# Utilizing process modeling and graph rewriting as a formal approach for algorithmic design of modular precast structures

B. Harder & S. Esser & A. Borrmann
*Chair of Computational Modeling and Simulation, Technical University of Munich, Munich, Germany*

ABSTRACT: This paper proposes a formal approach for the use of graph rewriting systems to achieve algorithmic design for modular precast structures. Design tasks have historically been the responsibility of engineers and architects, and providing computer-based tools to assist during the design process can help reduce costs and overhead. One possible method of expressing evolving design development is graph rewriting. In order to leverage graph rewriting rules so that they incrementally modify a model, process modeling is employed to control the application of said rewriting rules. Managing which rewriting rules are applied in a certain sequence during the design process is crucial for the validity of the resulting model. Furthermore, pre- and post-processing of the individual modules ensured that geometrical as well as topological conditions were satisfied. An implementation exploring the viability of this approach was developed using Rhino and Grasshopper, as well as an internally developed rule engine and algorithm structure. The implementation demonstrated the consistency and scalability that can be achieved by employing graph rewriting systems for algorithmic design of modular precast structures.

## 1 INTRODUCTION

Managing large-scale construction projects has become progressively complex, with surging demand in the construction sector and an ever-increasing number of guidelines, regulations, and building codes. Developing potential design options compliant with all boundary conditions has so far been the task of engineers and architects, who do a majority of the work of manually developing the design while achieving regulation compliance. Engineers develop a potential design solution in today's design workflows and run various simulations and checks to validate their draft against given criteria. If deficiencies are detected, the design is modified and fed into the entire checking pipeline again. Yet, the automation of these tasks promises a significant reduction in cost and overhead.

The question of what approach should be chosen to achieve this algorithmic design is crucial. On the one hand, usable and recognizable structures that satisfy all requirements must be created. On the other hand, having an algorithm that explores all viable alternatives within the realm of possibility during the planning process helps to single out designs that best suit the project's particular needs. However, its operation may be overwhelming for an end user. Graph rewriting systems (GRS) as a method of capturing and applying changes to a model have recently been explored (Vilgertshofer, 2022; Vilgertshofer, 2017) and provide the consistency mentioned above.

While partial automation of the work necessary to produce the designs can help tremendously (Preidel, 2020; Abualdenien et al., 2021), a complete algorithm that not only aids the engineer/architect during planning but actively develops the designs by itself has yet to be achieved. Here, it is crucial to find a suitable balance between full-scale automation that may take on a larger set of tasks on the one hand and providing appropriate interfaces for the user to modify the design process on the other. “Black-Box” approaches that leave no option to influence the result or tune the behavior is generally undesirable due to decreased possibilities exposed to the user. Therefore, this paper focuses on applying a graph rewriting system to encapsulate and apply engineering knowledge by representing it with the help of rewriting rules. In more detail, we explore if and how design procedures facilitating modular design of high-rise precast structures can be represented and executed upon with the help of graph rewriting rules. Additionally, the paper explores the conjunction of the rewriting system with a process model, which controls the application sequence of rules to achieve a desired construction layout.

## 2 RELATED WORK

### 2.1 *Precast structures*

Precast structures use concrete modules cast in advance off-site and installed on-site without the need for any in-situ concreting. Modules reduce costs and overhead, especially during construction, and allow for the modules' adaptability. This is because of the more precise fabrication of modules in conditions independent of weather. Much research on precast structures is being conducted today (Chen et al., 2024; Auer et al., 2023). The main disadvantage is knowing the precise dimensions and characteristics of the modules in advance, which is why a method is necessary to consistently determine these properties.

### 2.2 *Graphs*

Many researchers have already explored the use of graph systems to represent design information in the Architecture, Engineering, and Construction (AEC) sector (Kolbeck et al., 2022). Generally, graphs are constructs consisting of vertices and edges that connect said vertices (Diestel, 1996), with any interconnected subset of the edges and vertices being called a subgraph. The interconnective nature of graphs allows them to represent complex, interconnected information.

Depending on the system chosen for storing and interacting with such graphs, vertices and edges can be labeled, providing context information for the specific item. The vertices and edges can also carry additional properties that specify the object or relationship. A graph adhering to these concepts can also be called a Labeled Property Graph (LPG). To describe objects within the design of a built asset, it is crucial to be able to append further information to the vertices.

Graphs are powerful structures that can represent the relationships between entities and methods for modeling the topology of geometric objects and finding patterns and subgraphs (Diestel, 1996). In conjunction with the semantics of LPGs, these capabilities provide the groundwork for modeling and representing the building model for our approach.

### 2.3 *Graph rewriting*

Algorithmic or automated design relies on being able to make certain changes to the design at certain stages of the design process. Graph rewriting provides such a method, originating from formal grammar. Essentially, graph rewriting relies on the definition of rewriting rules (sometimes called transformation rules) that can delete, modify, and add vertices and edges to a graph. These rules are, at their core, defined by their *left-hand-side (LHS)* and their *right-hand-side (RHS) (*Rozenberg, 1997), as seen in Fig. 1. The LHS is a graph pattern that is searched for in the main graph. Should a matching subgraph be found, it is replaced with the RHS of the rewriting rule. Once this has happened, the rewriting process is complete.

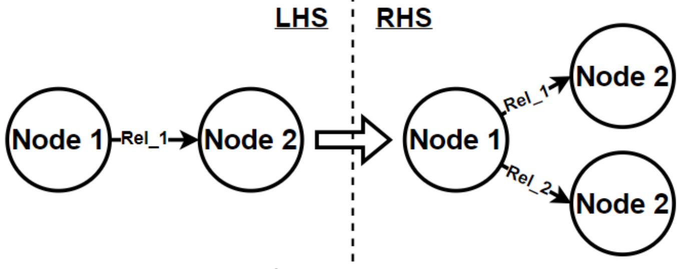

Figure 1. An example of a graph rewriting rule. The LHS is matched in the main graph and is then replaced by the RHS.

Critical here is the LHS with its graph pattern. It allows us to precisely define the necessary context in which the change is supposed to happen. This pattern matching allows us to ensure that changes only happen in specific parts of the model and consequently encapsulate engineering and design knowledge (Vilgertshofer, 2022). Just as an engineer knows that a column needs a foundation (or something structurally similar) to stand on, a rewriting rule can be defined that expresses this fact.

The usability of graph rewriting techniques in engineering domains has already been greatly researched, not only in the AEC domain (Campbell, 2009; Sangelkar & McAdams, 2017; Wang et al., 2020). A relevant selection will be explained in further detail in the next subsection.

### 2.4 *Graphs and their applications to describe geometric shapes of built structures*

Various research has been conducted in the scope of graph rewriting rules that modify geometric representations. Tessmann & Rossi (2019) introduced a method using modular units and topological interlocking to create structures. Their approach, implemented through a Grasshopper plugin called WASP, aggregates parts by aligning them to constrain all degrees of freedom, enabling load-bearing capabilities. This combinatorial design process sequentially combines basic parts into discrete assemblies, transforming objects so their interface planes face each other. The method employs explicit sequence descriptions, stochastic procedures, or gradient field-driven aggregations to arrange parts. This approach yields reversible joint modular assemblies, challenging conventional parametric design by offering a more sequential workflow where instructions are continuously executed until achieving the desired form.

Vilgertshofer & Borrmann (2017) utilize graph rewriting rules to automate infrastructure project planning across multiple levels of detail (LoD). Different LoDs are employed in various planning stages and domains. To streamline changes across LoDs, they use the GrGen.NET framework to consistently apply modifications. Challenges include the complex and error-prone manual definition of dependencies between models within conventional parametric environments (Vilgertshofer & Borrmann, 2017). Additionally, graph systems enable the independent representation of engineering knowledge regardless of CAD systems.

Kolbeck et al. (2023) investigate the application of graph systems for modular bridge structures, focusing on adaptable precast modules to enhance scale effects and mass customization for optimized production and planning. Unlike the approach discussed here, they utilize graph transformation directly, bypassing the need for rewriting rules. Changes in parameters are translated into graph transformations via a steering sketch, resembling conventional parametric modeling within a graph system framework. The authors suggest the feasibility of adapting graph grammar for their approach in future work.

Esser et al. (2022) advocate for graph-based systems and graph rewriting to perform version control of Building Information Models (BIM) models based on their underlying object structures. They propose the interaction with BIM data represented in the IFC data model as a graph, allowing model changes to be reflected through graph alterations. By converting the file-based model representations into graphs before and after modifications, differences between both versions are analyzed to generate incremental patches. Like graph rewriting rules, these patches facilitate an asynchronous cooperative workflow rather than jointly collaborating in a central model accessible to all project stakeholders. This approach is particularly beneficial for large-scale projects involving multiple contractors in the planning process and showcases the strengths of representing building models with a graph.

Abualdenien & Borrmann (2021) utilize a Parametric Building Graph (PBG) to identify patterns in BIM models for potential application across projects. Objects, relationships, and contextual information within a BIM authoring tool are captured to create rewriting/transformation rules. This enables the transformation of different projects into graph representations, facilitating pattern matching and deployment of architectural and engineering detail knowledge between projects. They highlight the significant cost and performance impact of successfully transferring detailing changes from one model to another.

### 2.5 *Research Gap*

The use of graph systems in the AEC domain has clearly identifiable advantages, as the previously mentioned scientific publications have detailed. The plethora of use cases enabled by the graph representations, graph rewriting, and transformations articulate the flexibility, interoperability, and scalability of using such systems.

Tessmann & Rossi (2019) explicitly used interface-based rewriting rules to algorithmically create their aggregations. Their methods of managing which and when rewriting rules are applied partially relied on explicit sequence definitions, meaning that the order in which the rules are to be applied is explicitly specified. Capturing and representing design procedures in this manner has thus already been demonstrated. Yet, their method does not allow for detailed control of intermediary and final results, i.e., the specification of fundamental parameters such as the number of levels or plot outline. This paper explores this method further by wrapping the sequence definitions within a process model that regulates the flow of the rewriting algorithm and provides the ability to directly modify assembled components. This way, the fulfillment of necessary conditions can be ensured with greater reliability.

## 3 METHOD

Fig. 2 illustrates the overall approach. Structures can be created by the sequential application of transformation rules based on a predefined set of components, rule definitions, and a process model. Our approach aims to result in an algorithm that can reliably and consistently produce models of built structures based on user-defined parameters. This is achieved by defining a component library with geometric parameters and their respective connecting interfaces. Based on this library, a catalog of rewriting rules is created that specifies how the individual components assemble. The component library and the rule catalog are wrapped inside an algorithm following a process model where the start state is set; rule sequences are defined and executed upon. Finally, the graph is interpreted to be read as a fully realized 3D model.

### 3.1 *Component libraries*

Fundamentally, components are used as the core building blocks of the design. Since this approach is embedded within the context of modular precast structures, such basic components are suitable for defining the structure's modules. The used components can be seen in Fig. 3:

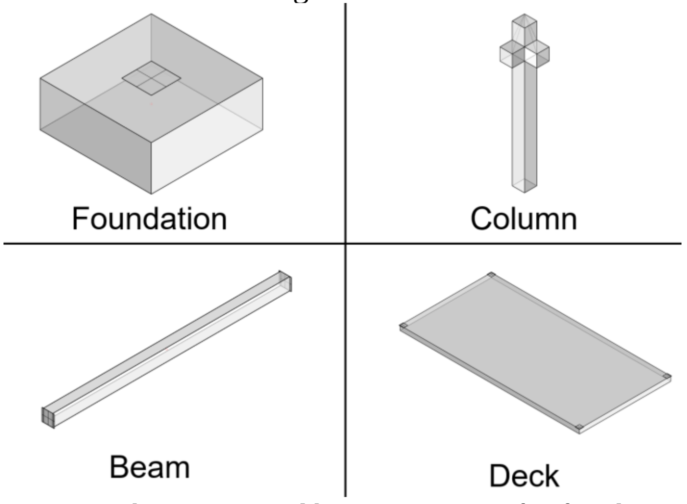

Figure 3. The component library consisting of a foundation, column, beam, and deck. Each component also carries geometric parameters as well as connecting surfaces (interfaces).

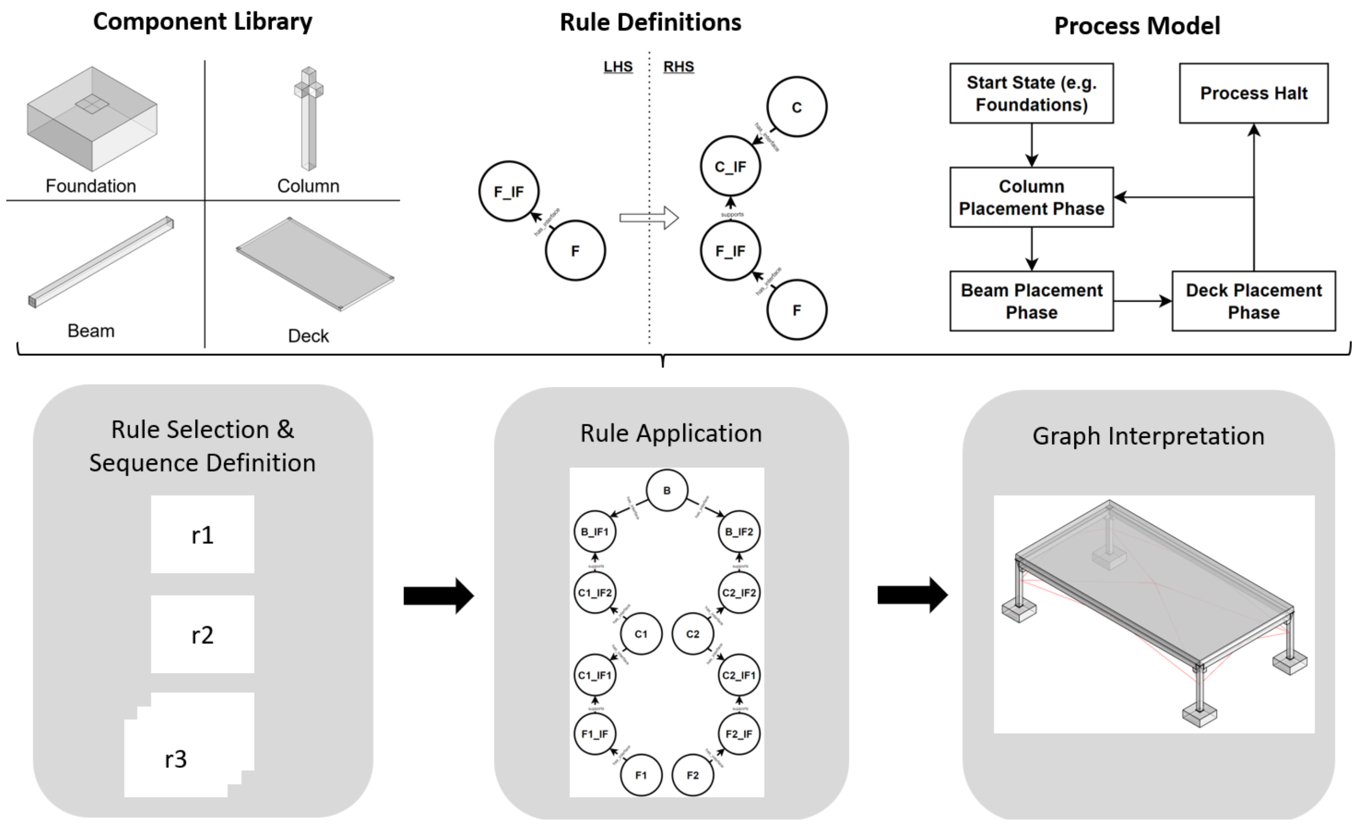

Figure 2. General overview of the approach.

The components include the necessary geometric information as well as geometric parameters that influence the dimensions of a component, such as length, width, and thickness. Furthermore, connection interfaces are defined for each of the components that indicate surfaces to which other components can connect, akin to what Tessmann & Rossi (2019) have done. A foundation, for example, may have its top surface defined as an interface where other components connect (such as a column) and so forth. For this approach, a small library of four parts was chosen to simplify the process and implementation as a whole. The selection of components still allows us to create basic structures, as can be seen in Fig. 4:

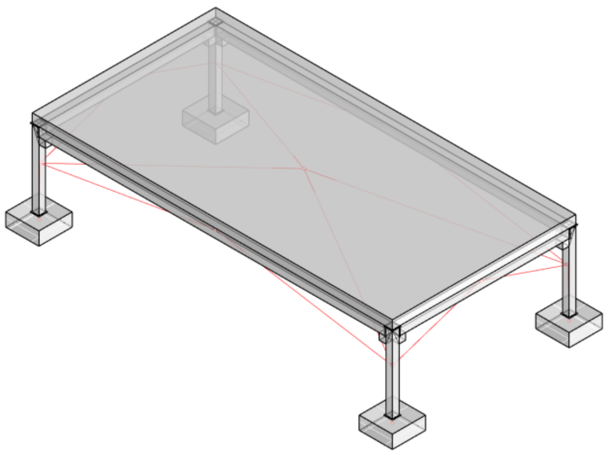
Figure 4. A basic modular structure made using the component library above (Fig. 3)

## 3.2 *Graph representation*

This approach uses an LPG to represent the various components and their connections with each other. Considering a structure like the one in Fig. 5, we can describe it through the means of a graph by creating a node for each component (e.g., for a column) and establishing relationships between the nodes/components that interface with each other for example, the column that stands on a foundation. A visualization of the graph representing the structure in Fig. 5 can be seen in Fig. 6.

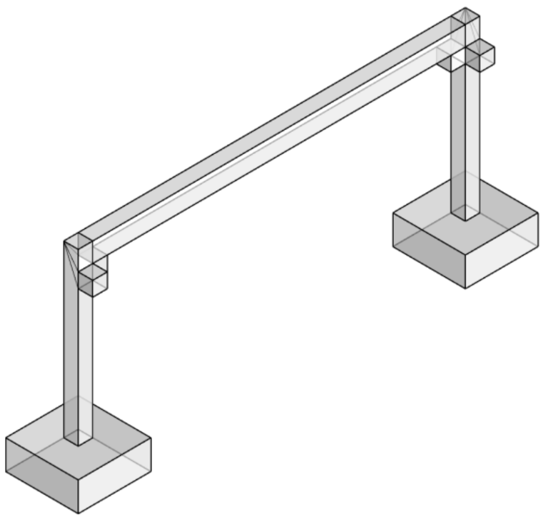
Figure 5. A simple structure consisting of two foundations, two columns, and a beam.

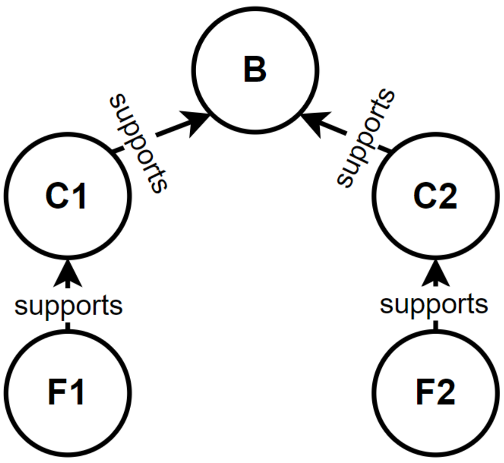

Figure 6. The previous structure represented by a graph. The beam (B) is connected to two columns (C), which are each connected to a foundation (F). All relationships carry the label *'supports'*.

Semantically, the relationships between the nodes (or the components they represent) are labeled with '*supports*', indicating their structural dependency.
Modular components within a model are suitable for being represented by nodes since their properties, such as their geometry and connectivity, can be accurately depicted by nodes inside an LPG. As mentioned earlier, this approach also uses a component library for the modules, defining the amount and nature of each component's interfaces. As previously discussed, the possible connections of a *component* were broken down into a certain number of specific *interfaces*. This allows us to model the graph's structure more granularly by also representing these interfaces with their own nodes. While a component's connection at its core is defined by the two components it connects, we can modify that definition by postulating that the interfaces are connected to each other and that each interface can only be connected to one, and exactly, one other interface. The interfaces are a core part of the component and thus, the node. This definition of a connection between two components ensures that the type of connection is always known (since it is tied to a specific interface). Since this changes the schema of our graph, an updated visualization can be seen in Fig. 7.
Here, the additional nodes represent the interfaces of the components. Columns and foundations do not have a direct connection to each other but are indirectly linked via their interfaces, the same as the beam. This way of increasing the granularity of the model and its graph representation allows us to make a more accurate description. Finding out which interfaces of a component are open or closed is easily done. It also offers us the groundwork to consistently and reliably apply changes to the graph by employing graph rewriting techniques.

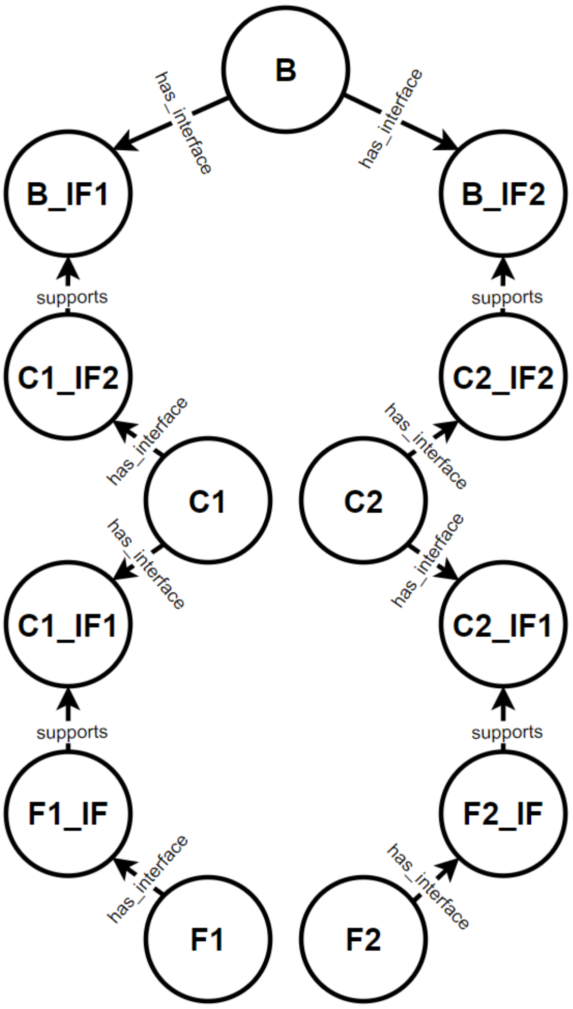

Figure 7. Two foundations (F1, F2) each connected with a column (C1, C2) via their respective interfaces (F1_IF, F2_IF, C1_IF, C2_IF). The connection from the columns to the beam (B, B_IF1/2) is analogous.

### 3.3 *Graph rewriting as a method of incrementally applying changes to the graph*

This approach makes use of graph rewriting as the primary technique to change the graph and, consequently, the model. To address the previously mentioned fact that rewriting rules can encapsulate engineering knowledge, let us look at the simple example in Fig. 8.
When wanting to add a column to the model, the graph also changes accordingly. This can be expressed as a rewriting rule that precisely defines that a column interface with its respective column node should be added to an open interface of a foundation. This rule is visualized in Fig. 9. This way, it can be ensured that a column may only connect to a foundation when this rule is applied. Further defining rules to include all possible connections the components can make, leads to a rule catalog that can express any possible design decision at any part of the project.

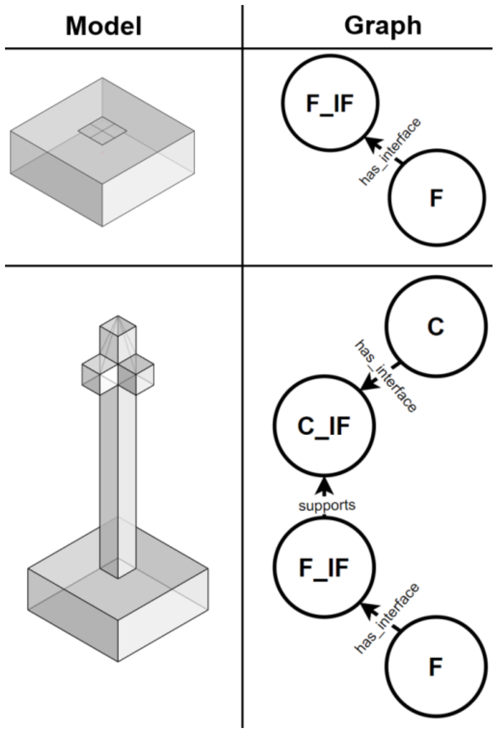

Figure 8. A graph representation of the model only containing a foundation (and its interface), as well as a graph representation of a model also containing a column.

This flexibility, consistency, and granularity of rewriting systems make them a strong candidate for the basis of automated design and form the foundation for this approach. With enough correctly defined transformation rules that describe the engineering knowledge to an adequate degree, the groundwork for incrementally applying changes to a model is laid out. Yet, while useful as a unit of change, the rules themselves still need a framework that controls *what* rule is applied at *which* point during the process.

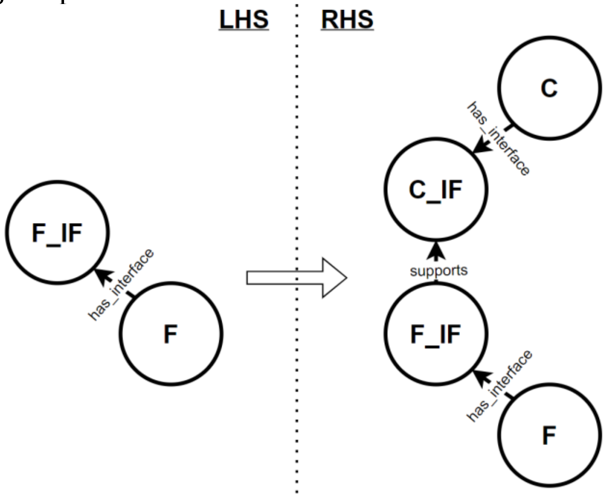

Figure 9. A rewriting rule that specifies a column (C) being added to a foundation (F) via their interfaces (C_IF, F_IF)

### 3.4 *Process modeling*

An explicit definition of the sequences in which the rewriting rules are applied is necessary to provide a method for users to interact with and understand the algorithm and ensure that the context for specific rules exists at a given point during the design process. A subdivision of the design process into smaller, more manageable parts can be achieved using a process model: Given a certain start state of the design (or *start symbol* to speak in graph terms), we can aggregate rule applications into packages that encapsulate certain steps during the design, as seen in Fig. 10:

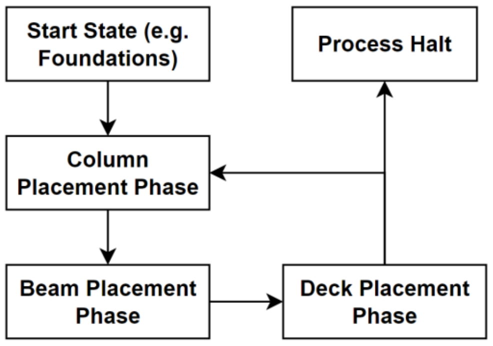

Figure 10. The process model begins with a defined start state and proceeds to process the various design phases. Design phases may loop, such as when more columns are placed upon the deck after it has been added to the model.

For example, given a start state of four foundations in the model, placing a column on each can be aggregated into one. This, in turn, provides us with the rule application sequence that takes place during this design phase. The placement of beams on these columns, as well as the deck on top of those, can also be aggregated into these design phases. Since the placement of the beams using rewriting rules necessitates columns with open interfaces, it is advisable to locate this design phase after the columns have been placed.
Furthermore, encapsulating these steps gives us the ability to directly specify the components' parameters that we have defined in the library, all at once. For example, if the height of the first floor is supposed to be 3 meters, the height parameters of all columns placed during their respective design phase can be set to 3 meters. This allows for greater control over the intermediary as well as the final results of the model, specifically to satisfy various geometrical conditions. The process model also gives us a structure in which the start state can be easily defined and set according to the user's wishes, e.g., a certain number of foundations.

### 3.5 *Geometrical and topological conditions*

Graphs, at their core, are purely topological constructs. This, in turn, means that rewriting rules only specify the topological context in which they can be applied. A rewriting rule may specify that a beam needs two columns, each with open interfaces to be placed. If successful, the resulting graph contains just that: A beam connected to two columns via its interfaces. Yet, one crucial condition has so far been

ignored: The beam needs to have the correct length to be supported by the columns. A rewriting rule cannot explicitly articulate this since it only concerns topological conditions. The geometrical conditions need to be taken care of differently.
During each design phase, the parameters of the components can be set to a specific value. By calculating the correct value in advance (pre-processing), the geometric condition can also be satisfied, with the beam having the correct length.

### 3.6 *Graph interpretation and conversion*

Once the various design phases have been processed and the final graph is finished, it needs to be interpreted and converted to a model. The model should contain the entire geometry of the placed components as well as the information concerning the connections between them. The latter is easily retrieved by investigating the relationships between the nodes inside the graph. If two interfaces share a relationship they are connected. As for geometry, there are a multitude of ways to store it and retrieve it from the graph. Geometric representations carry a significant amount of complexity, and they could be represented by an entire subgraph attached to the node of a specific component. Alternatively, the node can store a reference to the geometric information that describes the component. This can, for example, be stored and manipulated inside its own geometric system, such as a CAD program. This way, the modification of the components' geometry is offloaded to a different system so that the graph system only needs to consider the fulfillment of geometric conditions instead of the actual manipulation.

## 4 PROTOTYPICAL IMPLEMENTATION

To assert the viability of this approach, an implementation was developed using Rhino 7, Grasshopper, and the programming environment Microsoft .NET. Grasshopper and Rhino provide the necessary methods of visualizing, storing, and manipulating the model's geometries. Rhino offers an external compute library that can interface directly with .NET so that geometrical modifications can be made directly inside the program. At the same time, Grasshopper provides a visual computing language with which it is simple to create parametric geometries and store them inside Grasshopper files. Grasshopper also defines the connecting surfaces of the components which are later used as interfaces.
The program itself was developed using C# and .NET, with self-implemented classes representing the graph, nodes, and the process model. Inside the program, the rule catalog is defined, followed by the definition of the start state and the various design phases. Here, the various design phases are defined, each with a specific rule sequence as well as parameter setting. In the beam installation phase, for example, the rule of adding a beam to two columns is set to be executed a certain number of times, followed by setting the length parameter accordingly. This is done for all phases, after which the rewriting process begins. Each phase is executed successively, with the rewriting algorithm being invoked according to the rule sequence defined within. At the same time, the geometric parameters are changed within the grasshopper module. The graph can be converted to a model at the end of each phase, to check for consistency and errors.

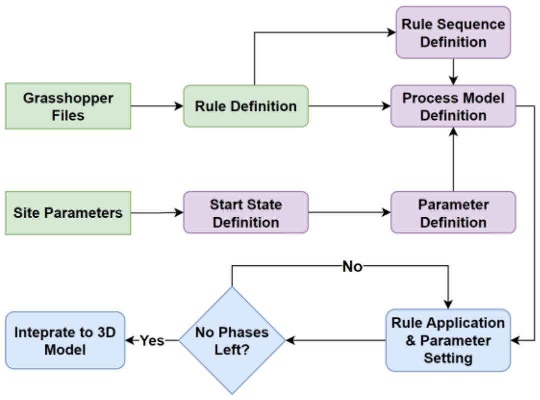

Figure 12. Overview of the algorithm. The rules and the start state are defined with the site parameters (plot size, etc.) and the grasshopper files (component definition). After parameters and rule sequences have been defined, the algorithm processes every design phase until there are none left. At this point, it interprets the graph and converts it to a 3D model.

## 5 RESULTS AND LIMITATIONS

The implementation successfully produces simple yet recognizable and usable structures. The process model, which provides control over geometric parameters and the applied rules, contributes the necessary framework so that the approach remains transparent and governable. A few resulting models of the algorithm can be seen in Fig. 13 below.

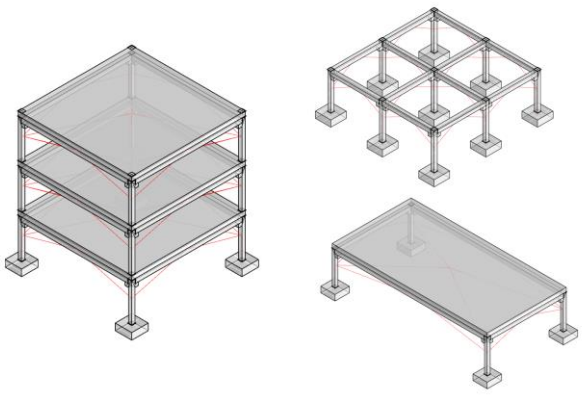

Figure 13. Various example resulting models of the algorithm. With different plot dimensions, number of levels, and fields. The red lines between the components indicate an established connection.

As for the limitations, the achieved geometric structures are of a low degree of detail. Issues may arise when especially the connections between the components become more detailed. The geometric conditions that need to be fulfilled for a proper joint, including bolts and more, may become increasingly difficult to manage with the approach of pre-processing the component's geometry. In general, the pre-processing step, while useful at first glance, does not fully describe the conditional geometric relationship between the components. To briefly describe an alternative approach, one could define the parameter implicitly as opposed to imperatively: Instead of calculating the length of a beam in advance and then setting it accordingly, one could define the length of a beam to always be the same distance as the distance of the two columns it is supported by. This kind of condition could be expressed by *predicates* and *directives*, a concept belonging to the *sortal grammars* (Stouffs, 2019), that can describe non-topological conditions that have to be met on either side of the rewriting rule.

## 6 CONCLUSION AND FINAL THOUGHTS

Graph rewriting techniques have proven that they are a powerful method of expressing and applying changes to a model. Yet, exploring methods on how to best implement them for automation purposes in construction has shown that while they are useful as a unit of change/modification, a framework in which they are applied in a controlled and consistent manner is necessary to achieve desirable outcomes, in this case, usable structures. At the same time, though, this framework also provides an interface for the user to directly influence the algorithm and maintain transparency and control over the final result, thus preventing a ‘black box’ with no transparency for the end-user.

For precast and prebuilt structures the approach seems especially promising since modular architecture is well suited to be represented by graphs, while their construction phases are easily represented by rewriting rules. Further development of this kind of approach may have significant implications for the prebuilt construction sector.

## 7 ACKNOWLEDGEMENTS

The authors gratefully acknowledge the financial support of the Deutsche Forschungsgemeinschaft (DFG) in the frame of the programs SPP 2187 “Adaptive modularized constructions made in a flux” (Project Number 423969184) and Transregio 277 “Additive Manufacturing in Construction—The Challenge of Large Scale” (Project Number 414265976).

## 8 REFERENCES

Abualdenien, J., & Borrmann, A. (2021). PBG: A parametric building graph capturing and transferring detailing patterns of building models. Proc. of the CIB W78 Conference.

Auer, D., Bos, F., Olabi, M., & Fischer, O. (2023). Fiber Reinforcement of 3D Printed Concrete by Material Extrusion Toolpaths Aligned to Principal Stress Trajectories. Open Conference Proceedings, 3. https://doi.org/10.52825/OCP.V3I.759

Campbell, M. (2009). *A Graph Grammar Methodology for Generative Systems*. http://hdl.handle.net/2152/6258

Chen, K., You, B., Zhang, Y., & Chen, Z. (2024). Automatic lift path planning of prefabricated building components using semantic BIM, improved A* and GA. Engineering, Construction and Architectural Management. https://doi.org/10.1108/ECAM-11-2023-1119

Diestel, R. (2017). Graph Theory (Vol. 173). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-662-53622-3

Esser, S., Vilgertshofer, S., & Borrmann, A. (2023). Version control for asynchronous BIM collaboration: Model merging through graph analysis and transformation. Automation in Construction, 155, 105063. https://doi.org/10.1016/J.AUTCON.2023.105063

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, June). Graph-based mass customisation of modular precast bridge systems. Proc. of the 30th Int. Conference on Intelligent Computing in Engineering (EG-ICE).

Preidel, C. (2020). *Automatisierte Konformitätsprüfung digitaler Bauwerksmodelle hinsichtlich geltender Normen und Richtlinien mit Hilfe einer visuellen Programmiersprache.* (Doctoral dissertation). *Technische Universität München.*

Rozenberg, G. (1997). Handbook of Graph Grammars and Computing by Graph Transformation. Handbook of Graph Grammars and Computing by Graph Transformation. https://doi.org/10.1142/3303

Sangelkar, S., & McAdams, D. (2017) Automated graph grammar generation for engineering design with frequent pattern mining. *International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*. Vol. 58127. American Society of Mechanical Engineers.

Stouffs, R. (2019). Shape Rule Types and Spatial Search. In J.-H. Lee (Ed.), Computer-Aided Architectural Design. “Hello, Culture” (pp. 474–488). Springer Singapore.

Tessmann, O., & Rossi, A. (2019). Geometry as interface: Parametric and combinatorial topological interlocking assemblies. Journal of Applied Mechanics, Transactions ASME, 86(11). https://doi.org/10.1115/1.4044606/960599

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

Wang, X. Y., Liu, Y. F., & Zhang, K. (2020). A graph grammar approach to the design and validation of floor plans. *The Computer Journal*, *63*(1), 137-150.