Interactive Procedural Design Exploration for Modular Structures

by

John Huanshuo Rao

Submitted to the
Department of Architecture
in Partial Fulfillment of the Requirements for the Degree of

Bachelor of Science in Architecture
Bachelor of Science in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

May 2022

Signature of Author:_____

Department of Architecture
May 25, 2022

Certified by:_____

Caitlin T. Mueller
Ford International Career Development Professor
Associate Professor of Architecture
Associate Professor of Civil and Environmental Engineering
Thesis Supervisor

Accepted by:_____

Leslie K. Norford
Professor of Building Technology
Chair, Department Undergraduate Curriculum Committee

Interactive Procedural Design Exploration
for Modular Structures

by

John Huanshuo Rao

Submitted to the Department of Architecture on May 25, 2022
in Partial Fulfillment of the Requirements for the Degree of
Bachelor of Science in Architecture
Bachelor of Science in Electrical Engineering and Computer Science

Abstract

This thesis presents a grammar-based methodology for generating and evaluating structures that are constructed as aggregations of modular units. Using modular units as a building system can be more efficient for construction and potentially high performing structurally. Most of modular structures today are built in simple stacks which clearly advantages construction efficiency and the structural load transfer. However, other more complex configurations of modules might better address other important design factors such as daylight availability and the creative design intent of the architects. With the goal of expanding the design exploration process for modular structures, this thesis proposes a new methodology that integrates procedural design generation using shape grammars and structural performance evaluation using finite element analysis.

Algorithmically, this paper takes inspiration from recent advances in discrete modeling tools. Under the existing frameworks, aggregations can be generated following either stochastic procedures or deterministic procedures. However, using deterministic systems often yields expected results with limited diversity while using stochastic systems does not give designers direct control over the generation process. By controlling the stochasticity of the generation process based on user feedback and performance goals, the methodology proposed in this thesis generates design options that follow specific design intent yet provides unexpected results.

Thesis Advisor:
Caitlin T. Mueller

Title:
Ford International Career Development Professor
Associate Professor of Architecture
Associate Professor of Civil and Environmental Engineering

Acknowledgments

This thesis would not have possible without the guidance and support from everyone who has been a part of my journey at MIT. First and foremost, I must express my immense gratitude for my academic and thesis advisor, Caitlin Mueller, who has inspired me throughout my time at MIT with her creativity and intellectual curiosity. During the development of this thesis, Caitlin has offered me thoughtful feedback and valuable advice. In the past five years at MIT, she has guided me through my academic work, career planning, and even personal life. I am truly grateful for all the insightful conversations we have had. Caitlin, thank you for always being there for me, not only as a great advisor, but also as a great friend.

I would also like to thank Keith Lee, Yijiang Huang, and everyone from the Digital Structures group for sharing and inspiring me with their work. In particular, I thank Keith for generously sharing his work and experience which helped me start this thesis. I thank Yijiang for his technical expertise that helped address many issues in the implementation process of this thesis. I am grateful for all faculty and staff of the Department of Architecture. I would not be where I am today without their support and guidance throughout the entirety of my time at MIT.

I am indebted to all my friends for their unconditional support and love that kept me going even during the roughest times. My experience at MIT would have been incomplete without the memories we have had together. Joyce, thank you for always being there for me and believing in me. To Catherine, Jackie, Caleb and Dongnyung, thank you for constantly stimulating my passion for design and making all the late nights in studio one of the most unforgettable parts of my last five years.

Most importantly, I would like to express my utmost gratitude for my parents, Junfeng Rao and Xin Zhou, and everyone in my family for their unending support and dedication to my education. Their unconditional love has given me the strength to push myself through all the challenges during my time at MIT. My mom cultivated my creativity at a young age while my dad taught me how to challenge myself intellectually. They have continuously inspired me with their wisdom and passion.

And finally, I thank Davey for always smiling during my toughest of times.

Table of Contents

Chapter 1

Introduction

This thesis presents a new computational methodology that enables designers to efficiently explore a diverse set of design possibilities for modular structures. This chapter introduces the motivation behind this research with a discussion around the historical interpretations of modularity in architecture and its present day challenges.

1.1 Interpretation of Modularity in Architecture

Modularity has been a constantly evolving concept in architecture since the mid-20th century. Empowered by the industrial revolution and driven by the demands of post-World War II period, building systems consisted of standardized units, "modules", became an area of interest in modern architecture. Notably, the *Metabolist* movement, launched by a group of young Japanese architects in 1960, introduced the notion of buildings as living organisms that evolve with the urban environment. Emphasizing the idea of dynamic, adaptable, and replaceable living space, the Metabolists envisioned systems of *megastructures* with hundreds of individual units attached and subject to frequent replacement (Lin, 2011).

Figure 1.1.1. Notable examples of modular architecture and some of the design concepts that are commonly associated with modularity in architecture.

Kisho Kurokawa's Nagakin Capsule Tower is perhaps one of the most well-known examples that materialized this vision, to some extent. Situated in middle of the bustling neighborhood of Ginza, Tokyo, this 14-story apartment building is comprised of 144 standardized units plugged into two larger shafts at varying angles and offsets. These capsules resemble shipping containers in size, and they were all equipped with built-in furnishings. In response to the rapid urbanization

around the world, Nagakin Tower suggested a new model of affordable micro-living in cities. Though Kurokawa initially intended the capsules to be replaced within thirty-five years, the plan was never realized. In the 50 years since its completion, not a single unit was replaced due to various financial and construction challenges. In recent years, the building has fallen into disrepair due to poor maintenance and became essentially uninhabitable. Controversy surrounding Nagakin Tower has perhaps gloomed its early glory and plans of its demolition have been underway for more than a decade (Lin, 2011). In the spring of 2021, it was announced that Nagakin Tower would eventually be demolished and replaced by a completely new building. Despite its impending doom, Nagakin Tower remains an inspiring image of modular design, with its aspirations to be a form of architecture that reacts and responds to the constantly shifting needs of the society (Dara & Sinclair, 2018).

Another exemplary interpretation of modular architecture is manifested by Mosh Safdie's Habitat 67. In comparison to the Nagakin Tower, Habitat 67 presents a different building system in which modules are stacked on each other to form complex and imaginative aggregations. Safdie envisioned Habitat 67 "as a system not a building" (Safdie, 1970). It challenged the "inhumane" approach to building characterless urban towers, and established an unprecedented model of quality urban living. The creative geometric arrangement of 354 identical modules created 146 residences of varying sizes and configurations that catered to the individual needs of the inhabitants. It addressed people's need for community, nature, light, and space. Safdie perceived modularity in architecture as an adaptable and expandable building system that adjusts to the humans inhabiting the space.

Though there have been many other interpretations of modular structures focusing on different design concepts such as the temporality and transformability of architecture, the system of stacking modules on top of each other seems to be the more widely adopted model than the *Metabolist megastructures*, "because, paradoxically, the rigorous megastructure-capsule distinctions offer little flexibility in terms of occupancy and structural expansion" (Lin, 2011). Constructing a secondary structural system would clearly reduce the financial and constructional

benefits of modular structures. Therefore, this thesis focuses on designing modular architecture as systems of independently stacked sub-assemblies.

1.2 Motivation

Nagakin Capsule Tower and Habitat 67 both embodied visionary design thinking and ideologies that were well-ahead of their time. However, contemporary modular architecture seems to be largely focused on the quantitative benefits (Dara & Sinclair, 2018). Using modular building system has become increasingly popular in addressing issues like post-disaster rebuilding and affordable housing crisis (Thompson, 2019). As a result, designs of modular structures seen today are often optimized for construction efficiency, costs, and structural performance, often resulting in monolithic stacking of boxes. Simply stacked forms can be the most intuitive solutions to address the quantitative goals, but more complex aggregations of modules might be more desirable options in terms of manifesting the creative intent of the designer and providing access to daylight and outdoor space for the inhabitants. The lack of overlap between these two design approaches can be exemplified by the contrast between the Huoshenshan Hospital in Wuhan and Habitat 67 in Montreal.

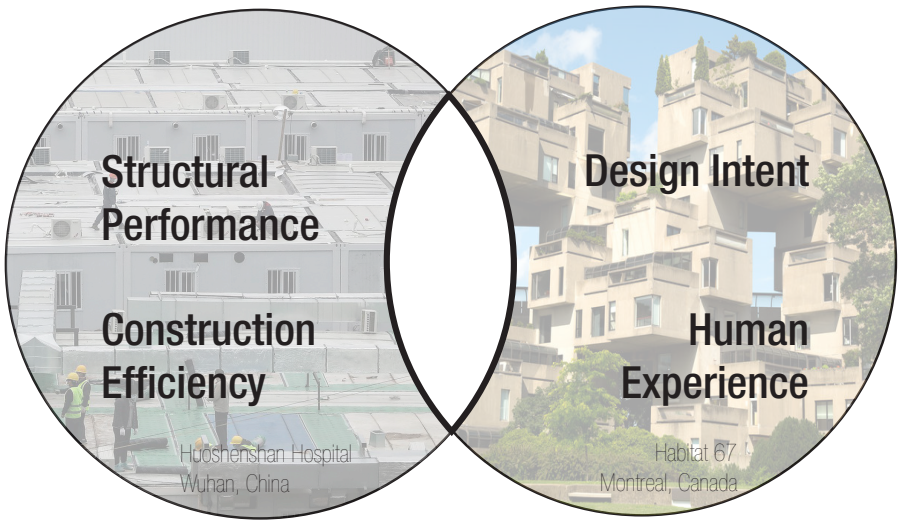


Figure 1.2.1. Lack of overlap between optimizing for quantitative and qualitative goals in modular design seen today.

Huoshenshan Hospital was designed and built in early 2020 to accommodate the intense demand for hospital care due to the outbreak of COVID-19 in Wuhan. Constructed in 9 days, Huoshenshan was the earliest and fastest architectural response to the global pandemic (Luo et al, 2020). Using prefabricated modular units not only enabled fast construction but also provided the flexibility to effectively organize the layout into multiple levels of contamination zones, preventing the spread of the virus. In this case, modularity served its purpose by efficiently addressing the need of the time. However, Huoshenshan Hospital can perhaps only be considered as a temporary design solution with little architectural significance. Its lack of emphasis on patient experience and aesthetic appeal hinders it from being converted to a permanent hospital. On the other hand, Habitat 67 is visually interesting and provided a novel urban living experience for the rapidly growing cities in the 1960s. Yet, the construction of the project took over three years to complete and came with an extremely high price tag (Safdie, 1970).

Building complex aggregations with simple standardized units poses both opportunities and challenges. It can be challenging to understand the structural behaviors of modular systems once the design of such system becomes more geometrically complex. For instance, if the system involves a lot of cantilevering or irregular rotations, it becomes difficult to understand and visualize the structural load transfers within the system. Difficulty optimizing structural efficiency is a factor that restricts the creativity of designers while designing with modular units. It is difficult for architects to balance quantitative goals and qualitative goals of a design without a thorough understanding of the structural behavior of modular systems. Although many structural analysis software tools allow designers to determine the performance of a particular design, they are often only used in the post-design rationalization process, independent of the creative design process. It is impractical to manually model and test all potential early design concepts. Especially, in a high-dimensional design space, there are many variables, such as the offset distances and the rotation angles between modules, that can have unexpectedly large impact on the structural performance of the whole system.

What if there was an opportunity to fully integrate the quantitative rationalization and qualitative

design exploration process? Is it possible to harness the generative power of computational design to automatically generate potential design solutions and structural feedback, allowing designers to spend more time on iterating through different concepts? The purpose of this thesis is to develop a new computational methodology that could help architects design, iterate, and evaluate the quantitative and qualitative characteristics of complex modular structures following specific design intent. The proposed methodology in this thesis also presents the opportunity to expand the creative capacity of designers by discovering new design possibilities. Fig. 1.2.2 illustrates the conceptual framework of the methodology.

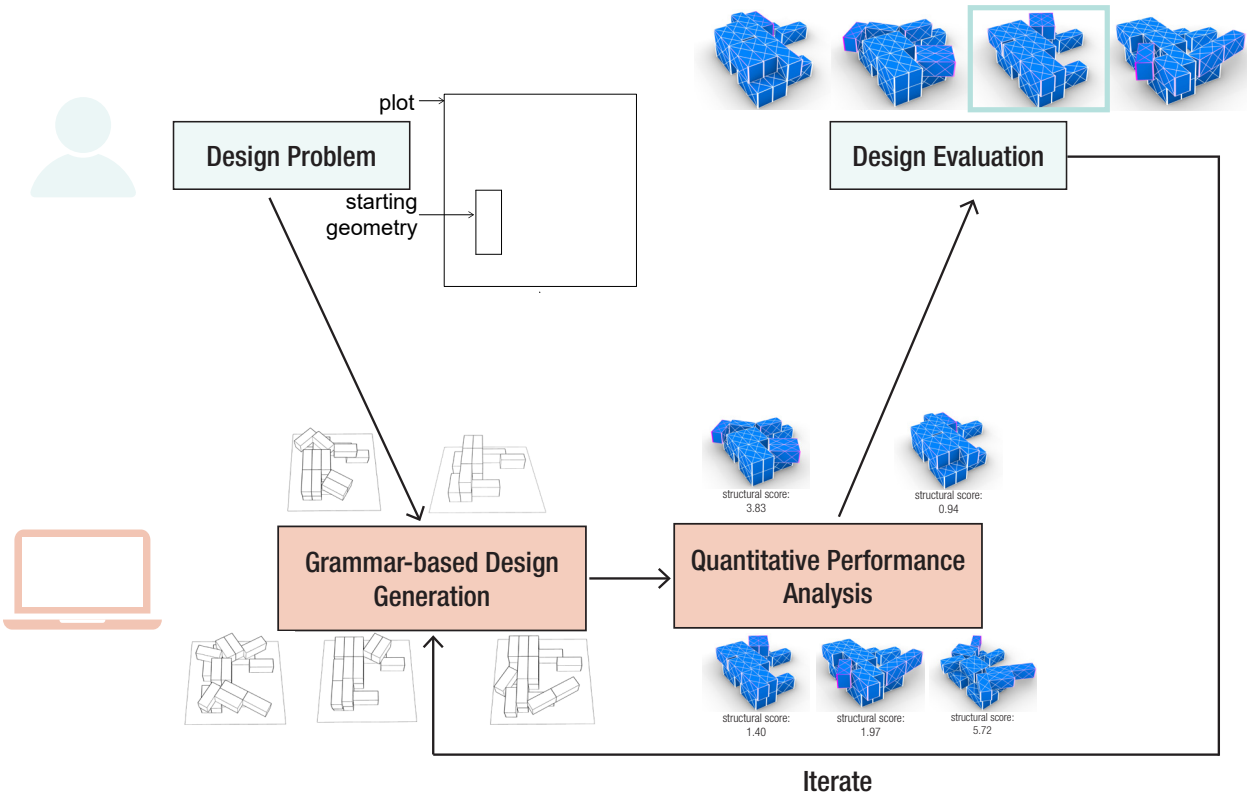


Figure 1.2.2. Conceptual framework of the methodology proposed in this thesis. This method focuses on establishing a collaborative and iterative process between human designers and computers.

Chapter 2

Background

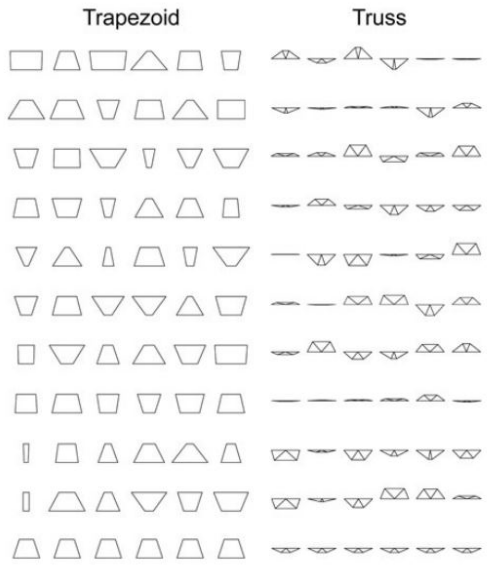
This chapter discusses existing research in grammar-based geometry generation methods and identifies specific needs for further research to develop a methodology that can be effectively integrated into the design exploration process for modular structures.

2.1 Related Works

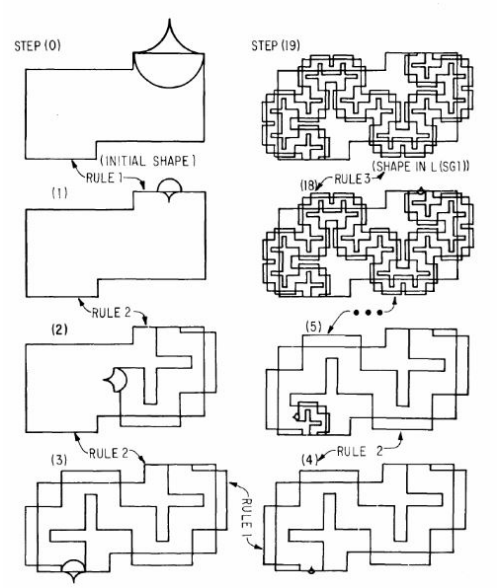
This section presents existing computational methods and tools used design and computer graphics that inspired the work in this thesis.

2.1.1 Common Paradigms of Computational Design

Existing computational design methods can be categorized into two common paradigms: the parametric approach, and the procedural approach (Fig. 2.1.1). In parametric design, specific aspect of a geometry can be manipulated through a set of parameters, or variables, and each design can be mathematically represented as a point in a design space. *Grasshopper*, a plug-in for 3D modeling software *Rhinoceros,* was developed around this idea in which users can design and edit complex shapes by adjusting defined parameters.



Parametric design study
(Brown & Mueller, 2019)



Generation of a shape using shape-grammar
(Stiny & Gips, 1971)

Figure 2.1.1.The two common paradigms of computational design: parametric methods and procedural methods (shape-grammar).

In the case of generating modular design options, the simplest approach to designing an efficient algorithm that computes different assemblies is to parameterize certain inputs and sample a large set of parameter points across the design space to output different design options. Using a conventional parametric design paradigm like this would be relatively easy to implement. However, parametric design solutions are often in the same family of limited variety, since the

number of solutions are limited by the number of parameters and the range of each parameter (Lee, Fivet, & Mueller, 2015). Though it is possible to define a parameter-based design space that covers broader diversity in possible solutions, with extensive expertise, it is often not practical to do so at the conceptual design stage before overall formal strategies have been decided upon (Mueller C. T., 2014). This is especially limiting in the case of exploring modular design options, since the question concerns both the geometry and the topology of the aggregations.

One way that effectively addresses this limitation is using a procedural approach, or *shape-grammar.* First introduced by George Stiny at MIT and James Gips from Stanford in the 1970s (Stiny & Gips, 1971), shape-grammar is defined as a set of rules that dictate the transformations of geometries. These rule sets can be used to describe a design language rather than a specific design. For example, one rule could specify how and when a module can be attached to another module. Using shape-grammars in place of a simple parameter-based framework has many advantages. Firstly, automatic form generation can be easily achieved following the desired logic and objectives (Stiny & Gips, 1971). Secondly, instead of defining large sets of parameters, grammars can concisely represent different classes of models. Since rules can be applied repeatedly and recursively, using a small set of rules can potentially generate infinite number of design possibilities. In contrast to generating design by adjusting predefined parameters, the diversity of solutions is not bounded by the range or size of the parameter set. Furthermore, even with very simple building blocks and a simple set of rules, it is still possible to find complex and diverse forms of unexpected designs if certain randomness is applied during the application of the rules (Brunn et al, 2021).

Implementing the generation procedure using shape-grammar also presents the opportunity to restrict generated design solutions to those that are constructable. Specifically, rules that dictate the transformation and aggregation of the structures can be designed to follow common construction logic, thus only producing solutions that are feasible to construct. In *Making Grammars,* Terry Knight and George Stiny discuss in detail how to expand the theories of shape grammars into the making of physical “things” (Knight & Stiny, 2015). Hence, adopting a

grammar-based system is the more suitable candidate for the purpose of this thesis.

2.1.2 Automatic Grammar Exploration

Grammars in existing architectural literature, are typically defined abstractly, and therefore designed to be applied by humans manually (Wonka et al, 2006). Rule-based systems, or grammars, are widely researched and explored in many other fields. In computer graphics, this approach is sometimes referred to as *procedural modeling*. A famous procedural modeling method, the L-system, was first proposed by biologist Aristid Lindenmeyer as a formal approach of describing the growth process of biological developments. Since then, it has been commonly extended to geometry generation of various types of models, such as plants, textures, buildings, and even cities. Using a procedural approach based on L-system to model buildings and cities allows the consideration of global goals and local constraints, such as building programs and zoning rules (Parish & Muller, 2001). *CGA shape,* another shape-grammar based modeling system, was developed to produce extensive architectural models with high visual quality and details, specifically for computer games and movies (Wonka et al, 2006).

Figure 2.1.2. Generation of a building using an extended L-system, as presented in (Parish & Muller, 2001).

Though efficient and robust, L-system based procedural modeling systems tend to focus on the automatic generation process of complex geometries that “resemble” scenes of building and cities rather than the creative design process of specific architectural forms. The algorithm, *Model Synthesis,* proposed by Paul Merrell and Dinesh Manocha (Merrell & Manocha, 2010) addresses this question, to some extent, by controlling the generation outputs to resemble user-defined input models (Fig. 2.1.3). However, this technique is most effective only when designers have clear visions for the overall topology of the modular aggregations. Furthermore, most existing procedural modeling techniques find new forms of geometry through a process of

"morphing" in which one shape is refined into another more detailed shape. This thesis, however, attempts to focus on generating complex forms by aggregating simple components, and develops a system that can be inherently integrated into an iterative process for designing real structures.

(a) Example Model (b) Generated Model

Figure 2.1.3. Model Synthesis example. The model on the left is a user-input model and the image on the right shows the generated output that resembles the user specified form (Merrell & Manocha, 2010).

Many recent developments in assembly-driven algorithms for designing spatial structures are closely related to the goal of this thesis. Expanded from a protein folding algorithm, *trussfold* is a growth-based, assembly-driven truss design algorithm recently proposed by Keith Lee at MIT (Lee & Mueller, 2021). Starting with a triangular lattice, the algorithm would search for the placement of next node and it would compute the performance scores, according to metrics defined by the initial algorithm, of all possible conformations growing from the input partial chain (Lee & Mueller, 2021). The resulting conformations would be categorized based on their quantitative performances and a set of the best performing solutions would be propagated into the next iteration, until a set of possible final solutions is found (Fig. 2.1.4). This thesis takes inspiration from this iterative growth procedure developed in *trussfold*. However, as discussed later in Section 3.4.2, each module in this thesis consists of a significant number of nodes and elements, hence the growth process is much more complex than the *trussfold* algorithm.

Algorithmically, this thesis also takes inspiration from recent advances in discrete modeling tools, such as *Wasp*, a framework developed to model objects as assemblies of discrete modular units (Rossi & Tessmann, 2019). Under the *Wasp* framework, aggregations can be generated following multiple different procedures: stochastic aggregation, explicit aggregation description,

geometry-driven aggregation, and field-driven aggregation (Rossi & Tessmann, *Designing with Digital Materials,* 2017). Each of these procedures has its own advantage and disadvantages. Developed as a plug-in for *Rhino Grasshopper*, *Wasp* allows users to define different forms of geometries as different modular units and the specific types of connections between each module. Using the stochastic generation procedure allows users to find diverse solutions of unexpected forms but it reduces user's control on the outcomes (Rossi & Tessmann, *Designing with Digital Materials,* 2017). Explicit descriptions allow users to manually apply rules to drive the form of generated results, but this process can be highly inefficient and tedious. Geometry and field-driven aggregations use user-defined global geometries or scalar fields to drive the rule selection and aggregation growth (Rossi & Tessmann, *Designing with Digital Materials,* 2017). However, after some experimentation with the *Wasp* system, it was found that using field-driven aggregation essentially reduces the problem to a parametric-based design space where the outcome of a specific geometry input is deterministic. Since the number of connection types between modules is limited, the diversity of possible outcomes is great greatly reduced. For instance, each specific rule states that one module can only be connected at a very specific point along one of its faces. Attempting to broaden the design space using this approach can result in the rule sets becoming overly complex and difficult to understand. Thus, further innovation and research is needed to expand these techniques to generate complex modular structure options that follow specific design intentions yet provide unexpected results.

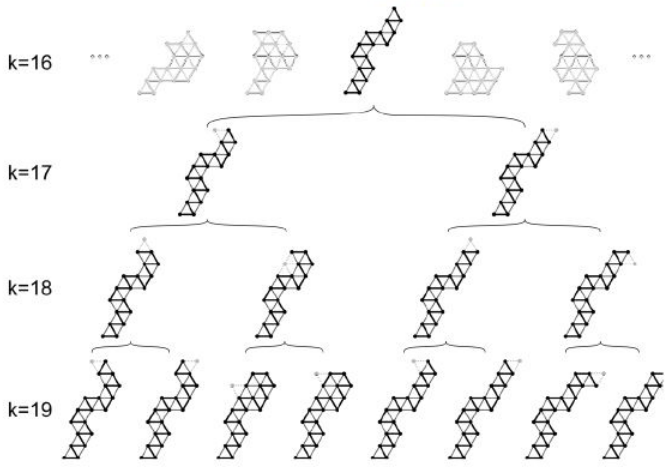


Figure 2.1.4. *Trussfold* - growth-based spatial truss design based on computational protein folding logic (Lee & Muller, 2021).

2.1.3 Human Interaction

As discussed in the previous section (Section 2.1.2), there are exiting computational tools that can efficiently generate geometries and optimize the topology based on quantitative performance but they are not intentionally designed for a collaborative experience between human designers and computers. Under these frameworks, more qualitative concerns are often only considered at the beginning of the generation process. *Human-robot collaboration*, a fabrication framework for the design and construction of spatial structures (Brunn et al, 2021), highlights the potential of enabling active collaboration between machines and humans during the creative process. Using two robotic arms to cooperatively to aggregate an unplanned structure made of a collection of spherical units, Brunn's paper describes a "design as you build" process in which the final form of the resulting structure is driven by both robotic inputs, as path-planning constraints, and human evaluation.

Caitlin Mueller and John Oschsendorf at MIT proposed an interactive evolutionary algorithm, *structureFIT,* that integrates user interaction to control the diversity of solutions for conceptual structural designs (Mueller & Ochsendorf, 2015). This technique highlights a promising way of giving users direct control and access to more than a single optimized solution. Taking inspiration from this algorithm, this thesis proposed a system where users can define their design preferences and a set of parameters to control the stochasticity of the generated designs. To encourage an iterative creative process, this system is designed to generate assemblies in a layer-by-layer construction process (detailed in Section 3.2.4), in which designers can evaluate the outputs and adjust the design problem accordingly between each layer of generation.

2.2 Research Gap and Opportunities

So far, this chapter has shown that existing work in grammar-based geometry generation methods are inspiring, yet not sufficient to achieve the goal of this thesis on their own. This

section summarizes the research goals of this thesis and potential ways to combine and expand on the different strategies presented in the chapter.

Based on the review of works discussed in Section 2.1, several main challenges can be identified:

- Parametric design systems and deterministic grammar systems can give designers direct control of generated design solutions, but they often lack the diversity that is critical at the conceptual design stage.
- Known procedural modeling algorithms can efficiently produce large number of geometries, but they are suited as a part of iterative creative design process of interesting architectural forms.
- Design optimization algorithms are normally driven by quantitative performance instead of promoting a collaborative process between human designers and machines.
- Aggregation-based form generation methods are not as widely researched, and their efficiency often becomes a limiting factor. For instance, generating large assemblies with *Wasp* can sometimes take up to several seconds per assembly, since it constantly attempts to solve geometric constraint problems, such as collision detection, using *3D meshes* of the modular units.

Attempting to resolve these challenges, this thesis presents a few specific research goals:

- Expand upon existing work in procedural modeling and other grammar-based systems for developing a computational approach to automatic explore design options for modular aggregations.
- Design a set of simple and easily understandable grammar rules that can create diverse and unexpected design options that respond to specific constraints.
- Define new metrics that can be used by designers to evaluate both quantitative and qualitative design factors such as structural performance, daylight performance, porosity, and geometric complexity.
- Develop strategies to allow designers to control a stochastic generation process and direct

the formal characteristics of design outputs.

- Implement an efficient generation system in which users can iteratively collaborate with computers throughout the entire creative process.

Chapter 3

Modular Aggregation Grammar

As discussed in Chapter 2, this thesis proposes a grammar-based generation method to increase the diversity of the output designs. This chapter introduces the overall framework of the proposed method and the definition of the grammar used in exploring the design space. Specifically, this chapter focuses on the underlying mechanisms of the system: setup of user specifications and design constraints, production rules of the grammar, geometric utilities for evaluating design outputs, and the definition of quantitative performance metrics.

3.1 System Definitions

How do the different components of the system interact with each other? How are design preferences and constraints defined through user inputs? How does the computer interpret the user inputs? This section presents the high-level framework and the definition of geometries throughout the generation process. Though this system is implemented considering modules of specific scale and form, this approach can be extended and adapted for modules of any scale and form.

3.1.1 Framework Overview

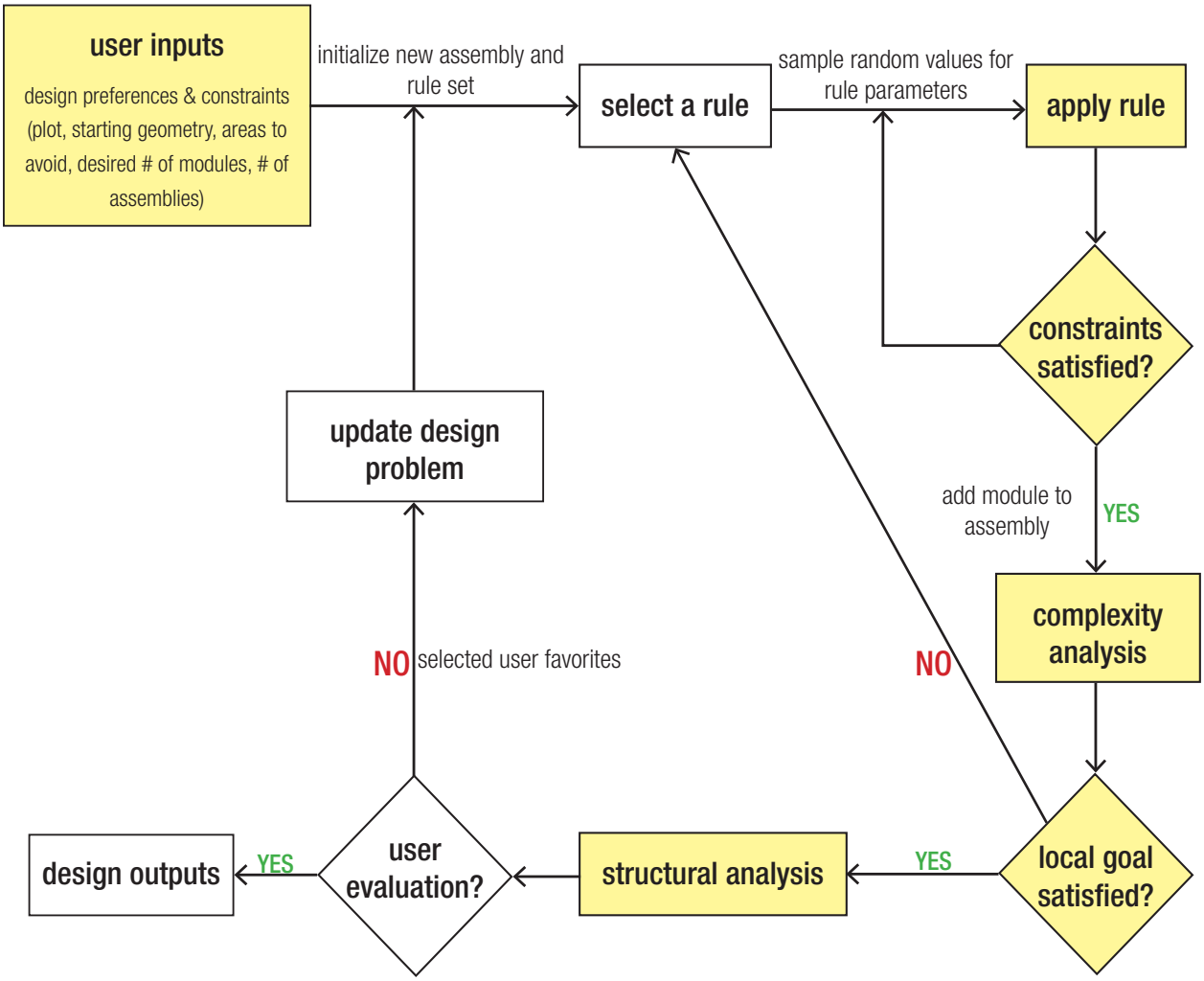


Figure 3.1.1. System flowchart. This diagram illustrates how different components of the system are connected at a high-level. This chapter focuses on discussing the highlighted components. Details of other components are presented in later chapters.

In contrast to some of the existing methods discussed in Chapter 2, the approach presented in this thesis attempts to build an iterative collaboration between human designers and computers. A user of the system would first set up the design problem for the system with specific inputs, as shown in Fig, 3.1.1. Section 3.1.3 discusses how users can specify these inputs and how they can influence the outputs of the system.

Once the system validates all user inputs, it would begin the generation through the process of selecting a rule, applying the selected rule, constraint checking, and quantitative evaluation. This process is repeated until the design goal is satisfied. Specifically, this method generates assemblies of modules through a layer-by-layer construction approach, meaning that each floor of the structure is completed before the generation of the next floor begins. In this case, the *local goal* is satisfied when the current floor contains the desired number of modules specified by the user. This approach is advantageous because it follows the common construction logic of modular structures and provides designers the opportunity to interact with the system by adjusting the design problem between each layer of generation to achieve desirable global form. Designers can iterate through different design concepts with different sets of user inputs until they are satisfied with the outputs from the system.

As shown in Fig. 3.1.1, this proposed method presents a stochastic approach of deploying the grammar. Since the rules are selected randomly, diverse sequences of actions can be generated with the same user inputs and only a small number of rules. Since each rule is also associated with a set of random parameters, a diverse range of designs can be automatically generated using the same rule sequence. This enables the system to explore large portions of the design space and potentially yield infinite number of possible design solutions. The process of selecting random rule and parameters are detailed in Chapter 4.

3.1.2 Geometry Definitions

This section discusses some of the definitions and terminologies commonly used in this thesis. A *module* is defined as a single modular unit. An *assembly* is defined as the group of all modules

forming an modular structure. How and where would a new *module* be added to the existing *assembly*? Applying an *aggregation-based* generation method requires some understanding of the spatial relation of individual units in the existing design. Visually interpreting the form and relationship between discrete geometries can be a simple task for humans. However, it can be difficult for computers to efficiently determine the relations within an assembly and apply the appropriate transformation. To simplify this process, the implementation of the system defines each module with a set of parameters in the *global coordinate system*: the *dimensions*, the *origin,* and the *rotation angle.*

Considering rectangular modules, the *dimensions* of each module describe its length, width, and height. By default, the dimensions of modules in the *starting geometry* are assigned to all modules in the same assembly. However, it is possible for users to specify new dimensions throughout the generation process as desired.

The *origin* of a module describes the *Cartesian coordinate* of one of the corners on its *bottom* face. As illustrated in Fig. 3.1.2, each module is initially modeled with its length along the *global y-axis;* its width along the *global x-axis,* and its height along the *global z-axis*. In this initial orientation, the *origin* is defied by its bottom left vertex of its bottom face. The *rotation angle* describes each module's *degree* of rotation, counter-clock wise around its *origin,* from the initial orientation with respect to the *positive y-axis* (Fig. 3.1.2).

This method of defining modules with parameters describing its relation to a *global coordinate system* can be generalized to accommodate modules of other forms or even assemblies of different types of modules, as long as the definition is consistent within an assembly. Since all modules can be described with a simple set of numbers, generating a new assembly at each step only involves computing the proper range for each one of these parameters. This computation process is presented in Section 3.2 and Section 3.3. Any design solutions can also be easily reconstructed as long as the parameters of all modules in the assembly of interest are properly recorded.

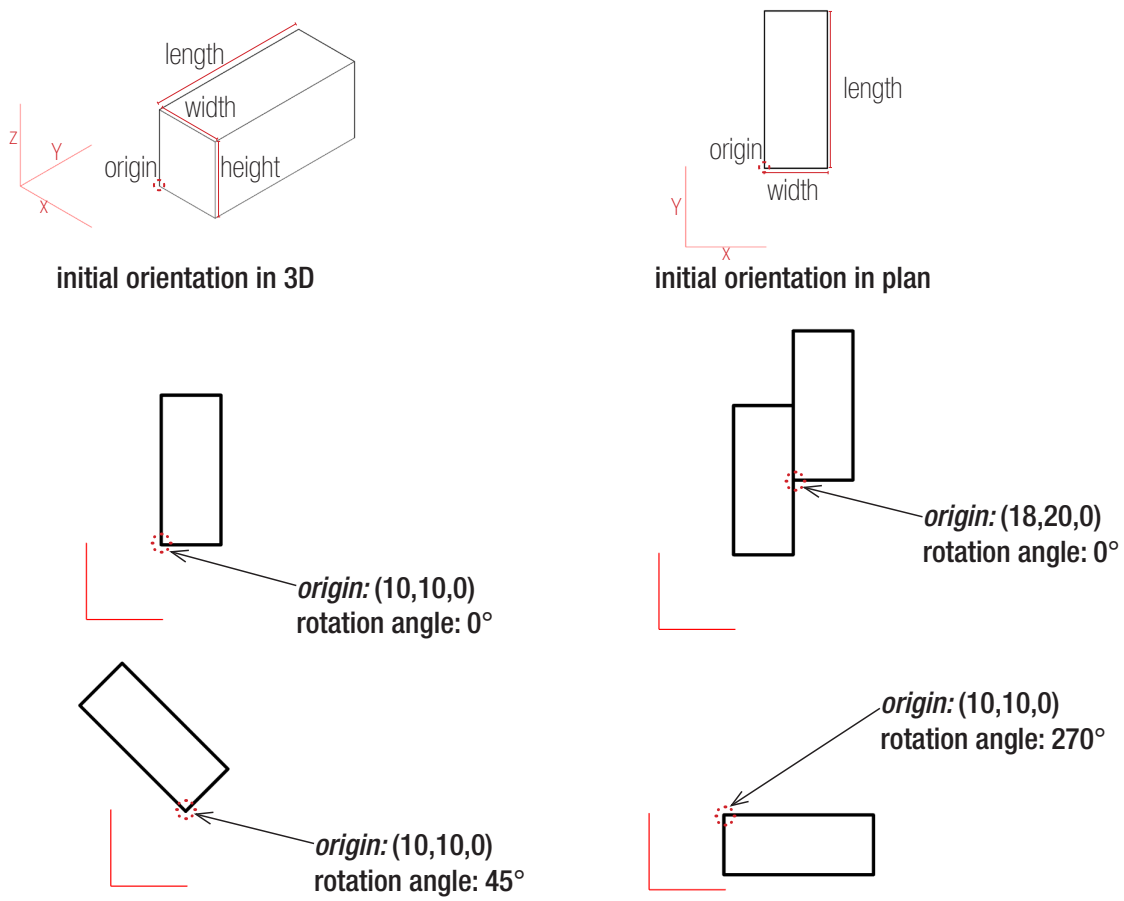


Figure 3.1.2. Definition of a module showing its *dimensions, origin,* and *rotation angle.*

3.1.3 User Inputs and Design Constraints

The design problem is specified by a set of user inputs(Fig. 3.1.1). The *plot* defines the boundary of the area in which the aggregation is allowed to grow. For designers, this can be used to describe the boundary of the site. The *starting geometry* is a group of one or more modules that describes the point at which the aggregation of modules would start forming. Because of how the *production rules* are applied sequentially (detailed in Section 3.2), the resulting aggregations of modules tend to grow around the starting geometry, given that all modules are within the defined *plot*. This gives designers the opportunity to consider different configurations of starting geometry based on the programmatic and circulation needs. For instance, the starting geometry can be used as the lobby space or the entrance to a building. The position of the *starting geometry* on the *plot* determines the positioning of building(s) on the given site.

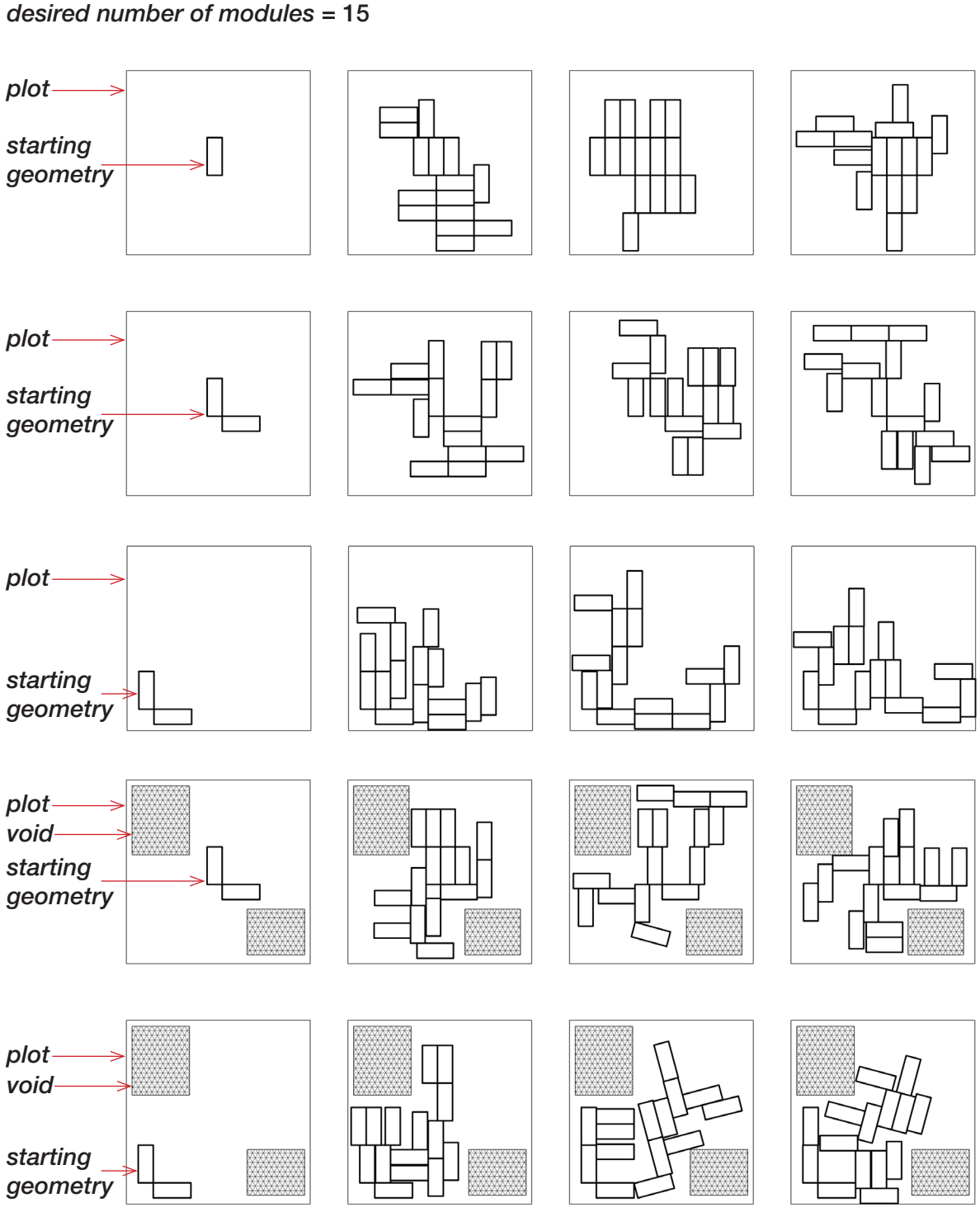


Figure 3.1.3. 2D plan projection of 1-layer generation outputs with *desired number of modules* set to 15. These designs demonstrate how user inputs can be adjusted to drive the form of resulting designs. *Starting geometry* of multiple modules tend to generate designs of similar characteristics while using a single module as a *starting geometry* produces less expected results. Specifying *void* can push the resulting designs to grow around certain areas on the *plot*. The positioning of *starting geometry* also clearly affects the growth pattern.

When considering how the resulting designs would respond to their specific site conditions, designers can also specify areas to avoid within the *plot* by defining the *void*. Defining specific *void* enables users to efficiently iterate different design concepts considering the topographic conditions or the planning of their site.

The desired number of modules defines the size of *each* aggregation layer. Sometimes, the system might not satisfy this number under the given design constraints, but the number of modules in outputs would never exceed the specified number. Fig. 3.1.3 demonstrates how these user inputs are used to drive the form of a one-story modular structure. Since the system can generate 100 assemblies of 15 modules in roughly *8 seconds* on a standard laptop, users can generate and evaluate hundreds of designs at once.

In addition to the design constraints describe by the *plot, void,* and *desired number of modules,* there are also several other design constraints defined within the grammar. First, all modules within a structure must be connected by overlapping faces or sometimes by only a corner. If the user wishes to explore design options of multiple structures on the same *plot*, multiple *starting geometries* can be defined, one for each structure. In this case, the resulting designs would feature a cluster of modules around each *starting geometry*, and modules within each cluster are always connected. Secondly, *none* of the modules can collide with any other module in the assembly. Lastly, all modules must be either grounded or supported by another module from the layer below. This constraint does not evaluate the actual structural performance of the system but is only designed to eliminate obviously infeasible options caused by “floating” modules. Section 3.3 presents the algorithms for how these design constraints are checked at every step during the generation process.

3.2 Production Rules

The *grammar* defines the set of shape transformations allowed during the growth of an assembly. In this proposed methodology, the grammar was implemented using a set of two *stochastic*

production rules. Each rule dictates how a new module is added to a group of existing modules to create a new assembly. Even though both rules are stochastic in nature, they each define a different design logic, leading to distinctive geometric patterns in generation outputs. In this paper, the two rules will be referred to as the *naïve rule* and the *extension rule*. This section details the underlying algorithm of these two rules and how they can be applied in combination to produce desirable designs.

3.2.1 Naive Rule

The *naïve rule* is a stochastic parametric production rule. Fig. 3.2.1 demonstrates the step-by-step process of how an *one-layer assembly* of five modules is generated using *only* the naïve rule. At each step, a new module is attached to the *boundary* of the existing assembly layer. As highlighted in Fig. 3.2.2, the boundary is defined as the group of line segments that encloses all modules of this layer in 2D projection (plan view). These line segments are computed using the intersection points between all module edges. The detailed algorithm of boundary computation will be discussed in Section 3.3.1.

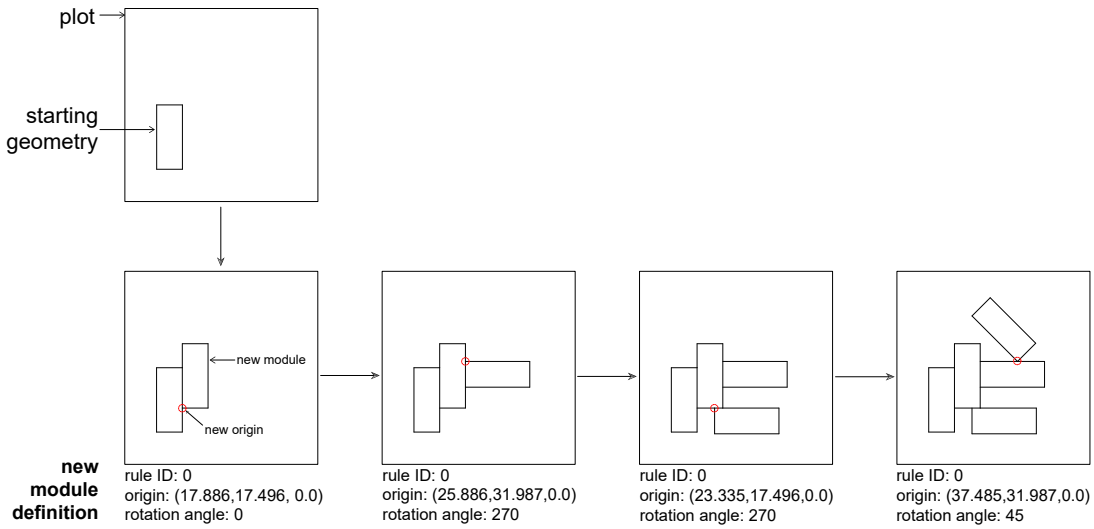


Figure 3.2.1. Production rule steps of generating a 5-module assembly using the *naive rule* only. At every step, a single module is added to a randomly selected location in the existing assembly.

When the naïve rule is applied, a random line segment on the boundary of a given group of modules is selected. Then a real-valued random *position parameter* in [0, 1] is selected using a probability distribution based on user input. The position parameter is used to find a point at

the normalized length along the selected line segment, as the *origin* location(Section 3.1.2) of the new module. As previously discussed in Section 3.1, in the implementation of the proposed methodology, a module can be defined using two parameters, the *origin* and the *rotation angle*. The rotation angle of the new module is selected from a set of discrete angles using another predefined probability distribution. If the location and orientation of the new module does not violate any of the design constraints, this new module is added to the given assembly. Otherwise, the algorithm repeats the same procedure until a valid location is found, or the defined maximum number of iterations has been reached.

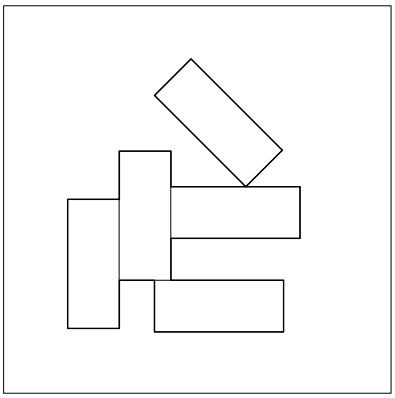


Figure 3.2.2. A 5-module assembly. The heavier line indicates the boundary of this assembly.

Note that the range of potential rotation angles for any new module always spans 90 degrees, but the start and end of this 90-degree range is constrained by the orientation of the selected boundary edge, as shown Fig. 3.2.3. The number of discrete rotation angles in the set is determined based on the user input, *angle step size*. For instance, if the angle range is [0º, 90º] and the step size is set to 15º, the set of discrete rotation angles to select from would be {0º, 15º, 30º, 45º, 60º, 75º, 90º}.

Since both the origin location and rotation of the added module is randomly selected at each step, applying the naïve rule introduces randomness to both the location and directionality of growth. Every time the naïve rule is applied, three random variables are selected from three distinct sets, thus the total number of possible outputs is determined by the size of these three sets. Considering a case where the user starts with an assembly of only one rectangular module and the angle step size set to 15º, there would be *4* possible choices for boundary edge selection and *7* possible choices for rotation angle. And, if the system rounds all the position parameters to

two-decimal places, there would be *100* possible points on each edge. This could produce *2,800* different design outputs. As the assembly grows, the number of possible design outputs grows exponentially and becomes essentially unbounded. A generation process using only this rule would likely produce results that cover a diverse range in the design space, providing designers with a large number of diverse design options to consider. However, if the user were interested in exploring design options that follow a specific pattern, using only the naïve rule would not be sufficient. Because of this highly randomized procedure, it is difficult to control the outputs form the naïve rule so that they formulate a consistent design language that meets the intent of the designer.

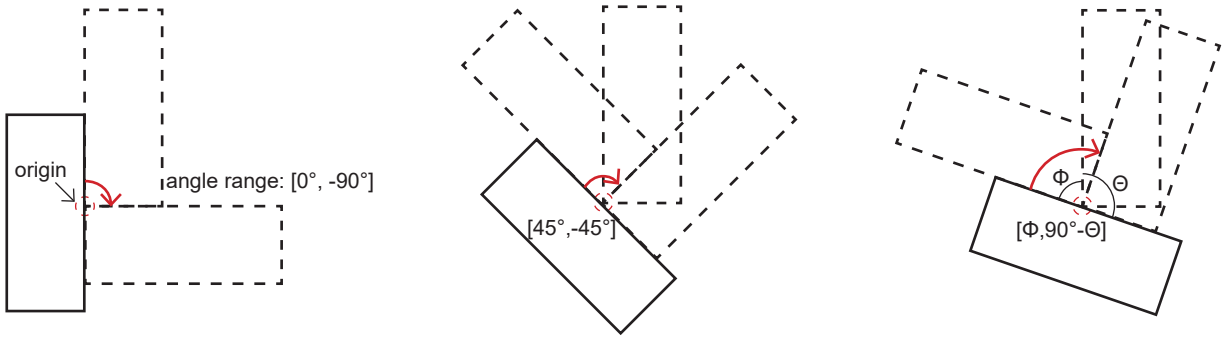


Figure 3.2.3. Relationship between the rotation angle range and the orientation of an existing module. The range of potential rotation angle always spans 90° but the actual range is determined by the orientation of the existing module.

3.2.2 Extension Rule

The extension rule was designed to limit the amount of randomness in the generation process. Fig. 3.2.4 demonstrates how the extension rule is applied recursively to generate a *one-layer assembly* of five modules. At each step, a random module along the boundary is selected, then a new module is added by extending the selected module from one of its edges that is on the assembly boundary. Unlike the naïve rule, the extension rule only introduces randomness to the location of growth, because it does not rely on any other random parameter. The directionality of growth will always be orthogonal to the global orientation of the existing assembly. Since there is only one possible extension from any given edge of a module, the number of the possible outputs using the extension rule is always bounded by the number of extendable edges on the assembly boundary. Note that an edge is defined as extendable only if it covers the full length or

full width of a module.

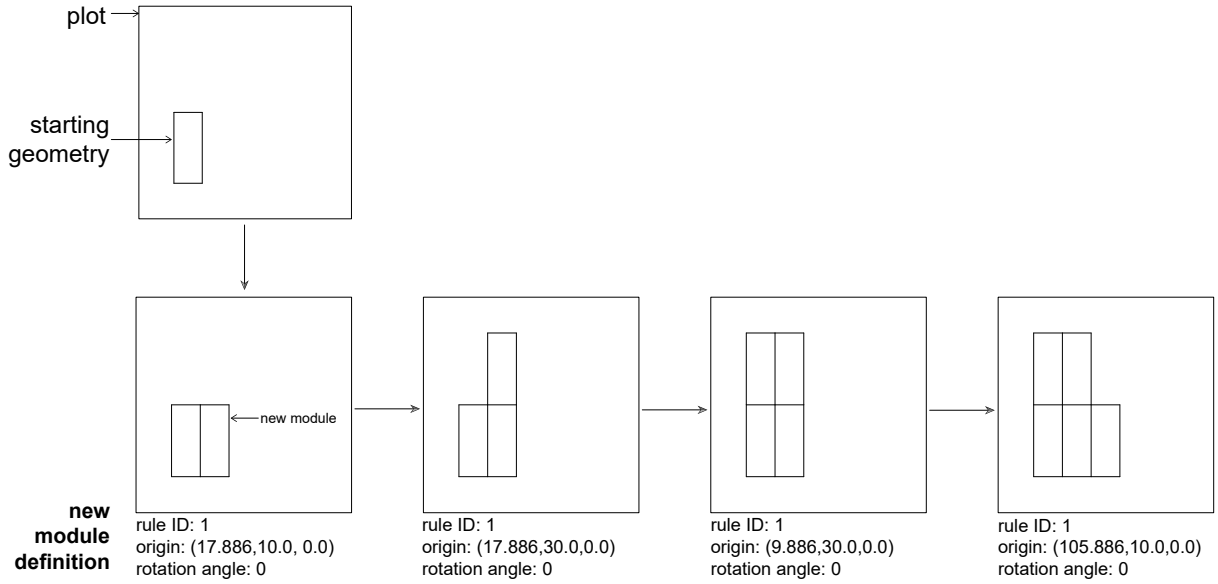


Figure 3.2.4. Production rule steps of generating a 5-module assembly using the *extension rule* only. At every step, a new module is added by extending a randomly selected module in the existing assembly.

In contrast to the naïve rule, only one random variable is selected every time the extension rule is applied, and the size of the set of variables increases only by a small constant amount with every new addition to the assembly. Considering the case of a user starting with an assembly of a single rectangular module and using only the extension rule, the number of possible outputs is initially *4*. This number increases by *at most 2* every time a new module is added to the assembly, as shown in Fig. 3.2.5.

Therefore, the number of possible outputs only increases linearly as the assembly grows. If the user were to generate assemblies with only the extension rule, the resulting designs would only cover a relatively small portion of the design space and most of the outputs would likely to have very similar design patterns.

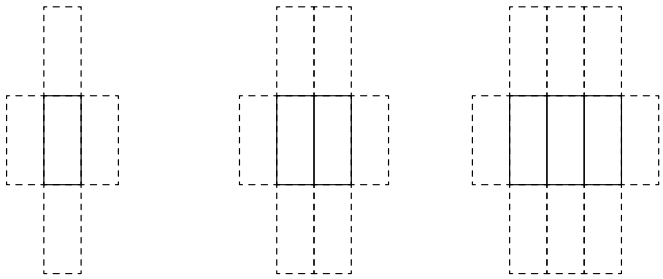


Figure 3.2.5. The number of possible extensions increases by 2 as an assembly grows.

3.2.3 Combination of rules

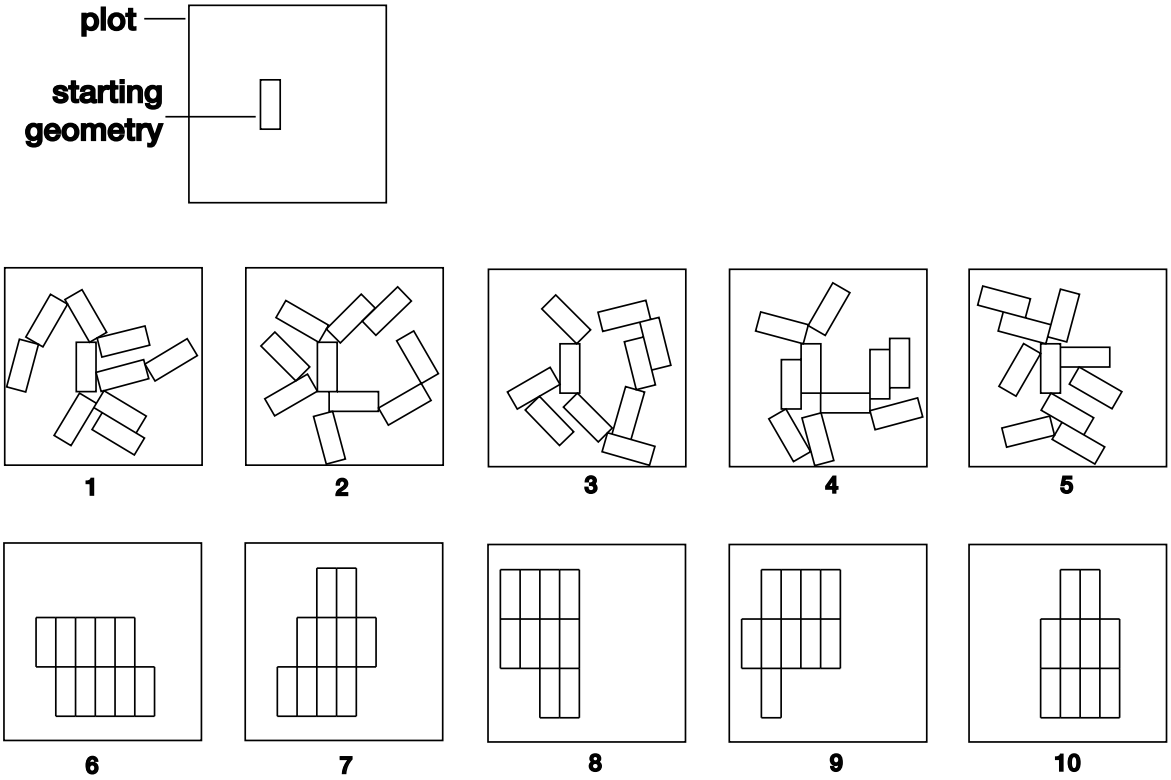


Figure 3.2.6. Generation outputs of 10 module assemblies. Design #1 to #5 are generated using only the *naïve rule* , and design #6 to #10 are generated using only the *extension rule.* These sets of results demonstrate the distinct design pattern of two production rules.

Fig. 3.2.6 shows selected design outputs from generating one-layer assemblies of 10 modules. The first row of designs are generated using only the naïve rule while the bottom row shows the outputs of using only the extension rule. All the random variables used in these examples were generated using a *uniform probability distribution* (discussed in Section 4.1). Visually, these two sets of results each clearly defines a distinctive design pattern. Aggregations using only the naïve rule produces complicated sprawling structures while the ones generated with only the extension rule have more compact and rigid configurations. If both rules are applied throughout the generation process, the resulting designs are expected to exhibit an interesting mixture of characteristics seen in both sets of designs seen above. This section illustrates the qualitative behavior of design outputs generated from specific combinations of the naïve and extension rule. A sequence of rules will be denoted as a list of 0s and 1s, representing the naïve rule and the extension rule respectively. For example, [0,1,0,1] represents the sequence where the two rules are applied in alternating order.

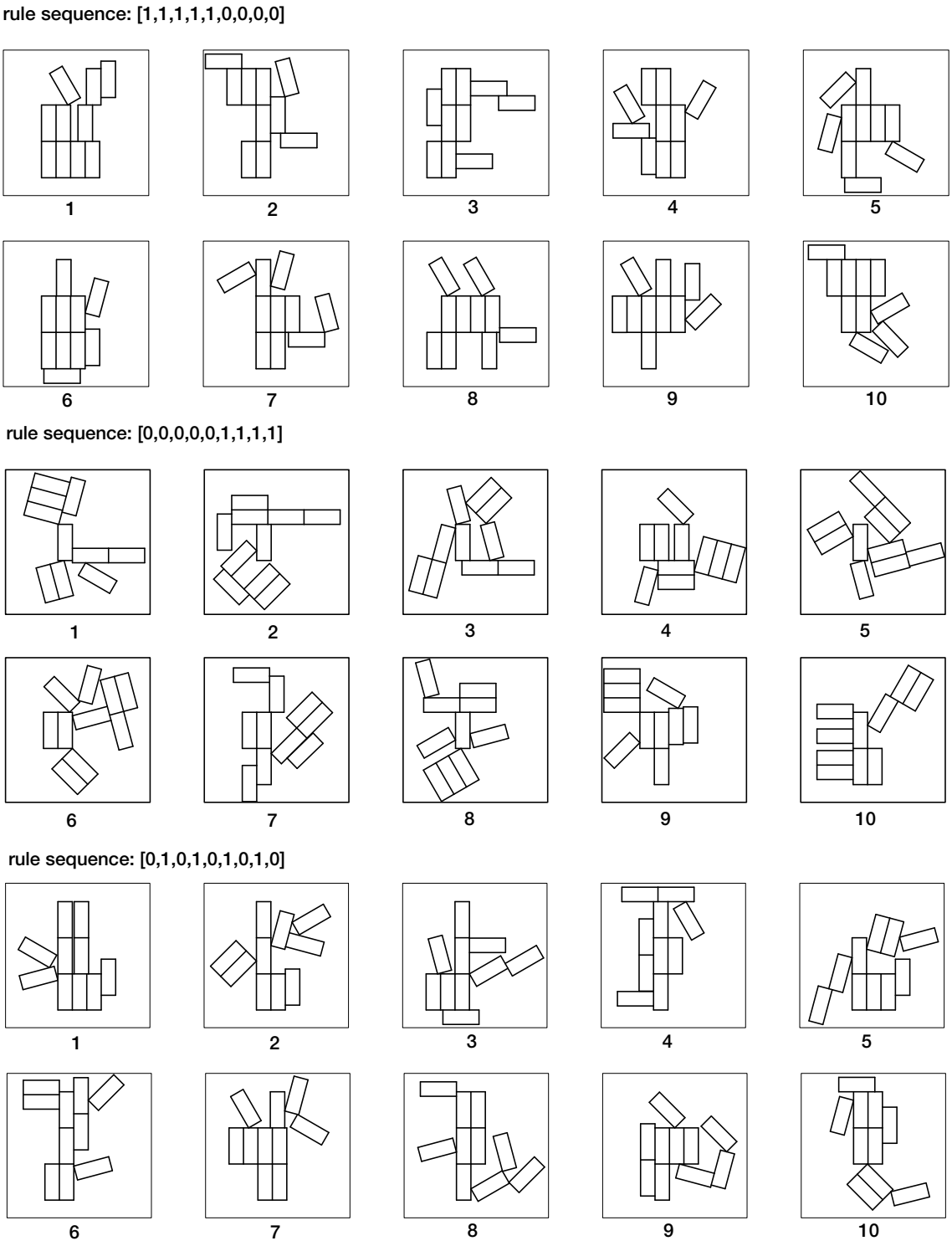


Figure 3.2.7. Generation outputs of 10 module assemblies using different rule sequences. These sets of results demonstrate that the visual characteristics of design outputs can be controlled by splicing different rule sequences.

Fig. 3.2.7 shows ten selected designs from applying the different splicings of rule sequences to the same plot and starting geometry setup as the previous example. Unsurprisingly, similar

visual characteristics can be observed across each set of results. The pattern seen in sequence [1,1,1,1,1,0,0,0,0] can be described as having a few “unstructured” modules radiating from a central mass of modules “structured” on an orthogonal grid. On the other hand, generation from rule sequence [0,0,0,0,0,1,1,1,1] show less uniform characteristics across the designs. However, these designs can be characterized as having multiple clusters of “structured” configurations. These clusters are not always formed on a single grid like seen previously, and the relationship between each clusters is unique. Because of the unique spatial relations within each assembly, this also demonstrates that applying the production rules in a defined sequence still allows designers to explore a diverse set of design options.

Interestingly, all of the designs from the alternating rule sequence [0,1,0,1,0,1,0] show a consistent growth pattern except design #5 and #9. The majority of these designs are “structured” along a central vertical axis with a few modules breaking from the grid in multiple directions. The behavior of design #5 and design # help illustrate that the system can sometimes output quite unexpected results.

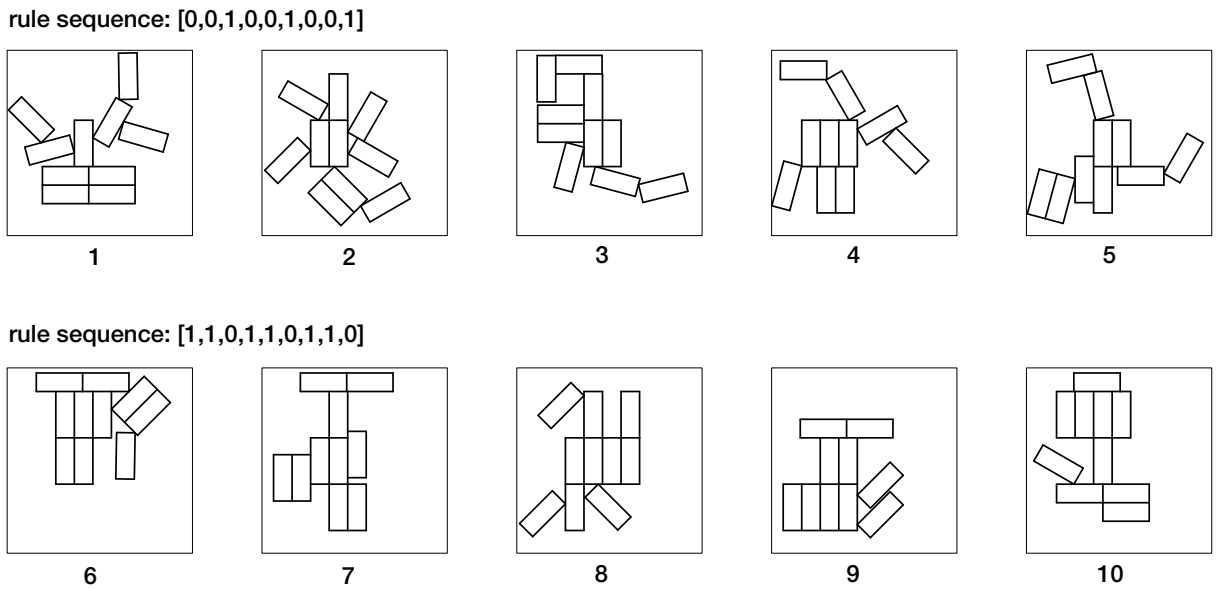


Figure 3.2.8. Generation outputs of 10 module assemblies using rule [0,0,1,0,0,1,0,0,1] and [1,1,0,1,1,0,1,1,0]. These results show that the frequency at which each of the rules is applied has a large impact on the global form of the assembly configurations.

To further illustrate the possibility of deriving a desired design pattern in the generation outputs by splicing rule sequences, Fig. 3.2.8 shows two sets of results that exhibit drastically different behaviors. Design #1 through #5 are generated by applying the sequence [0,0,1,0,0,1,0,0,1].

Design #6 to #10 are produced with the sequence [1,1,0,1,1,0,1,1,0]. As expected, the resulting designs show more "structured" growth patterns when the extension rule is applied more frequently. Even though design #1 through #5 do not necessarily formulate a consistent design language, they still demonstrate a mixture of the visual characteristics previously.

In summary, examples in this section demonstrates that specifying the sequence of rules can be an effective way of controlling the *topology* of the design outputs. Applying the extension rule introduces "structure" to the configurations of modules while the naïve rule allows the assembly to "sprawl". In practice, it can be difficult for designers to experiment with all combinations of rule sequencing. This thesis proposes a method of generating random sequences of rules based on user preferences. The proposed method utilizes probability distributions computed from user inputs to determine which production rule to apply to a given state of an assembly. More discussion around this method will be detailed in Chapter 4 of this thesis.

3.2.4 Construction by Layer

The production rules discussed so far only consider the transformation within a *single-layer assembly.* This methodology proposes a *layer-by-layer* construction logic for generating multi-story structures. To increase user interaction throughout the aggregation process, designers can iterate on the designs of partial structures by generating a single layer of the assembly at a time and select their favorite designs as the starting geometry base for the next layer. Once a geometry base is specified, a parametric production rule is applied to initialize the generation of a new layer by directly extruding up a set number of modules from the previous layer. Then, the rest of the generation procedures within each new layer follow the same production rules discussed in the previous sections. Using a parameter to control the amount of direct extrusions enables designers to consider their programmatic needs and vertical circulation designs between each iteration. Design samples shown in this section demonstrate that this construction by layer approach is a promising way to incorporate designer preferences into different stages of a generative design process. Considering the design #3 of rule sequence [1,1,1,1,1,0,0,0,0] from

Fig 3.2.7 as the base geometry, Fig 3.2.9 shows how a *1-layer assembly* transforms into different starting geometries to start a *second layer* growth and a set of selected outputs from each starting geometry.

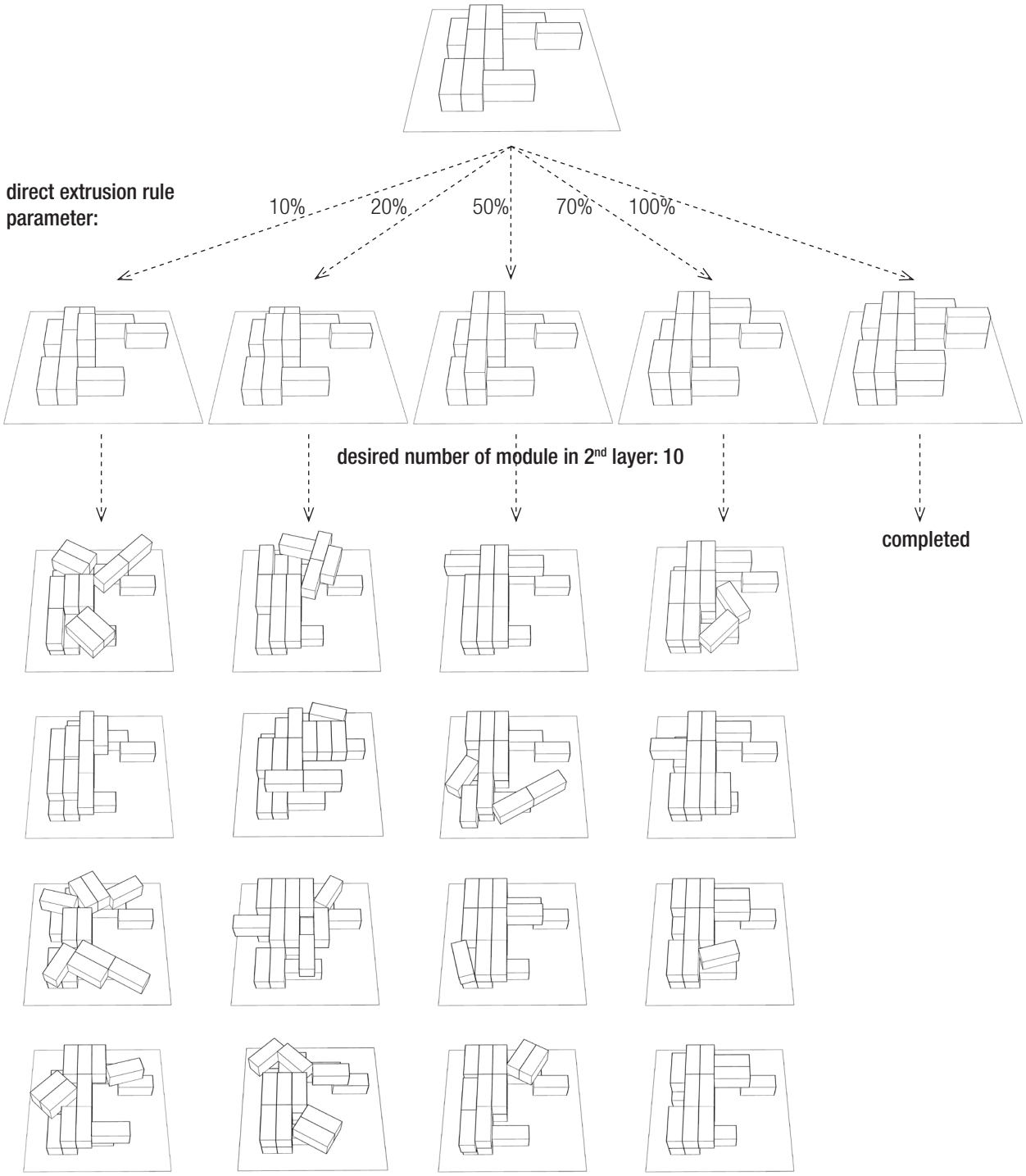


Figure 3.2.9. 1-layer assembly to 2-layer generation process. These results show that direct extrusion rule parameter can influence the diversity of final design outputs.

The outputs shown above are generated applying an alternating rule sequence to the same design problem setup as the selected 1-layer geometry base. Alternatively, designers can also adjust the behavior of each layer individually by adjusting the user inputs between each iteration. Chapter 4 and Chapter 5 will discuss how different user inputs, such as the design constraints and the probability distributions of the random variables, can be used to drive the global form of the generated aggregations.

3.3 Geometric Utilities

This section describes the high-level implementation of some of the geometric utilities used in the generation process, including boundary computation and design constraint checking. Since these computations occur whenever a new module is added to a given assembly, the performance of these algorithms can have significant impact on the overall performance of the system. The methods discussed here are implemented with the assumption that all modular units are vertical extrusions of two-dimensional polygons (sometimes referred to as 2.5D objects). Hence, all geometric properties are computed in two-dimensional space to improve the efficiency of the system. Adapting the implementation to accommodate three-dimensional modules that are not simple extrusions 2D shapes is non-trivial and was not explored in this thesis.

3.3.1 Boundary Computation

As introduced in Section 3.2.1 (Fig. 3.2.2), the boundary of a given assembly layer is the collection of module edges that encloses all modules of the same layer in two-dimensional space. Drawing the outline of a set of polygons is an inherently simple problem for humans, but it can be difficult to generalize the computation procedure algorithmically. Many approaches were considered during the implementation of this thesis.

Considering rectangular modules, the 2D projection of a module is a set of four edges and four

vertices. Mathematically, the term *convex hull* can be used to describe the minimal convex polygon containing all these rectangle vertices. Many algorithms, such as the famous "Gift Wrapping" algorithm, have been proposed to efficiently find the *convex hull* of given set of 2D points. However, the boundary of an assembly is *almost never* a convex polygon. Another approach to solving the boundary computation problem was to compute the *concave hull*. As the name suggests, the term *concave hull* describes the *concave* polygon that containing a set of points. Computing the concave hull from a set of points is much more complicated and little work has focused on concave hull algorithms in comparison to convex hull algorithms (Asaeedi et al, 2017). An early implementation tested in this thesis was based on the "Swing Arm" algorithm proposed by Galton and Duckham (Galton & Duckham, 2006). However, it was noted later that, in some cases , the concave hull of all module vertices also differs from the boundary of an assembly, as shown in Fig. 3.3.1. In these cases, the boundary of an assembly layer cannot be described by a single concave polygon but multiple connected polygons. This resulted in design outputs that are not fully connected within itself, since an inaccurate boundary was used to compute location of new modules in the process described in Section 3.2.1.

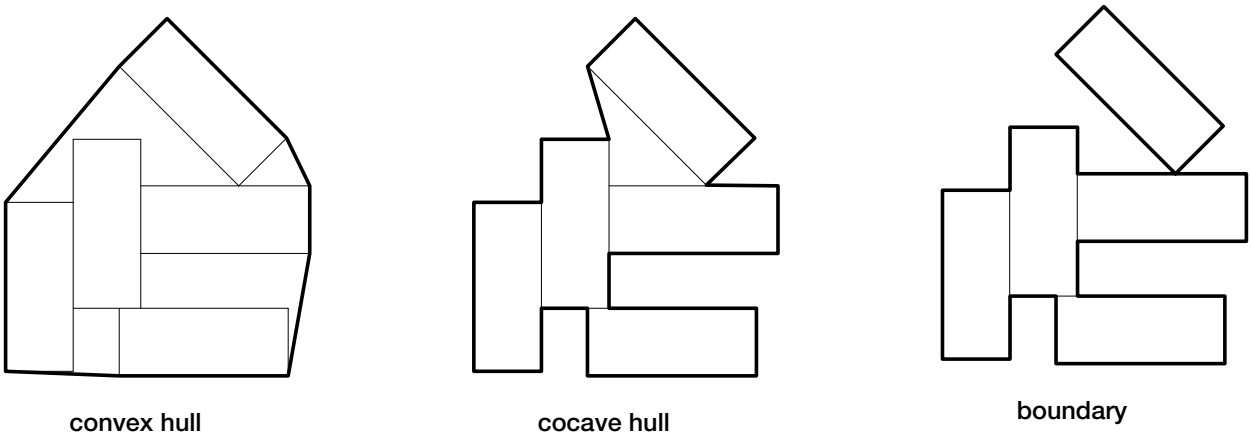


Figure 3.3.1. Convex hull vs. concave hull vs. assembly boundary. Using known concave hull algorithms sometimes approximates the boundary well but does not guarantee that every line segment is a part of a module edge.

Since the desired boundary only contains line segments that are on the edges on the modules, a new method of computing the boundary was implemented using the edges of all the modules instead of only using the vertices. An outline of this implementation is given below in Algorithm 1. Fig. 3.3.2 illustrates the different cases of intersection events described in the algorithm.

Algorithm 1 Boundary computation

Inputs: Edges - an array of all module's bottom edges in the assembly layer
Outputs: Boundary – an array of line segments representing the boundary of an assembly layer
Initialize a boundary set output ← {}
Initialize a set tracking indices of edges that completely overlap, O ← {}
Initialize a set tracking indices of edges that have partial overlaps, P ← {}
Initialize a map mapping edge indices to overlap intersection points, Pts ← {}
N = the number of edges in Edges
for i ← 0 to N-1 do
  if i is in O, **then**
    this edge is not on the boundary, continue onto the next edge.
  for j ← i+1 to N do
    if j is in O, **then**
      this edge is not on the boundary, continue onto the next edge.
    Check intersection of $edge_i$ and $edge_j$
    if there is an intersection event between the 2 edges, **then**
      case 1: the intersection event occurs only at a point, **then**
        both edges could potentially be on the boundary, continue onto the next edge
      case 2: the intersection event of these 2 edges is an overlap, AND
      the overlap spans the entire length of either or both edges, **then**
        that edge(or both edges) is not on the boundary, add the index of that edge to O
        continue onto the next edge
      case 3: the intersection event of these 2 edges is an overlap, AND
       the overlap is only spans a part of either or both edges
        add the index of the edge that has a partial overlap to P
        add the start and end points of the overlap to the corresponding entry in Pts
  end for
end for
for n ← 0 to N do
 if n is not in O or P, **then**
    $edge_n$ is on the boundary, add $edge_n$ to output
 if n is in P, **then**
    only a part of $edge_n$ is on the boundary, specifically the non-overlap part of the edge
    sort entries in Pts[n] by distance from the start point of $edge_n$ in ascending order
    for pt_i in Pts[n] do
      if pt_i is the first point, **then**
        make a new line segment from start point of $edge_n$ to pt_i
      if pt_i is the last point, **then**
        make a new line segment from pt_i to end point of $edge_n$
      otherwise
        make a new line segment from pt_i to pt_{i+1}
      add the new line segment to output
end for
return output

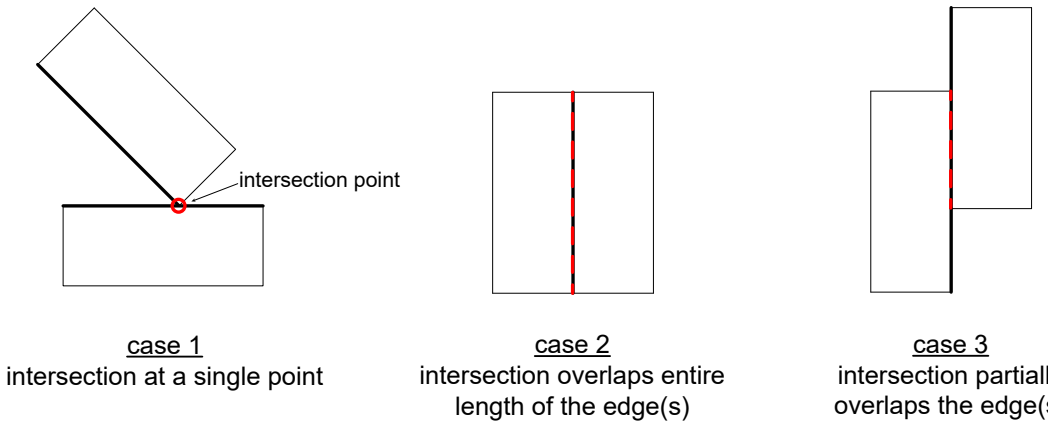


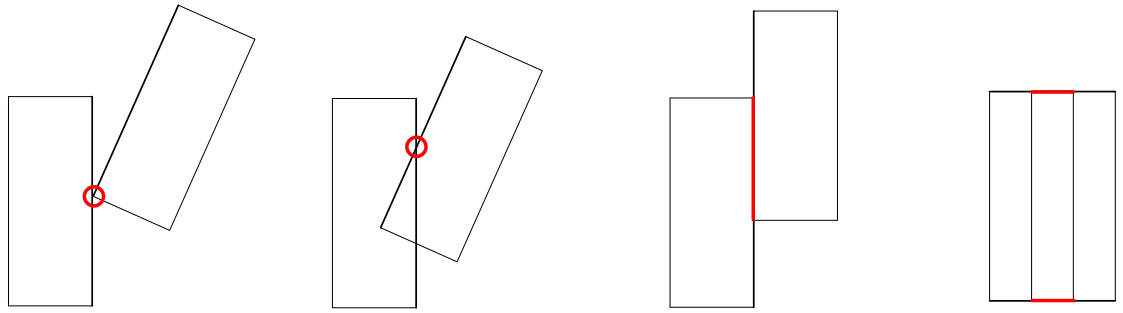
Figure 3.3.2. Three different cases of intersection events as described in Algorithm 1. The heavy dark lines indicate the selected edges, and the red lines describes the intersection events.

Since the system is iterating through all the edges and checking the intersections with every other edge, the time complexity of this algorithm is $O(n^2)$ where n is the number of module edges in 2D. The time complexity upper bound of the "Swing Arm" algorithm is also $O(n^2)$ (Galton & Duckham, 2006), so the algorithm used in this implementation has similar worst-case performance as other concave hull algorithms. However, efficient data type designs in an object-oriented programming language can significantly improve the performance of this algorithm. For instance, the implementation of this thesis makes use of an object class that updates and stores the boundary at every step throughout the generation process. In this case, the initial computation follows the procedure outlined in Algorithm 1, but every time a new module is added it only needs to check for intersections between the newly added edges and the previously computed boundary edges. Therefore, the time complexity of updating the boundary is $O(n)$ since adding a single module adds a constant number of edges. This approach achieves linear time complexity at the expense of memory usage. More discussion on the implementation of different classes will be detailed in Section 5.

3.3.2 Collision Detection

As discussed in Section 3.1.4, one of the design constraints for the generation of assemblies is that no module can collide with another module. The previous prototype implemented in Rhino Grasshopper uses the Wasp plug-in's built-in 3D mesh collision detection checking. Collision checking with 3D objects was found to be a performance bottleneck of the system's overall

efficiency. An effective approach to resolve this issue is checking for possible collisions in 2D, assuming that all modular units are vertical extrusions of 2D shapes. If two polygons have no overlapping areas, their 3D extrusions do not collide. Algorithm 2 outlines the implementation for checking whether a new module collides with another module in the existing assembly. Fig 3.3.3 (collision cases) shows that different cases described in Algorithm 2.



case 1a	case 1b	case 2a	case 2b
2 modules DO NOT collide, IF edge intersections occur at an endpoint	2 modules collide, IF edge intersections occur at a point that's NOT an endpoint	2 modules DO NOT collide, IF number of edge overlaps <= 1	2 modules collide, IF number of edge overlaps > 1

Figure 3.3.3. Different cases of intersections events as described in Algorithm 2. If intersections events occur in two of the scenarios described, the 3D extrusion of these shapes would collide.

Algorithm 2 Collision Checking

Inputs: M - new module for collision checking; A - an array of existing modules in this layer
Outputs: True if new module does not collide with another module in the assembly
False otherwise
newEdges = all bottom edges of M
maxDist = 2 × the length of the longest edge in newEdges
for $module_i$ in A do
  calculate the Euclidean distance D from the origin of M to the origin of $module_i$
  if D > maxDist **then**
    M does not collide with $module_i$, continue onto to next module in A
  existingEdges = all bottom edges of $module_i$
  initialize an overlap edge counter C ← 0
  for $edge_A$ in newEdges do
    for $edge_B$ in existingEdges do
      Check intersection of $edge_A$ and $edge_B$
    if there is an intersection event between the 2 edges, **then**
      case 1: the intersection event occurs only at a point
        if this intersection point is NOT an endpoint of either $edge_A$ or $edge_B$ **then**
          M and $module_i$ intersect
          return False
      case 2: the intersection event of these 2 edges is an overlap **then**
        C += 1
        if C > 1 **then**

Algorithm 2 Collision Checking (continued)

```
                M and module_i overlap
                return False
        end for
    end for
end for
return True
```

Since this algorithm iterates through all the modules in the given assembly once, and each module has a constant number of edges, the time complexity of this collision checking procedure is *O(n)* where *n* is the number of modules in the given assembly layer.

3.3.3 Other Constraint Checking

Before a new module can be added to the assembly, it is also critical to ensure that the proposed location of the new module is within the bounds of the user define plot and outside of the void. Like the previous operations discussed in this section, checking these constraints can also be efficiently done in 2D. The implementation of this system includes two versions of these constraint checking mechanism. The first, and the more efficient, approach is to consider a module within the bounds of the plot if all of its bottom vertices and its centroid are contained by the curve of the plot when projected in the *XY-Plane.* Similarly, a module is out bounds of the voids when these point projections are not contained by any curves of the void. Note that this is not a strict checking of these conditions since a module might have small areas that are still inside the voids or outside the plot in some edge cases. A more robust, but slightly less efficient, version uses a similar logic as the one described Algorithm 2. Instead of only checking if a given curve contains a set of points, this algorithm checks for specific intersection events between all the edges of a module and the input curves. Since each implementation could potentially lead to very different solutions, the system allows users to select whether or not a strict checking is required.

When the growth of the assembly takes place above the grounded layer, the system also ensures that the new module is supported by at least another module in the assembly layer below. In the proposed methodology, a module is considered to be supported if it has *any* overlapping

area with any module in the assembly layer below. This operation is also implemented in 2D using a similar logic as the collision checking algorithm discussed in Section 3.3.2. Instead of checking for intersections with modules of the same assembly layer, support condition checking uses Algorithm 2 to check for intersections with modules from the layer below the new module. This procedure eliminates structurally infeasible designs by ensuring that every module in the design outputs is either grounded or supported by another module. This method is able to produce a diverse range of solutions in terms of both visual characteristics and quantitative performance because it does not place any constraints on the amount of the overlapping areas between module. Allowing some poor-performing partial structures to aggregate is critical since these partial structures, especially early in the generation process may eventually lead to well-performing final designs. Further structural performance analysis of the solutions is included at end of each iteration to allow designers to evaluate the quantitative performance goals of their favorite designs.

3.4 Quantitative Analysis

One of the goals of this thesis is to help designer find modular structure design solutions that are both visually interesting and quantitatively well-performing. As briefly introduced in Chapter 1, structures that are consisted of repetitively stacked modules are expected to be more advantageous in terms of structural performance. More complex configurations can often better communicate the creative intent of the architects and address other important design factors such as porosity and daylight availability. This section introduces the two main numerical metrics that are used to evaluate the structural performance and complexity of the aggregations through the generation process.

3.4.1 Complexity Score

This thesis presents a method of characterizing the *topology* of an assembly with a numerical metric, complexity score. It is a measurement how complex the configuration of modules is

within each layer of a given assembly, relative to the number of modules. Each individual layer in an assembly has its own complexity score, and the mean complexity score of all assembly layers can be used to evaluate the configuration of the entire assembly. The calculation of complexity score for each assembly layer makes use of the *boundary* computed for the given layer. As defined in Section 3.3.1, the boundary of an assembly layer describes the overall shape of the module configuration, and the total length of the boundary defines the perimeter of this shape.

Considering an assembly layer consisted of two rectangular modules, the *simplest* configuration is when two modules are position directly adjacent of each other, forming another large rectangle in plan. To create a more *complex* configuration of two adjacent modules, one could position the two modules with a certain offset so that the plan projection of the two modules only share an edge *partially*, as shown in Fig. 3.4.1. The complex configuration is occurs when the two module edges of the two modules only intersect at a single point, meaning that the two rectangles in plan do not share any edges at all. Note that the total boundary length of these configurations increases as the configuration gets more complex. This demonstrates that boundary length is an effective approximation of a configuration's complexity. The proposed evaluation method uses the ratio between the total length and total floor area to quantify the complexity of a given assembly layer, as given in Eq. [3.4.1] below.

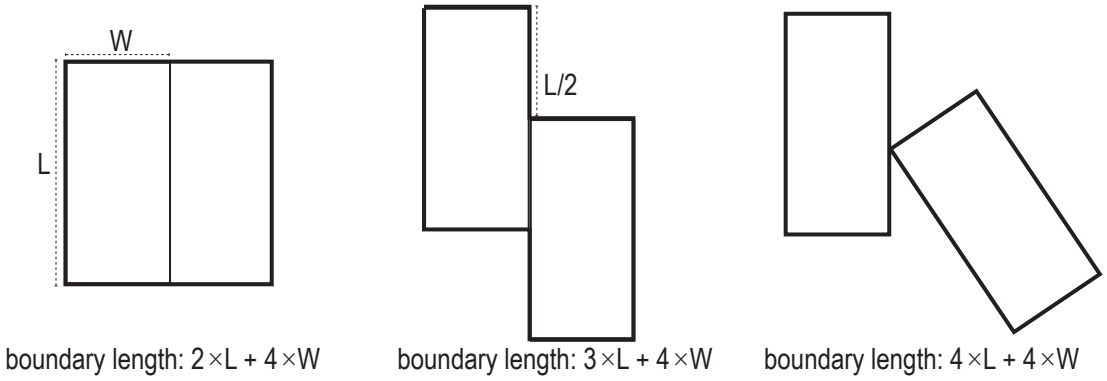


Figure 3.4.1. A few configurations of 2 modules in plan. The heavier line shows the boundary of these assemblies, which demonstrates that the total length of the boundary can be used to measure the complexity of a given assembly.

$$C_R = L_{boundary} / A_{total} \quad \text{Eq. [3.4.1]}$$

where C_R is the raw complexity score of a given assembly layer; A_{total} is the total floor area of the assembly layer, and $L_{boundary}$ is the total length of the boundary. This metric can be computed in linear time, since the boundary can be computed in linear time, as described in Section 3.3.1.

To account for assemblies of different sizes, the complexity score needs to be so that this metric would fall within the same numerical range, [0, 1], for all assemblies. This would also give more information about how complex a given assembly layer is in comparison to other possible configurations. To normalize the complexity score, the system needs to predict the maximum and minimum possible complexity score for an assembly of a certain size. As discussed previously, the most complex assembly here can be defined as the configuration of modules in which none of the modules share any edges. Therefore, the maximum complexity score can be calculated using Eq. [3.4.2]. Similarly, the minimum complexity score can be calculated by approximating the minimum boundary length of an assembly of a given size. Considering rectangular module, a square has always has the smallest perimeter of any rectangle with a given area. Hence, the most "compact" configuration of modules would result in a square-shaped assembly. Even though it is not always possible to arrange modules into a perfect square, depending on the dimension and number of modules in the assembly, using Eq. [3.4.3] serves as an efficient method of approximating the minimum complexity score.

$$C_{max} = (n \times L_{module}) / A_{total} \quad \text{Eq. [3.4.2]}$$

where C_{max} is the maximum possible complexity score used to determine the normalized complexity score; n is the number of modules in the given assembly, and L_{module} is the perimeter of a single module.

$$C_{min} = 4 \times \sqrt{A_{total}} / A_{total} \quad \text{Eq. [3.4.3]}$$

where C_{min} is the minimum possible complexity score used to determine the normalized complexity score, and A_{total} is the total floor area of the assembly layer. The numerator of this equation gives the perimeter of a square of given area A_{total}. This is only an approximation of the minimum possible complexity score, which isn't always achievable.

$$C_{normalized} = (C_R - C_{min}) / (C_{max} - C_{min})$$ Eq. [3.4.4]

where $C_{normalized}$ is the normalized complexity score of an given assembly layer and C_R is the raw complexity score given by Eq. [3.4.1]. C_{max} and C_{min} are given in the equations above. The metric quantifies how complex is a given assembly configuration relative to the number of modules in the assembly. A higher value indicates the assembly is more complex, or less "structured".

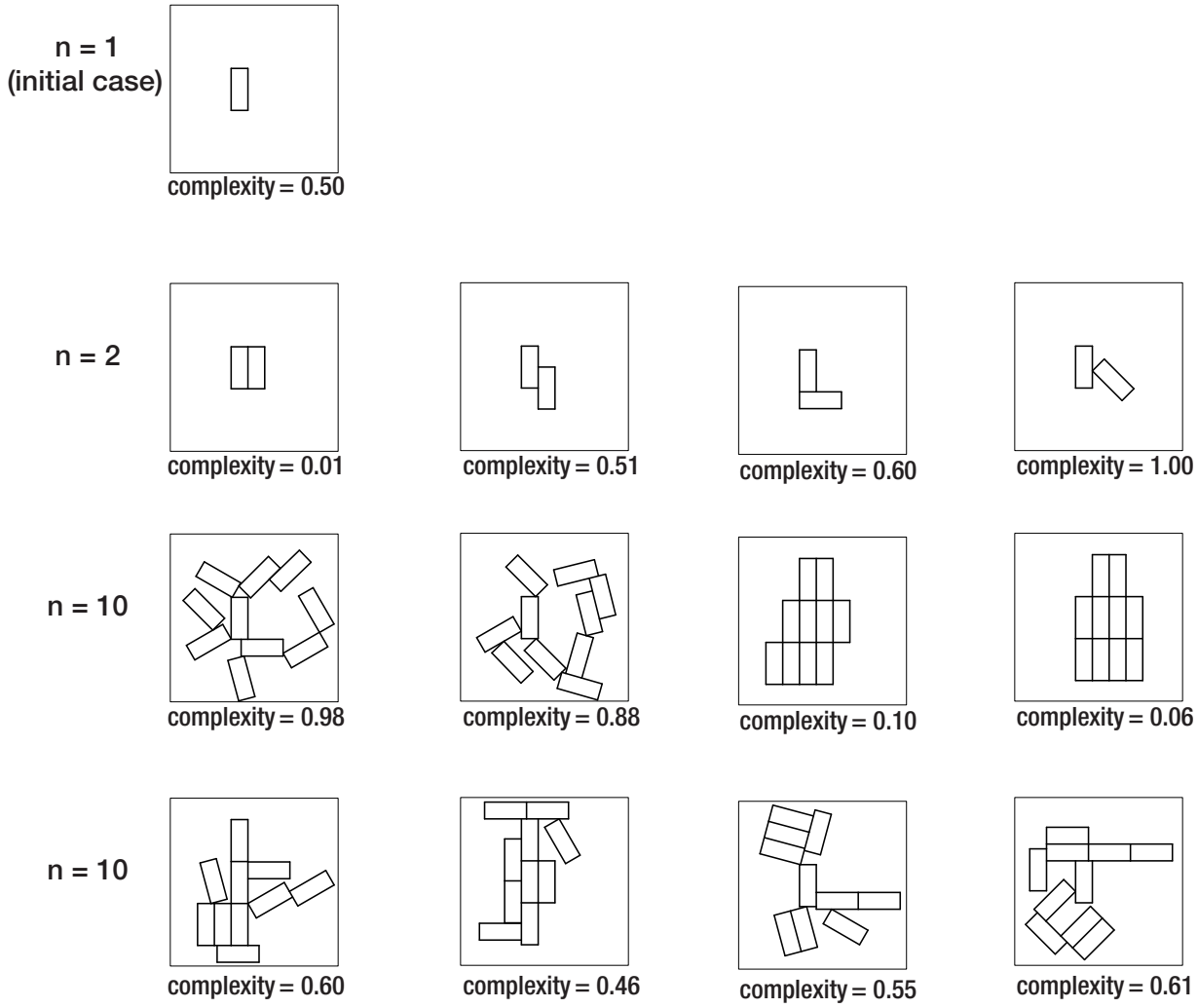


Figure 3.4.2. 1-layer design samples shown with their complexity scores. Assemblies with similar topology demonstrate comparable complexity score.

Fig. 3.4.2. shows the complexity score of a few small assemblies as well as a series assemblies selected from Section 3.2.3. These examples demonstrates that assemblies with a relative low complexity score are more "structured" than those with higher complexity scores, as expected. As designers iterate on different early design concepts, it might be important to consider how the

complexity of an aggregation could affect spatial relations within a design. Higher complexity scores are often associated with more "porous" designs that offer a more interesting and novel experience for the inhabitants of the building. Since complexity score is calculated using the total boundary length of the assembly, it directly correlates to the potential total façade area. Designs with longer façade bring opportunities to introduce additional openings in the building envelope for daylight to enter. Furthermore, designs with lower complexity scores tend to have larger depth between exterior walls, which means that there is a higher percentage of *non-daylit* area than designs with higher complexity scores and shallower floor depth (Fig. 3.4.3). Hence, this methodology proposes that the complexity score of a given design can also be interpreted as the *heuristic* value for daylight performance. This could enable designers to explore more possibilities in the early design phase without performing daylight simulations on all of them.

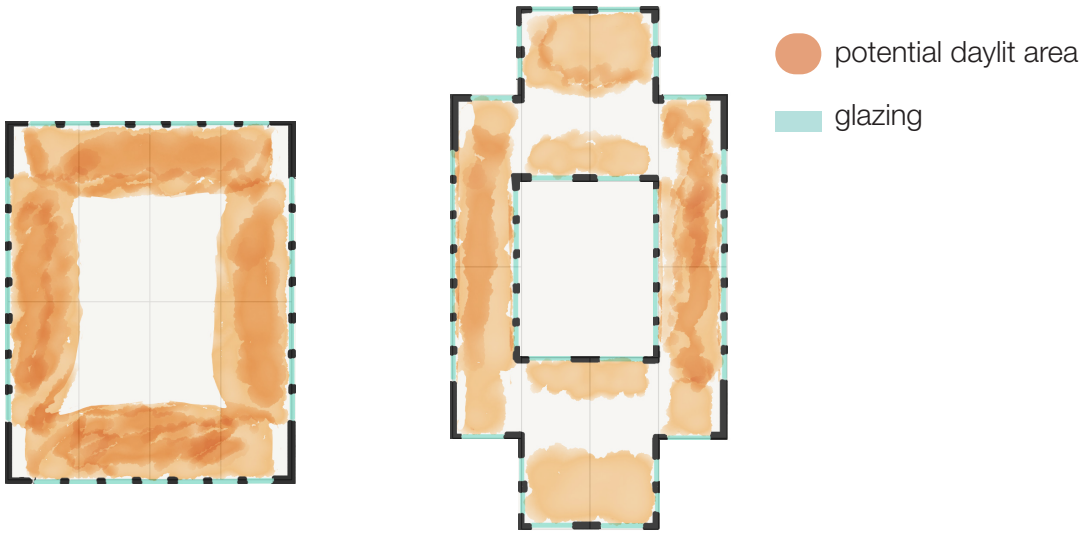


Figure 3.4.3. Illustrated comparison of potential daylit area between a simple and a more complex configuration. Daylight depth are estimated based the rule of thumb: 2.5 times of the window head (Reinhart, 2018). Both consisted of 8 modules, the more complex assembly on the right could perform much better in terms of daylighting.

However, overly complex configurations might sometimes make it difficult to accommodate specific programmatic needs and circulation designs. Considering multi-story structures, complex configurations might also disadvantage construction efficiency and structural load transfers. Therefore, it is critical to balance the complexity score of the generation outputs with other evaluation metrics when designing modular structures following this methodology. Chapter 4 will discuss how the aggregation *grammar* can be controlled by designers to produce designs with desirable complexity scores.

3.4.2 Structural Performance Score

At the end of each iteration, structural analysis can be performed on each generated modular structure to allow designers to consider the structural performance of different design concepts. In the implementation of this thesis, structural analysis is integrated into the system using *Karamba3D*, a parametric structural engineering analysis tool. This section details the specific structural model and loading conditions used to analyze the performance of modular structures discussed in this thesis. Under this methodology, designers may also specify their own structural model and loads to get more accurate feedbacks for the specific material and construction of the modular unit of their choosing. This thesis specifically considers the case of using shipping containers as modules and each assembly is abstracted as a system of steel frames. Each module in an assembly is represented as a system of finite number of beam elements and nodes. Fig. 3.4.4 shows a *20-feet High Cube* shipping container (20 feet by 8 feet by 9.5 feet) represented with 44 nodes and 64 elements.

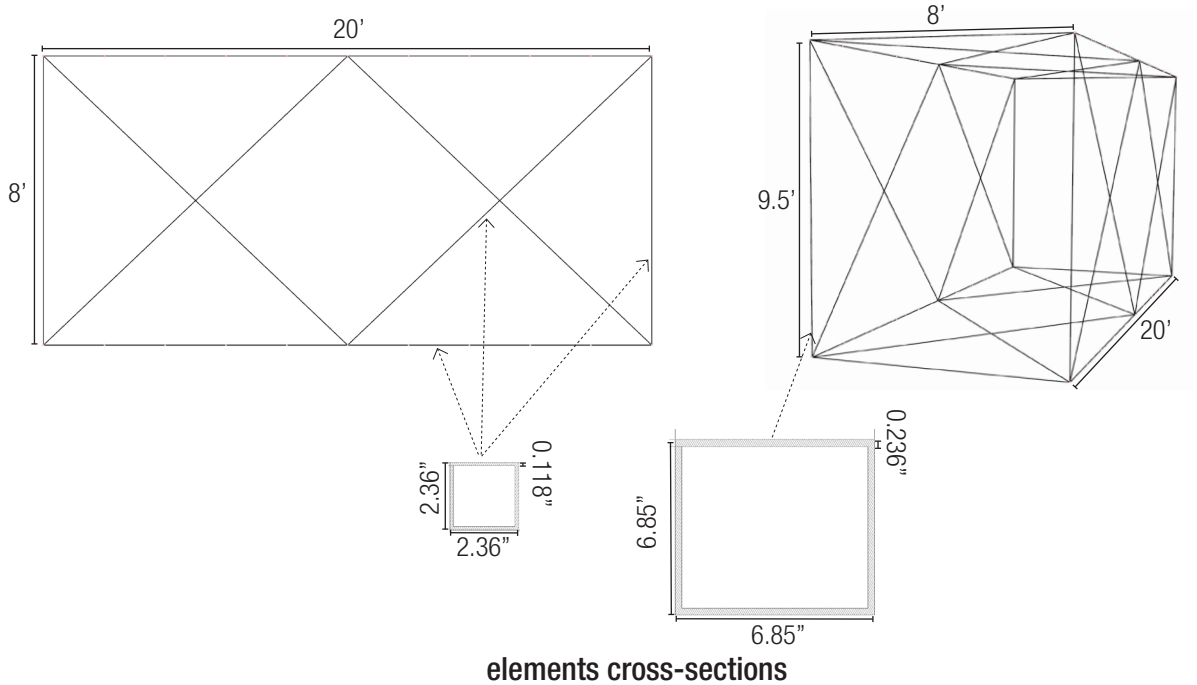


Figure 3.4.4. A few configurations of 2 modules in plan. The heavier line shows the boundary of these assemblies, which demonstrates that the total length of the boundary can be used to measure the complexity of a given assembly.

Note that each face of the module is modeled as a bracing system with the exception of the smaller walls at the ends of the module. No structural elements were placed on these faces to

account for the possibility that these faces may be commonly used for large openings such entrances into the building. In reality, the faces of a shipping container are much more complex structurally. Since a more accurate structural model of shipping containers would significantly increase the time it takes to perform finite element analysis. This bracing system is used to approximate each module as a rigid body.

A total downward load of *2,300 kilogram (6.173 kips)* is split between all nodes to simulate self-weight (Bernardo et al, 2013). Fig. 3.4.5 demonstrates the loading condition and structural behavior of a few simple stacks of modules using this structural representation.

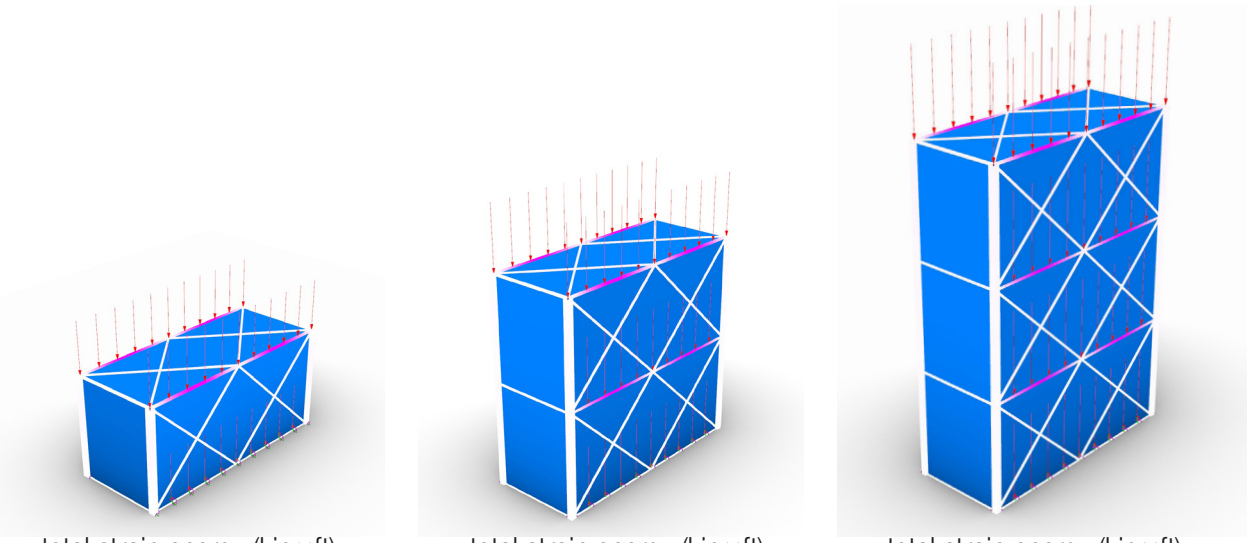


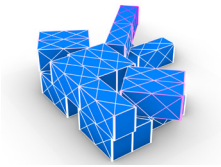
Figure 3.4.5. Structural analysis of simple stacks of modules. The arrows indicate the point load applied at each node. The purple gradient on the structural elements indicate the areas of largest displacement.

The total strain energy describes the elastic energy stored in the deformed system. When comparing two structural systems under the same loads, a smaller value of total strain energy denotes a structure with less deformation, hence, a better performing structure. However, as demonstrated in Fig. 3.4.5, the number of modules in a system can also have a direct impact on the total strain energy. To compare the relative structural performance between assemblies of different number of modules, this methodology defines the structural performance score of an assembly as given in Eq. [3.4.5]. If two assemblies have similar total strain energy but different number of modules, the assembly with more modules would have a lower structural performance

direct extrusion parameter: 100%

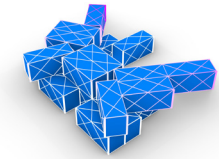
layer 2 complexity: 0.38
structural score: 0.98

direct extrusion parameter: 10%

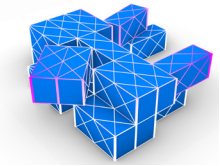


layer 2 complexity: 0.53
structural score: 2.26

layer 2 complexity: 0.20
structural score: 0.94

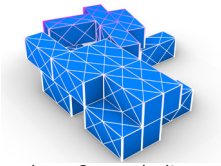


layer 2 complexity: 0.68
structural score: 5.72

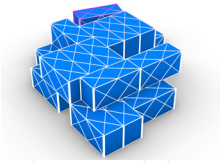


layer 2 complexity: 0.48
structural score: 2.46

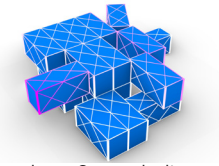
direct extrusion parameter: 20%



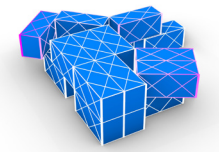
layer 2 complexity: 0.40
structural score: 3.87



layer 2 complexity: 0.32
structural score: 4.61



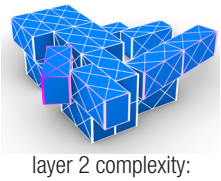
layer 2 complexity: 0.47
structural score: 3.33



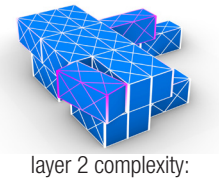
layer 2 complexity: 0.49
structural score: 3.83

direct extrusion parameter: 50%

layer 2 complexity: 0.32
structural score: 2.47



layer 2 complexity: 0.58
structural score: 1.97

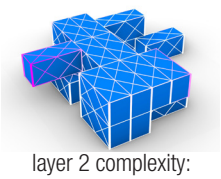


layer 2 complexity: 0.37
structural score: 2.03

layer 2 complexity: 0.35
structural score: 1.40

direct extrusion parameter: 70%

layer 2 complexity: 0.55
structural score: 5.81



layer 2 complexity: 0.41
structural score: 2.38

layer 2 complexity: 0.45
structural score: 1.49

layer 2 complexity: 0.31
structural score: 1.29

Figure 3.4.6. Structural analysis of the selected 2-layer assembly designs from Fig. 3.2.9. As expected, these designs show that there are clear correlations between the complexity score and structural performance score. The gradient on the structural elements indicate their relative displacement, where darker color indicates larger displacement. This visualization of displacement could also help designers adjust the generated designs to achieve better structural performance.

score. This metric defines lower structural performance scores as better-performing in terms of relative structural efficiency since having more modules also means that there is a larger total floor area in the assembly.

$$S = E / n \times 100 \qquad \text{Eq. [3.4.5]}$$

where S is the structural performance score; E is the total strain energy of the structural system, and n is the total number of modules in this assembly.

To demonstrate how designers could use this method to evaluate the design outputs based on the structural performance score, Fig. 3.4.6 presents the structural analysis results of the 2-layer assemblies previously discussed in Section 3.2.4. All these designs were generated using the same ground layer base geometry but different *direct extrusion rule parameter.* Despite the visual diversity of these results, many of them have relatively comparable structural performance score. In addition to evaluating the quantitative feedback of the outputs, users can also evaluate the structural performance visually based on the relative displacement of each elements. As seen in some of the examples with large cantilevering, a few *poorly* placed modules can have a large impact on the overall structural performance score. This encourages designers to interact with the generation outputs to achieve more desirable solutions by adjusting the placement of modules. Seeing the visual and quantitative performance feedback at the end of every iteration can also help designers build a better understanding of the structural behavior of modular aggregations.

Comparing the input parameter and the performance scores, the larger extrusion parameter value tend to produce designs that perform better structurally since directly stacking a second layer advantages load transfer. However, the second design form the *10% extrusion parameter* set was intentionally selected to demonstrate that this relationship is not always guaranteed. Though rare, some outliers are expected because of the stochastic nature of the generation method, . Additionally, one may also observe that there is a clear correlation between the complexity score and structural performance score as expected. More discussion around how characterize this relationship and balance these numerical metrics in design outputs will be detailed in Chapter 4 and Chapter 6.

Chapter 4

Algorithmic Grammar Control

This chapter presents a method of controlling the deployment of the grammar to satisfy the creative intent of designers. As seen in Chapter 3, the form of the design outputs not only depends on user inputs such as the *plot* and *void,* but are also determined by the sequence of rules applied during the generation process. Section 4.1 in this Chapter discusses a method of selecting a production rule and parameters at each aggregation step in a stochastic yet controllable way. Section 4.2 provides a detailed overview of how this stochastic rule selection algorithm is embedded into the overall framework of the system introduced in the previous chapter. Design samples in this chapter demonstrate how the proposed methodology can enable designers to explore diverse design options with desirable quantitative and qualitative characteristics.

4.1 Stochasticity Control

Instead of applying transformations to existing geometries manually, using a *grammar-based* aggregation method presents the opportunity to algorithmically explore a large portion of the design space. This allows designers to efficiently iterate through different design concepts and discover unexpected possibilities. Automatic grammar deployment can be implemented with a number of different approaches. Using *explicit aggregation description,* where users would specify the complete sequence of rules to be applied, allows full control over the generation process. As presented in Section 3.2.3, this approach can be used to effectively control the growth pattern in the resulting designs. However, experimenting with explicit sequences of rules can quickly become tedious and laborious (Rossi & Tessman, 2017). *Stochastic aggregation method* allows the system to randomly select a rule to apply at each step, hence allowing more open and unexpected growth process that can be challenging to control.

When using a stochastic approach, it is critical to consider the probability distribution that determines the likelihood of a random choice. In probability theory, an *uniform distribution* describes a statistical function in which all possible values are equally likely to be selected, hence the outcome is arbitrary. Eq. [4.1.1] and Fig. 4.1.1 describe the general *probability mass function* of discrete uniform distribution.

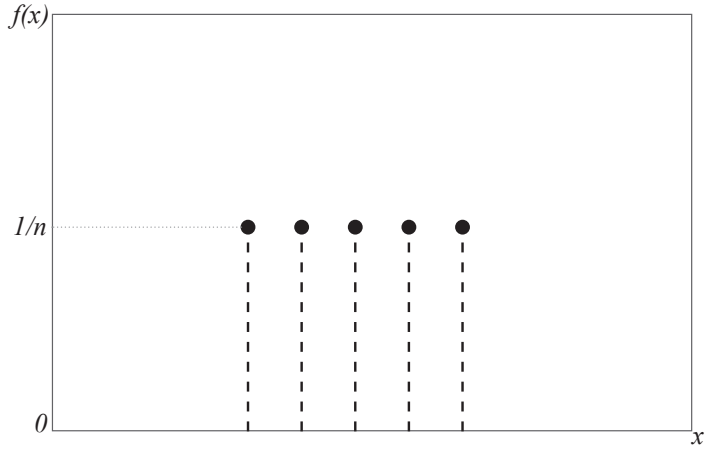


Figure 4.1.1. Probability mass function of discrete uniform distribution. *a* is the lower bound of possible values while *b* is the upper bound. *n* represents the total number of possible values. If selecting a production rule using uniform distribution, *n = 2, a = 0 (naive rule), b = 1 (extension rule).* In the case of rotation angle, these parameter would depend on the *angle step size* and computed angle range (Section 3.2).

$$f(x) = 1/n \qquad \text{Eq. [4.1.1]}$$

where $f(x)$ describes the probability of a discrete random variable x, and n denotes the number of all possible values in the set.

Since the two main production rules, the *naive rule (rule 0)* and the *extension rule (rule 1),* each produce results with distinct characteristics (Fig. 3.2.6), arbitrarily selecting a rule at every step based on *uniform* probability distribution is unlikely to lead to designs with desirable architectural quality. Additionally, if *the naive rule* is chosen, the *rotation angle* and the *position parameter* of the new module also need to be selected (Section 3.2.1). Selecting all these parameters using *uniform* distribution is not an effective approach to generate designs with similar design languages. Instead, this section proposes a method of sampling from a *normal distribution* based on the current state of the assembly and user preferences. The general form *probability density function* of a normal distribution is described below.

$$f(x) = (1/\sigma\sqrt{2\pi}) \times e^{-(1/2)((x-\mu)/\sigma)^2} \qquad \text{Eq. [4.1.2]}$$

where $f(x)$ is the probability density of a real value x, μ is the mean or *expectation* of the distribution, and the σ is the standard deviation. The variance of the distribution is σ^2.

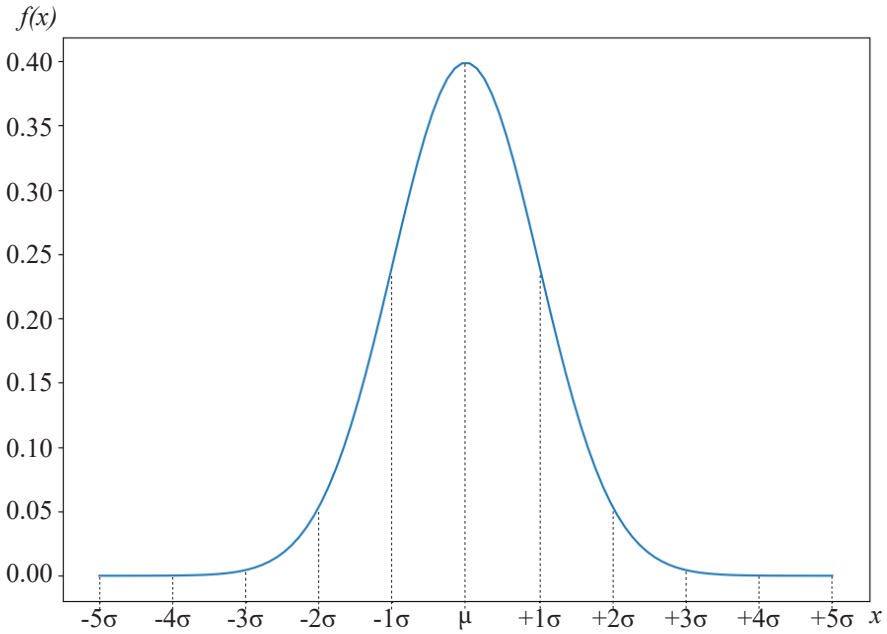


Figure 4.1.2. Probability density function of a normal distribution. $f(x)$ is the probability density of a real value x, μ is the mean, or *expectation,* of the distribution, and the σ is the standard deviation.

Normal distributions, or also known as Gaussian distribution, are important in statistics as they often accurately describe the expected distribution of random values of natural phenomena (Lyon, 2014). For the purpose of this thesis, normal distributions' unique properties can give user direct control of the probability distribution of the random sampling process at each step, by adjusting the *standard deviation (σ)* or the *mean* (μ). For instance, when selecting the *position parameter*, keeping the *mean* at *0.5* and setting the *standard deviation* to a relatively small value is more likely to result in placing a new module around midpoint of an *existing* module's edge. This is a promising way of controlling the stochasticity of the generation process while still allowing the system to explore a diverse set of possible designs. This section presents how different values of standard deviation can affect different aspects of the design outputs, quantitatively and qualitatively.

4.1.1 Rule Selection

As previously shown in Chapter 3, applying the *naive rule* tend to result in sprawling structures that have higher *complexity score* while the *extension rule* produces more compact configuration of modules with relatively low complexity score. Designs with high complexity scores are more visually interesting and perform well in terms of porosity. However, when considering the constructibility and structural performance of multi-story buildings, very complex configurations are not as desirable. Designs with lower complexity scores can sometimes exhibit more desirable architectural qualities, such as providing more flexible and functional spaces, but they are often less creative solutions, and do not address other design concerns well. The goal is to balance the qualities produced by these two rules to achieve complexity scores in a desirable range.

Instead of selecting a random rule arbitrarily based on a given probability distribution, the proposed method uses the complexity score of the current assembly to make a more informed choice. At every step in the generation process, a random *benchmark score* is selected using a specified *normal* probability distribution. As the shown in Fig. 4.1.3, the *mean (μ)* of this probability density function is set to *0.5.* Users can adjust the likely range of this randomly selected benchmark score by adjusting the *standard deviation (σ)* of the probability distribution,

which will bed referred to as the *rule control factor* throughout this thesis. For instance, setting the rule control factor to *0.1* would likely yield a benchmark score around 0.5.

The complexity score of the current state assembly is then compared to the selected benchmark score when deciding which rule to apply. If the current complexity score is *lower* than the benchmark score, the *naive rule* will be applied. Otherwise, the *extension rule* will be applied. This means that the likelihood of applying the naive rule is much higher when a higher benchmark score is selected, vice versa.

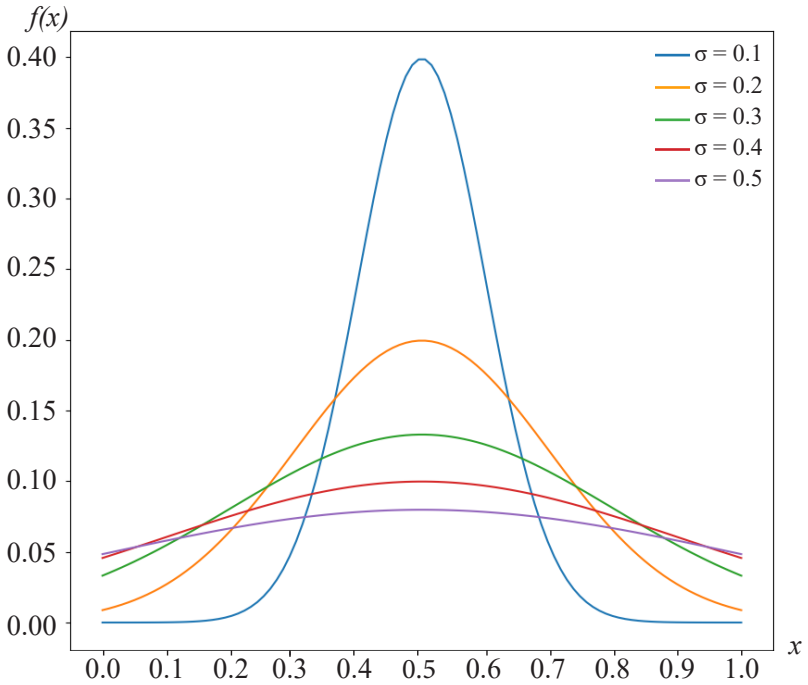


Figure 4.1.3. Overlay of probability density functions with different *rule control factor (σ)*. *f(x)* describes the probability of selecting a benchmark scores *x* (defined in Section 4.1.1). Using control factor of 0.1 would most likely yield a benchmark score between 0.4 and 0.6 while the probability is *almost* uniformly distributed when the control factor is set to 0.5.

As demonstrated in Fig. 4.1.4, smaller values of rule control factor form a more deterministic system where the complexity score of the design outputs are likely to be lower, On the other hand, larger values would result in a more diverse range of complexity scores. Setting the control factor to *0* would change the probability distribution to a *uniform* distribution, in case the user wishes to experiment with a “completely” stochastic aggregation process. Visually, the design samples in Fig. 4.1.5 show that smaller rule control factors produce compact configurations with similar design language while larger control factors lead to more "chaotic" configurations that are

similar to the pattern previously seen in Fig. 3.2.8 (when the explicit rule sequence [0,0,1] was applied repeatedly).

The results presented in this section demonstrate how designers can control the growth pattern of modular aggregations without explicitly specifying the sequence of rules to apply at each step. Section 4.1.2 and 4.1.3 further discuss how this method can be used to select other parameters to generate specific forms that follow the creative intent of designers.

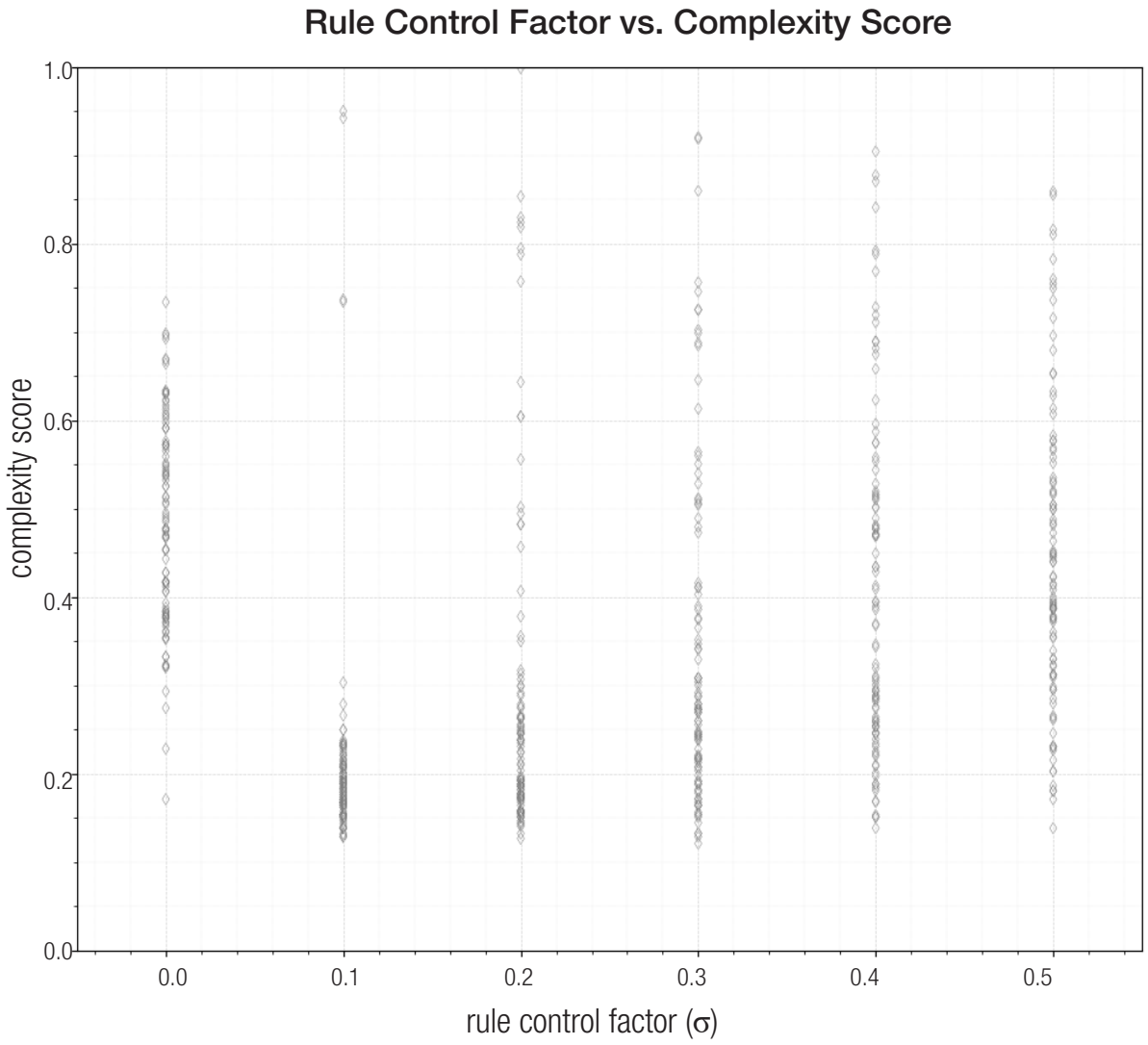


Figure 4.1.4. Complexity vs. rule control factor data based on 600 generated design samples. 100 designs are evaluated for each rule control factor (Section 4.1.1). The *rotation angle* and the *position parameters* (Section 3.2.1) are randomly selected from *uniform* distribution to best demonstrate the effect of rule control. Control factor of 0 denotes sampling rule from a *uniform* distribution. As expected, lower values of control factors result in concentrated score in the lower range while higher values have a wider range of scores.

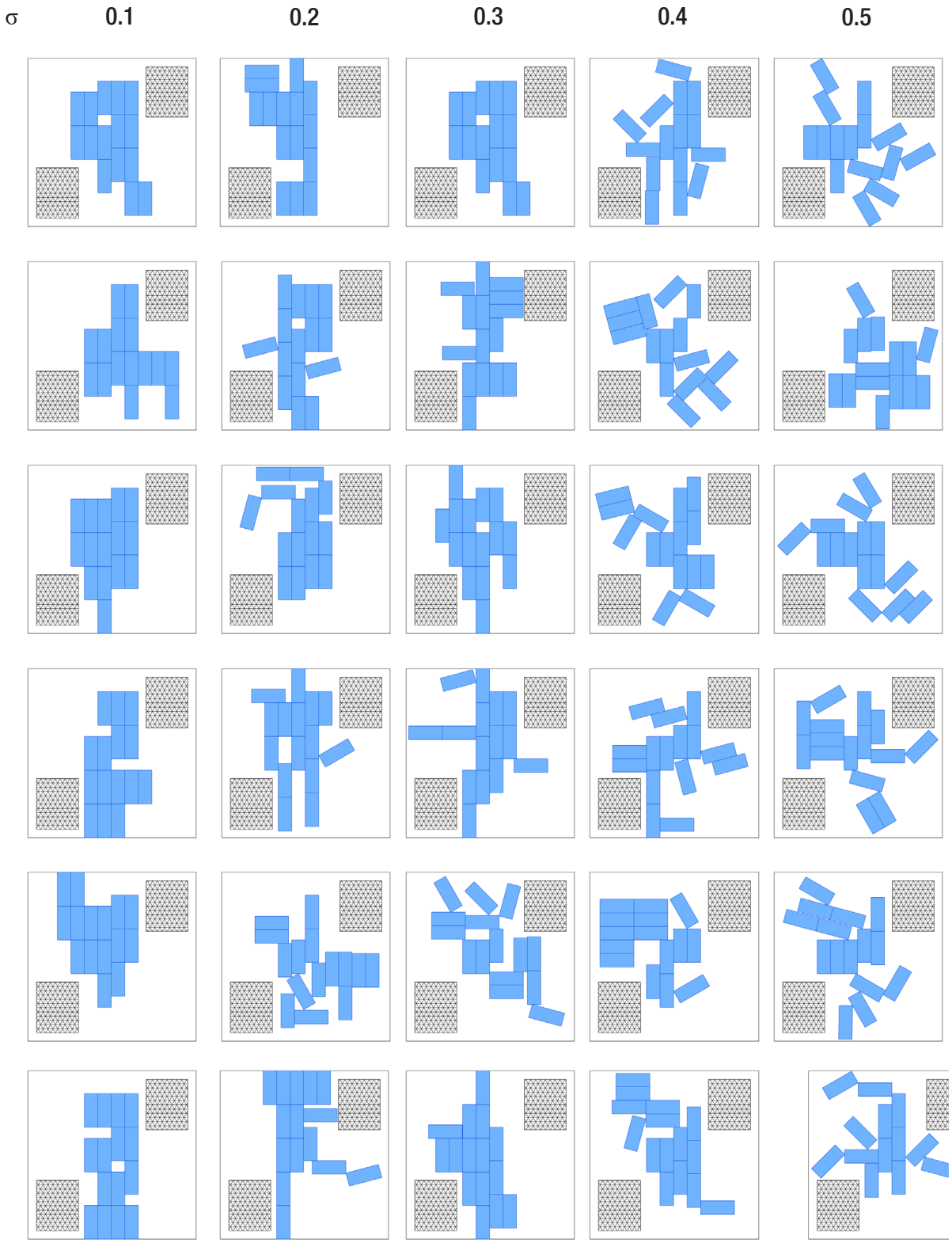


Figure 4.1.5. Design samples of 1-layer assembly of 15 modules generated using different rule control factor (σ) and uniformly sampled *rotation angle* and *position parameter*. These designs demonstrate how different σ values can affect the qualitative characteristics of the design outputs in addition to the numeric *complexity score* shown in Fig. 4.1.4.

4.1.2 Position Selection

Recall Section 3.2.1, when the *naive rule (rule 0)* is selected during the generation process, the system has to select two other random values: the *position parameter* and *rotation angle*. This section focuses on the sampling method of *position parameter*, which determines the point on an *existing* module edge where the new module's *origin* (Section 3.1.2) would be. The method is similar to the rule selection procedure described in the previous section. Every time the naive rule is chosen, a random variable representing the *position parameter* is sampled using a *normal distribution* based on specified *position control factor (μ)* (Fig. 4.1.6). The *mean (μ)* of this normal distribution is centered at *0.5*. The position parameter describes the point at the normalized length along a given line segment (Section 3.2.1). Setting the mean to *0.5* means that new modules are more likely to be attached to the midpoints of existing module edges. This choice of mean value can be adjusted by designers to explored different topology of assembly and address their own design concerns. Based on experiment results, attaching new modules at the midpoints produces more interesting spatial qualities. Offsetting the modules can create functional *negative spaces* for circulation or outdoor access.

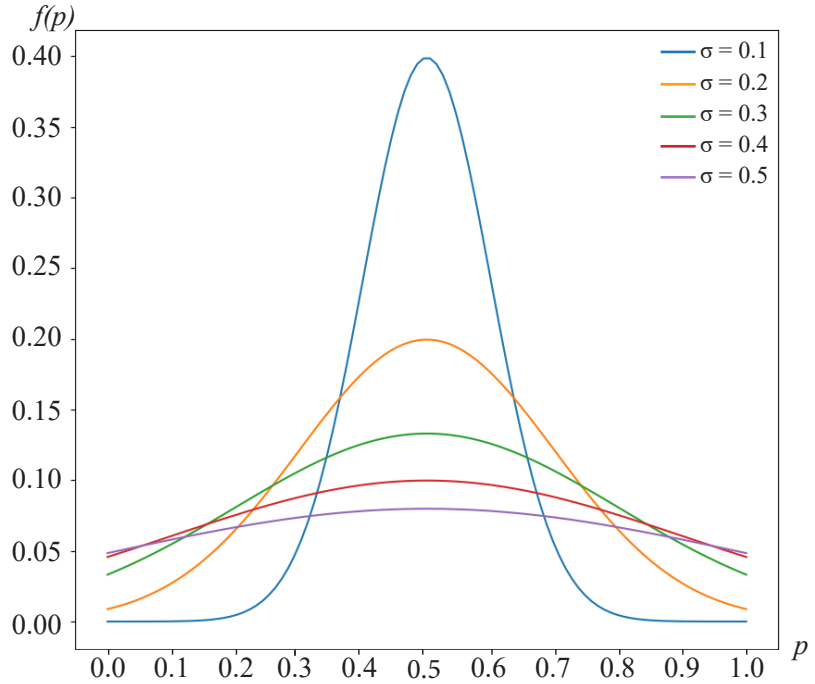


Figure 4.1.6. Overlay of probability density functions with different *position control factor (σ)*. *f(p)* denotes the probability density for a particular *position parameter p.* When the normal distribution is centered at 0.5, the placement of new modules are more likely to be around the midpoints of an existing module edge.

Fig. 4.1.7 shows the complexity scores of 600 designs generated using different position control factors and applying only the naive rule. The *rotation angle* of these designs are constrained to {90°, 0°}. In contrast to the relation seen in the rule selection process (Fig. 4.1.4), there is no clear correlation between the numerical score and the position control factor. The concentration of scores on the higher end of the scale reflect the behavior of applying only the naive rule. However, Fig. 4.1.8 shows that the organization of modules can be related to their position control factor visually.

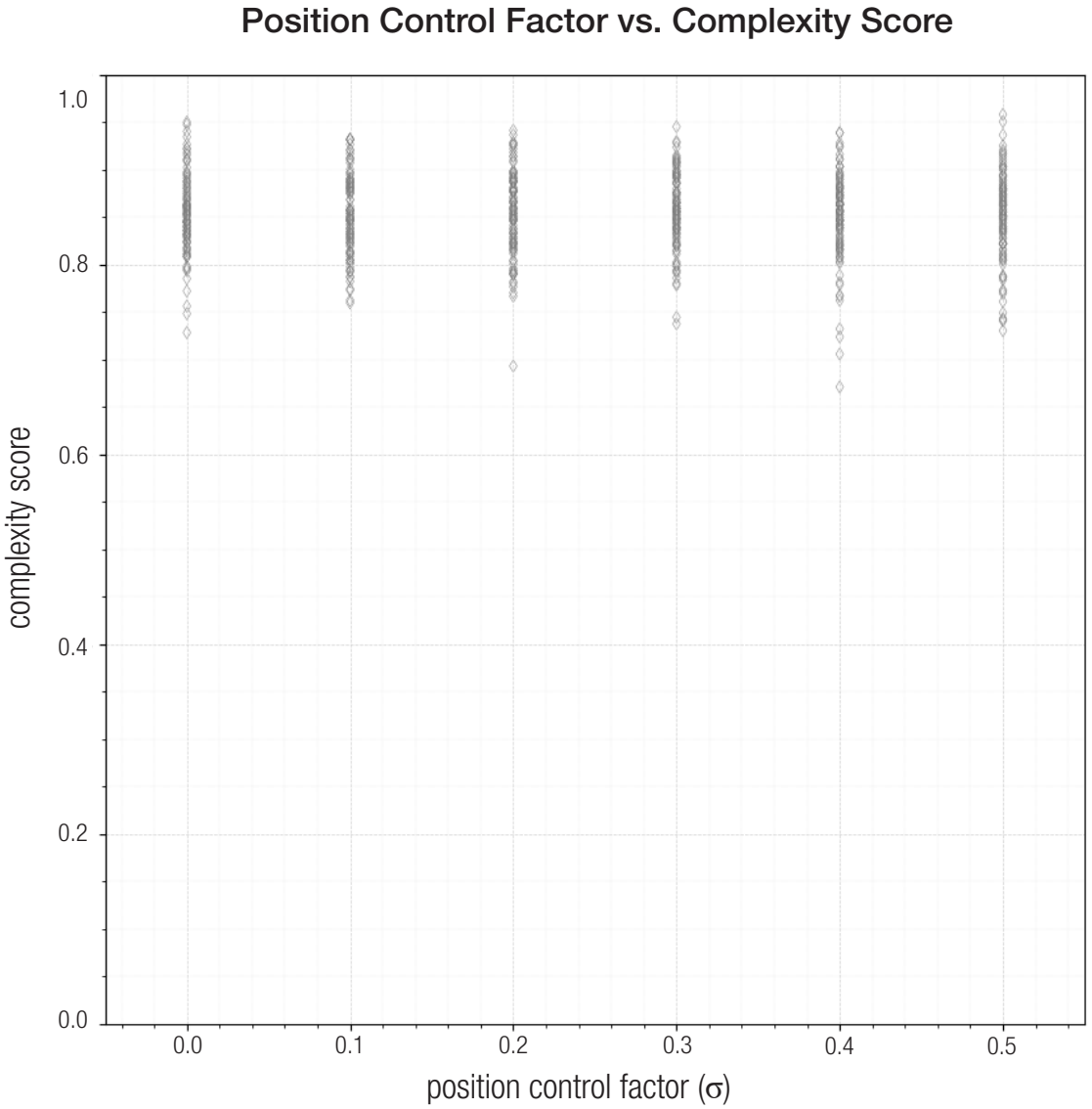


Figure 4.1.7. Complexity vs. position control factor data based on 600 designs gnerated with only the *naive* rule. 100 designs are evaluated for each position control factor (σ) (Section 4.1.2). The *rotation angle* (Section 3.2.1) is randomly selected from *{90°, 0°}* best demonstrate the effect of position control. Control factor of 0 denotes sampling position parameter from a *uniform* distribution. Complexity score are concentrated on the higher end because only the naive rule was applied.

Figure 4.1.8. Design samples of 1-layer assembly of 15 modules generated by applying only the naive rule with different position control factor (σ) and uniformly sampled *rotation angle* from {90°, 0°}. Even though the correlation is not as obvious, one can still observe that the spatial qualities and organization of the assemblies seem to vary with the control factor. Lower σ values force all modules to be connected around the center of their edges.

4.1.3 Rotation Angle Selection

In addition to the *position selection* procedure discussed in Section 4.1.2, when the *naive rule (rule 0)* is selected during the generation process, the system has to choose the *rotation angle* (detailed in Section 3.1.2) which determines the angle between the new module and the existing module it would be attached to.

To generate more feasible architectural outputs, rotation angles are always chosen from a specific set of discrete angles, as discussed in the Section 3.2.1. Each angle in the set of possible rotation angles is associated with its own unique set of *real* numbers. Whenever the naive rule is chosen, a random number is selected using the specified *normal distribution*, and then the corresponding rotation angle would be applied to the new module. The *mean (μ)* of this normal distribution is centered at the values associated with orthogonal rotations. Non-orthogonal angles are disfavored in the aggregation procedure because they are more difficult to construct and sometimes produce unusable corner spaces in resulting designs. Considering the set of possible rotation angles {0º, 15º, 30º, 45º, 60º, 75º, 90º}, the normal distribution would be centered around 90° and 0°. Fig. 4.1.9 illustrates the probability of each of these angles being chosen when the *angle control factor (σ),* the standard deviation of the normal distribution, is set at *0.1.* Increasing the angle control factor would significantly increase the likelihood of applying non-orthogonal rotation at each aggregation step.

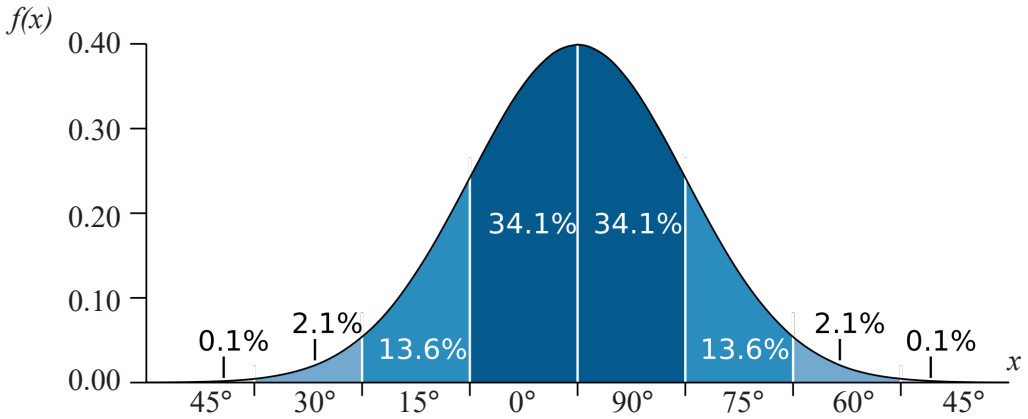


Figure 4.1.9. Probability density function with *angle control factor (σ)* set to 0.1 as discussed above. *f(x)* describes the probability of selecting a particular real number *x*. Each number *x* is associated with a discrete rotation angle to be applied. Lower values of angle control factors strongly favors orthogonal rotations over other possible angles.

Fig. 4.1.10 compares the complexity scores of designs generated using different angle control factor. Control factor of *0* means that the rotation angle is sampled from a *uniform* distribution. Since the angle control factor is not considered when applying the extension rule, these designs were generated by applying only the naive rule. As seen previously, applying only the naive rule lead to designs with extremely high complexity score. Though, the correlation between the complexity score and the control factor is not as obvious as seen in the rule selection process (Fig. 4.1.4), there is still a slight upward trend in the distribution of the scores as the control factor value increases.

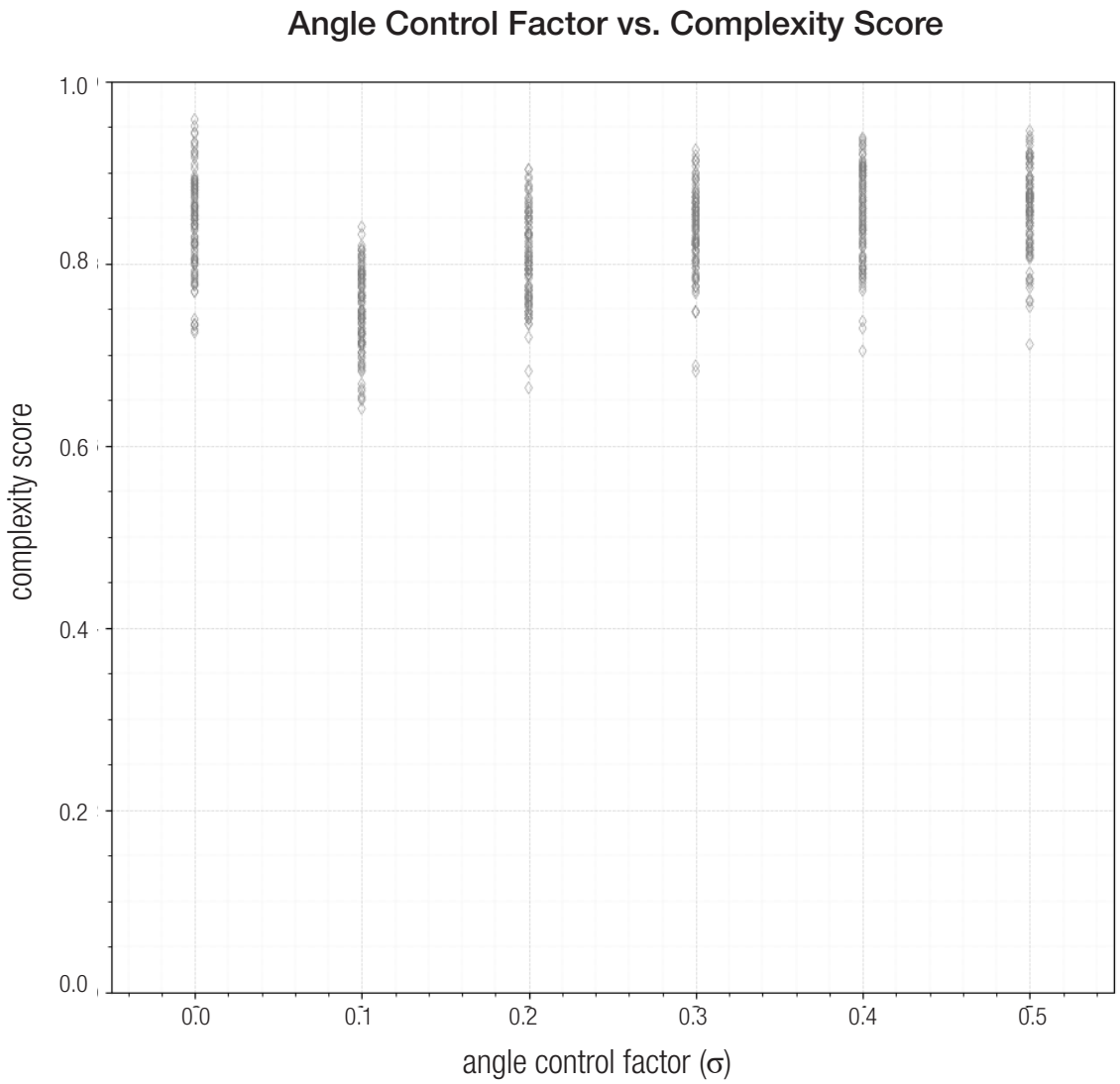


Figure 4.1.10. Complexity vs. angle control factor data based on 600 generated design samples. 100 designs are evaluated for each angle control factor (discussed in Section 4.1.3). These aggregations were generated using only the naive rule, and sampling the *position parameter* from a *uniform* distribution. Control factor of 0 denotes sampling rule from a *uniform* distribution. The correlation between the control factor and the complexity score is less prominent as seen in Fig. 4.1.4, but obvious visual correlation between the two can be observed in Fig. 4.1.10.

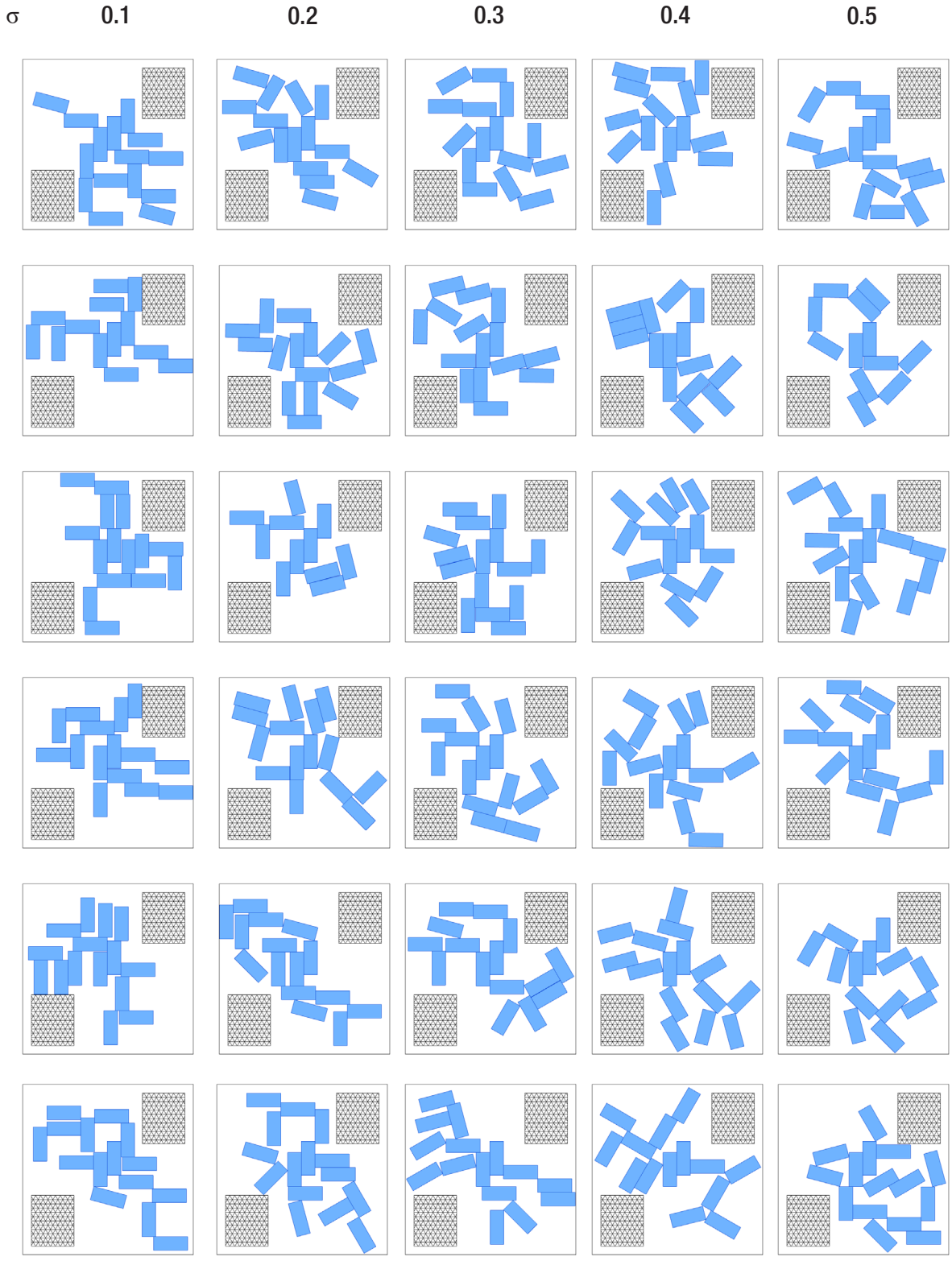


Figure 4.1.11. Design samples of 1-layer assembly of 15 modules generated applying only the naive rule with different angle control factor (σ). The *position parameter* is sampled from a *uniform* distribution. Visually, the relations between modules are influenced by the angle control factor. Designs generated using smaller values of angle control factor lead to more orthogonal rotations between modules while higher σ lead to patterns that are not structured on a orthogonal grid.

Similar to behaviors discussed in Section 4.1.2, the effect of angle control factor is more prominent visually. Smaller angle control factor clearly lead to more "structured" design outputs. When the control factor is set to 0.1, almost all modules are organized on a clear orthogonal grid. Using angle control factor of 0.5 produced many results that seemed to be growing in a circular pattern. Interestingly, as seen in Fig. 4.1.8 and Fig. 4.1.11, both position and angle control factors seem to influence the directionality of growth and how the assembly interacts with the *void* (hatched areas in the figures) defined by the user (Section 3.1.3). This provides the opportunity for designers to consider how they want the structures to respond to their site conditions, such as the topography and existing infrastructures.

The stochasticity control methods presented in these sections demonstrate a promising way to allow designers control the *complexity* and growth pattern of the generated results while exploring a diverse set of possible solutions. Instead of specifying explicit rule sequences, designers can potentially formulate their desired design language by experimenting with different combinations of control factors. Chapter 5 further explores the relation between different values of control factors and other quantitative measurements, specifically structural performance.

4.2 Algorithm Overview

This section presents an overview of how the stochasticity control methods discussed in the section 4.1 are integrated into the generation algorithm of the whole system. Fig. 4.2.1 summarizes the algorithm framework of the proposed automatic stochastic generation method. As previously illustrated in Fig. 3.1.1, designers first formulate the problem by specifying the required user inputs. In addition to the user specifications introduced in Section 3.1.3, designers may also specify the control factor values, or select the default values, to initialize the generation process. Step 3 to step 10 are repeated until each assembly reaches a termination condition, either satisfying the desired number of modules of the current layer or the system fails to find a valid placement within the maximum number of iterations allowed. Design outputs are displayed for the user once all the assemblies are completed. Users can then evaluate the designs, adjust design problem specifications as desired, and select their favorites as the base geometries for

the generation of next layer. On average, generating 100 assemblies of 15 modules takes around 8 seconds. The efficiency of the generation process allows designers to efficiently repeat this process and iterate on different design ideas.

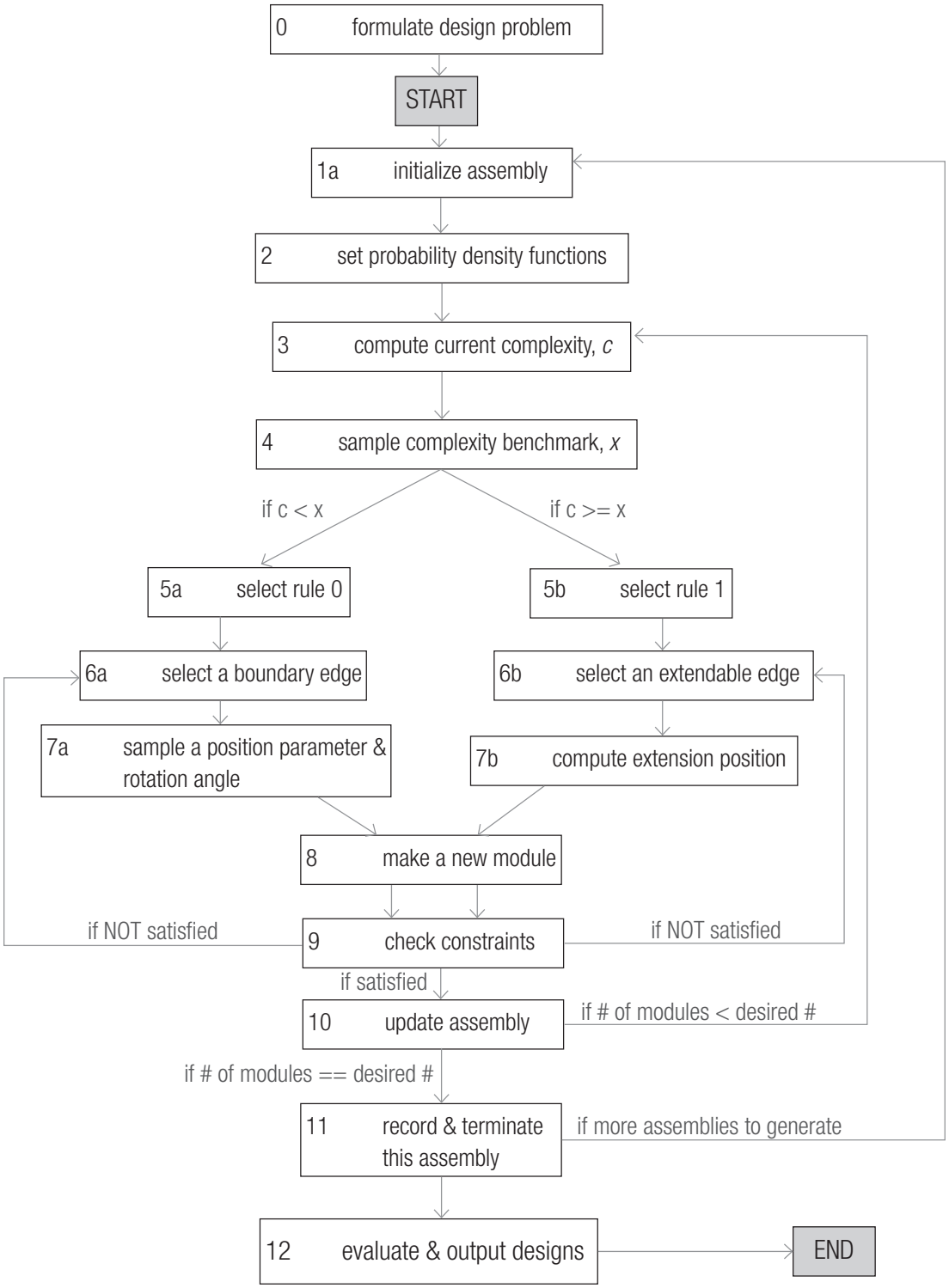


Figure 4.2.1. Framework of the proposed stochastic aggregation algorithm.

Chapter 5

Case Studies

Chapter 3 and Chapter 4 presented a grammar-based aggregation methodology that allows designers to directly interact with the generation process of modular structures. This chapter presents additional results to demonstrate the ability of this system at balancing the qualitative and quantitative design goals. A few design examples are also included in this chapter to show the potential usage of this generation system for solving challenging design problems.

5.1 Tradeoff Between Complexity and Structural Performance

Generally, more complex configurations of modules often lead to more desirable qualitative traits but they are expected to perform worse structurally when stacked in multiple layers. The balance of qualitative and quantitative design goals can be treated as a multi-objective optimization problem, using the structural performance score and complexity score (defined in Section 3.4). This section formulates a specific design problem to explore the *Pareto frontier* of these two numerical metrics. Fig. 5.1.1 illustrates the user inputs and the ground layer structure selected as the base geometry to initialize the generation of the second layer.

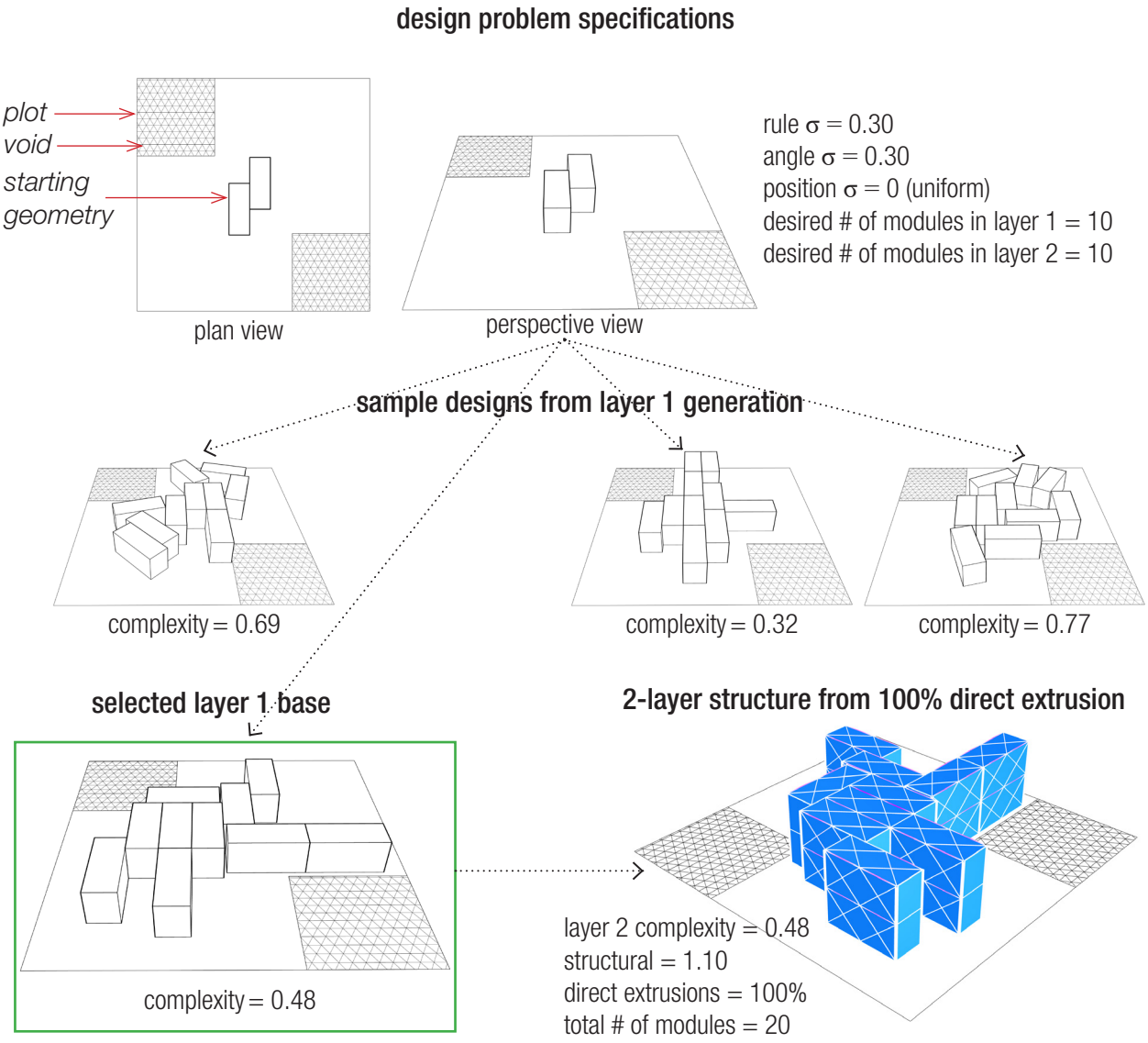


Figure 5.1.1. Design problem formulation for exploring the Pareto frontiers for the complexity score and structural performance score of 2-layer assemblies. The performance of the 100% directly extruded 2-layer structure is shown as base case. Lower structural score indicates better structural performance.

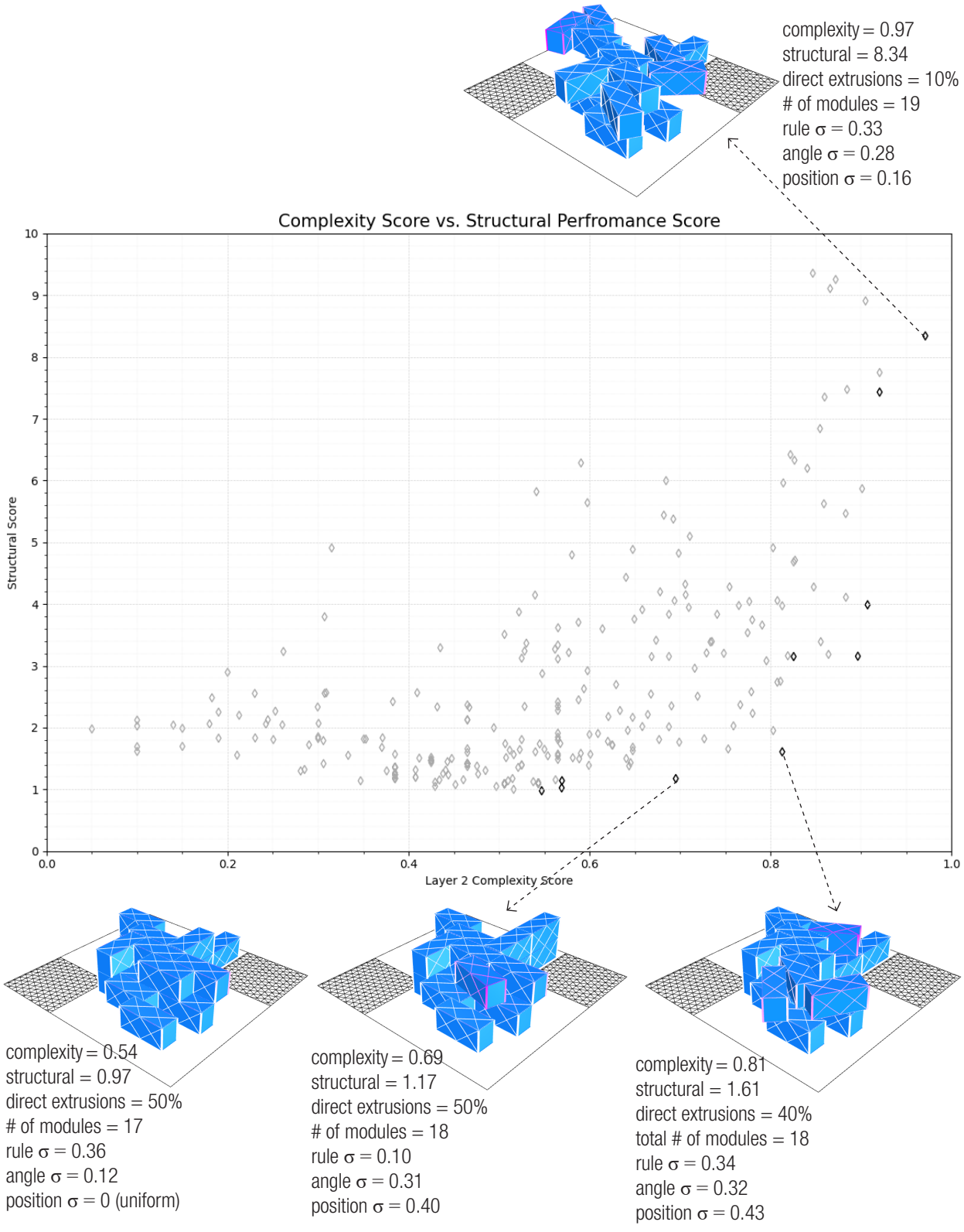
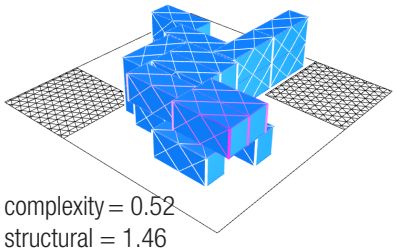


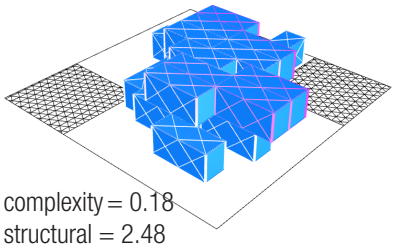
Figure 5.1.2. Complexity score vs. structural performance score of 300 2-layer designs illustrating the Pareto tradeoff. The Pareto optimal solutions are indicated with darker outlines in the plot. These data show that many solutions have similar level structural performance as that of the directly extruded assembly shown in Fig. 5.1.1.

In multi-objective optimization, *Pareto frontier* is defined as a set of non-dominated solutions in which all objectives are considered equally optimal and no objective can be improved without sacrificing the others. Fig. 5.1.2 shows the structural performance score and complexity score of 300 designs of 2-story modular structure generated by randomly sampling control factor (σ) values (Section 4.1). The data shows expected correlation between the complexity and structural performance, and the Pareto tradeoff is clear. However, it is also evident that simple configurations with lower complexity scores do not necessarily lead to the best structural performance. In fact, the best structural performing solutions are concentrated between 0.4 and 0.5 on the complexity score axis. This shows that the designs with similar complexity score in each layer tend to perform better since the ground layer complexity score in these designs is 4.8 (Fig. 5.1.1). Designs with similar complexity score tend to have more overlapping areas and less cantilevering, which again shows that the amount of cantilevering can have a large impact on the structural performance.

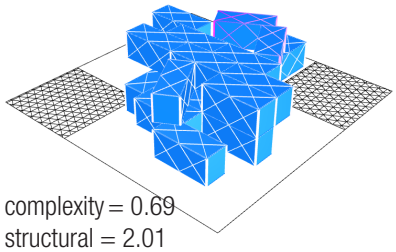
Additionally, this case study also demonstrates that many more interesting aggregations can have similar, sometimes even better, structural performance as the directly extruded assembly. Note that the Pareto optimal designs shown in Fig. 5.1.2 do not have the desired number of modules as specified by user input. Designers are given the freedom of deciding how they wish to consider these different factors when selecting their favorite designs. Analyzing the correlation between qualitative and quantitative performance of a large variety of designs numerically and visually can further help designers understand the behavior of modular structural systems and develop desirable design solutions. Fig. 5.1.3 shows a few additional designs near the Pareto frontier.



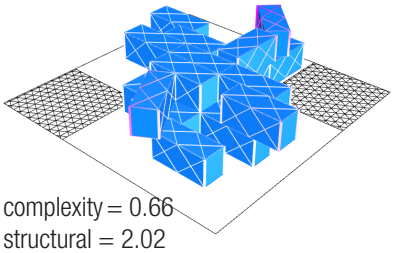
complexity = 0.52
structural = 1.46
direct extrusions = 60%
# of modules = 20
rule σ = 0.09
angle σ = 0.12
position σ = 0 (uniform)



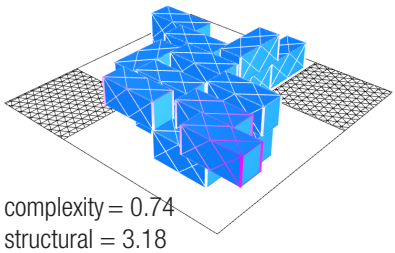
complexity = 0.18
structural = 2.48
direct extrusions = 20%
# of modules = 20
rule σ = 0.24
angle σ = 0.34
position σ = 0.1



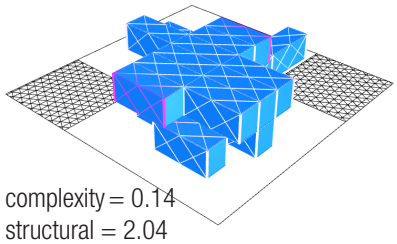
complexity = 0.69
structural = 2.01
direct extrusions = 60%
# of modules = 20
rule σ = 0 (uniform)
angle σ = 0.26
position σ = 0 (uniform)



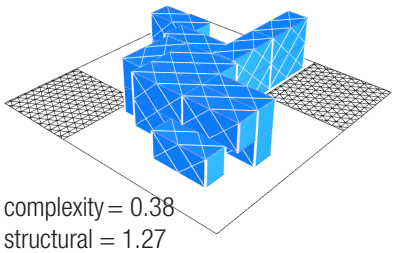
complexity = 0.66
structural = 2.02
direct extrusions = 30%
# of modules = 20
rule σ = 0.49
angle σ = 0.41
position σ = 0.56



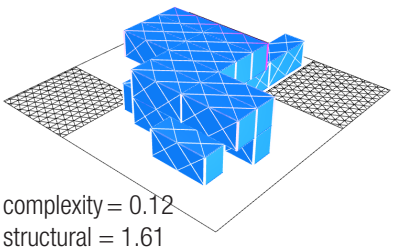
complexity = 0.74
structural = 3.18
direct extrusions = 30%
# of modules = 20
rule σ = 0.27
angle σ = 0.09
position σ = 0.11



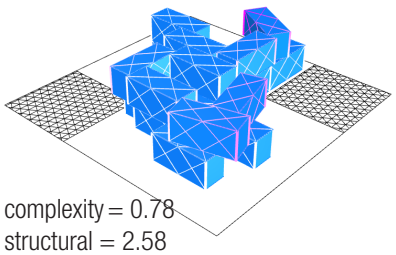
complexity = 0.14
structural = 2.04
direct extrusions = 10%
# of modules = 20
rule σ = 0.10
angle σ = 0.20
position σ = 0.38



complexity = 0.38
structural = 1.27
direct extrusions = 70%
# of modules = 20
rule σ = 0.12
angle σ = 0.33
position σ = 0.47



complexity = 0.12
structural = 1.61
direct extrusions = 10%
# of modules = 20
rule σ = 0.05
angle σ = 0.33
position σ = 0.47



complexity = 0.78
structural = 2.58
direct extrusions = 30%
# of modules = 19
rule σ = 0.43
angle σ = 0.23
position σ = 0.10

Figure 5.1.3. Additional designs near the Pareto frontier shown Fig.5.1.2.

5.2 Design Example - "Grow Bigger"

To demonstrate how this generation system can be used to find design solutions that follow very specific global forms specified by the designer, this section presents a design example that grows "bigger" as it aggregates up in a "tree form". This particular form presents interesting spatial qualities and porosity that cannot be produced through simple stacking of modules. Formally, this design takes inspiration from Habitat 67, seeking to provide sufficient daylight and outdoor space by arranging the modules with various offsets and rotations. Quantitatively, the structural performance of this particular design scored in the top 10% of 80 designs generated for each layer. Fig. 5.2.1 illustrates how the form of this design was derived by manipulating the user inputs between each iteration of a new layer.

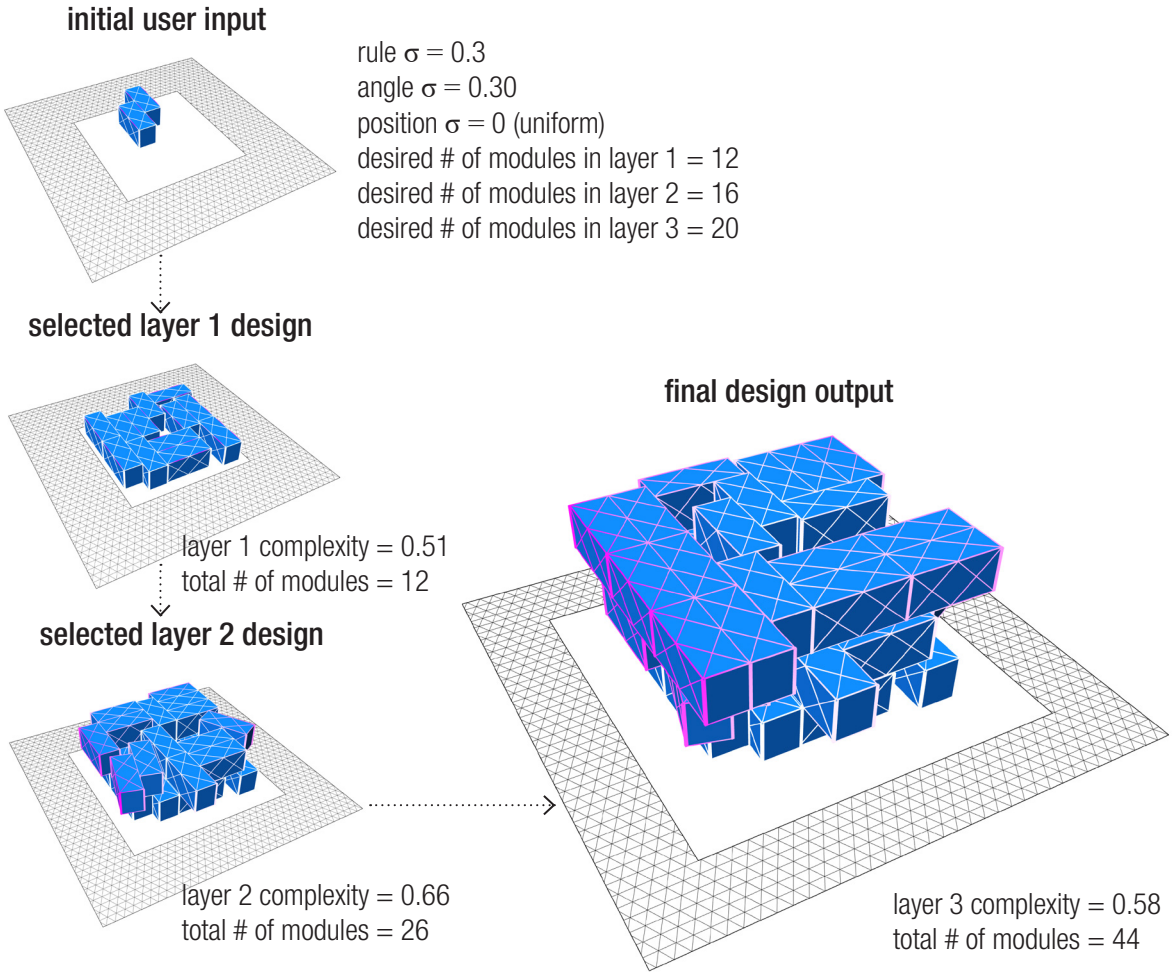


Figure 5.3.1. Generation sequence of "Grow Bigger". The area of the *void* (hatched area) is reduced and the desired number of modules was increased between the generation of each layer to allow the new layer to grow beyond the boundary of its previous layer.

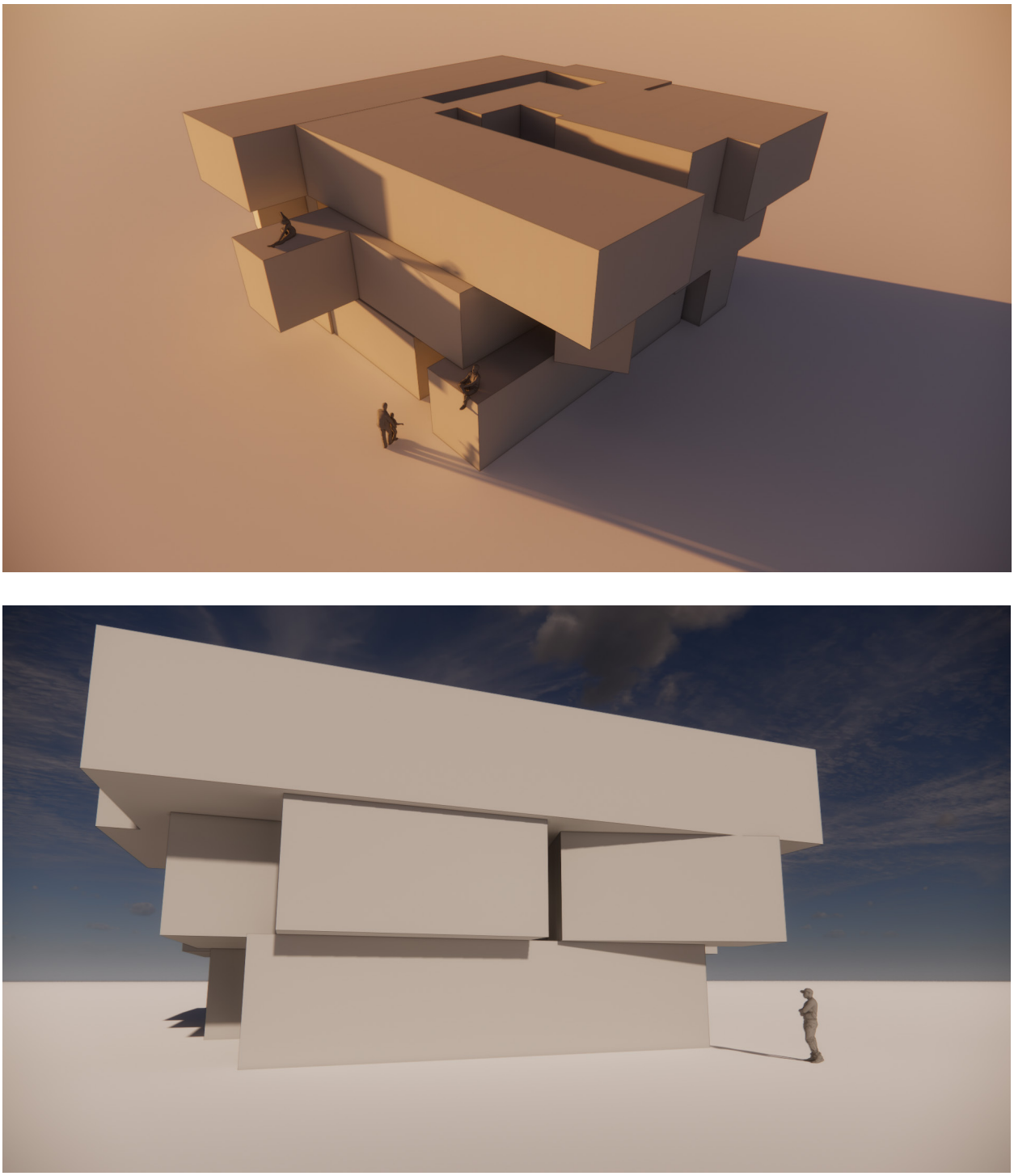


Figure 5.3.2. Massing model renderings of "Grow Bigger". The lack of materiality in these renders suggests another interpretation of the design as a heavier and more monumental structure rather than an aggregation of light-weight steel boxes.

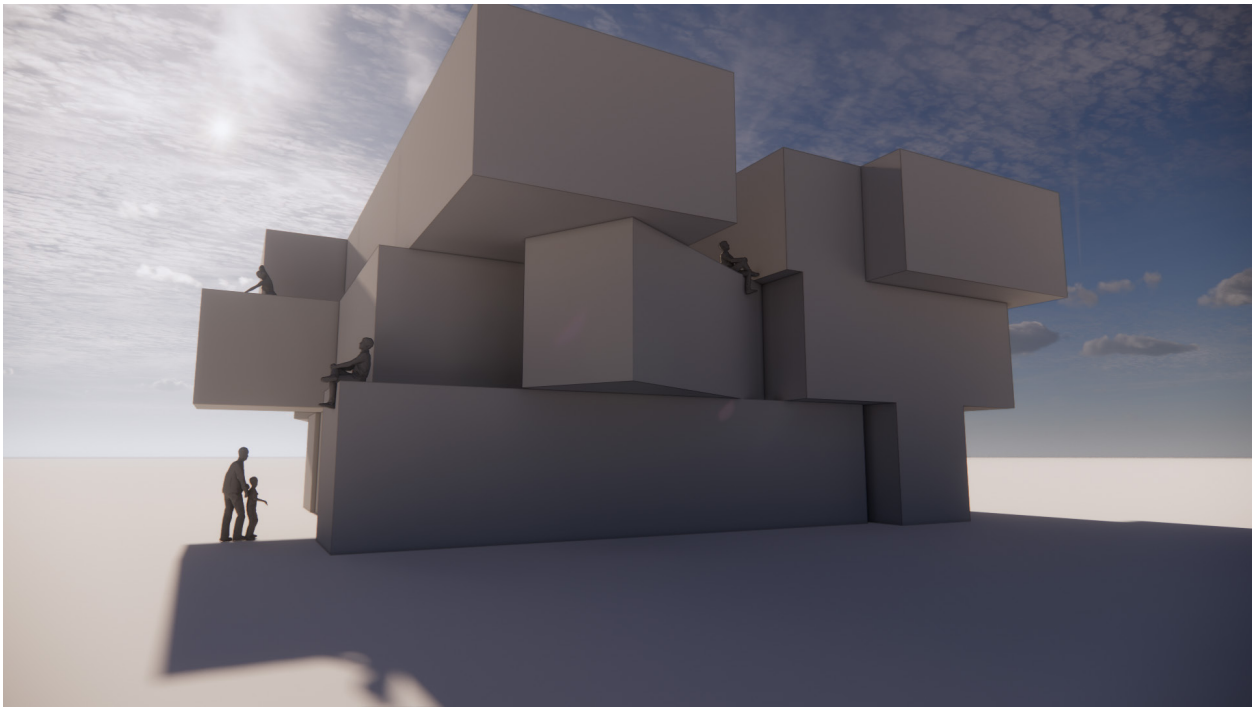


Figure 5.3.3. Massing model rendering of "Grow Bigger". The varying spatial relations between the modules create function negative spaces within the structure.

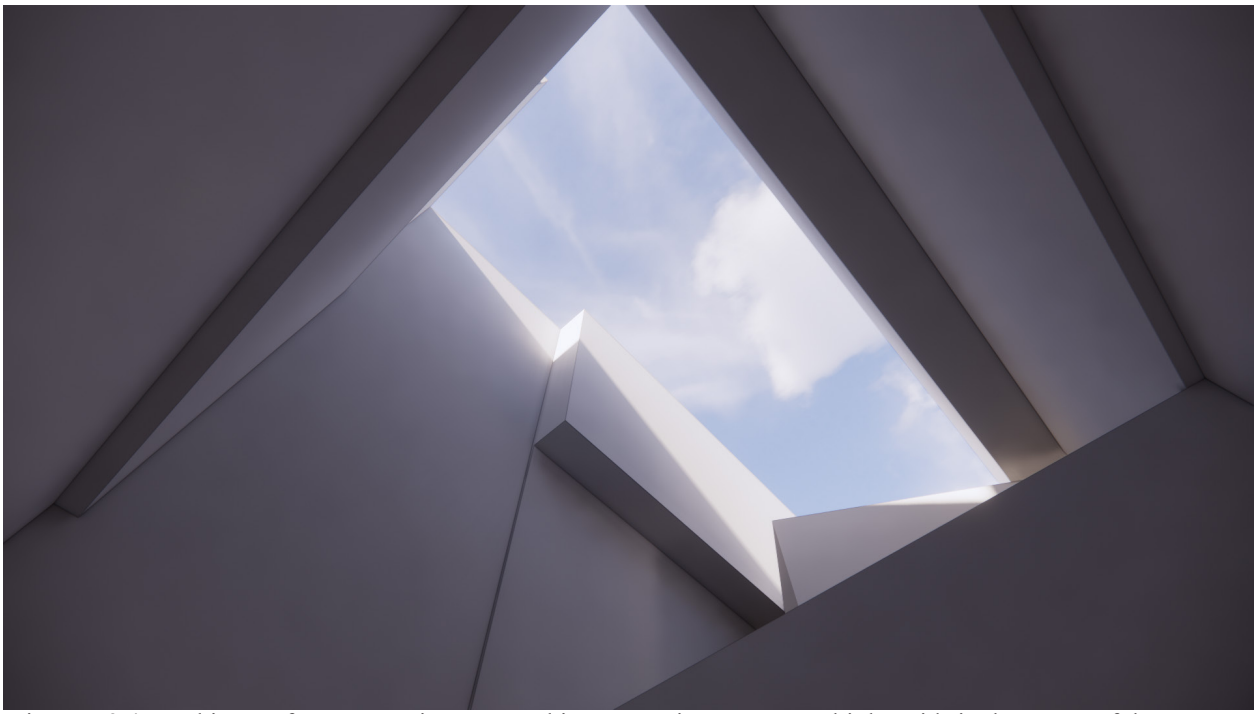


Figure 5.3.4. Looking up from a negative space. This aggregation creates multiple voids in the center of the building, forming more permeable edges between the interior and exterior space.

5.3 Design Example - "Grow Around"

This section presents a playful design example demonstrating how the generation algorithm can be used to find design solutions that navigate around especially constraining site condition. As shown in Fig. 5.3.1, the geometric user inputs consists of a rectangular *plot* with the MIT logo specified as the *void.* Multiple single-module starting geometries are placed around the site. Interestingly, these separate structures start to join and fill up the negative space on the site as the assembly grew larger. Throughout the generation process, all modules were placed strictly outside the logo. This example also illustrates this aggregation method's ability to generate design solutions that respond to very specific and challenging circumstances at an urban scale.

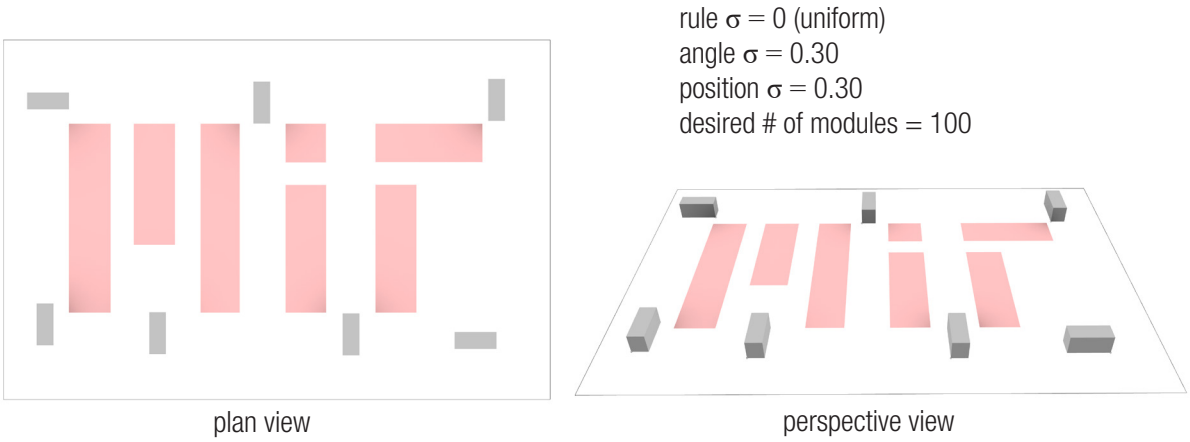


Figure 5.3.1. Design problem specification before initializing the generation procedure. The problem is designed to show how the system can navigate around complex site conditions. The MIT logo is specified as the *void*.

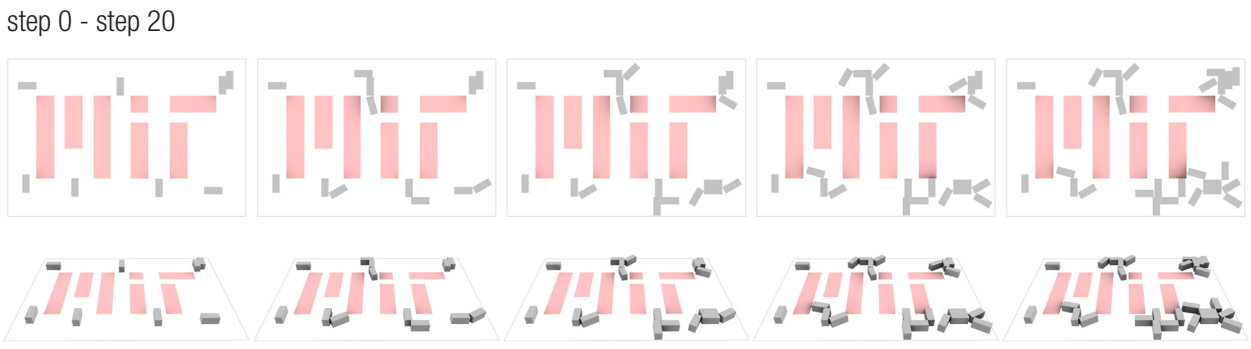


Figure 5.3.2. Generation sequence of "Grow Around". Partial solutions are displayed at every 5 steps.

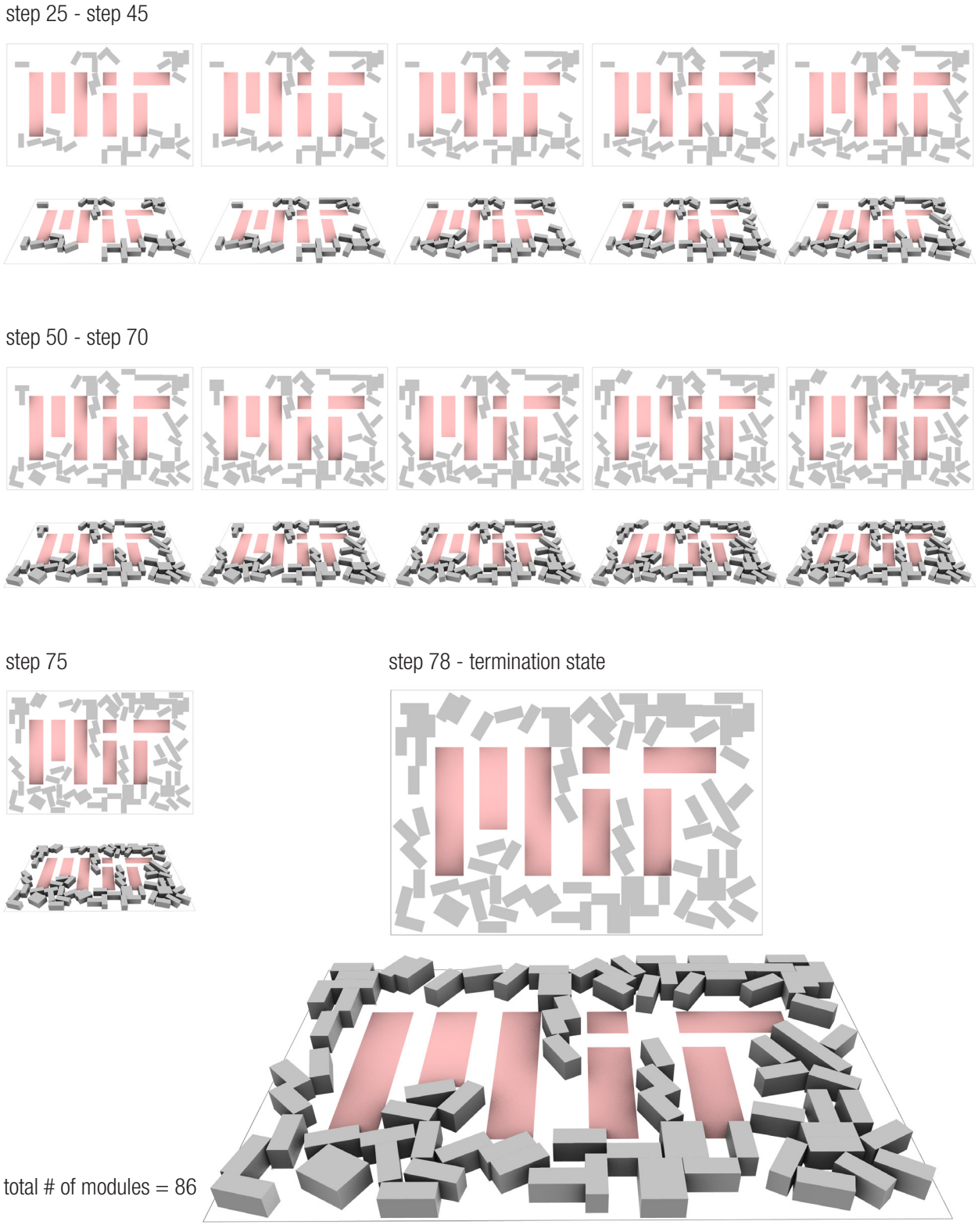


Figure 5.3.3. Generation sequence of "Grow Around". Partial solutions are displayed at every 5 steps. At step 78, the procedure was terminated because the algorithm was no longer able to find a valid placement for a new module within the specified *maximum number of iterations*. Although the final output did not reach the desired number of modules (100), it was able to populate most of the space around the MIT logo.

Chapter 6

Conclusion

This chapter summarizes research work developed in this thesis, highlights the potential impact of this new computational methodology, presents important directions for future work, and offers concluding remarks.

6.1 Summary of Contribution

This thesis presents a new aggregation-based generation methodology for exploring modular design possibilities. With an iterating by layer approach, the aggregation process promotes designer intervention at every stage. The grammar system presented in this thesis consists of a simple and easily understandable set of rules that can define infinitely diverse aggregation topologies.

This thesis improves upon existing stochastic grammar exploration systems by giving designers the control over the stochasticity of the generation process while maintaining the diversity of outputs, allowing discovering unexpected design possibilities. By allowing users to specify design preferences through geometric and numerical inputs, this generation system computes design solutions that responds to a variety of constraints and the creative designer intent.

In addition to commonly used structural performance metrics, this thesis also introduces a new numerical metric, *complexity score* (Section 3.4), to quantify other traditionally unformulated qualitative design goals in early design stage, such as porosity and visual characteristics. A series of designs are selected to demonstrate how geometrically complex modular aggregations could have better daylight performance and exhibit more interesting spatial qualities. The potential tradeoffs between structural performance and geometric complexity are explored in Section 5.1 of this thesis.

The implemented system demonstrates good performance in terms of speed. The algorithm(Section 3.3) and software design (Appendix A) allow geometric constraints and properties be computed in linear time. Thus, designers can generate a large number of potential designs within a few seconds.

Finally, this thesis presents a few case studies demonstrating the potential use of this generation methodology and how it may be adopted in helping designers solve different types of design challenges. Through systematic explorations, designers may find modular design options that well balance the qualitative and quantitative performance goals.

6.2 Potential Impact

This new methodology is a more efficient and systematic approach than the trial-by-error method commonly practiced in the design industry today. Giving designers the ability to explore more diverse design options and evaluate their respective performance can help address designers' concerns about the quantitative benefits of modular structures. This could enable more experimentation of different forms that further advance the discourse around modularity in architecture and broaden the applications of modular structures. This method can potential be extended to address design problems of different scales, and even in fields beyond architectural design. It can be generalized to solve any challenges concerning complex aggregations of simple components, in urban design, structural design or mechanical engineering. For instance, the overall organization of cities can be interpreted as aggregations of communities, land parcels, and transportation networks. The grammar rules and evaluation metrics may need to be redesigned and adapted accordingly to describe new constraints, but the fundamental framework could remain similar.

6.3 Limitations and Future Work

This thesis demonstrates the potential of this new modular design exploration methodology, but there are limitations that should be considered for future work. Real world design problems normally involve more quantitative and qualitative design factors that cannot be described by the two metrics defined in this thesis: the structural performance and complexity score (Section

3.4). Though construction sequencing is considered in the algorithm, there is no defined metric evaluating the cost of construction. Some designs may be easily constructed with a single crane and minimal labor. Others might be impossible to construct without more complicated setup. This might be an especially important for designer to consider when designing for sites that are inaccessible by heavy machinery. It is also critical to note the distinction between the process of assembly and disassembly. Certain designs might be easy to assemble but challenging to disassemble or even to replace a few modules. Since modular structures are often favored for temporary designs, the financial, environmental, and labor cost of disassembly is equally important as that of the assembly process.

Further research is also needed to develop methods to contextualize the performance score of each design. Even though the current complexity metric can be used as an indication of the amount of negative space in a design, not all negative spaces are equally valuable. Certain spacings between modules might be function as circulation space or patio spaces while others might provide no real functions. The performance of each design solution must also be considered with its social, environmental or community context. For instance, spacing for patios or courtyard may be less desirable in colder climate. This is certainly a challenging task to accomplish computationally. It would require developing other systems that allow designers to validate the “scoring of designs” during the aggregation process.

Additionally, developing optimization algorithms to search for optimal solutions based on designer preferences is an important direction for future research. A recurrent neural network (RNN) is a subset of machine learning algorithms that are used for problems involving sequential data or time series data, such as language translation and speech recognition. Under the proposed methodology, each design can be abstracted as a sequence of data, including the sequence of product rules applied and of the specific rule parameters selected. Training sequence models using large datasets of generation outputs may be a promising approach to search for the design solutions that have the best performance in certain user-specified categories or predict sequences of grammar rules that might lead to designs satisfying designers’ qualitative expectations.

Designing more case studies and usability tests with more complex design problems involving module of different materials and forms would provide more guidance expanding the capabilities of this methodology.

6.4 Concluding Remarks

Technological advancement in the 20th century has enriched the world with a new form of architecture that adapts and reacts to the shifting needs of the society through modularity. However, despite the unprecedented development of computing power and fabrication technology in the world today, expressive forms of modular architecture seem to have been largely replaced by tedious stacks of boxes. This change of course in modular design is partially due to the difficulty of finding visually interesting solutions that also meet the intense quantitative performance demands. By proposing a procedural design exploration methodology, this thesis intends to show that it is possible for designers to liberate their creativity by taking full advantage of computation in a human-machine collaborative design process.

Appendices

Appendix A

Implementation

This appendix gives a high level overview of the implementation of the system. The grammar exploration and assembly generation procedures are implemented in *Python* and integrated into *Rhino Grasshopper* environment as a series of components using *Hops,* a *Grasshopper* component allowing external functions to be added as *Grasshopper* definition. Visualization of generation outputs and structural analysis are scripted in *Grasshopper* using *Karamba3D,* a parametric structural analysis tool. Six *Python* classes are defined to operate at different levels of the grammar exploration process:

- *module*
    - properties – stores data for each individual modular unit;,e.g. *origin, angle, dimensions,* etc.
    - methods – modular level operations, e.g. *getBottomEdges(), getVertices(),* etc.
- *assembly*
    - properties – stores data for each generated assembly, e.g. *modules, history, boundary, controlObject,* etc.
    - methods – assembly level operations, e.g. *computeBoundary(), checkAllConstraints(), computeComplexty(),* etc.
- *control*
    - properties – stores data for stochasticity control process, e.g. *ruleControlFactor, angleControlFactor, positionControlFactor,* etc.
    - methods – stochastic procedure operations, e.g. *getRandomRule(), getPositionParameter(), getRotationAngle(),* etc.
- *rules*
    - properties – stores the definition of rules, e.g. *ruleID, canApply,* etc.
    - methods – rule operations; e.g. *applyRuleZero(), applyRuleOne(), etc.*
- *history*
    - properties – stores the generation history of each assembly, e.g. *currentStep, rulesApplied*
    - methods – history recording operations; e.g. *recordCurrentStep(), etc.*
- *grammar*
    - properties – stores high level data for entire generation process, e.g. *allAssemblies, maxIteration, positionControlFactor,* etc.

- methods – highest level generation operations; e.g. *initializeGeneration(), generateSingleLayer(), generateMultiLayer()* etc.

As discussed in Section 3.3 of this thesis, the object oriented system design allows faster computation (at the expense of memory) since different geometric properties and data are stored and updated accordingly at every step of the generation process. Fig. A.1 illustrates how these six different classes are integrated into the system.

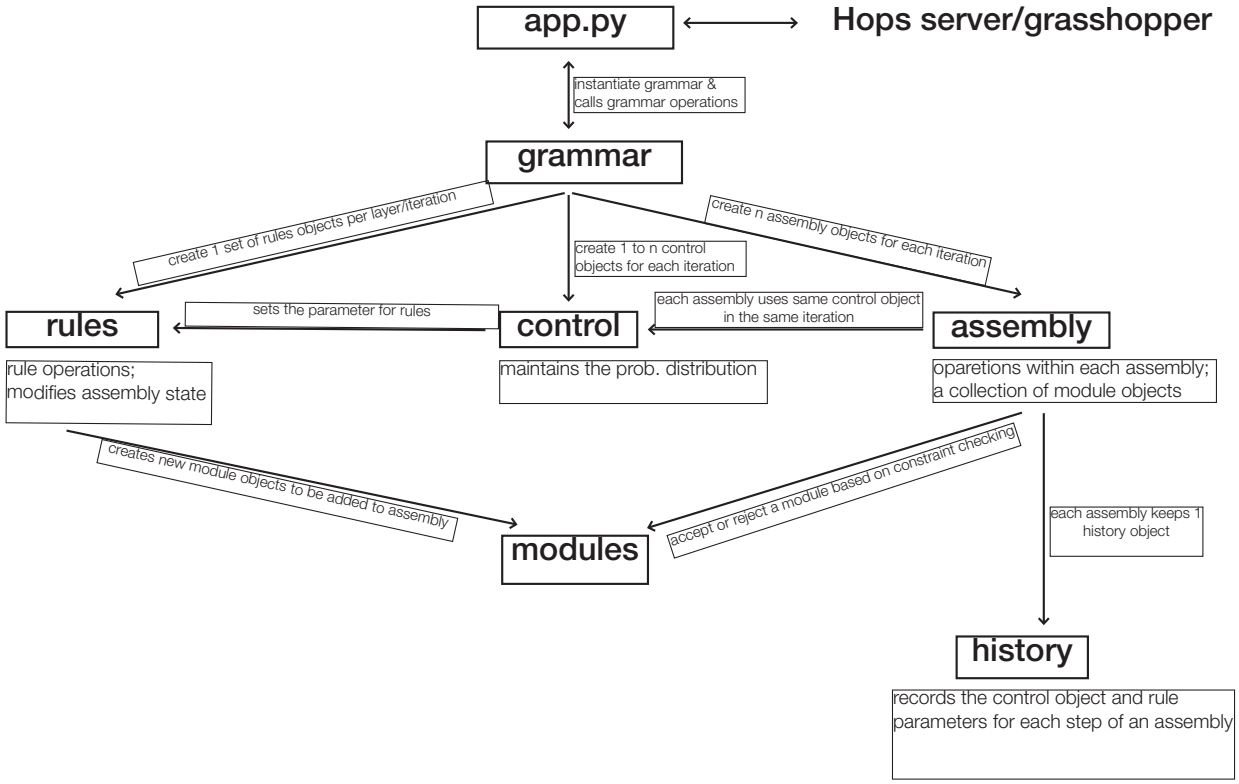


Figure A.1. High-level integration of the system. *app.py* is the entry point that interfaces with *Hops* to start the application.

Appendix B

Bibliography

Asaeedi, S., Didehvar, F., & Mohades, A. (2017). Alpha-Concave Hull, a Generalization of Convex Hull. *Theoretical Computer Science*, 18-59.

Bernardo, L. F., Oliveira, L. A., Nepomuceno, M. C., & Andrade, J. M. (2013). Use of refurbished shipping containers for construction of housing buildings: details for structural project. *Journal of Civil Engineering and Management*.

Boafo, F., Kim, J.-H., & Kim, J.-T. (2016). Performance of Modular Prefabricated Architecture: Case Study-Based Review and Future Pathways. *Sustainability, 8*(6), 558.

Brown, N., & Mueller, C. (2019). Quantifying diversity in parametric design: a comparison of possible metrics. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing, 33*, 40-53.

Brunn, E. P., Ting, I., Adriaenssens, S., & Parascho, S. (2021). Human–robot collaboration: a fabrication framework for the sequential design and construction of unplanned spatial structures. *Digital Creativity*, 320-336.

Dara, C., & Sinclair, B. R. (2018). Liberating Architecture: A Critical Review of the Landscapes of Innovation + Advancement in Modular Design + Construction. *11th International Symposium on Architecture of 21st Century: In Search of New Paradigms.* Baden-Baden, Germany.

Galton, A., & Duckham, M. (2006). What Is the Region Occupied by a Set of Points? *Geographic Information Science. GIScience 2006. Lecture Notes in Computer Science. 4197*, pp. 81-98. Springer, Berlin, Heidelberg.

Knight, T. W. (1993). Color Grammars: The Representation of Form and Color in Designs. *Leonardo, 26*(2), 117-124.

Knight, T., & Stiny, G. (2015). Making grammars: From computing with shapes to computing with things. *Design Studies, Vol. 41*, 8-28.

Lee, J., Fivet, C., & Mueller, C. T. (2015). Modelling with Forces: Grammar-Based Graphic Statics for Diverse Architectural Structures. *Modelling Behavior*, 491-504.

Lee, K. J., & Mueller, C. T. (2021). Adapting computational protein folding logic for growth-based, assembly-driven spatial truss design. *Proceedings of the IASS Annual Symposium.* Guilford, U.K.

Lin, Z. (2011). Nakagin Capsule Tower Revisiting the Future of the Recent Past. *Journal of Architectural Education (1984-), 65*(1), 13-32.

Luo, H., Liu, J., Li, C., Chen, K., & Zhang, M. (2020). Ultra-rapid delivery of specialty field

hospitals to combat COVID-19: Lessons learned from the Leishenshan Hospital project in Wuhan. *Automation in Construction, 119*.

Lyon, A. (2014). Why are Normal Distributions Normal. *The British Journal fo the Philosophy of Science, 65*, 621-649.

Merrell, P., & Manocha, D. (2010). Model Synthesis: A General Procedural Modeling Algorithm. *IEEE Transactions on Visualization and Computer Graphics, 17*(6), 715-728.

Mueller, C. T. (2014). *Computational exploration of the structural design space.* Cambridge, MA: Ph.D. Dissertation, Dept. Architecture, MIT.

Mueller, C. T., & Ochsendorf, J. (2015). Combining structural performance and designer preferences in evolutionary design space exploration. *Automation in Construction 52,*, 70-82.

Navaratnam, S., Ngo, T., Gunawardena, T., & Henderson, D. (2019). Performance Review of Prefabricated Building Systems and Future Research in Australia. *Buildings , 9*(2), 38.

Parish, Y. I., & Muller, P. (2001). Procedural modeling of cities. *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (pp. 301–308). New York, NY: Association for Computing Machinery.

Reinhart, C. (2018). Massing Studies. In *Daylighting Handbook I* (pp. 91-125). Cambridge, MA: Building Technology Press.

Rossi, A., & Tessman, O. (2017). Geometry as Assembly: Integrating design and fabrication with discrete modular units. *eCAADe 35*, (pp. 201-210).

Rossi, A., & Tessmann, O. (2017). Aggregated Structures: Approximating Topology Optimized. *Proceedings of the IASS Annual Symposium.* Hamburg, Germany.

Rossi, A., & Tessmann, O. (2017). Designing With Digital Materials. *Proceedings of CAADRIA 2017.* Hong Kong: The Association for Computer-Aided Architectural Design Research in Asia.

Rossi, A., & Tessmann, O. (2019). From Voxels to Parts: Hierarchical Discrete Modeling for Design and Assembly. In L. Cocchiarella (Ed.), *ICGG 2018 - Proceedings of the 18th International Conference on Geometry and Graphics. ICGG 2018. Advances in Intelligent Systems and Computing. 809*, pp. 1001-1012. Springer, Cham.

Safdie, M. (1970). *Beyond Habitat.* Cambridge, MA: The MIT Press.

Stiny, G., & Gips, J. (1971). Stiny, George and James Gips. "Shape Grammars and the Generative Specification of Painting and Sculpture. *IFIP Congress*.

Tavernier, I., Cambier, C., Galle, W., & De Temmerman, N. (2021). A Conceptual Framework for Interpretations of Modularity in Architectural Projects. In J. Littlewood, H. R., & L. Jain (Ed.), *Sustainability in Energy and Buildings 2020. Smart Innovation, Systems and Technologies. 203*, pp. 127-137. Springer, Singapore.

Thompson, J. (2019). Modular Construction: A Solution to Affordable Housing Challenges.

Cornell Real Estate Review, 17.

Wonka, P., Muller, P., Haegler, S., Ulmer, A., & Van Gool, L. (2006). Procedural Modeling of Buildings. *ACM Transactions on Graphics*, 614-623.

Zhao, A., Xu, J., Konakovic, M., Hughes, J., Spielberg, A., Rus, D., & Matusik, W. (2020). RoboGrammar: graph grammar for terrain-optimized robot design. *Proceedings of SIGGRAPH Asia 2020.*