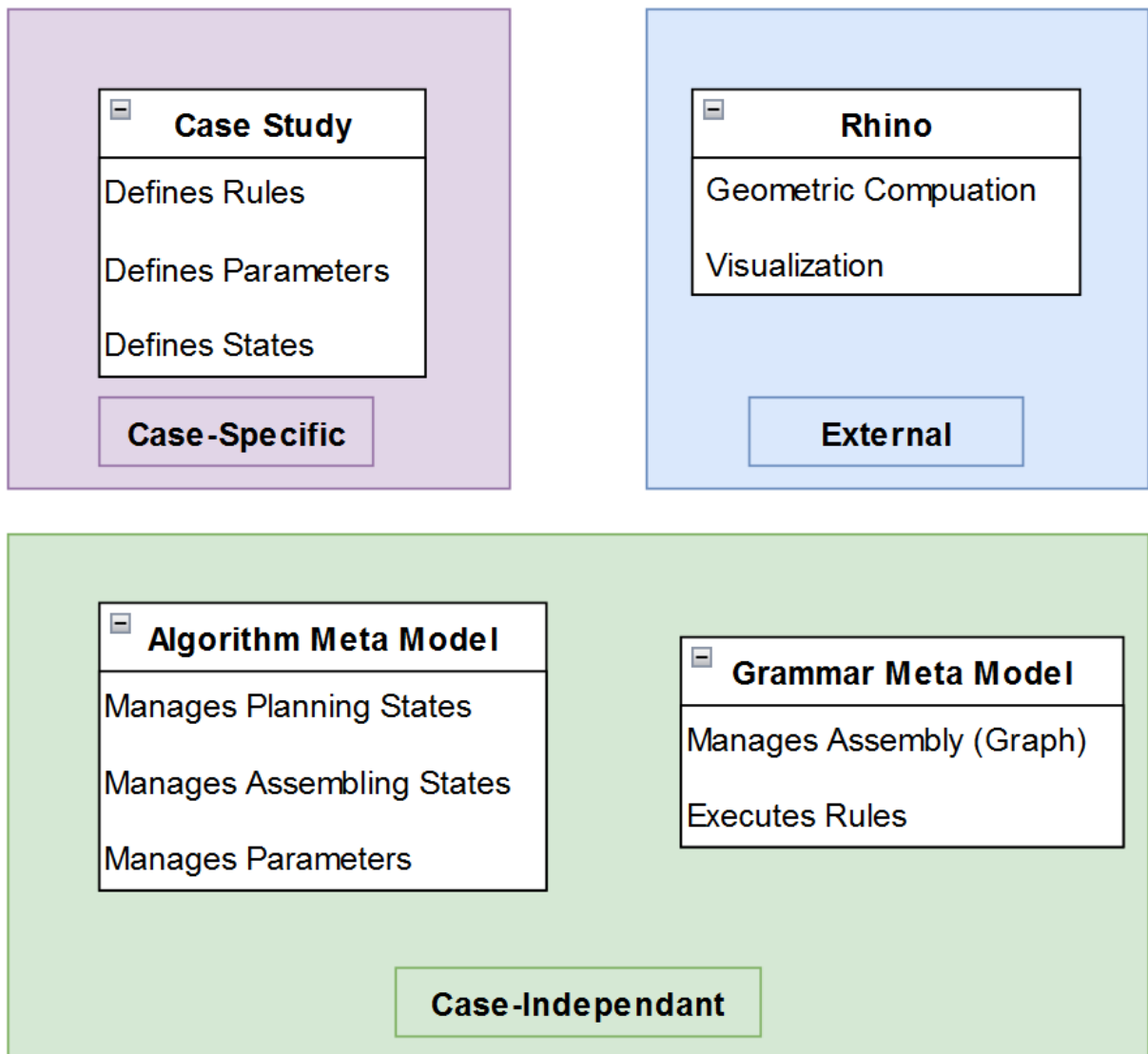


Figure 4.6: General overview of the projects software architecture.

as well as two methods: one which returns all open interfaces in the assembly (meaning all interfaces that are not connected to another one) and a method to serialize it to a .3dm file using Rhino compute, readable by Rhino 7. It is important to consider here that the assembly only contains a 'flat' list of components, but no direct information about their relationship to each other. That information is stored in the components themselves. It is also worth noting that from here on, any expression mentioning the assembly or modifications to it actually refers to changes made to a graph.

Parts and Components

A `Part` object contains general information as in a name, but also details the amount and type of connections the part has as well as its available input parameters. Also, a file-path to its corresponding grasshopper script is saved here. The path is necessary to invoke two methods which populate our parameters and connections by fetching the information from the grasshopper script. When a `Component` object is created on the basis of a `Part`, it can

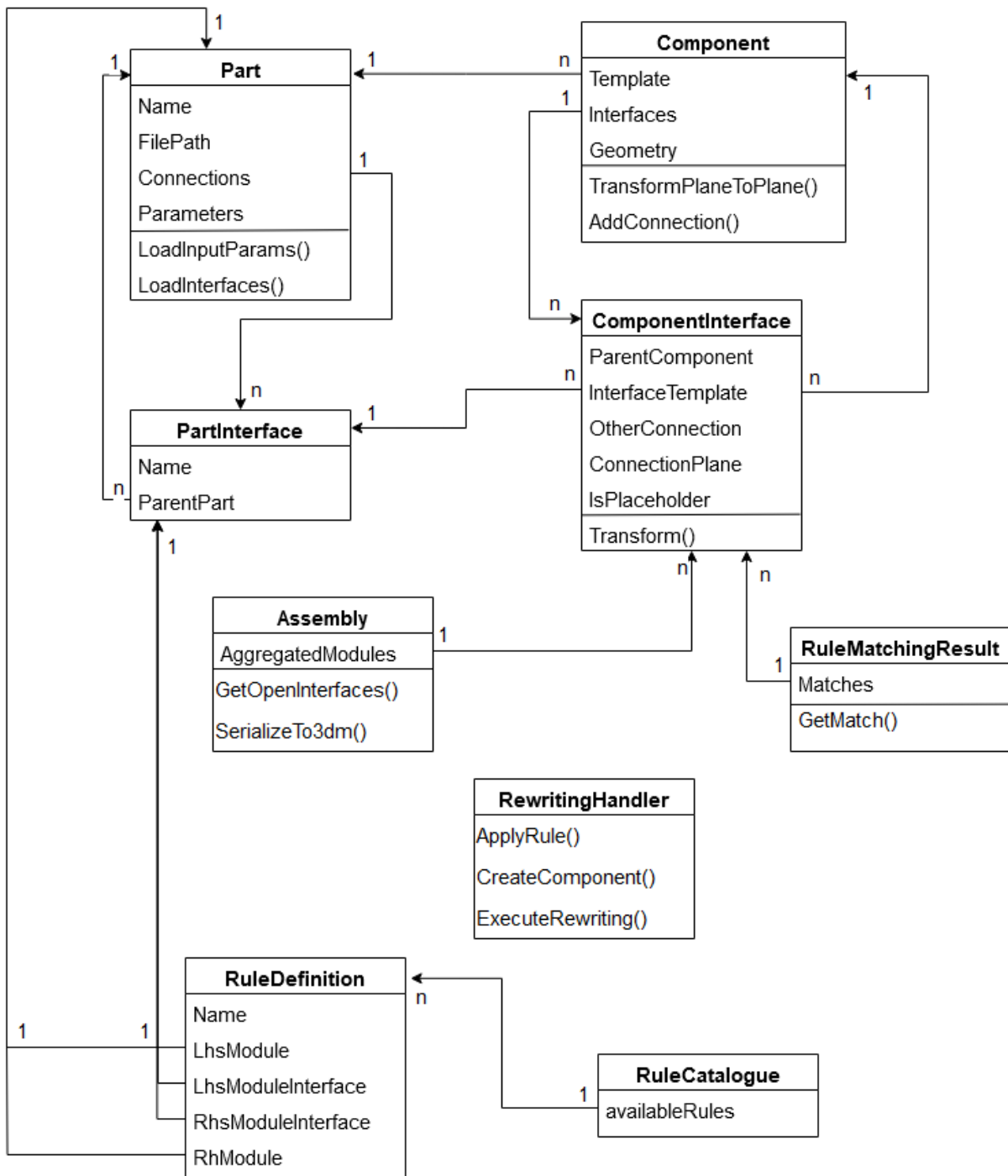


Figure 4.7: The architecture of the grammar meta model.

also obtain the geometry from the grasshopper file. Since our grasshopper scripts include parameters, components created from the same part may differ in their geometry or other aspects. Components also include methods for adding connections or modifying their geometry. Similarly, the connections of Parts and Components are of a Part- and ComponentInterface type, with, again, the ComponentInterface being created from the PartInterface and only the former containing the geometry. Important here is, that the ComponentInterface also has a property called `OtherConnection`. This can be any other ComponentInterface and determines whether an interface is connected to another one. Remember that information about the

connections between components was not stored directly in the assembly. Instead, they are stored here. The `ComponentInterfaces` are designed in such a way that one interface can either be connected to a placeholder, meaning that it is not connected anywhere, or to one (and only one) other `ComponentInterface`.

Rule Definitions

The above mentioned part-component abstraction lets us easily create components in our assembly based on a given 'blueprint', but also lets us elegantly define rule definitions. As described in the previous chapters, transformation rules need to be given a left-hand- and well as a right-hand-side. This means that we can define our rules by referencing certain parts and their interfaces which should be matched in the assembly (LHS), and then referencing a part and their interfaces which should be added to the assembly (RHS). Defining rules this way allows for a quick creation of them which is convenient when setting up new case studies. One drawback of this implementation however is that it only allows for matching to solely one type of component, and subsequently only adding one component. Should a certain interface be able to interface with more than one kind of component, we need to define multiple rules. The decision to keep these rules as simple as possible stems from also wanting to keep the rule engine as a whole simple and robust. Note that rules themselves are not defined here, only the class with which they are created. Rule definitions can be aggregated inside a rule catalogue, which we can hand over to other parts of the software to have all defined rules within a case study readily available.

Rewriting Handler

This class is where rules are applied to the graph, or, in other words, modifications are made to the assembly. Here, all formerly described concepts are used to do the following:

- Retrieve a matching open interface via a `RuleMatchingResult`.
- Create a component and transform it so that both interfaces align.
- 'Close' both interfaces by setting the `OtherConnection` of each interface to the other one.
- Check if other connections can be closed after transformation.

Worth noting is the `RuleMatchingResult`, which takes a single rule definition and the assembly as an input and can return an open interface matching with the rules LHS in the assembly with the help of the method `GetMatch`. Once a rule match is found, the new component is added and transformed as it is done in Andrea Rossi's WASP plugin for grasshopper. Finally, the last part corresponds to the connection-check mentioned in the last chapter.

4.3.2 Algorithm Meta Model

The algorithm meta model contains class definitions concerning the process model. As already mentioned, the process model exists to control the flow of the algorithm during runtime, meaning that while anything concerning graph rewriting is stored in the grammar meta model, the actual rewriting process is invoked from within here. Again, this part of the program is also on the meta-level, meaning that it is not system-specific but generally applicable instead.

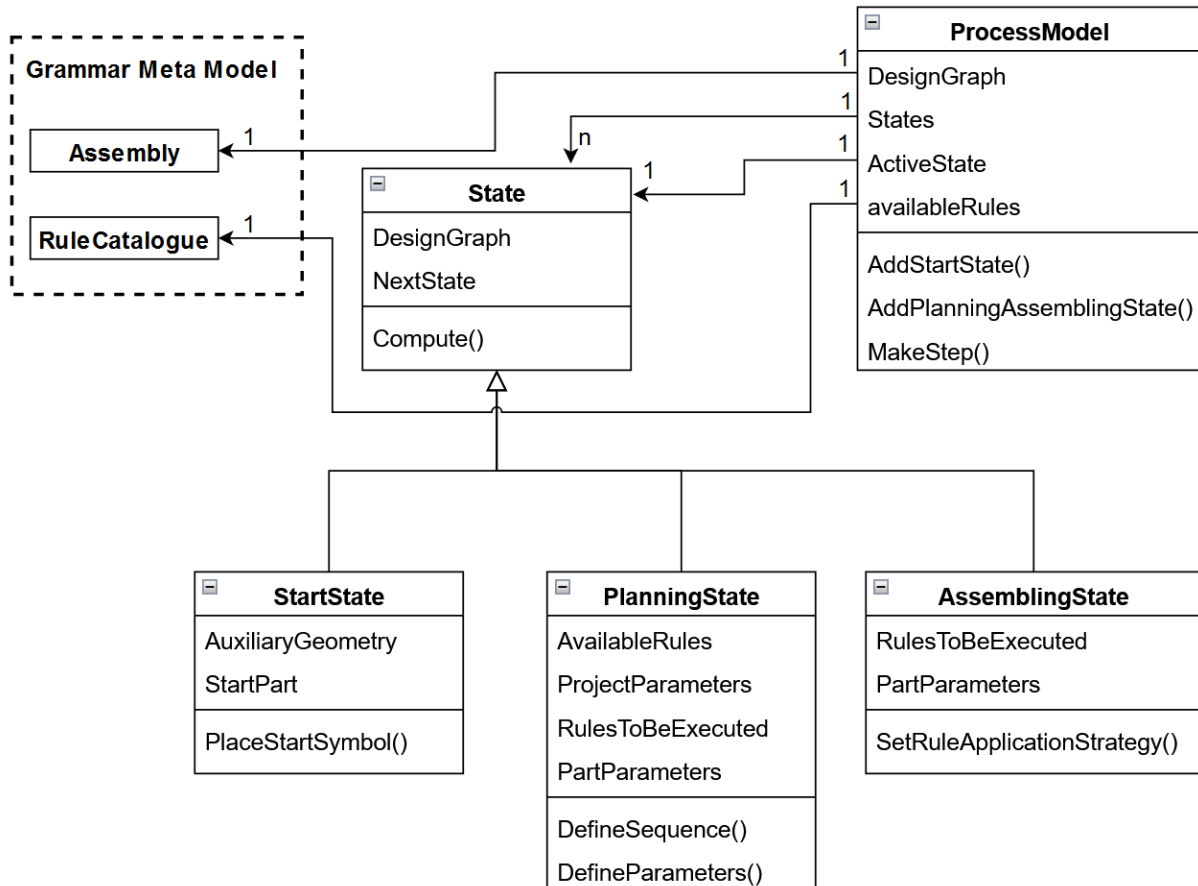


Figure 4.8: The architecture of the algorithm meta model.

States

The various kinds of states have already been described and explained in [chapter 3](#), and the same conditions apply here for the implementation. Furthermore, a state must always have a `compute` method, which executes all steps necessary to either make modifications to the assembly or prepare for it. This is guaranteed by implementing an abstract `State` class which all other state classes inherit from, mandating that each must implement a `compute` method. Modifications to the assembly/graph can only happen during the `StartState` as well as the `AssemblingState`, which is handled by the `compute` method each time. Meanwhile the `compute` method sets and handles parameters and rule sequences in the case of a `PlanningState`. There, these sequences and parameters are immediately handed over to the succeeding `AssemblingState`. The separation of states where 1: commands are 'queued' and 2: commands

are executed (`PlanningState` and `AssemblingState` respectively) was implemented to better abstract the process modeling as well as provide a clear interface for the case study to input its use case-specific data.

Process Model

The `ProcessModel` class wraps the states into a single data structure. It contains the list of states that must always follow a certain pattern. After a `StartState` where the start symbol is defined there always follows a `PlanningState` where sets of rules and parameters are defined. After a `PlanningState` there must always be an `AssemblingState` where these instructions are executed. After the initial `StartState` there can be as many `PlanningState-AssemblingState` pairs added to the process model as the desired result needs. Wrapping the applications of transformation rules inside a process model pattern gives us more control over what and when rules are applied, which is quintessential for our rule application framework mentioned in the previous chapter.

4.3.3 Case Study

The case study is the part of the program where the concepts and functionalities of the algorithm and grammar meta model are used to create a model. It involves a main entry point in which all necessary parts, rules, and parameters are defined as needed by the use case, then wraps the rules that are to be applied and their parameters into objects either of type `PlanningState` or `StartState`. Finally, after defining the necessary states, each state can be computed which invokes the rule engine.

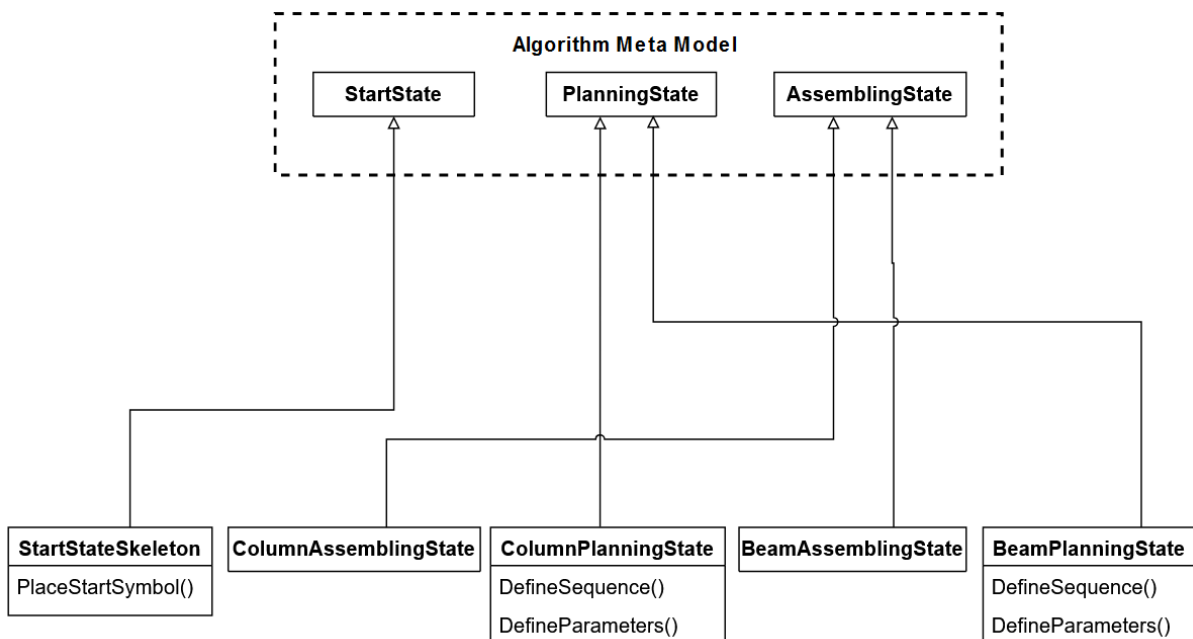


Figure 4.9: Example structure of a case study.

Program

The `Program` file is the entry point for the entire software. It also serves as a hub for creating parts, rule definitions, parameters and the process model. To start, all necessary parts such as foundations, columns, beams and more are defined. On the basis of these, rule definitions are created and then added to a rule catalogue. Next, the parameters such as plot geometry, story height and number of fields are set and stored in a dictionary. With parts, a rule catalogue and parameters, our process model can finally be set up.

Process Model Specification

This subdirectory contains class definitions all inheriting from either the `StartState` or `PlanningState` classes. While the previously explained classes provide the general framework, the methods of setting start symbols as well as defining rule sequences and parameters are actually declared here. For example, if we want to place a single foundation as our start symbol, we would do this within the `PlaceStartSymbol` method of a `StartStateSkeleton` class (inheriting from `StartState`). If we want to define rules that are to be applied, we define those within the `DefineSequenceOfRuleApplications` method of a `PlanningStateColumns` class (inheriting from `PlanningState`).

While most of the logic concerning the creation of the model is handled in the process model and rule engine, one crucial part is actually dealt with here. The collision checking algorithm detailed earlier only works when the basic geometry of the components is correct, i.e. the length of the beam is defined correctly and fits perfectly in between two columns. The logic concerning this correct parametrization of the parts is a portion of the process model specification, inside the `PartParametersPerRule` method to be exact. This needs to be handcrafted by the developer for each separate `PlanningState` as well as case study.

Detailed parameters and rule definitions for each case study are explained in the next chapter.

4.4 Algorithm Overview

To get into more detail on how the different software components behave and what they do during runtime, let us now consider the overall algorithm flow of the program. As seen in [Figure 4.10](#), the entry point is always inside the case study by defining the parts based on our grasshopper definitions. Based on these parts and their connections, we can start defining our case-dependant set of rules and parameters, which then lets us outline our process model with all necessary start and planning states. With the process model set, we can start invoking the rule engine by calling the `MakeStep` method inside a loop. This method computes the current state inside the list of states until the end is reached, where the resulting graph is serialized to a `.3dm` model.

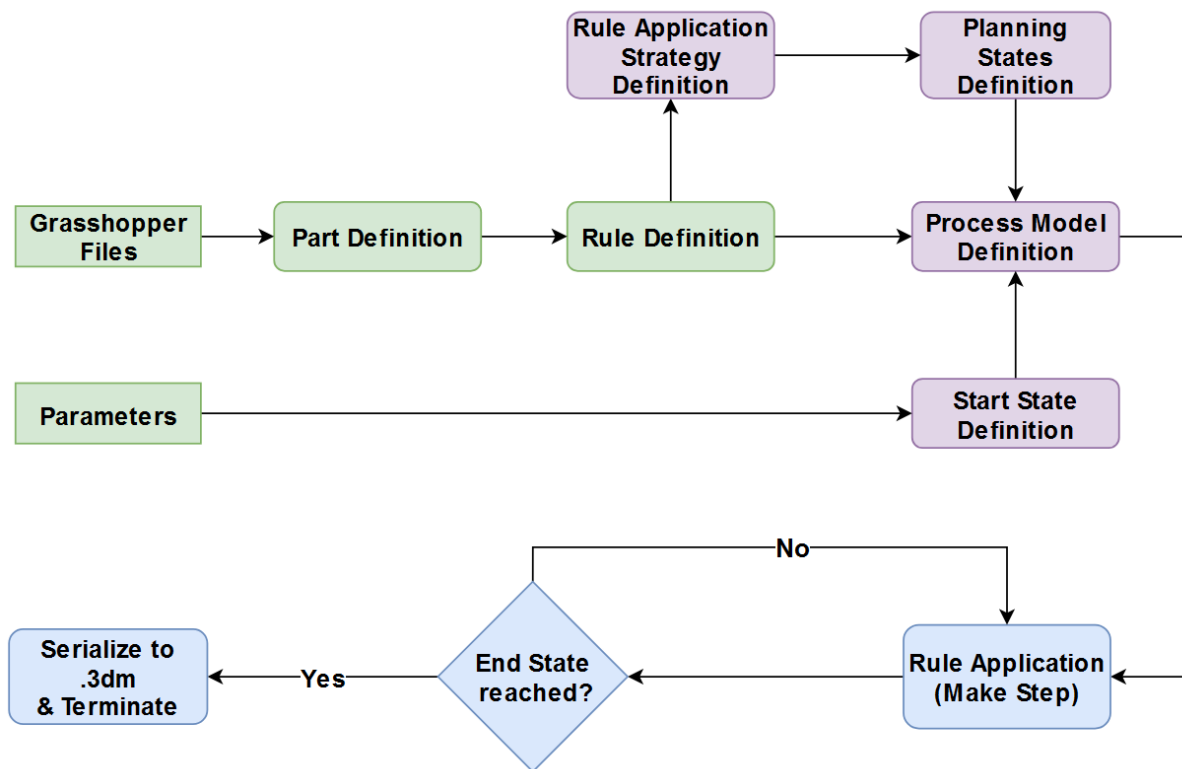


Figure 4.10: General program flow of the algorithm during runtime.

4.4.1 Details of Select Parts of the Program

Rule Engine

Of course, the most crucial and interesting part of this program lies within this `MakeStep` method and all that it entails, as seen in [Figure 4.11](#). As mentioned before, it always invokes the computation of the current state, while simultaneously shifting the current state to the next in the list. Should the current state be a `StartState`, the start symbol as defined in the case study is added to the assembly. This could be, for example, the placement of the foundations of a building. Note that there can only ever be one `StartState` within the process model and it has to be the first state that is computed. Should the current state be a `PlanningState`, which will always follow after a `StartState`, the rule application strategy/order as well as the necessary part parameters are set and prepared for assembly. To name just one example, one could set a number of rules which specify that columns should be placed on top of the foundations, and that these columns need a certain height corresponding to a pre-defined story height. Yet, no changes are made to the assembly during this state. This happens if the current state is an `AssemblingState`, where the rule application strategy of the previous `PlanningState` is actually executed with the help of the rule engine. Each rule is applied in the exact order specified during the definition of the process model, by first checking the existing assembly for matching open interfaces, then placing a component compliant with the previously set parameters in the assembly at the global origin, then transforming that component so that the two matching interfaces face each other. We place a column within the model (at the origin), change its height

and then move and rotate that column so that its bottom surface/interface matches with the top surface/interface of a foundation. After completing the assembling step, the next `PlanningState` is computed and the loop continues until there are no more states left to compute.

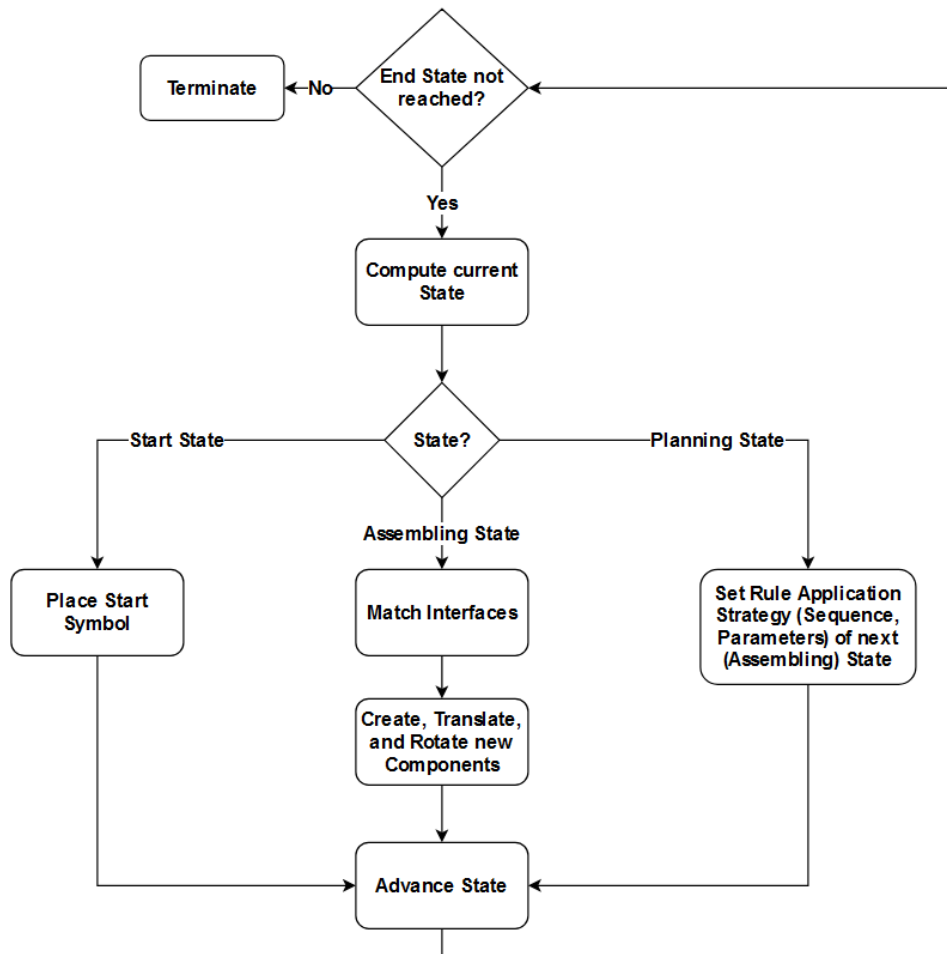


Figure 4.11: Iterative calling of the rule engine.

Start State Definition

Pivotal for the resulting model is how and in what order the modifications to the assembly are made. This is where another part of the rule application framework comes into play. The process model alone already gives us control over when certain rules are applied but what it cannot do is let us directly define how, or to phrase it better, where the rule is applied. For example, we want to apply a rule where a column is placed on a foundation. The process model lets us determine that the rule is applied right after the start symbol (the foundations) are placed. Yet, since there may be multiple foundations that have open interfaces, the column could be placed on any of the foundations with the resulting model being fully compliant with what we have set up in our case study.

When looking at the rewriting algorithm more closely, specifically the `RuleMatchingResult`, the first available matching interface is returned when we want to check if a rule can be applied to the graph. As a reminder, the `RuleMatchingResult` determines whether there are patterns in the

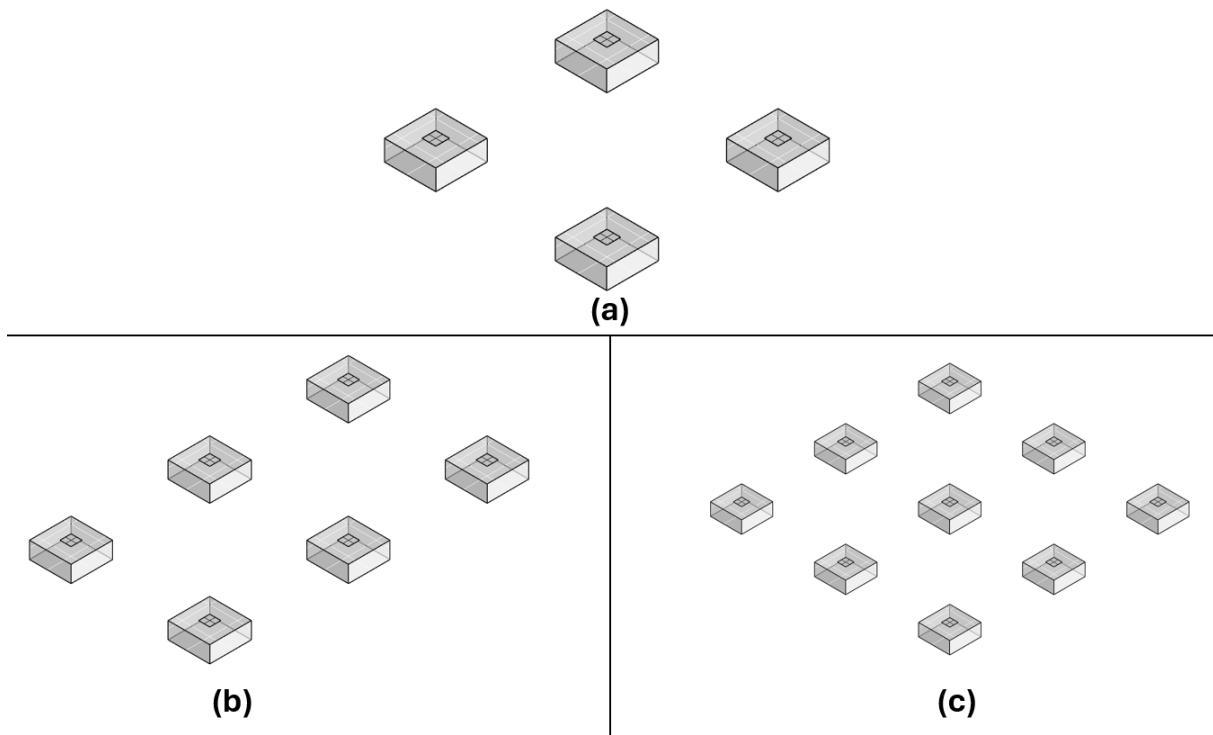


Figure 4.12: Appropriate start state definitions for (a) 1x1 fields, (b) 1x2 fields, (c) 2x2 fields.

graph that match to the specified left-hand-side of the rule, and returns the first valid interface. When searching through the graph, the components are stored in the same order in which they were placed originally. Using this fact, we can introduce a certain predictability to where the rule is applied. Staying with our example, the foundations which act as our start symbol were added in accordance to an algorithm that places them in counter-clockwise order starting from the global origin. This means that the foundation at the origin point will always be the first to be returned when looking for a component of 'type' foundation inside the assembly. Once a column is placed on top of this component, the interface is no longer open, which in turn means that the next best interface would be returned when searching for yet another LHS match. In our case, it would be the interface of the next foundation in counter-clockwise order. We can make use of this technical detail by examining the start symbol closely and defining our rule application strategy accordingly. As to why we would want that kind of control over where the rules are applied, we can reiterate what was explained in the previous section on the basis of our example: Let us say we have different types of columns that can all be placed on a foundation, but also want to exactly specify on which foundation a certain type of column should be placed. The order of the foundations was defined in our `StartState`, and with that (predictable) knowledge we can set our rule application strategy in such a way that the correct type of column will be placed on the correct foundation, as seen in [Figure 4.13](#). However, it cannot be stressed enough how significant and crucial the correct definition of the start symbol is when approaching it this way. Should the algorithm which creates the start symbol be in any way invalid or faulty, any subsequent change made by rule application may also be compromised.

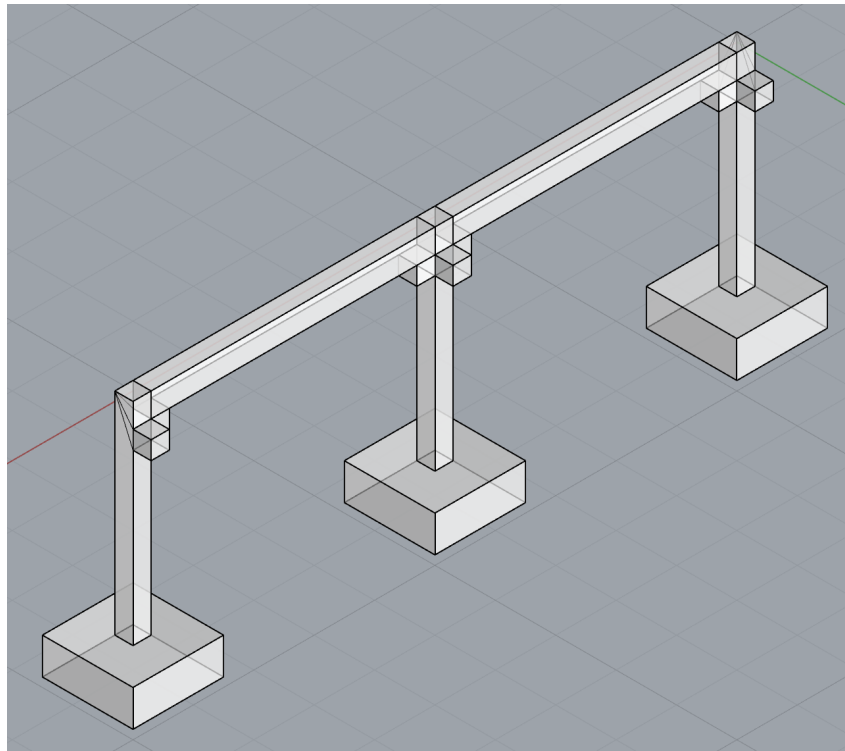


Figure 4.13: Different types of columns may be necessary even when placing them on the same type of foundation. In this case, the second column was placed using a completely different rule.

Rule Matching

We have now spoken about how the `RuleMatchingResult` behaves when it actually finds a matching open interface. Yet, there may also be the case where the LHS of a rule does not match anywhere in the graph. Of course, if there is no match then the rule can not be executed at all, since the RHS cannot attach anywhere. In this case, this step is simply discarded and the algorithm moves on to the next rule in the rule application strategy. The alternative would be to halt the program at this point. Yet, this would mean that which, and especially how many rules are applied needs to be meticulously determined and defined by the developer before the program is executed. This is cumbersome and unnecessary and also the reason why the former approach was ultimately chosen. Discarding invalid rules gives us the advantage that we can set "wrong" or "too many" rules in our application strategy without needing to worry whether the program will halt. We pay for this luxury with a potential loss in performance since the program is not being as productive. Yet what we gain in runtime performance when not doing this we lose many fold in development time.

Connection Checking Algorithm

Furthermore, a sub-algorithm for connection checking has also been implemented. As detailed in the last chapter, this serves the function of reviewing the geometry for other possible connections after a rule has been applied. Since the rules we have defined for our use cases all follow a certain schema (look for a single open interface, attach a single vertex and edge to it), we can check afterwards whether any other open interfaces now geometrically match.

4.5 Results

4.5.1 One Field Skeleton With Multiple Stories

Being the simplest of the case studies, the results were successful. In [Figure 4.14](#) we can see that an assembly was created that fulfils our topological and geometrical conditions. The red lines between the components indicate that a connection exists between them (inferred from the `otherConnection` property), and when examining closely, the geometry is correct at all points: beams and decks are at the right length, interfaces align with each other ([Figure 4.16](#)). The start symbol is created algorithmically within the context of a `StartState` and places four foundations at each corner of a user-defined rectangular plot. Each foundation had to be rotated by 90 degrees compared to the previous one, since the columns that are placed on top also need to be rotated accordingly. In summary, a process model very similar to the one used in [Figure 3.8](#) was used, with one additional step to add the deck. Once these states were correctly defined, they could be packaged into a loop that can repeat that same process for the next level. This can scale indefinitely ([Figure 4.15](#)).

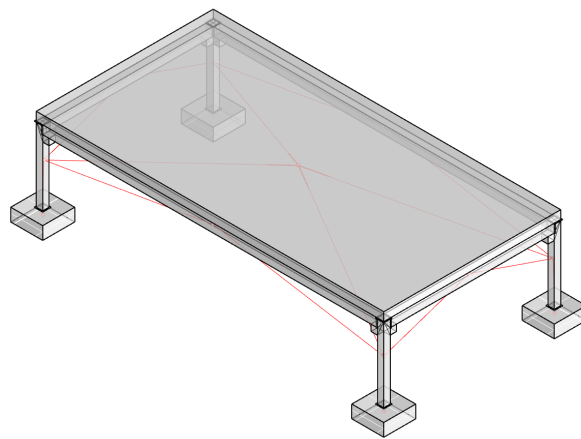


Figure 4.14: Mode resulting from defining one story and unequal side lengths. The red lines between the parts indicate that they share a connection (e.g. a column is connected to a beam).

The graph grammar, also known as the rule catalogue, is also fairly straightforward to set up and define. A summary of the rules we need to achieve this result is provided in the following. In total, eleven rules had to be defined.

- `ColumnOnFoundation`: Place a column on the foundation.
- `BeamOnColumnConsole`: Two Rules that place a beam on a console. One for each console/beam interface combination.
- `DeckOnColumn`: Four rules that connect the bottom deck interfaces with the top interfaces of the columns. Since each interface of the deck is numbered, four rules need to be created.
- `ColumnOnDeck`: A set of rules that places columns on top of the deck. Similar to the previous set of rules, four rules needed to be created.

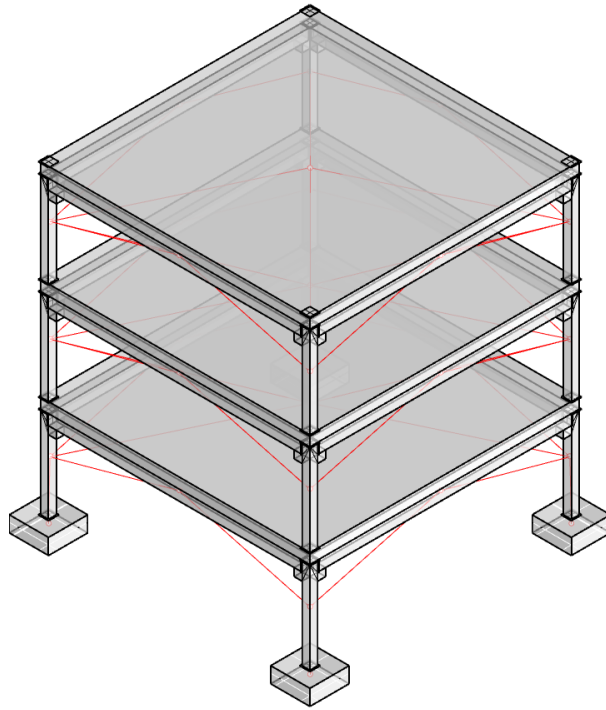


Figure 4.15: Resulting model using a square plot and four stories.

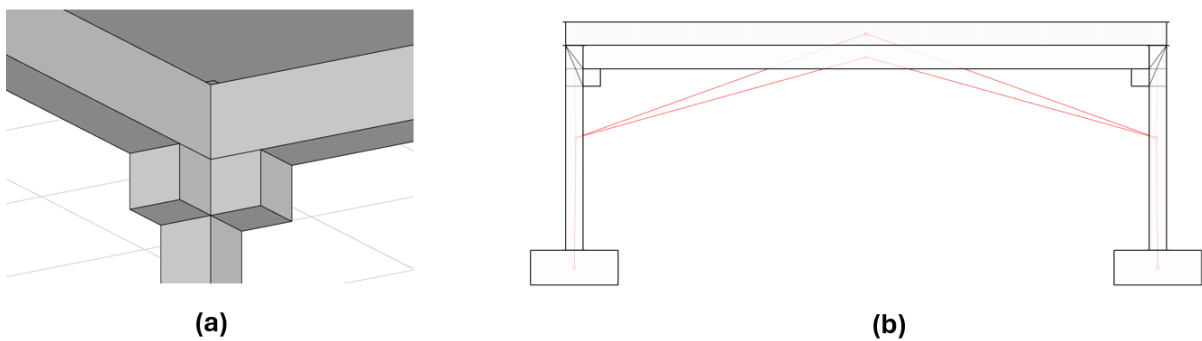


Figure 4.16: Detail view showcasing that beams, decks and joints are correctly dimensioned.

4.5.2 Two Field Skeleton With One Story

This case study introduced a significant amount of complexity to both the start symbol definition as well as the rule catalogue. Large parts of the previous case study could be kept the same, but the algorithm that defines the start symbol needed to be significantly extended. It had to not only place foundations at each corner off the plot, but also in between them. The decision was made to also solve this with a scalable algorithm that not only works with two fields but also more. Here again, the correct orientation of the foundations was crucial and can be seen in [Figure 4.17](#). Furthermore, more column parts were introduced since we now needed to place columns with three or more consoles ([Figure 4.19](#)), which in turn meant that we needed to introduce significantly more rule definitions than before. In addition to the rules from the previous case study, the following needed to be appended, for a total of twenty:

- `ColumnOnFoundation`: Analogous to the previous case, with the addition of two more column types, resulting in a total of three rules.
- `BeamOnColumnConsole`: Analogous to the previous case. Since there are now two more potential console interfaces, the total rules amount to eight.
- `DeckOnColumn`: With two by two fields, up two a number of nine interfaces on the bottom lead to nine rule definitions.

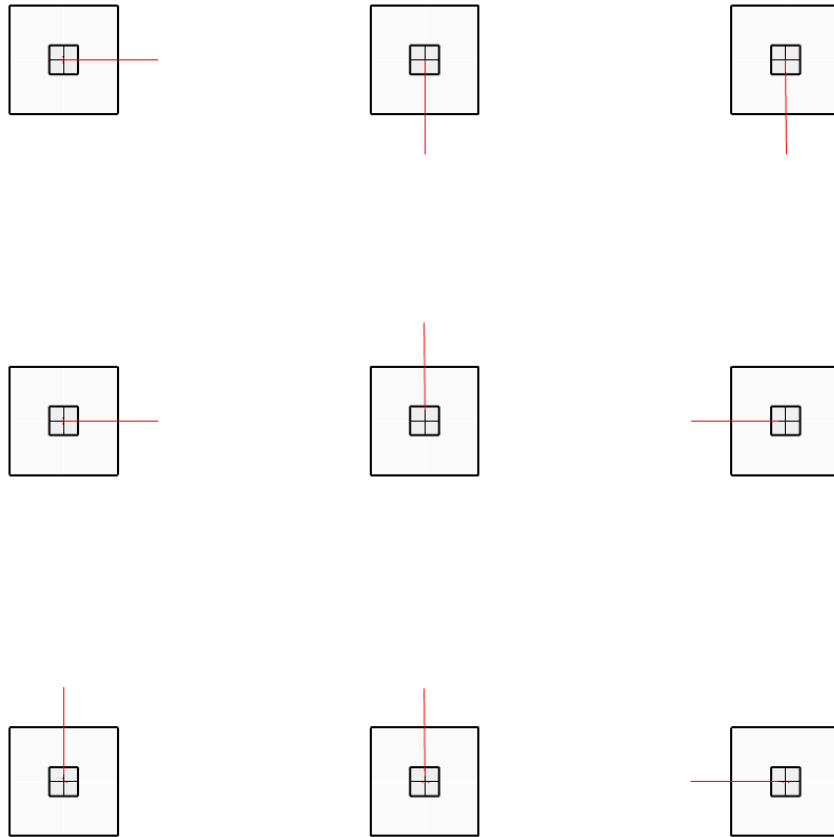


Figure 4.17: The orientation of the foundations and their interfaces indicated by red lines. While the `StartState` itself may not change significantly (square geometries), the orientation is paramount for the subsequent placement of the columns.

4.5.3 Multi-Field, Multi-Story Skeleton

Extending the previous case even more, an attempt was made to now scale the multi-field skeleton upwards by adding support for multiple stories. While successful (Figure 4.20), the necessary setup was extensive, with large parts of the rule sequences needing to be defined manually. The choice of rule definition schema led to there being a drastic increase in the amount of rules necessary, since there was a higher count of parts and interfaces for this case. Specifically, the deck had to be adapted to include multiple top interfaces for multiple fields, which all needed their own rule definition to be able to connect to the appropriate columns. Large catalogues of rule definitions are difficult to handle, and also make sequence definitions for the process model more difficult. The following rules were included in the catalogue:

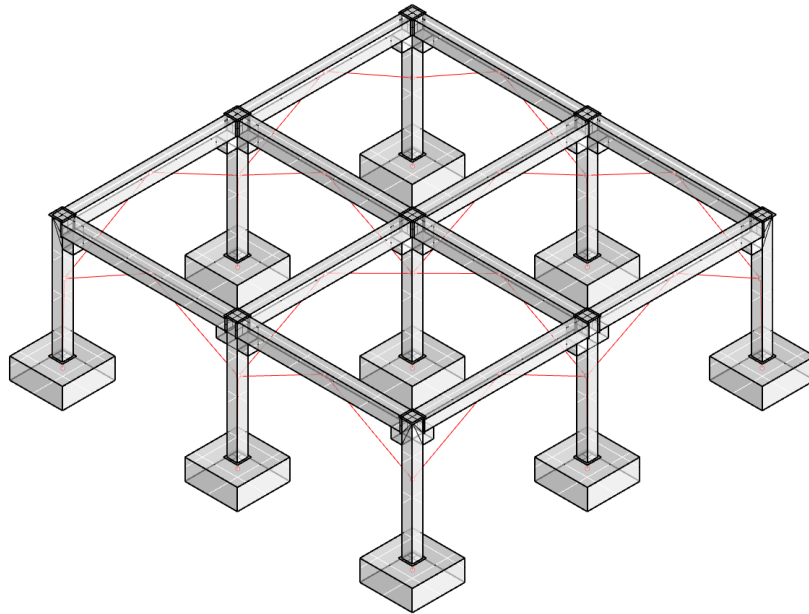


Figure 4.18: Resulting model for two fields using the above mentioned rule catalogue.

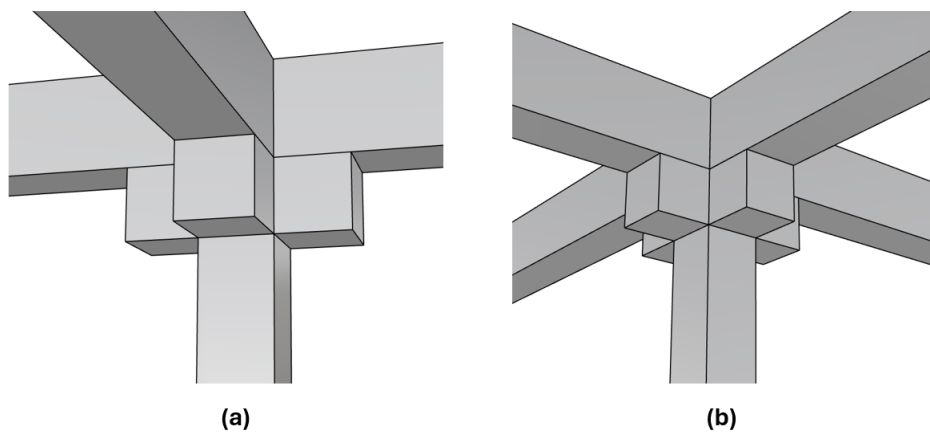


Figure 4.19: Images detailing the columns with three or four consoles.

- `ColumnOnFoundation`: Analogous to the previous case. Total of three rules.
- `BeamOnColumnConsole`: Analogous to the previous case. The total rules amount to eight.
- `DeckOnColumn`: With three types of columns needing to be placed on top of the deck with 9 interfaces, 27 additional rules need to be defined.

This case study has shown that the rule sequence definition, while pivotal, is difficult to automate and often has to be defined manually. Which is, of course, not the goal of this thesis. Reasons as to why this was necessary will be discussed in the following section and chapter.

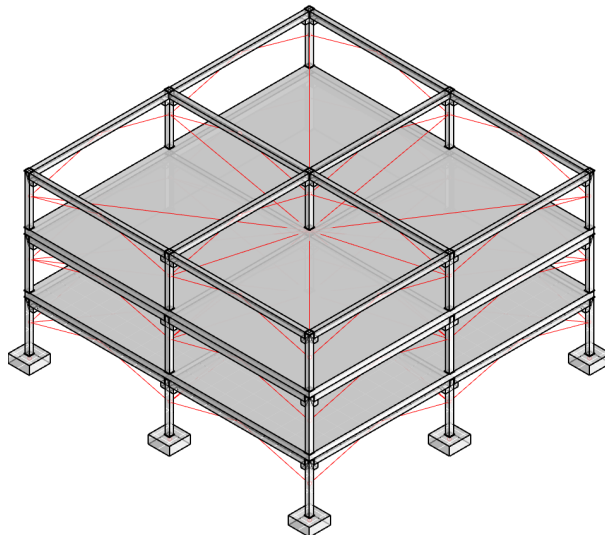


Figure 4.20: Example result using two fields and three stories.

4.6 Implementation Issues

4.6.1 Part Parameters in Grasshopper

Grasshopper allows fine-grained parametrization of parts, but also does not come without its drawbacks. Consider the following example: A column can have anything between two and four consoles to place beams on. If the column is placed at the corner of a building it will most likely have two, while if it is placed somewhere in between it will have three or four. Of course, we could also parametrize this inside the grasshopper script by simply adding something like a number slider as an input. This would allow us to dynamically change the number of consoles depending on the requirements. This is not too difficult to implement in Grasshopper itself, but a problem arises when trying to interface with the script inside our program: Should we define the number of consoles to something lower than four, some components inside the grasshopper script throw an exception as seen in [Figure 4.21](#), since they do not receive their expected inputs. For example, if we would want to add a `NumberOfConsoles` parameter to our column grasshopper script, the error would be thrown when the parameter is too small, as not every component receives its expected inputs. In this case, certain components having to do with the duplication of data do not receive any data to duplicate. Rhino still manages to visualize the geometry adequately, but rhino compute starts throwing errors when parsing the Grasshopper script, making the program unable to correctly define part parameters. While it may be possible to resolve this issue, creating individual scripts for all the different types instead is solution applied here. Yet, this leads to more parts and, more importantly, more rules needing to be defined. This problem is not limited to columns either, a deck component is supposed to have a different amount of interfaces on its top depending on the amount of fields where set.

An attempt could be made to fix this issue by instructing the grasshopper scripts to return lists instead of singular items. This way, one could parse the returned lists for the desired parameters

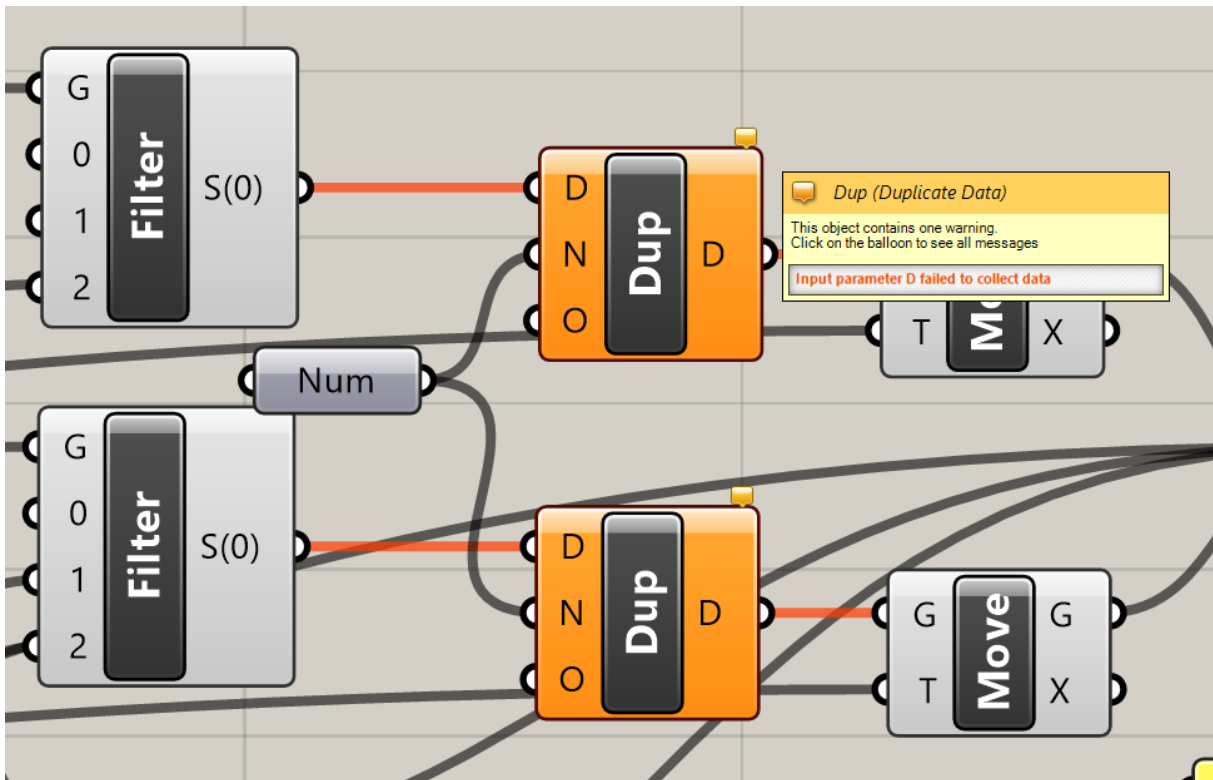


Figure 4.21: Error in a grasshopper script: An input parameter is set in such a way that certain components do not receive their expected input. While Rhino may still be able to visualize the geometry without any problem, Rhino compute cannot.

and interfaces and most likely remedy the errors. Currently, the implementation suffers greatly from this issue.

4.6.2 Grammar Definition and Debugging

While our rule definitions were kept as simple as possible, they exhibit a significant disadvantage once the desired outcome becomes more complex. Since parts were categorized into sub-parts such as columns with different amounts of consoles or decks with differing amounts of interfaces, the amount of rules necessary to define to achieve a reliable grammar becomes increasingly extensive with each new part type we introduce. Any column type must be able to connect to each interface of the deck part. Assuming three column types (two, three, and four consoles) and a three-field construction (resulting in sixteen interfaces on the deck) we must define 48 rules simply to place columns on top of a deck. The considerable amount of rules makes debugging quite tedious.

Chapter 5

Discussion

Graph rewriting as an approach for algorithmic design or automated planning offers great advantages compared to conventional planning methods as well as other algorithmic design approaches. But the manner in which it is implemented, especially the intricate and small details of the rule engine and process model, is crucial regarding the capabilities and extendability of the approach. To start off, let us first answer the research questions we have articulated:

5.1 Research Questions

5.1.1 How easily can algorithmic design be achieved with the help of graph rewriting rules within the context of modular precast structures?

The approach benefits greatly from a modular construction method as is present in structures using precast modules. Since each module can easily be represented by a node including its node properties, the connections between these modules can also be represented using edges. Within the context of precast structures we can adapt the model to a graph system using these nodes and edges without too many layers of abstraction (see [Figure 2.6](#)). Furthermore, graph rewriting rules act as a very suitable method of adapting engineering knowledge since the rules can represent the low-level concepts of construction in a computer- but also human-readable manner (VILGERTSHOFER, 2022). With low levels of abstraction it is also easy to create the rewriting rules similar to [Figure 3.2](#) and in turn, create a proper grammar for the type of structure.

Apart from the adaptation of modules into a graph system, the geometry of precast modules also carries significant advantages. The geometry of precast parts can easily be parametrized in such a way that often by only changing a single parameter the geometrical conditions discussed in [chapter 3](#) can be satisfied.

To summarize, using precast modules adds little abstraction to the graph adaptation process and provides simple geometries to parametrize. These in combination make precast structures a suitable candidate for graph rewriting as a method to achieve algorithmic design.

5.1.2 How detailed does the surrounding framework for rule application need to be to create usable structures?

This approach employed a rule application framework to manage when and where certain rules are applied. This was done to ensure that the rules were executed at certain points during the assembling process leading to the creation of usable structures. In our example of skeletons

made out of precast modules this was achieved by defining rule sequences and parameters within the various `PlanningStates` defined for the process model, as detailed in [chapter 3](#). In general, everything defined within the process model states is not strictly part of the graph rewriting process.

Without the definition of rule sequences, the result does not resemble an actual real-world usable structure, as seen in [Figure 5.1](#). This means that at least for this implementation the surrounding framework is crucial for the success of the approach. Let us briefly summarize which steps of the program are part of the surrounding framework and which ones are part of the rule applications:

- Surrounding Framework: Project parameter definition, start symbol definition, rule sequence definition, part parameter definition.
- Rule Engine: Rewriting process

As one can see, the implementation relies heavily on the surrounding framework (especially the process model) to achieve the desired results. While this may have yielded some usable structures, having to make use of the framework to this extent was not the original intention. Many of the problems encountered during development, such as connecting a beam to two columns, correct definition of part parameters, and post-processing of newly added components were solved algorithmically within the surrounding framework instead of in the rule engine. For future research, methods on how to solve these problems with graph rewriting must be explored.

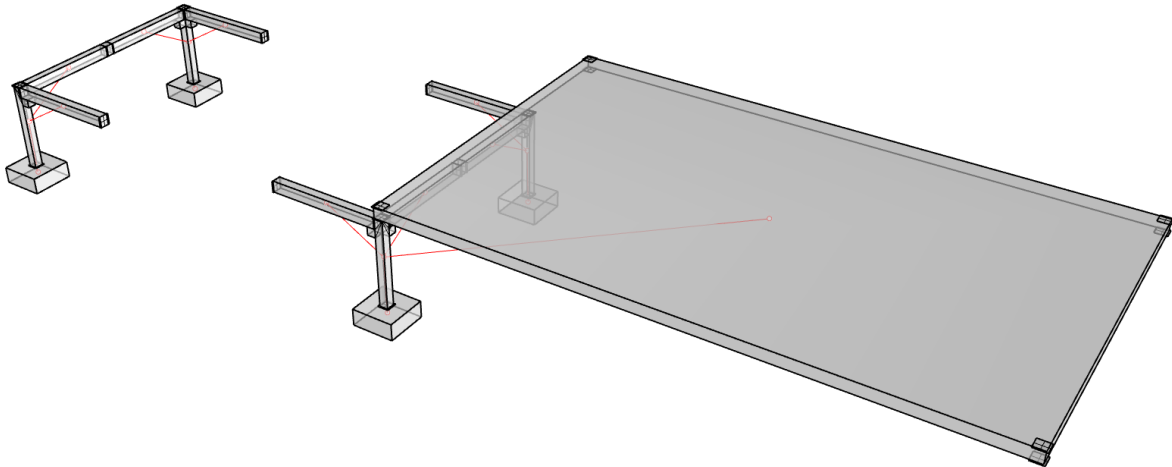


Figure 5.1: Example of a resulting model using no process model setup.

Let us conclude this answer with a small, experimental thought: Rewriting rules were originally intended to formally describe the grammar of languages (CHOMSKY, 1956). Using the rules, the grammar of an entire natural language (e.g. English) could be formally described, without the help of any 'surrounding framework' at all. Transferring this thought over to our graph rewriting approach in the built environment, there may exist a graph grammar for precast structures that is so meticulously defined that it always results in a valid precast structure - without the need of any surrounding framework. But, whether the definition of such an extensive grammar is feasible is an entirely different question.

5.1.3 How well can this approach be adapted to other fields of civil engineering?

The approach makes use of various software components that each fulfil a different purpose. The rule engine handles the graph rewriting process by applying the rules, whose sequence and parameters were defined inside the process model, which was defined in the respective case study. Importantly, the rule engine and the process model, or *Grammar Meta Model* and *Algorithm Meta Model* as they were called in the implementation, are entirely case-independent and can, in theory, be used for any kind of construction as long as it can be described using rewriting rules.

We have established that any structure made out of precast modules is suitable for this approach (e.g. bridge construction), yet what about other methods of construction, such as infrastructure? When planning a road for example, the first part that is drawn/planned is the alignment, the central axis from which everything else builds off of. Such an alignment consists of various sections such as straights, circular arcs, and clothoids, each defined by a set of parameters. If we consider these sections to be our parts or modules we can also define rules, with a few examples being given in [Figure 5.2](#).

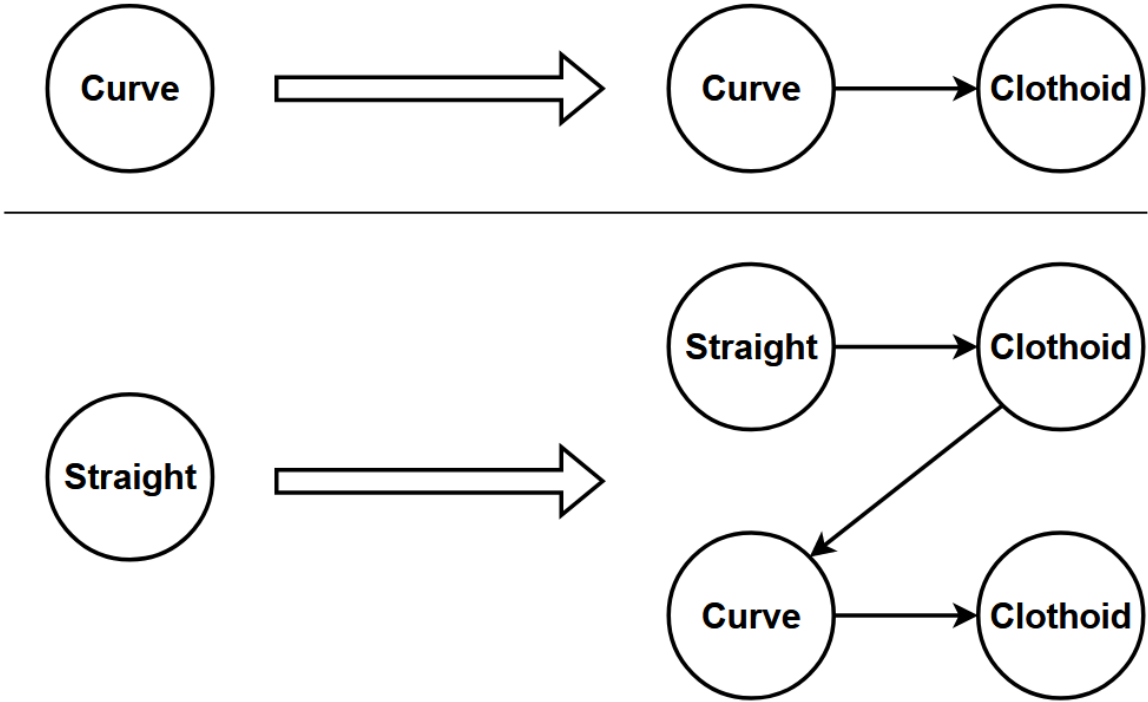


Figure 5.2: Basic examples of what rule definitions could look like in infrastructure planning.

With the crucial parts of the approach being case-independent, it can be confidently said that graph rewriting has the potential of being applied many fields of civil engineering, as KOLBECK et al., 2022 have already described. The only limiting factor being, how well the knowledge can be represented using rewriting rules.

5.2 Drawbacks and Improvements

Some drawbacks of the implementations as well as potential fixes were already discussed in the [chapter 4](#). Now, the scope of these issues will be broadened to better discuss the areas of improvement that this approach has for potential future research.

5.2.1 Rule Definitions

One area mentioned in the last chapter was that of the rule definitions. The simplistic approach taken in the implementation made the handling and application of rules rather easy, up to a certain point of complexity. The example given in [Figure 3.4](#) illustrates the complexity quite well: While simple situations such as columns being placed on foundations are adequately represented using the rule schema as seen in [Figure 3.2](#), the method crumbles as soon as the circumstances become even slightly more complex. This is already the case when we want to place a beam on top of two (already placed) columns. One must remember that rewriting rules serve as the main way of adapting the real-world engineering knowledge into a computer-readable form. This in turn means that the rule must be able to represent such knowledge adequately. Rule definitions as in [Figure 3.2](#) can not represent the fact that a beam *must* be supported by at least two columns. The workaround employed by this approach was to check for other connections after a rule has been applied. Yet, this also necessitated that the geometric conditions (the length of the beam must equal the distance between the columns) must be met *before* the rule was applied. Which, in turn, resulted in a slew of pre-application modifications that must be made. It is this mixture of pre-processing and post-processing that undermined the graph-centric approach and made it more imperative than originally intended.

To stay with our beam-length example, the rule definition can of course be adapted to represent the situation adequately. In [Figure 5.4](#) it is shown that a beam can only ever be added to the graph if there are two columns with open interfaces. This represents the engineering knowledge more closely. As for the geometric conditions, one could implement the shape grammar STOUFFS, 2019 employs by using *predicates* and *directives*. Predicates being conditions that must be fulfilled for a rule to be applied that cannot be simply represented in the LHS pattern. Our geometrical conditions would be such an example. *Directives* on the other hand, specify a certain value that must be set upon rewriting that cannot be declared by the shape of the RHS. The solution proposed in our method by applying a connection check sits somewhere in the middle: On the one hand, we determine that the geometric conditions must be satisfied *before* rewriting is started, on the other hand our specific value (the connection between the second beam-interface and the second column) is established after rewriting has been completed. Making the approach more closely resemble the concepts STOUFFS, 2019 has introduced, one could explore the viability of making the length parameter of the beam dependent on values of the column nodes. Instead of determining the length during pre-processing and making the second connection during post-processing, the beam can be (topologically) connected to both right when the rule is applied ([Figure 5.4](#)) with its length *inferred* afterwards. This can be achieved by further extending the rule definition to include that the node properties can

be dependant of the node properties of other connected nodes (Figure 5.5). This approach to rewriting rules, while not yet tested, could prove to provide more elegantly the necessary tools to create a suitable graph-grammar for the specific usecase and must be further researched.

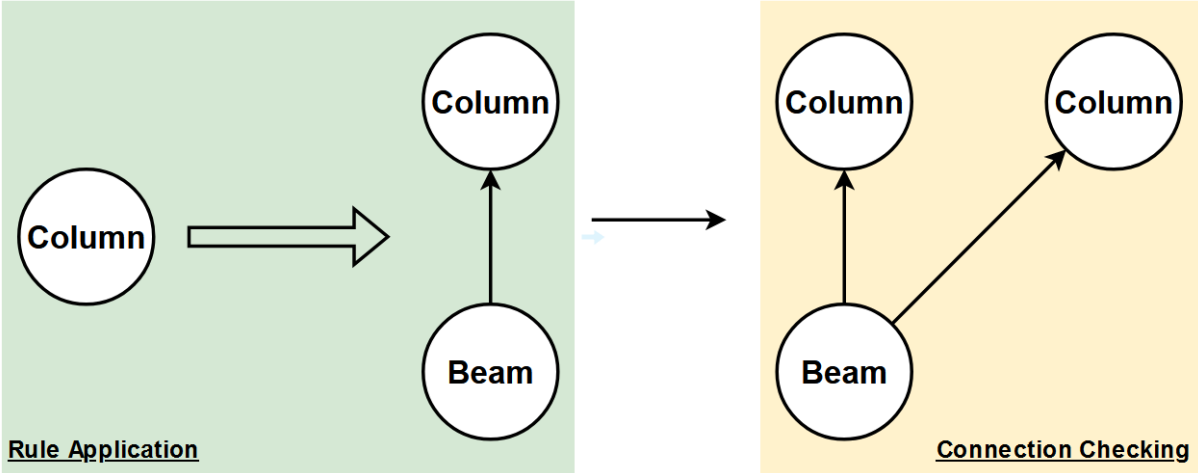


Figure 5.3: How a connection between the beam and its two neighboring columns is established according to the method used in the implementation. After the rule application, a connection check is issued and the relationship is established.

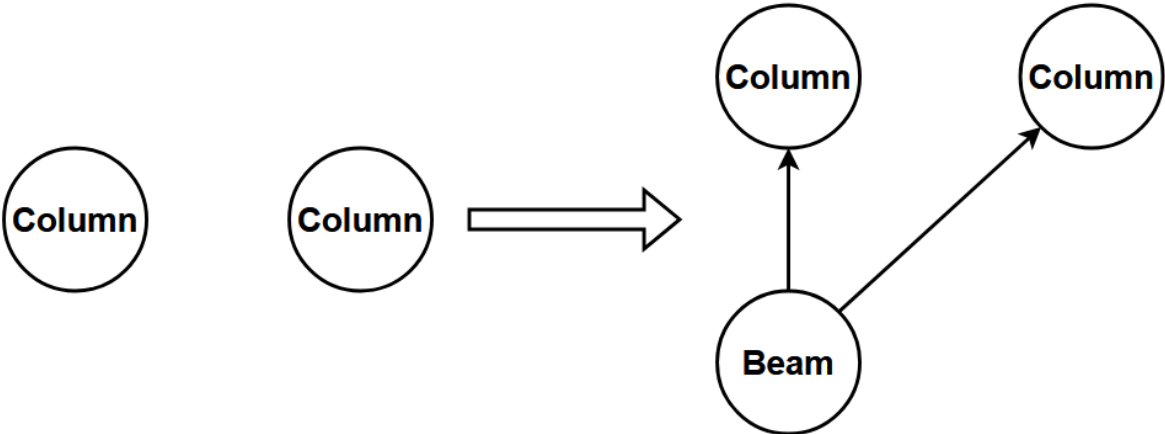


Figure 5.4: This new proposed rule definition for the placement of a beam resembles engineering knowledge more closely.

Hierarchical Rule Definitions

By implementing a more general way of defining rules as mentioned above, meaning the possibility of defining a larger and more complex LHS and RHS, further capabilities can also be added to this approach. One of these is what ROSSI and TESSMANN, 2019 call *hierarchical rules* that aggregate multiple parts into one 'aggregated part' which can then also be part of a rule definition. A great example for this could be the need for multiple storeys in a building. Assuming each level is structurally the same (columns, beam, decks), one level could be 'copy-pasted' to the next. Within a rewriting context, this works by aggregating these parts into one. At the same

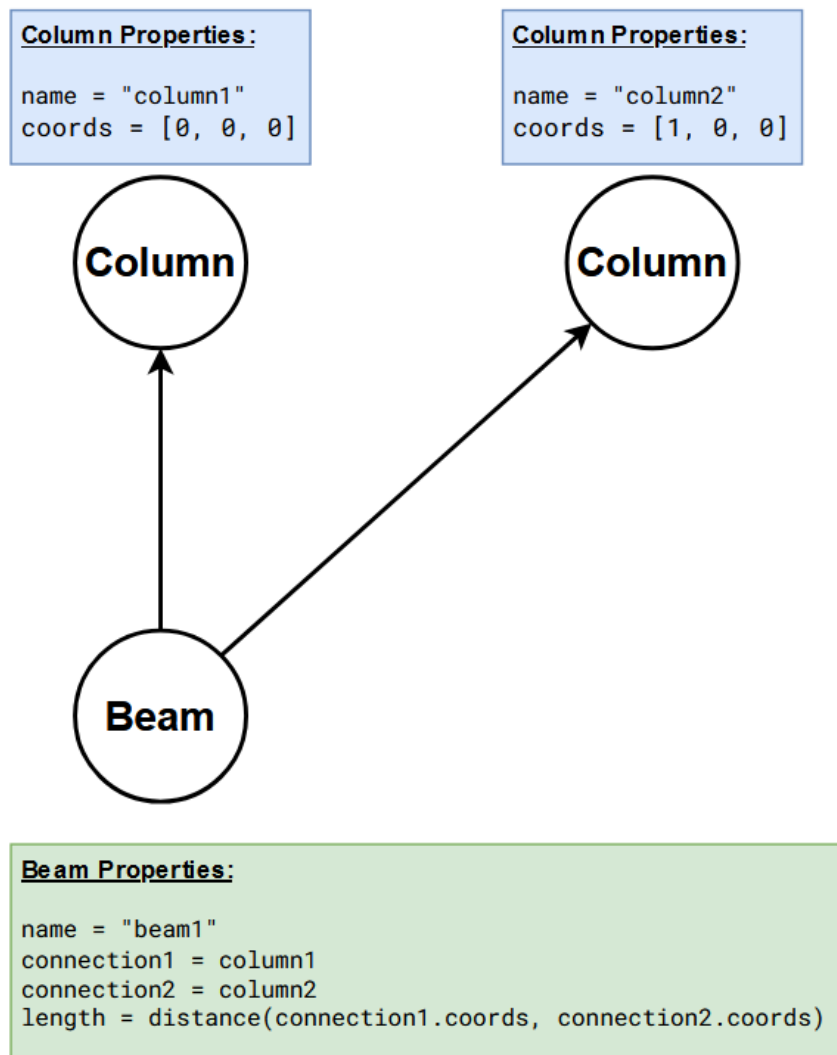


Figure 5.5: The node properties could be dependant on their neighboring nodes. Here, the length of the beam is conditional to the coordinates of the two columns.

time, we define a *hierarchical rule* which postulates that this aggregated part can interface with itself (Figure 5.6).

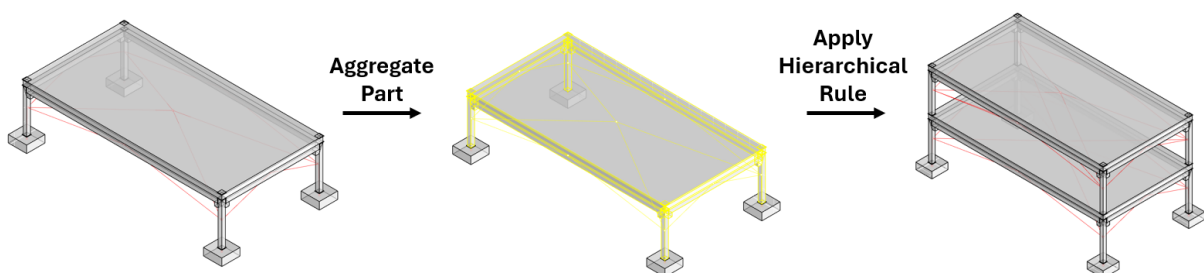


Figure 5.6: Example application of a hierarchical rule. First, a set of parts is aggregated into one. Then the rule is applied which specifies that this aggregated part can be placed on top if it self.

5.2.2 Start State Specification

We have determined that a correct and robust start symbol is the foundation (figuratively and literally) of the entire algorithm. This leads to the necessity of creating an algorithm that can deliver said start symbol consistently. Since user-defined parameters may change on a project by project basis, the algorithm needs to be dynamic in such a way that it can easily handle such dynamic changes. While certainly anything but impossible, it does create the need to define larger parts of the model imperatively.

5.2.3 Planning State Specification

Similarly, we run into a related problem when specifying our planning states of the respective case study. While the rule application strategies of some planning states are as simple as adding the same rule certain amount of times, other times it is more intricate to specify the correct rules at the correct position. Again, we need to meticulously determine when and how a rule should be applied. This puts more emphasis on creating the model 'manually' by burdening the developer with creating appropriate algorithms apart from the rule engine. The amount of work should be reduced by the implementation of larger, more complex rule definitions that alleviate the need for meticulous rule sequence definition. If, for example, a single rule can place all columns on the same level, the sequence for that step in the process model is kept simple and short.

The last two subsections correspond to an issue mentioned in [chapter 3](#). On the one hand, it is desirable to reduce the amount of 'manual' work necessary to create a correct result and instead let the rewriting rules do most of the work. On the other hand, we have established that some amount of control is paramount and comes with the responsibility of acting on it. In this balancing act, implementation choices must be carefully considered as they could either compromise model validity or make the rewriting system progressively obsolete.

Chapter 6

Conclusion

Graph rewriting systems as a method for algorithmic design promise to be a fitting solution compared to imperative approaches such as ABUALDENIEN et al., 2021. Rewriting rules specifically manage to encapsulate engineering knowledge in an intuitive way in order for humans to be able to create, read and understand them. Also, with rewriting and process modeling being universally applicable, it creates a solid base for algorithmic design in the future. But rewriting rules alone may never yield the desired results. The framework (or in the case of this implementation: the process model) for rule applications that facilitates the rewriting process must be able to control the flow of the algorithm appropriately so that usable structures get created. Strong rule engines and rewriting systems exist already today, but an approach on how to use them for design synthesis to reliably and consistently output such designs has not yet been formulated to the necessary extent. But the method presented in this thesis has made another step in this direction.

Graph rewriting in combination with a process model setup may make it possible to develop a 'design engine' one day that is generally applicable and capable of modeling any kind of building, design or product, similar to what EICHHOFF et al., 2016 propose. As long as its users provide a valid grammar as well as a functional process sequence, the core software components of the rule engine as well as the process model can, in theory, create any kind of design. To take it even a step further, current regulations, guidelines, and codes could also be adapted into a graph grammar style alongside their natural language versions. Architects and engineers may take these 'regulation grammars' and build upon them to not only algorithmically create their design, but also make them regulation compliant in the same step. While this may be years, if not decades away, the implications it has on the way we design and model products of any kind even today are both crucial and paramount.

Appendix A

Files

All files necessary to run the implementation are provided in the Sync & Share folder "Masterthesis Benedict Harder". Rhino 7 is required.

References

- ABEL, D. (2013). *Petri-netze für ingenieure: Modellbildung und analyse diskret gesteuerter systeme*. Springer-Verlag.
- ABUALDENIEN, J., & BORRMANN, A. (2021). PBG: A parametric building graph capturing and transferring detailing patterns of building models. *Proc. of the CIB W78 Conference 2021*.
- ABUALDENIEN, J., HARDER, B., & CLEVER, J. (2021). A prescriptive parametric model supporting performance-based design exploration at the early stages. *Proc. of the 32th Forum Bauinformatik*.
- ALLEN, E., & IANO, J. (2019). *Fundamentals of building construction: Materials and methods*. John Wiley & Sons.
- AMBLER, S. (2024). Uml sequence diagrams: An agile introduction [Accessed: 29/02/2024]. <http://agilemodeling.com/artifacts/sequenceDiagram.htm>
- ANTONIOU, F., & MARINELLI, M. (2020). Proposal for the promotion of standardization of precast beams in highway concrete bridges. *Frontiers in Built Environment*, 6. <https://doi.org/10.3389/fbuil.2020.00119>
- BAUMANN, T., FREBER, P., SCHÖBER, K., & KIRCHNER, F. (2016). Bauwirtschaft im wandel: Trends und potenziale bis 2020 [Accessed: 24/02/2024]. *München: Studie Roland Berger GmbH. und UniCredit Bank AG*. https://www.rolandberger.com/publications/publication_pdf/roland_berger_hvb_studie_bauwirtschaft_20160415_1_.pdf
- BJÖRK, B., & LAAKSO, M. (2010). CAD standardisation in the construction industry—a process view. *Automation in construction*, 19(4), 398–406. <https://doi.org/10.1016/j.autcon.2009.11.010>
- BORRMANN, A., FLURL, M., JUBIERRE, J., MUNDANI, R., & RANK, E. (2014). Synchronous collaborative tunnel design based on consistency-preserving multi-scale models. *Advanced Engineering Informatics*, 28(4), 499–517. <https://doi.org/https://doi.org/10.1016/j.aei.2014.07.005>
- BORRMANN, A., KÖNIG, M., KOCH, C., & BEETZ, J. (2018). Building information modeling: Why? what? how? In *Building information modeling: Technology foundations and industry practice* (pp. 1–24). Springer International Publishing. https://doi.org/10.1007/978-3-319-92862-3_1
- CHOMSKY, N. (1956). Three models for the description of language. *IRE Transactions on Information Theory*, 2(3), 113–124. <https://doi.org/10.1109/TIT.1956.1056813>
- DIESTEL, R. (1996). *Graphentheorie*. Springer.
- EICHHOFF, J., BAUMANN, F., & ROLLER, D. (2016). Two approaches to the induction of graph-rewriting rules for function-based design synthesis. *Proceedings of the ASME Design Engineering Technical Conference, 1B-2016*. <https://doi.org/10.1115/DETC2016-59915>
- ESSER, S., VILGERTSHOFER, S., & BORRMANN, A. (2022). Graph-based version control for asynchronous bim collaboration. *Advanced Engineering Informatics*, 53, 101664. <https://doi.org/10.1016/j.aei.2022.101664>

- FOLEY, J. (1996). *Computer graphics: Principles and practice* (Vol. 12110). Addison-Wesley Professional.
- GRGEN. (2024). Grgen.net: Transformation of structures made easy. [Accessed: 29/02/2024]. <https://grgen.de/>
- HELMS, B. (2013). *Object-oriented graph grammars for computational design synthesis* (Doctoral dissertation). Technische Universität München. München.
- HELMS, B., EBEN, K., SHEA, K., & LINDEMANN, U. (2009). Graph grammars - a formal method for dynamic structure transformation. *11th International DSM Conference*, 93–103.
- HELMS, B., & SHEA, K. (2012). Computational synthesis of product architectures based on object-oriented graph grammars. *Journal of Mechanical Design*, 134(2), 021008. <https://doi.org/10.1115/1.4005592>
- ISAAC, S., SADEGHPOUR, F., & NAVON, R. (2013). Analyzing building information using graph theory. *ISARC. Proceedings of the International Symposium on Automation and Robotics in Construction*, 30, 1.
- KIM, M., MCGOVERN, S., BELSKY, M., MIDDLETON, C., & BRILAKIS, I. (2016). A suitability analysis of precast components for standardized bridge construction in the united kingdom. *Procedia Engineering*, 164, 188–195. <https://doi.org/10.1016/j.proeng.2016.11.609>
- KOLBECK, L., VILGERTSHOFER, S., ABUALDENIEN, J., & BORRMANN, A. (2022). Graph rewriting techniques in engineering design. *Frontiers in Built Environment*, 7, 1–19. <https://doi.org/10.3389/fbuil.2021.815153>
- KOLBECK, L., VILGERTSHOFER, S., & BORRMANN, A. (2023). Graph-based mass customisation of modular precast bridge systems. *Proc. of the 30th Int. Conference on Intelligent Computing in Engineering (EG-ICE)*.
- MÖLLER, T. (1997). A fast triangle-triangle intersection test. *Journal of graphics tools*, 2(2), 25–30.
- MUENZER, C., & SHEA, K. (2017). Simulation-based computational design synthesis using automated generation of simulation models from concept model graphs. *Journal of Mechanical Design*, 139(7), 071101. <https://doi.org/10.1115/1.4036567>
- MÜNZER, C. (2016). *Constraint-based methods for automated computational design synthesis of solution spaces* (Doctoral Thesis). Zürich, ETH-Zürich. <https://doi.org/10.3929/ethz-a-010603411>
- MÜNZER, C., SHEA, K., & HELMS, B. (2013). *Automated parametric design synthesis using graph grammars and constraint solving*. <https://doi.org/10.1115/DETC2012-70313>
- NEO4J. (2024). Neo4j native graph database [Accessed: 06/02/2024]. <https://neo4j.com/product/neo4j-graph-database/>
- NGUYEN, D.-C., JEON, C.-H., ROH, G., & SHIM, C.-s. (2024). Bim-based preassembly analysis for design for manufacturing and assembly of prefabricated bridges. *Automation in Construction*, 160, 105338. <https://doi.org/10.1016/j.autcon.2024.105338>
- OBERGRIESSER, M. (2016). *Entwicklung von digitalen Werkzeugen und Methoden zur integrierten Planung von Infrastrukturprojekten am Beispiel des Schienen- und Straßenbaus = digitale Werkzeuge zur integrierten Infrastrukturbauwerksplanung : Am Beispiel des Schienen- und Straßenbaus* (Doctoral dissertation). Technische Universität München.

- ORGANIZATION, O. S. D. (2024). About the business process model and notation specification version 2.0.2 [Accessed: 29/02/2024]. <https://www.omg.org/spec/BPMN/2.0.2/>
- PEFFERS, K., TUUNANEN, T., ROTHENBERGER, M., & CHATTERJEE, S. (2007). A design science research methodology for information systems research. *Journal of management information systems*, 24(3), 45–77. <https://doi.org/10.2753/MIS0742-1222240302>
- PHILLIPS, D. (2014). Tessellation. *Wiley Interdisciplinary Reviews: Computational Statistics*, 6(3), 202–209. <https://doi.org/10.1002/wics.1298>
- PREIDEL, C. (2020). *Automatisierte konformitätsprüfung digitaler bauwerksmodelle hinsichtlich geltender normen und richtlinien mit hilfe einer visuellen programmiersprache* (Doctoral dissertation). Technische Universität München.
- ROLLAND, C. (1998). A comprehensive view of process engineering. *Advanced Information Systems Engineering: 10th International Conference*, 1–24.
- ROSSI, A. (2024). Wasp - a combinatorial toolkit for discrete design [Accessed: 27/02/2024]. <https://www.food4rhino.com/en/app/wasp>
- ROSSI, A., & TESSMANN, O. (2019). From voxels to parts: Hierarchical discrete modeling for design and assembly. *ICGG 2018-Proceedings of the 18th International Conference on Geometry and Graphics: 40th Anniversary-Milan, Italy, August 3-7, 2018*, 1001–1012. https://doi.org/10.1007/978-3-319-95588-9_86
- SARCAR, M., RAO, K., & NARAYAN, K. (2008). *Computer aided design and manufacturing*. PHI Learning Pvt. Ltd.
- SCHULTZ, C., BHATT, M., & BORRMANN, A. (2017). Bridging qualitative spatial constraints and feature-based parametric modelling: Expressing visibility and movement constraints [Towards a new generation of the smart built environment]. *Advanced Engineering Informatics*, 31, 2–17. <https://doi.org/10.1016/j.aei.2015.10.004>
- SHAH, J., & MÄNTYLÄ, M. (1995). *Parametric and feature-based cad/cam: Concepts, techniques, and applications*. John Wiley & Sons.
- ŚLUSARCZYK, G., ŁACHWA, A., PALACZ, W., STRUG, B., PASZYŃSKA, A., & GRABSKA, E. (2017). An extended hierarchical graph-based building model for design and engineering problems. *Automation in Construction*, 74, 95–102. <https://doi.org/10.1016/j.autcon.2016.11.008>
- STOUFFS, R. (2019). Predicates and directives for a parametric-associative matching mechanism for shapes and shape grammars. *Blucher Design Proceedings*. <https://api.semanticscholar.org/CorpusID:202608860>
- STROUD, I. (2006). *Boundary representation modelling techniques*. Springer Science & Business Media.
- STRUG, B., GRABSKA, E., & ŚLUSARCZYK, G. (2017). A graph-based generative method for supporting bridge design.
- TESSMANN, O., & ROSSI, A. (2019). Geometry as interface: Parametric and combinatorial topological interlocking assemblies. *Journal of Applied Mechanics*, 86(11), 111002. <https://doi.org/10.1115/1.4044606>
- VAJNA, S., WEBER, C., ZEMAN, K., HEHEBERGER, P., GERHARD, D., & WARTZAK, S. (2018). *CAX für Ingenieure*. Springer Vieweg Berlin, Heidelberg. <https://doi.org/10.1007/978-3-662-54624-6>

- VILGERTSHOFER, S. (2022). *Kopplung von graphersetzung und parametrischer modellierung zur unterstützung des modellbasierten entwerfens und der erstellung mehrskaliger modelle* (Doctoral dissertation). Technische Universität München.
- VILGERTSHOFER, S., & BORRMANN, A. (2017). Using graph rewriting methods for the semi-automatic generation of parametric infrastructure models. *Advanced Engineering Informatics*, 33, 502–515. <https://doi.org/10.1016/j.aei.2017.07.003>
- WEISHENG, L., TAN, T., JINYING, X., JING, W., KE, C., SHANG, G., & FAN, X. (2021). Design for manufacture and assembly (dfma) in construction: The old and the new. *Architectural Engineering and Design Management*, 17(1-2), 77–91. <https://doi.org/10.1080/17452007.2020.1768505>
- ZHU, J., WU, P., & LEI, X. (2023). lfc-graph for facilitating building information access and query. *Automation in Construction*, 148, 104778. <https://doi.org/10.1016/j.autcon.2023.104778>