REVIEW
published: 01 February 2022
doi: 10.3389/fbuil.2021.815153

Graph Rewriting Techniques in Engineering Design

Lothar Kolbeck, Simon Vilgertshofer, Jimmy Abualdenien and André Borrmann*[†]

Chair of Computational Modeling and Simulation, TUM School of Engineering and Design, Technical University of Munich, Munich, Germany

Capturing human knowledge underlying the design and engineering of products has been among the main goals of computational engineering since its very beginning. Over the last decades, various approaches have been proposed to tackle this objective. Among the most promising approaches is the application of graph theory for representing product structures by defining nodes representing entities and edges representing relations among them. The concrete meaning of these structures ranges from geometry representations over hierarchical product breakdowns to functional descriptions and flows of information or resources. On top of these graph structures, graph rewriting techniques provide another powerful layer of technology. By enabling the formal definition of rules for transforming graph structures, they allow on the one hand side to formally capture the engineering development process. On the other hand, the assembly of rewriting rules into graph grammars allows for an exhaustive search of the solution space of the engineering problem at hand. In combination with search strategies, an automated optimization of the design under given constraints and objectives can be realized. The paper provides an overview of the current state-of-the-art in graph rewriting and its applications in engineering design, with a focus on the built environment. It concludes with a discussion of the progress achieved and the missing research gaps.

Keywords: graph grammars, spatial grammars, graph theory, design synthesis, graph rewriting

Edited by:
Vagelis Plevris,
Qatar University, Qatar

Reviewed by:
Jiansong Zhang,
Purdue University, United States
Linzi Fan,
Sanjiang University, China

***Correspondence:**
Lothar Kolbeck
lothar.kolbeck@tum.de

†ORCID ID:
André Borrmann
orcid.org/0000-0003-2088-7254

Specialty section:
This article was submitted to Computational Methods in Structural Engineering,
a section of the journal Frontiers in Built Environment

Received: *15 November 2021*
Accepted: *28 December 2021*
Published: *01 February 2022*

Citation:
Kolbeck L, Vilgertshofer S, Abualdenien J and Borrmann A (2022) Graph Rewriting Techniques in Engineering Design. Front. Built Environ. 7:815153. doi: 10.3389/fbuil.2021.815153

1 INTRODUCTION

With the advance of modern information technology, computers take over work that was considered to be reserved for humans. Initially, repetitive and error-prone tasks in data processing were significantly accelerated, while computation today complements human intelligence in domains requiring creativity. To this end, graphs have proven their capabilities and flexibility to provide the necessary representations for numerous real-world problems. As a data structure, graphs provide a rich foundation to represent engineering product models with entities of arbitrarily abstract and concrete meaning. In the same context, the manipulation of graphs was investigated for decades and proven to be powerful for complex problems in various domains. Most prominently, the method of graph rewriting is well established to capture manipulation patterns in the form of rules.

Solving a design problem by graph rewriting methods requires twofold: a graph representation of particular world entities and rewriting rules that operate on this model to manipulate its nodes, edges, and their attributes. The rules formalize domain knowledge by declaring design processes as graphlets consisting each of a conditional and a rewriting pattern. Once formalized, mature software frameworks ensure an efficient application of the rules to evolve the design representation. Graph rewriting allows an engineer to define a design not directly as an end result, but in terms of a

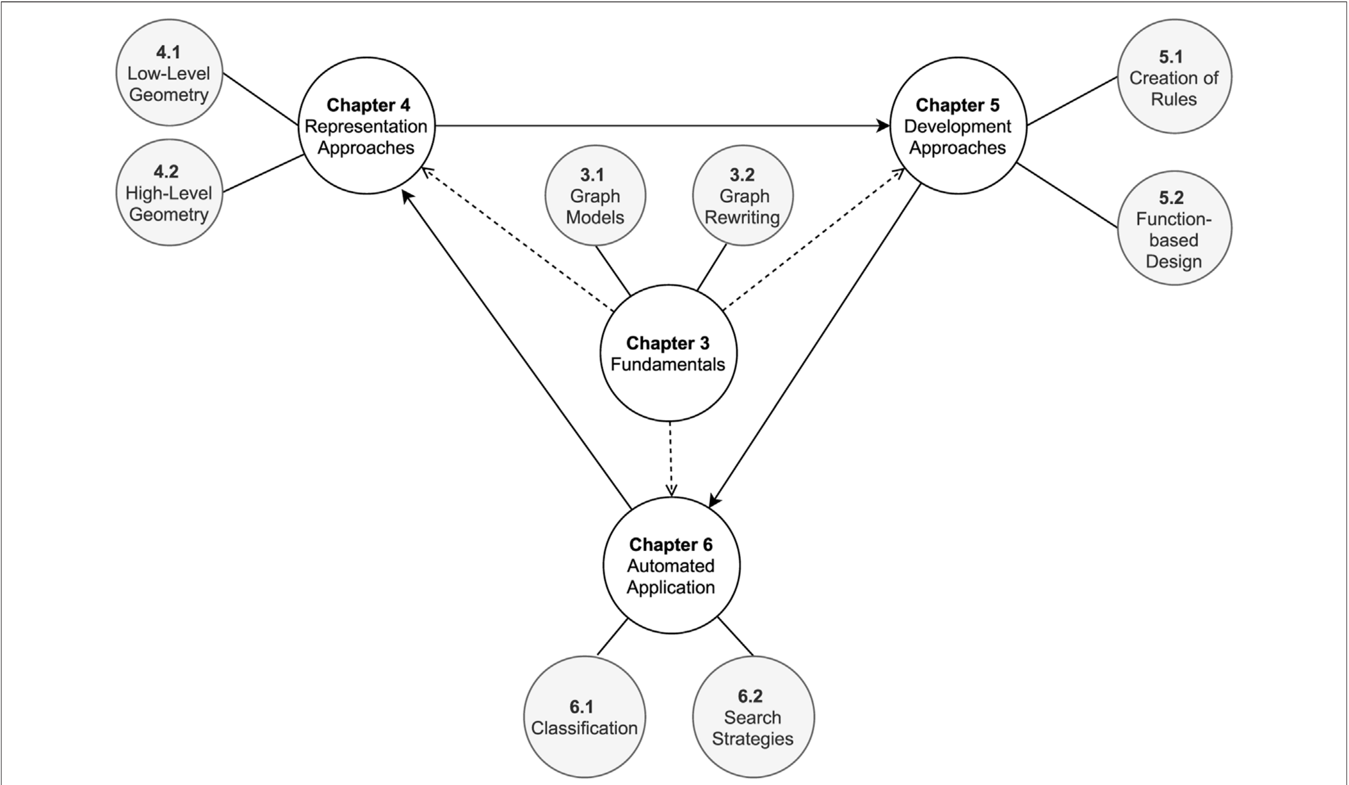


FIGURE 1 | Visualized outline of the review, following the cycle inherent to rewriting methods: The representation of a problem, the formulation of rules, and the application of rules.

procedural construction history. Arranging the rules in a flow they may be applied in, an engineering product can be gradually synthesized. In order to generate an illustrative variety of a prior unknown design, the engineer may vary the rules selected, the matches chosen, and variable parameters along that flow. The exploration of such a solution space may suggest creative, new ideas and then may be further restricted. This is achieved by either specifying the generation process or by directed stochastic search.

Originally, all rewriting methods emerged from linguistics (Chomsky, 1959), with fundamental branches dealing with shapes (Stiny, 1980), biological modeling (Lindenmayer, 1968), and later graphs (Nagl, 1979). Thanks to this background, a large part of the terminology used is related to linguistics. A *grammar* is formed when a set of rewriting rules, a *rewriting system*, is complemented by a start symbol, sometimes called *axiom*, a set of non-terminal symbols, and a set of terminal symbols. The terminal and non-terminal symbols are often referred to as *vocabulary* of the grammar. A defined grammar may be applied to exhaustively generate all possible different combinations of rule applications and parameters, constituting the *language*, the constituted solution space. The linguistic theory of *formal grammars* is rich and provides mathematical notations, terminology, and taxonomy available to many specific applications. Yet, the abstractness and extensiveness of the field also motivated researchers to enter specialized debates. In engineering design, expert systems were an early attempt to give a framework to the use of rewriting rules for design tasks. The research greatly decreased in the 1990s and is widely inactive today, due to various reasons, ranging from the difficulty of expert system maintenance and extensibility (Puppe, 1990).

Instead, several other frameworks emerged around the millennium. Cagan et al. (2005) achieved to encompass several schools of thought and several strategies for automated design synthesis along a simple, generic framework. The four key steps are the *representation* of the problem, the *generation* of solutions, their *evaluation*, and finally the *guidance* of the subsequent cycle of search. This framework is agreed to cover wide ranges of automation efforts under the collective term *computational design synthesis* (CDS). As one important stream, Chakrabarti et al. (2011) distinguished grammar-based synthesis. In parallel to the harmonization attempts of CDS, Rudolph (2002) motivated the so-called *graph-based design languages* as a powerful graph- and graph rewriting based methodology. Certainly complying with the broad definition of CDS, the field of graph-based design languages can be seen as a specification of the method. However, it is based on a stricter mathematical treatment of design objects and the formal design process (Riestenpatt and Rudolph, 2019).

In this review paper, we aim to break down this broad and extensive research field to the essential technological and conceptual questions. At a first glance, graph rewriting methods and the covering frameworks may appear very abstract. Yet, treating distinctly the

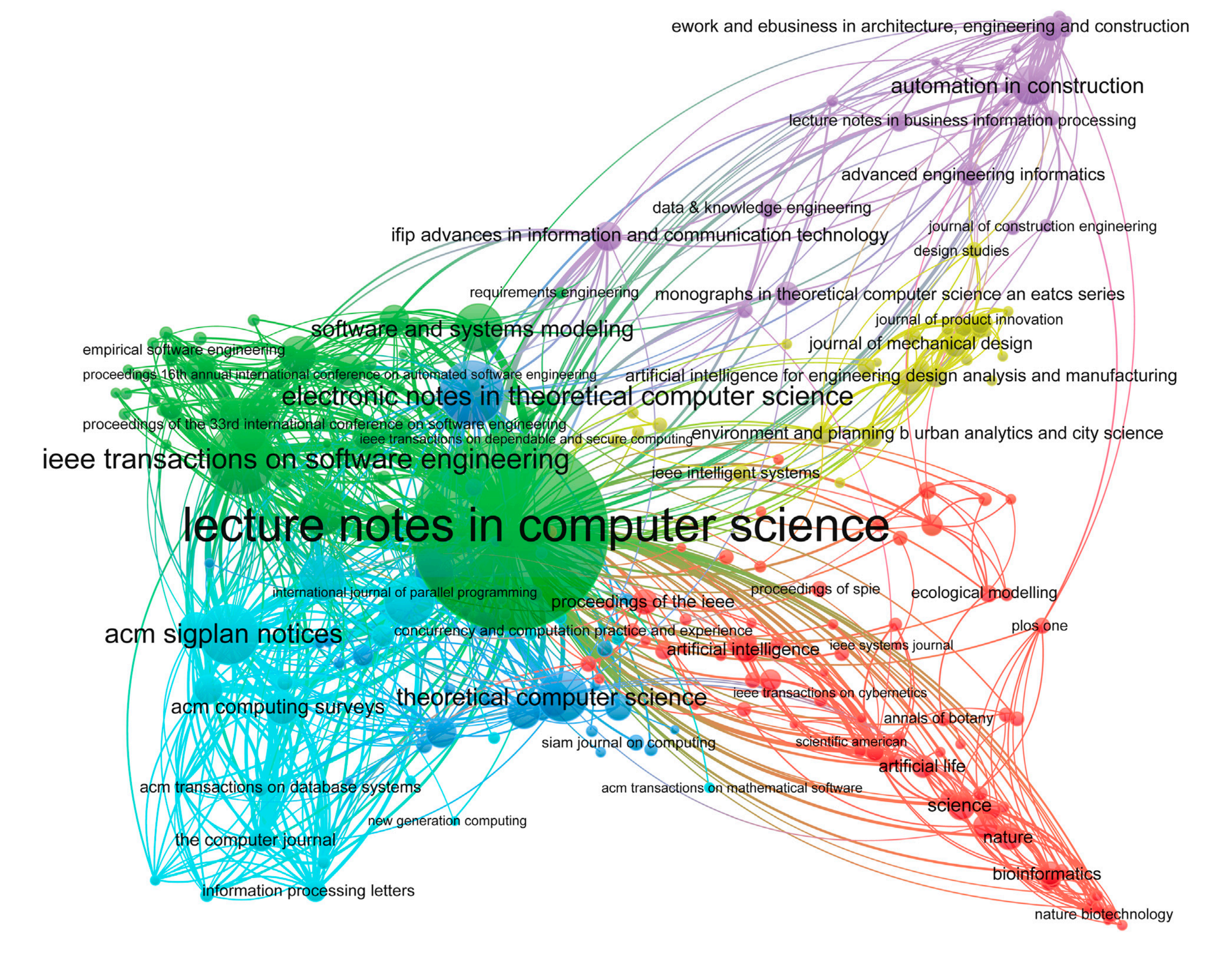

ework and ebusiness in architecture, engineering and construction
automation in construction
lecture notes in business information processing
advanced engineering informatics
data & knowledge engineering
ifip advances in information and communication technology
journal of construction engineering
design studies
requirements engineering
monographs in theoretical computer science an eatcs series
journal of product innovation
software and systems modeling
journal of mechanical design
empirical software engineering
proceedings 16th annual international conference on automated software engineering
artificial intelligence for engineering design analysis and manufacturing
electronic notes in theoretical computer science
proceedings of the 33rd international conference on software engineering
ieee transactions on dependable and secure computing
environment and planning b urban analytics and city science
ieee transactions on software engineering
ieee intelligent systems
lecture notes in computer science
international journal of parallel programming
proceedings of spie
ecological modelling
proceedings of the ieee
acm sigplan notices
concurrency and computation practice and experience
plos one
artificial intelligence
ieee systems journal
acm computing surveys
theoretical computer science
ieee transactions on cybernetics
annals of botany
siam journal on computing
scientific american
artificial life
acm transactions on database systems
acm transactions on mathematical software
science
new generation computing
nature
the computer journal
bioinformatics
information processing letters
nature biotechnology

FIGURE 2 | Bibliography analysis of citations between publication sources. Label and circle sizes correspond to the total number of documents. These results were collected by searching Scopus[1] for the keywords *Graph Rewriting*, *Engineering*, and *Design*. The visualization was performed using VOSViewer[2].

essential steps of representation, rule definition, and the later application of rules, we aim to make comprehensible associated concepts. Following the outline shown in **Figure 1**, we hope to communicate the relevancy, potentials, and challenges to a broader audience. The review starts with a bibliography analysis where we attempted to quantify the scientific interest to graph rewriting methods, with special attention to the building sector. To make current developments comprehensible to readers with little prior knowledge, we introduce the fundamentals of graph theory and graph rewriting in chapter 3. Chapter 4 draws attention to the various approaches for forming a graph representation of an engineering model with semantic and geometric meaning[1]. Supports to the development and organization of rewriting rules are introduced in chapter 5. Finally, chapter 6 focuses on issues of automatically applying rewriting rules for design generation while chapter 7 concludes the review by summarizing potentials and shortcomings for further research[2].

2 BIBLIOGRAPHY ANALYSIS

The terms *Graph Rewriting* and *Graph Transformation* appear in the literature along with the keywords *Engineering* and *Design* since the 1970s. Although combining multiple keywords reduces the search scope, there are numerous engineering and design domains that have investigated graph rewriting for their challenges. **Figure 2** depicts the citation network between journals when searching for those keywords combined. The journals of computer science and software engineering (appearing at the center) are the most dominant journals for this field, where graph rewriting was used as a technique for

[1]https://www.scopus.com/

[2]vosviewer.com

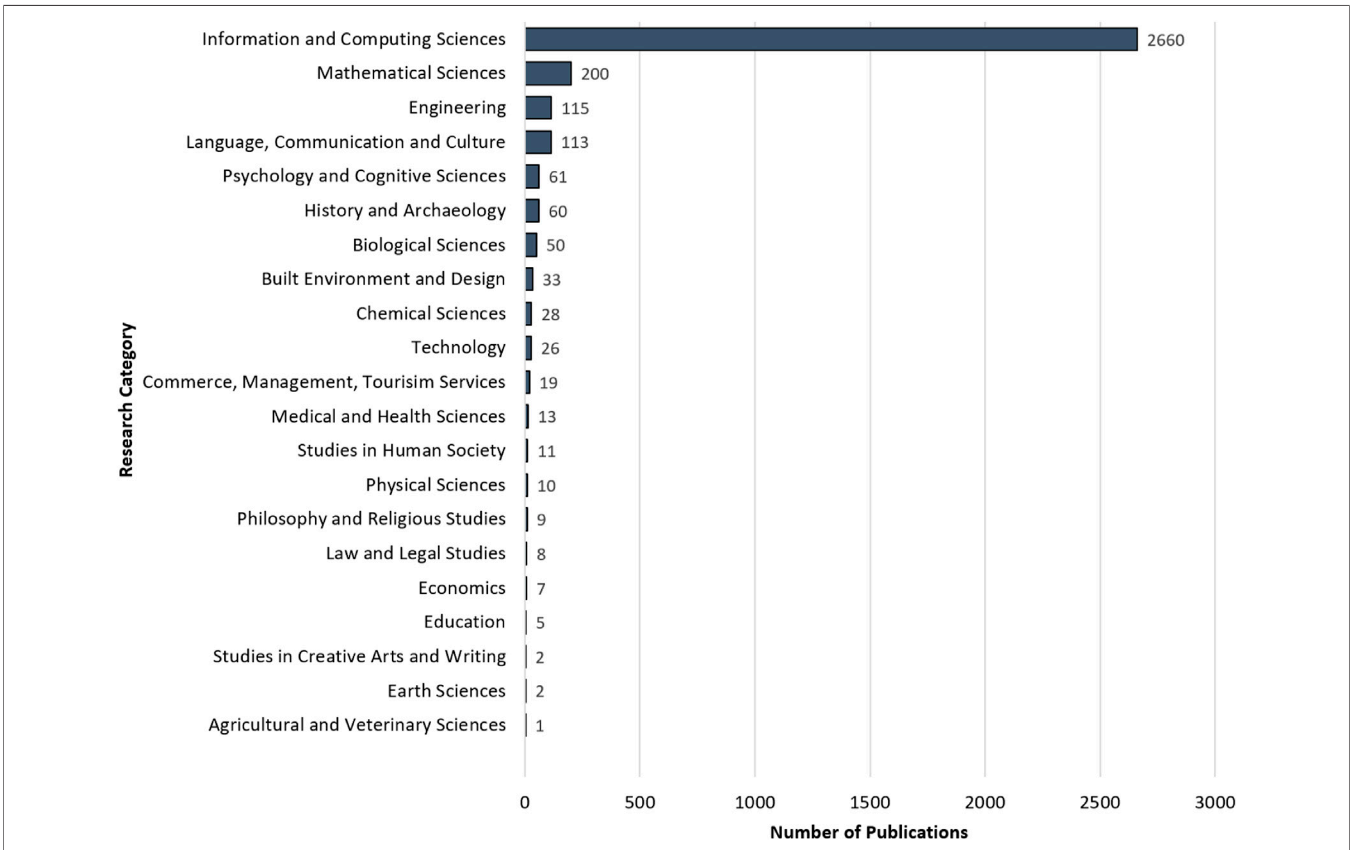


FIGURE 3 | An ordered grouped list of publications according to their research categories. These results were collected by searching Scopus[3] for the keywords *Graph Rewriting*, *Engineering*, and *Design*.

manipulating data structures, source codes, and more. Zooming out of the center, other application domains appear, such as biotechnology, business and engineering.

To get more insights into the involved research domains, **Figure 3** shows a grouped list of search domains and their corresponding publication counts. The field of *Information and Computing Sciences* provides a high percentage of the publications, as graph rewriting was introduced and developed by this domain. The rest of the domains typically adapt and apply the techniques developed in computer science to their particular challenges. Among others, engineering is ranked third, with 115 publications, whereas built environment and design is ranked eighth.

From this broad overview in engineering and design, a more detailed literature analysis was conducted with a special focus on publications that additionally include *Building Information Modeling* as a keyword. The first papers that appeared in the literature that included both *Graph Rewriting* and *Building Information Modeling* are from 2010 (Tratt, 2010). Afterward, there is a trend in increasing the number of relevant publications per year, as shown in **Figure 4**, where 23 relevant papers were published in 2018 alone and 58 in total until the year 2020.

The conducted bibliography analysis provides the necessary ground for the reviewed publications in the following sections. In this regard, the fundamentals of graph representations, graph rewriting, and the used software frameworks are described[3].

3 FUNDAMENTALS

3.1 Graph Representations

3.1.1 Theoretical Specifications

All graphs $G = (V, E)$ have in common that they consist of a set of nodes or vertices V and edges E, with each edge being represented by an ordered or unordered list of nodes. This generic definition encompasses a variety of specifications. In engineering design, it is commonly implied a typed and attributed graph, sometimes referred to as *property graph* (Robinson et al., 2015). Types, sometimes referred to as *labels* or *tags*, can serve to specify different categories of objects within a system. In an architectural context, this might serve to distinguish different room types (Langenhan et al., 2013), different building elements (Abualdenien and Borrmann, 2021), or load-bearing elements and their joints (Vestartas, 2021). Attributes in turn can serve to store relevant data about

[3]https://www.scopus.com/

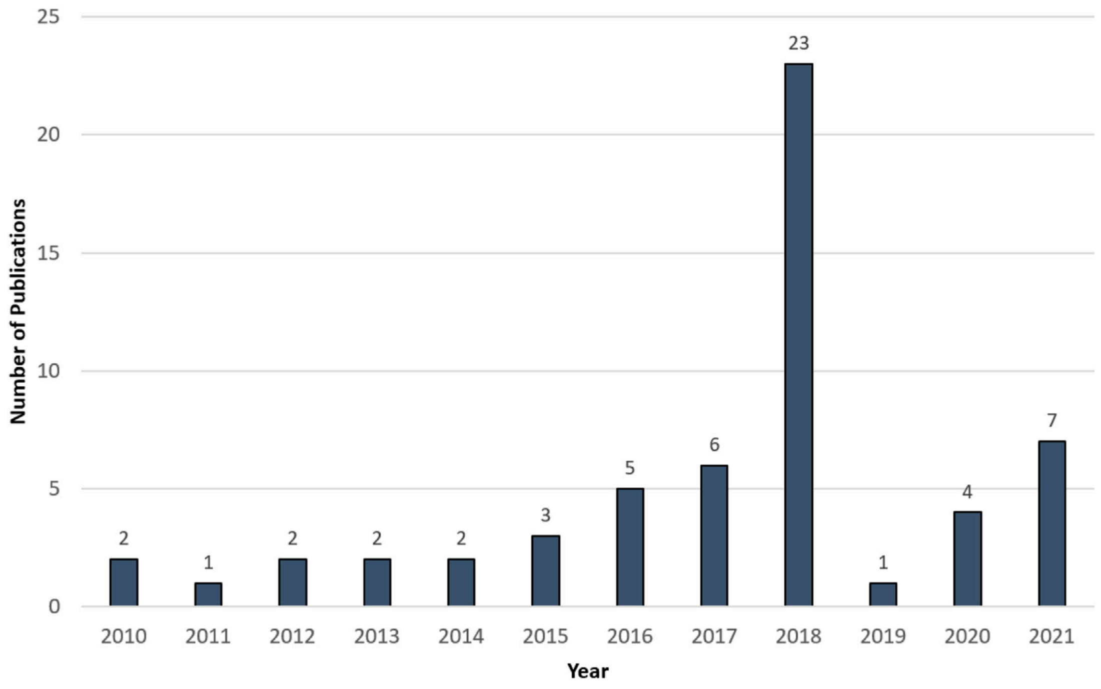


FIGURE 4 | An ordered grouped list of publications according to their publication year. These results were collected by searching Scopus[4] for the keywords *Graph Rewriting* and *Building Information Modeling*.

the objects in the form of key-value pairs. This data might be as simple as an identifier, a reference to external data sources, or as complex as a parametrization of geometric descriptions, see **Section 4.2**. Thus, graph structures are an expressive means to represent engineering systems as a network of objects with rich semantic and geometric meanings. However, most engineers find it difficult to formulate and solve their problems employing graph theory, except for of well-known applications like path planning. A look at two key motivations for the use of the data structure explains the difficulty.

One key advantage of graph theory is its ability to formally and flexibly represent relationships between entities. Large networks of objects can be created and queried for certain patterns of relationship to analyze or manipulate the represented system. However, the scale of these networks easily gets overwhelming. Therefore, there are specialized graph types that facilitate structuring complex domains and thus are better suited for certain applications[4]. As one example, *trees*, are special graphs that may serve to simplify analysis and manipulation of problems with an inherently hierarchical structure. Such a hierarchy can be applied to model the spatial structure of buildings, for example, where a site may comprise multiple buildings, each building may comprise multiple stories and each story comprises a number of spaces or rooms (Wonka et al., 2003; Grabska et al., 2012). To give a second example, *port graphs*, sometimes labeled composition graphs (Strug et al., 2022), are special graph types that distinguish two types of nodes: Object nodes and connector nodes. The connector nodes, the ports, restrict how the object nodes are allowed to be coupled to each other. This may be extremely useful for tasks that formalize assembly processes, as in the context of chemical reactions, bond graphs (Helms and Shea, 2012), or the design of segmented structures (Rossi and Tessmann, 2017a; Kolbeck et al., 2021). Many more specifications originate from the need to efficiently depict and manage complex networks, even leading to complex combinations like hierarchical hypergraphs (Drewes et al., 2002) and others currently experimented within engineering (Strug et al., 2022). Encountering all those specifications may easily overdemand a learning person, whereas all specifications branch off from the simple and comprehensible notation given above.

A second key advantage is the flexibility of graphs to adapt to arbitrary levels of scale and abstraction. For example, nodes and edges can represent geometric vertices and edges, but may also stand for complex objects such as walls or a building story. To illustrate the range of semantics a graph model can have in a design context, we discuss diverse applications in the building sector in the next section. Thereby, we highlight the characteristics of graph models used for the transformation of design by a comparison to the ones used for analysis tasks.

3.1.2 Applications in Architecture and Civil Engineering

An old stream of research aims to depict engineering systems as a graph to perform efficient analysis of the data structure. The well-

[4]https://www.scopus.com/

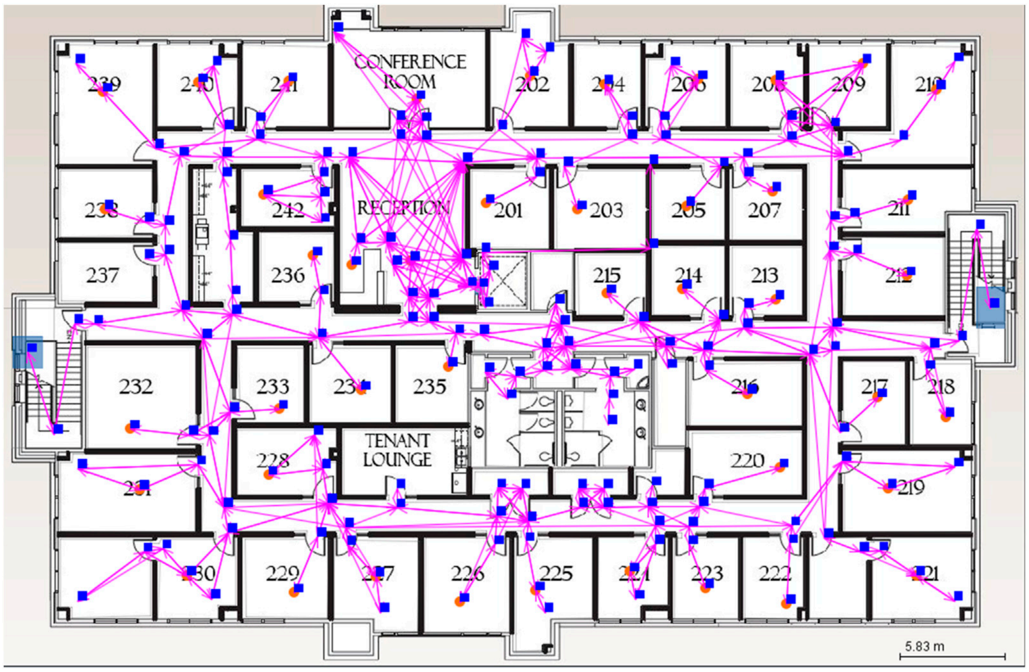


FIGURE 5 | Generation of a navigation graph from building geometry (Kneidl et al., 2012).

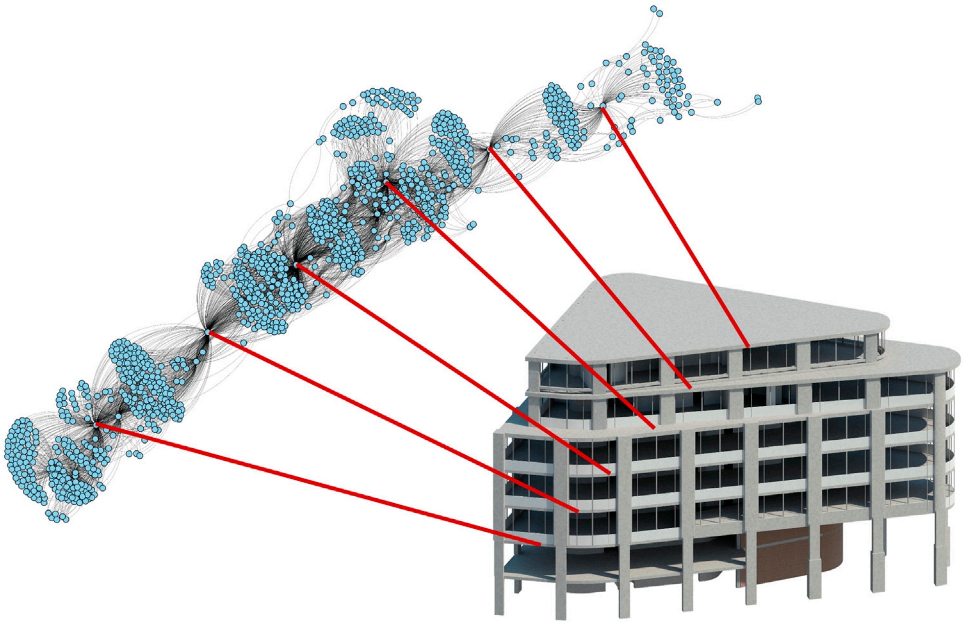


FIGURE 6 | A precedence relationship graph represents the order of erection of individual building components and their mutual dependencies (Braun et al., 2015).

known Dijkstra algorithm or the A* algorithm for path planning is familiar to most engineers. For a building, this can be adopted by translating architectural rooms and their mutual accessibility into a navigation graph, see **Figure 5**. A similar graph model of a building may be used to suggest architects preferable room layouts when dividing a floor into spaces (Langenhan et al., 2013). Equally, structural aspects of construction can be represented. Vestartas (2021) used a graph model to describe the different joints of crooked wooden beams. Braun et al. (2015) recorded the precedence relationships of construction

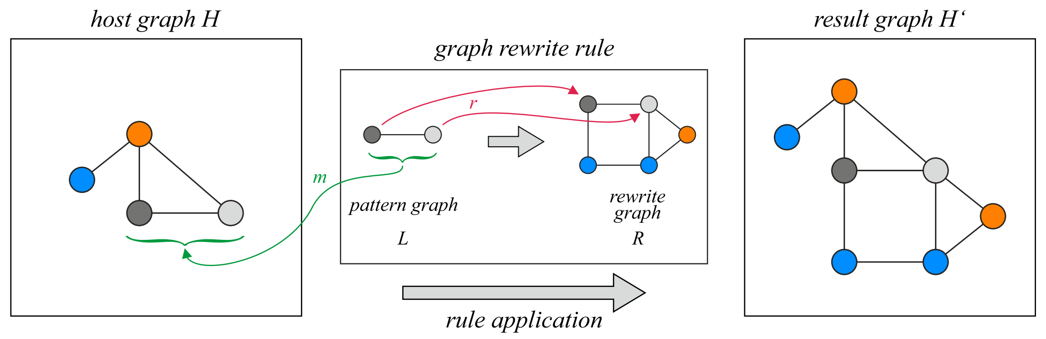


FIGURE 7 | A graph rewriting rule, graphically described in the middle, is applied to the host graph on the left side. The result graph is depicted on the right side. Here, the preservation morphism *r* is defined by the mapping of the grey nodes.

execution in a graph, where every node represents a column, wall, or floor and edges represent precedence relationships. The resulting graph model, depicted in **Figure 6**, was generated through a spatio-temporal analysis of the building construction process.

From a formal point of view, graph models for the dynamic manipulation of a design are identical to the ones for the analysis of a static design. However, a crucial difference is that design activities require a much greater amount of topological and geometrical adaptivity. Property graphs commonly enable topological extensibility while dynamic geometric transformations pose a less common challenge. In concrete terms, it is comparably easy to construct the system entities and their relationships for a coffee machine (Tonhäuser and Rudolph, 2017) or a bridge (Slusarczyk and Strug, 2017). Still, a valid topological and semantic configuration does not guarantee to make the components form a valid and harmonic assembly, without collisions and gaps at emerging interfaces. In comparison, a graph structure that merely analyzes a static design must capture the exact geometry only once. Without the need to dynamically change it, the geometry may even be stored and referenced employing external databases, for example in a point cloud format (Braun et al., 2015; Vestartas, 2021).

The characteristics of graph structures for dynamic manipulation of design is subject to further discussion in chapter 4. Since a representation for design is strongly linked to the manipulation mechanisms applied to evolve it, the next sections attempt to give a fundamental understanding of graph rewriting methods before.

3.2 Graph Rewriting

3.2.1 Theoretical Specifications

Graph rewriting, also referred to as graph transformation, describes the process of manipulating a graph structure by adding, removing, and altering nodes and edges, steered by declaratively defined rules. Each rule consists of a *left-hand side* (LHS or pattern graph) and a *right-hand side* (RHS or rewrite graph). Providing a *host graph* and a set of rules, the matches of the LHS in this host graph can be identified and be replaced by the RHS to generate the result graph as depicted in **Figure 7**. A *preservation morphism r* can be defined in order to specify that parts of the LHS are matched to the RHS to ensure that they are preserved.

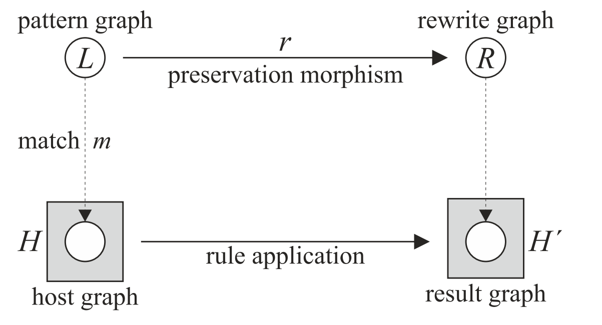


FIGURE 8 | Basic principle of graph rewriting using the Single Pushout Approach according to Geiß (2008) and Jakumeit et al. (2010).

Several characteristics can increase the expressiveness of a rewriting rule. Rules may be more concise by including attributes and multiple labels per node. For example, a graph representation of a building may enable a matching for objects labeled both "wall" and "load-bearing," with an attribute "height" at a certain value. Further, a rule may be context-sensitive, meaning that it specifies conditions that exclude possible matches depending on the surrounding of the pattern. These application conditions may concern the left or the right side of the rule (Habel et al., 1996). Rules can be defined to be more flexible for a wider range of applications by defining them parametrically, computing variables instead of fixed values. In order to span a wide solution space in a generative application of rules, variable rule parameters may be stochastically chosen.

Rules, potentially defined with all described characteristics, need to be applied to a host graph (H). To this end, it is necessary to detect correctly typed and attributed matches for the LHS (or pattern graph), as well as their rewriting to produce a valid result graph (H') containing the RHS (or rewrite graph) inserted. Efforts have been undertaken to significantly improve the computational complexity of match detection algorithms (Geiß et al., 2006; Batz et al., 2008). As well, a variety of rewriting methods has been researched for decades. In distinction to the algorithmic

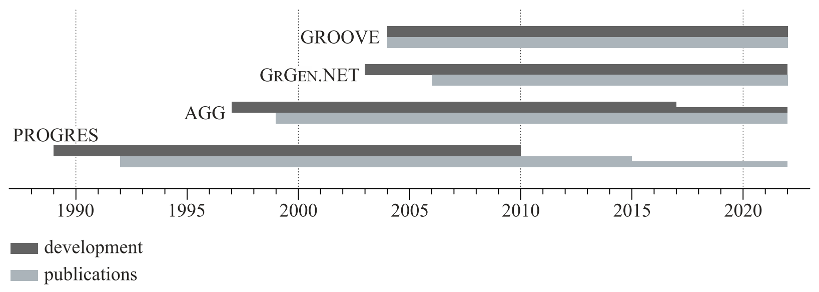


FIGURE 9 | Not all of the presented frameworks are still continued and maintained. This figure gives an overview of the times of development and reference and use in research projects. Smaller bars indicate that only small updates were published or references only list the tool without using it.

approaches, the prevailing algebraic approaches consider graph rewriting as a mapping problem between two algebras of nodes and edges. The main algebraic methods are the single pushout approach (SPO), shown in **Figure 8**, and the double pushout approach (DPO). The fundamental difference between both lies in the greater expressiveness of the SPO, which conducts the rewriting in a single step. On the other hand, the DPO introduces an intermediary *gluing graph* that allows a more restrictive avoidance of problematic situations, as for dangling edges in the result graph (Corradini et al., 1997). For further reading on the fundamentals of algebraic graph transformation approaches, we refer to Rozenberg (1997) and Ehrig (2006), while an application-oriented introduction to graph rewriting can be found in Heckel (2006).

3.2.2 Software Frameworks for Graph Rewriting

The modeling of graph representations and their transformations can be conducted with diverse software frameworks. The term *framework* describes an assembly of specialized tools. This includes at least an *interpreter* that processes human-readable descriptions of graph metamodels as well as graph rewriting rules and gives feedback in case of errors. A *compiler* then translates this into source code or libraries for further use. Most frameworks also have some sort of graphical user interface to display graphs and visualize rule execution. In this section, we give a short overview of the major frameworks that are available (see **Figure 9**). This list is not meant to be complete as there exist many further frameworks, although many of them are not maintained anymore. Extensive but rather outdated lists are provided in Nagl et al. (2003) and Rensink and Taentzer (2007). More recent comparisons were documented by Aouat et al. (2012), Bak (2015) and Kahani et al. (2019). Tools addressing graph transformation are also regularly presented amongst others in the annual *Transformation ToolContest*[5] that aims to evaluate and compare the expressiveness, usability, and performance of transformation tools for structured data.

A widely used graph transformation framework is the Graph Rewrite Generator **GRGEN.NET** for the .NET environment (Jakumeit et al., 2010; Jakumeit et al., 2021). GRGEN.NET offers declarative languages for graph modeling, pattern matching, and rewriting. GrGen allows users to define an object-oriented graph metamodel, a blueprint of the desired design representation, describing node and edge types including attributes and inheritance. The metamodel may also include connection assertions that define the allowed connections of nodes and edges in a graph. The frameworks offers many possibilities when defining rewriting rules including negative application conditions which may be applied with logical and iterative control of their application. Rewriting is generally based on the SPO approach, but also the DPO approach may be used. As GRGEN.NET creates . NET libraries for the graph metamodel and transformation rules defined in its own language, it can be easily used in custom projects. A major benefit is that the software including its documentation is regularly updated and well maintained. A quantification of the computational efficiency of GrGen can be found in (Geiß et al., 2006), including a relative comparison to the following two frameworks.

The Attributed Graph Grammar **AGG** is a rule-based visual language supporting an algebraic approach to graph transformation implemented in Java (Ermel et al., 1999; Runge et al., 2011). AGG allows the definition of attributed type graphs with inheritance. The defined graphs may be attributed by Java objects and types. A main feature is that the framework provides graphical editors for graphs and rules and a text editor for Java expressions including visual interpretation and validation. AGG is primarily based on the SPO approach but offers the possibility to enable rewriting based on the DPO approach. The main functionality of the framework is provided by a graph transformation engine that is independent of the graphical environment. Therefore, the transformation functionalities may also be used by other software. The last major update for AGG has been released in 2017, whereas a patch has been published in early 2021, so it can be considered to be maintained.

Another sophisticated and well established framework is **PROGRES** (PROgrammed Graph REwriting Systems) which is being developed at RWTH Aachen since 1989 (Schürr et al., 1995). It is based on directed, attributed and typed graphs, which can represent extensive and complicated issues in a clear and structured manner. PROGRES consists on the one hand of a specification language and on the other hand of a complex, integrated environment. The framework allows the specification of a graph schema with inheritance and edge cardinalities that can be used for type-checking of productions. Besides the graph schema, graph transformations can be specified graphically and textually. The PROGRES

[5]https://www.transformation-tool-contest.eu/aims_and_scope.html.

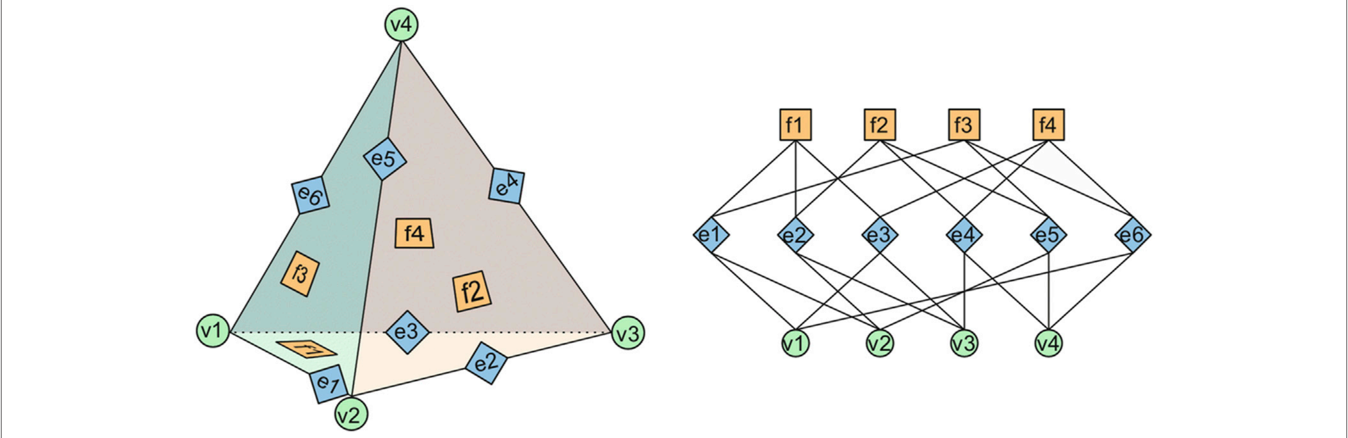


FIGURE 10 | A pyramid represented by a vertex-edge-face graph, illustrated geometrically (left) and topologically (right).

environment consists of three integrated frameworks. Graph schema and transformations can be defined in a syntax-controlled editor that highlights violations. The interpreter with a corresponding graph browser assists the user in debugging and the compiler automatically translates the specification into C or Java source code. However, PROGRES is not regularly maintained and has been last updated in 2005. While it is regularly referenced in recent articles giving an overview of graph rewriting frameworks, the last publication that describes its use in a project dates back to 2015.

The **GROOVE** tool set (Graph-based Object-Oriented VErification) is being developed since 2004 and is still regularly updated (Rensink, 2003; Ghamarian et al., 2012). With GROOVE simple graphs can be used for modeling the design-time, compile-time, and run-time structure of object-oriented systems. Therefore, it provides graph transformations as a basis for model transformation and operational semantics. GROOVE is a general-purpose graph transformation framework that uses simple labeled graphs and transformation rules based on the SPO approach. The framework is Java-based and provides an intuitive interface that allows graphical editing of rules and graphs.

Among this rich body of available alternatives, an engineer can deliberate the choice of a framework for specialized applications in design. This deliberation is a problem-specific evaluation of necessary and desirable characteristics in modeling, development, and execution. Building upon this fundamental understanding of both graph theory and the implications of rewriting, we address approaches to represent engineering products in the next chapter.

4 REPRESENTATION APPROACHES

Graph structures for design synthesis approaches require an efficient approach to both the representation and manipulation of the geometry of relevant design objects. In the aspect of identifying the crucial objects and linking them elegantly to a geometric representation, we see the key problem to the successful use of graph rewriting methods in design synthesis. Thereby, a first stream follows the idea of a very fine-grained representation and control of geometry, giving the graph an intuitive geometric meaning. The second stream attempts to further abstract objects, making it easier to define and describe transformations on a semantically higher level of abstraction. Both are discussed in the following sections.

4.1 Low-Level Representation of Geometry

An engineer familiar with computational geometry would likely associate the terms "graph" and "geometry" with well-known classical data structures. As such, the *vertex-edge-face* graph, illustrated in **Figure 10**, may be mentioned, used in boundary representation approaches. Two key advantages of such graph models with a strong linkage of topology and geometry may be highlighted:

They give a very fine-grained control of the geometry of objects, down to every single geodetic point. For an engineering model with objects in such a representation, efficient and detailed spatial-topological queries and consistency checks exist (Borrmann and Rank, 2009; Jabi et al., 2018). Another advantage to the low-level integration of topology and geometry is the geometric intuitiveness of a graph model. This is both beneficial to the development of rules, and the analysis of structures, e.g., when evaluating them by defining cost functions. Many applications illustrate these benefits. An early and widely known example is the optimal truss generation problem (Shea, 1997; Kaveh and Koohestani, 2008; Hooshmand and Campbell, 2016). Thereby, nodes commonly represent joins, while edges represent the beams of the truss. Describing and manipulating more complex structures is equally possible, as shown by the origami figures of Chen et al. (2019) or the walls and floors described in the architectural solid grammars by Heisserman (1994).

Despite not being specific to engineering, the implementation of shape grammars utilizing graph models must be mentioned in this context, too. For these implementations, graphs were recognized to have beneficial characteristics as a model of geometry. On the one hand, graphs were proven capable of implementing shape grammars that support the recognition of *emergent* shapes (Knight, 2003), i.e., to cope with ambiguous recognition problems as in the Sierpinski-triangle. More relevant to engineering applications is the question of topological-associative shape recognition (Grasl and Economou, 2013; Wortmann, 2013). Instead of depending on similarity transformations to geometrically match LHS patterns (Krishnamurti and Earl, 1992), graphs enable efficient queries for *topological* patterns of geometric entities. This enables the formulation of much more flexible and expressive rewriting rules. A rule defined for a quadrilateral may apply to any quadrilateral in a shape of a design.

This leads to two significant downsides of graph models in such a high resolution of geometry. First, what does a human intend when he defines a rewriting pattern that consists of a closed loop of four vertices and lines? Can the matching algorithm assume that orthogonal and parallel lines play a role or not? Did the human intend a void quadrilateral or may the pattern intersect itself or be intersected by other elements? Krishnamurti and Earl (1992) discuss why it is a very difficult task to capture the exact designer's intent. Recently, these questions were revisited by Stouffs (2019). Second, is it really necessary and desirable to have such high flexibility and control of geometry? It must be weighed up that this comes with the toll of defining rules that may easily become very complex.

These two problems have been known for a very long time and two main responses exist to remedy them. On the one hand, the introduction of an additional layer of abstraction, i.e., a user interface, could help humans to more intuitively express their intentions of a rule. Such an abstraction layer may be either graphical or textual (Dy and Stouffs, 2018), as further discussed in **Section 5.1**. Another remedy is to move from a geometrical to a more object-oriented view of design artifacts. If an architect wanted to formulate rules for the design of a house in natural language, likely very little would be explicitly stated about the relations between points, lines, and faces. Instead, the architect would reference columns, walls, and slabs (Mitchell, 1991). Many of those symbolic objects may be sufficiently described by a fixed or parametrized geometry. A column may be described by a point, a diameter, and a height. Instead of a large graph pattern with nodes representing points, lines and faces, a node with three key-value pairs may be precise enough for the scope of many practical problems.

This is the basic idea of a group of grammars for design that follow an object-oriented idea of design: instead of reasoning geometrically, *set grammars* discretize a design problem to a set of comprehensible entities. Those entities are processed in the grammar by rules that allow the set-theoretic operations of union, intersection, and difference among them. In the context of design problems, such entities are commonly tangible objects like stories, walls, or windows (Wonka et al., 2003), with complex geometry such as a parametrized solid (Alber and Rudolph, 2003). This enables the formulation of rules on a more natural level of abstraction and simplifies geometrical challenges as e.g. the shape recognition problem (Krishnamurti and Earl, 1992).

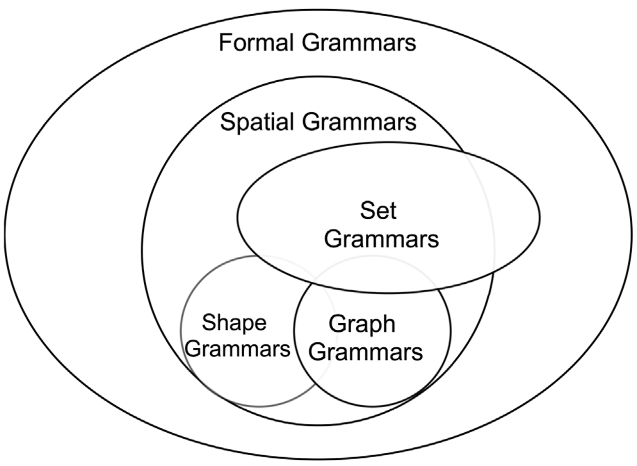


FIGURE 11 | Taxonomy of formal grammars utilized for engineering design problems.

However, the relations between shape grammars and set grammars often are unclear due to fuzzy terminology in the field (Lienhard, 2017). Many terms circulate and may be confusing as they treat very similar concepts. An early attempt to give a taxonomy to the various terms encountered is found in Krishnamurti and Stouffs (1993): all grammars with strong geometric implications may be subsumed under the term *spatial grammar*, including *L-systems* (Lindenmayer, 1968) as well as classical *shape grammars* (Stiny, 1980). Set grammars may be defined for a variety of objects, like strings, shapes, or graphs. Graphs recently were also used to implement shape grammars (Grasl and Economou, 2013; Wortmann, 2013). Our understanding of the different terms we summarized in **Figure 11**. The next section will focus on the intersection of graph grammars and set grammars.

4.2 High-Level Representations of Geometry

4.2.1 Set Grammar Approaches

In the former section, we discussed that many practical engineering problems allow the system to be represented as a composition of objects that can be sufficiently described by high-level geometric primitives. Thereby, one may roughly differentiate two types of object descriptions.

The first type associates an object to an individual from a set of strictly uniform geometries. This may be a column of a fixed diameter and height (Mitchell, 1991) or standardized, serially prefabricated parts. The expressiveness may be increased by discretizing the vocabulary as groups of the same object type, e.g., by introducing five columns with slightly different heights and diameters. Then, eventually, a wide enough solution space is opened up, while ensuring a geometrically elegant and comprehensible way of processing and interpreting the graph.

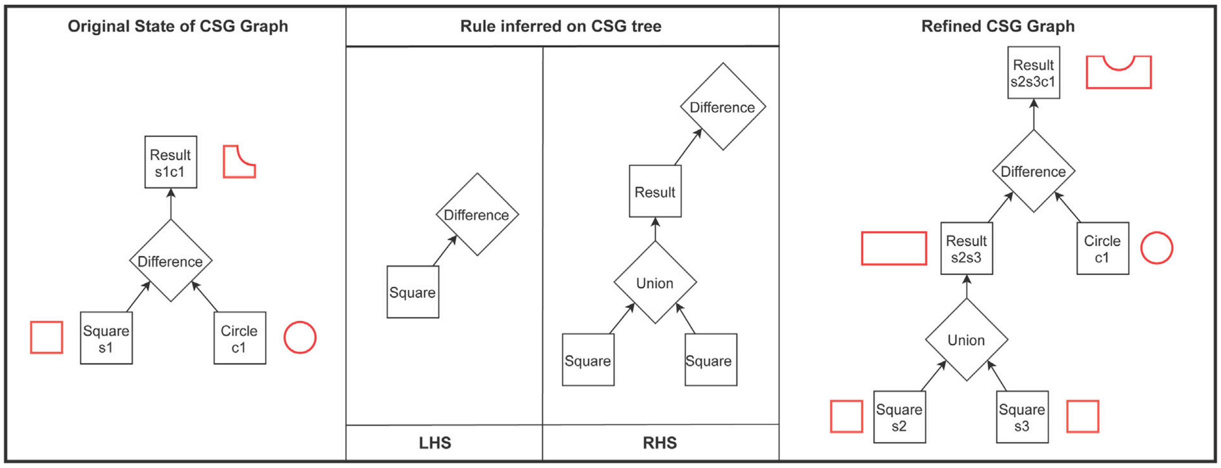


FIGURE 12 | Refinement of a CSG tree based on a graph rewriting rule.

Further, a sufficient degree of discreteness is certainly helpful to simplify the reasoning with rules (Peyshakov and Regli, 2003).

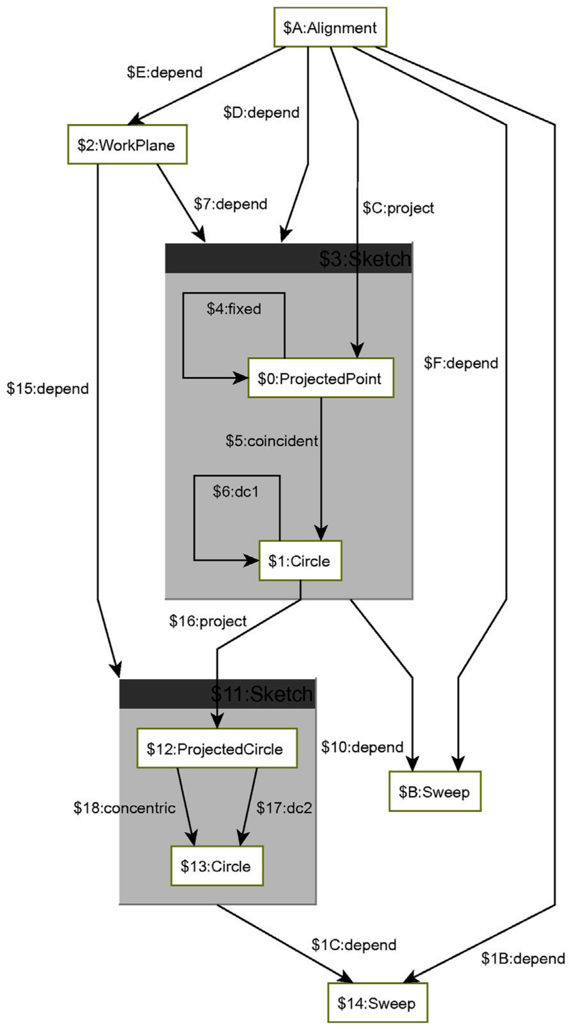
The second type relies on geometric descriptions with a parametrization defined by continuous or discrete variables. As a first example, Alber and Rudolph (2003) described the assembly of electricity pylons from a few, adaptive segments. As a second example, Vogel (2016) developed a method to construct exhaust filter systems by various, adaptive pieces of tubes and joints. One example from the building sector is the *split grammar* of Wonka et al. (2003), where buildings are efficiently generated by allowing every RHS pattern to be only a subset of a LHS object. In concrete terms, this results in a hierarchical "sculpturing" of buildings from a mass model over stories over walls to windows, following a strictly hierarchical order of the vocabulary. The split grammars were brought into a graph model by Lipp et al. (2008), with graph rewriting rules definable and applicable according to a concept by Patow (2012). To give a second example from the building sector, Abualdenien and Borrmann (2021) adopted the object parametrizations of commercial BIM software to capture design patterns in the context of high-rise buildings. Even though not using a graph data structure, Hoisl and Shea (2011, 2013) implemented the spatial grammar interpreter *Spapper* with a wide range of parametrized solid objects, ranging from tori to ellipsoids.

They mention, however, that the expressiveness of the interpreter could be further improved by including powerful procedures like sweeps or extrusions. This is an inherent restriction of the set grammar idea that defines vocabulary according to the objects present in the final design configuration (Hou and Stouffs, 2018). An alternative is to explicitly introduce *operations* that apply a certain transformation to the input objects. This idea, commonly known by techniques like constructive solid geometry (CSG), motivated another stream of graph structures for design discussed in the next section.

4.2.2 Explicit Description of Operations

The use of graph models for the geometric design of engineering products is, unconsciously, familiar to many engineers. In many undergraduate courses, the CSG approach is commonly taught. Intuitively, the technique allows applying Boolean operators (union, difference, intersection) on high-level geometric primitives. A procedural construction history based on CSG operations may be formally depicted as a directed, bipartite and hierarchical graph pointing toward one final geometry. The graph is bipartite because nodes can either represent objects or operations. Essentially, the visual programming interfaces of computer-aided drawing (CAD) software like Grasshopper Rhinoceros (Mc Neel and Associates, 2021) can also be represented as a directed graph, with nodes being either input objects or operations, performing imperative logic on the input objects. Because these procedural parametric models abide by the formal definitions of graph theory, they can also be refined by graph rewriting patterns. This basic idea of refining a procedural definition with a rule-based paradigm is illustrated in **Figure 12**. Therein, a simple CSG operation is manipulated, computing the difference between a square and a circle. The RHS pattern of the rule replaces the square with a rectangle. The rectangle is generated by a union operation that merges two new leaf nodes that each represent a square.

Procedural parametric modeling approaches draw their expressiveness from explicitly introducing operations as a part of the grammar vocabulary. The difference to other design grammar approaches is subtle but important. The greater part of grammars for design declare rules as a static *configuration* pattern before and after a rewriting step. When applying the rule, an interpreter derives from the difference between the sides the necessary procedures to take, i.e., manipulations, additions, and removals of objects. However, some operations may be much more intuitive to be stated in an *imperative* manner. To give an example from the building sector, Vilgertshofer and Borrmann (2017) distinguished "Sketch Nodes" and "Procedural Nodes" for



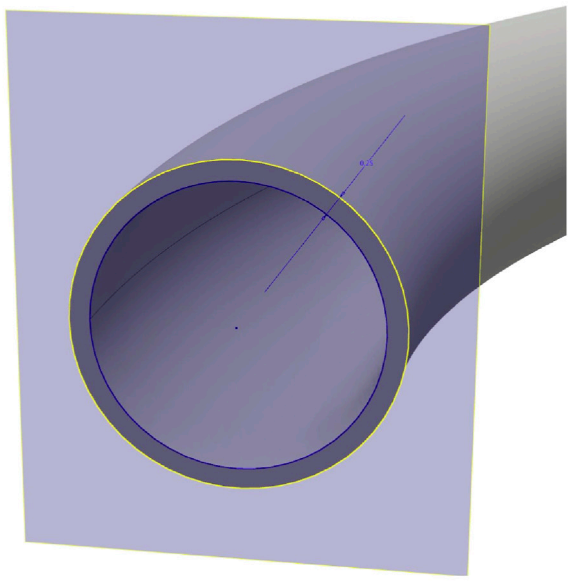


FIGURE 13 | A stage in the graph-driven design development process presented in Vilgertshofer and Borrmann (2017). Left-hand side: Graph structure consisting of entities and procedural operations, created by graph transformation. Right-hand: Interpretation of the graph by a parametric CAD system.

refining tunnel geometry, see **Figure 13**. The former described the composition of sketches, e.g. describing a tunnel profile. On the other hand, the procedural nodes enabled the transformation of geometry, e.g., by extrusion of a tunnel profile. These development steps are illustrated in **Figure 14**.

The introduction of explicit operations enables the straightforward integration of powerful features accessible by programming interfaces of CAD software. Silva et al. (2013) gave another example of a graph model that explicitly includes operations, in this case related to urban model generation. Thinking of a graph model as a network of objects or operations can make graph models in design much more expressive.

Finally, a noteworthy trend in the field of grammars and computational geometry is the investigation of mixed programming paradigms. Of course, it is more elegant to define a design problem within one of the various paradigms of procedural modeling. Yet, for considerations of efficiency, the combination of paradigms is worth further investigation. Many of the design grammars published partially needed to use imperative programming techniques (Hohmann et al., 2010). Leblanc et al. (2011) presented a modeling language based on CSG techniques, with imperative as well as rewriting characteristics. A second example, given by Hohmann et al. (2010), employs rewriting rules to refine commands of the stack-based, generative modeling language GML (Havemann, 2005). Certainly, these approaches are difficult to classify at a first glance. However, they may be a promising way to the pragmatic and widespread use of rewriting rules, specialized to the situations in which they are beneficial.

In the context of grammar-based design, any representation is just an important means for a purpose. This purpose is the development of a design from an initial state towards a goal state. This is performed by defining rules, which may be applied to incrementally evolve the design representation. Unarguably, the finding of appropriate and expressive rules is a demanding step. Therefore, we dedicate the next chapter to research treating the process of grammar development.

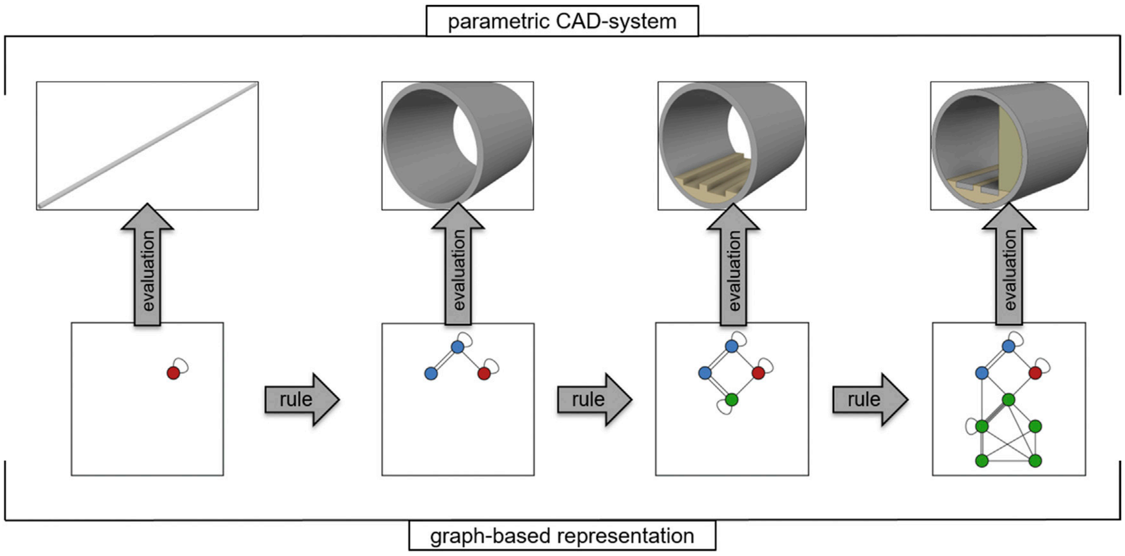


FIGURE 14 | Gradual Refinement of tunnel geometry by means of graph rewriting and according to a level of development concept (Vilgertshofer and Borrmann, 2017).

5 DEVELOPMENT OF GRAPH REWRITING APPROACHES

The variety of grammars published is commonly described as original pieces of handcraft. The representation chosen is demonstrated to have captured the essence of the problem and the rules to allow steering an efficient evolution of the system. Still, scientifically valid questions are "polemically" (Economou and Grasl, 2018) left out: why did the engineer choose these and not other rules? Did the developers follow certain guidelines to make the approach transparent and extensible? Is the approach transferable to other problems? Developing answers to such questions might be difficult, but is indispensable to make rewriting methods better understood and widespread. In this review, we distinguish two fields of research that attempt to make the development of grammars more transparent and streamlined: first, the facilitated creation of grammars and second, the idea of using rewriting methods within a standardized system modeling language, synthesizing a design solution from an abstract network of functions.

5.1 Facilitated Development of Rewriting Rules

In the practice of grammar development, a significant problem to be remedied is that domain experts are rarely familiar with the computational aspects of rewriting. A simple solution might be to let developers and domain experts collaborate, which was the strategy in expert systems research. Unfortunately, this strategy needed to be omitted, acknowledging that experts have limited time, motivation and that arising communication barriers may cause frustration (Puppe, 1990).

As a first alternative, one may attempt to not let domain experts define the grammar, but to generate the rules based on design artifacts they produced in the past. The problem of automatically "learning" or creating a grammar from a given dataset is referred to as *inverse procedural modeling* and is commonly assessed to be very complex (Puppe, 1990; Lienhard, 2017). Nevertheless, a modest amount of research has been classified by Lienhard (2017). One set of works attempts to adapt relatively generic template grammars to a given dataset, mostly in the context of facade parsing (Nishida et al., 2016). This approach likely has a limited potential to generally automate the development of grammars as any template grammar needs to be developed a priori and relatively specialized on the problems to be covered. Instead, the second class of research deals with the induction of set grammar vocabulary and rules with the help of statistical models. These approaches were considered to require very structured and annotated data structures, as in a hierarchical tree structure (Leblanc et al., 2011). Otherwise, it was thought to be impossible to identify the nodes representing the vocabulary of the grammar (Talton et al., 2012). However, recent efforts indicate that it might be possible to induce grammars for more unstructured datasets, including hierarchical and non-hierarchial, one- and more dimensional representations (Whiting et al., 2018). Despite these efforts, this research field is certainly still at a very fundamental level and inverse grammar modeling is not yet applicable to a wider range of engineering design problems.

A second alternative is to look for solutions that empower a wide range of domain experts to develop grammars. Besides guidelines to teach non-specialists good practices in the definition and organization of rewriting patterns (McCormack et al., 2008; Oster and McCormack, 2011), attention is drawn to provide less technical interfaces. As first example of this trend, Abualdenien and Borrmann (2021) recently proposed a method

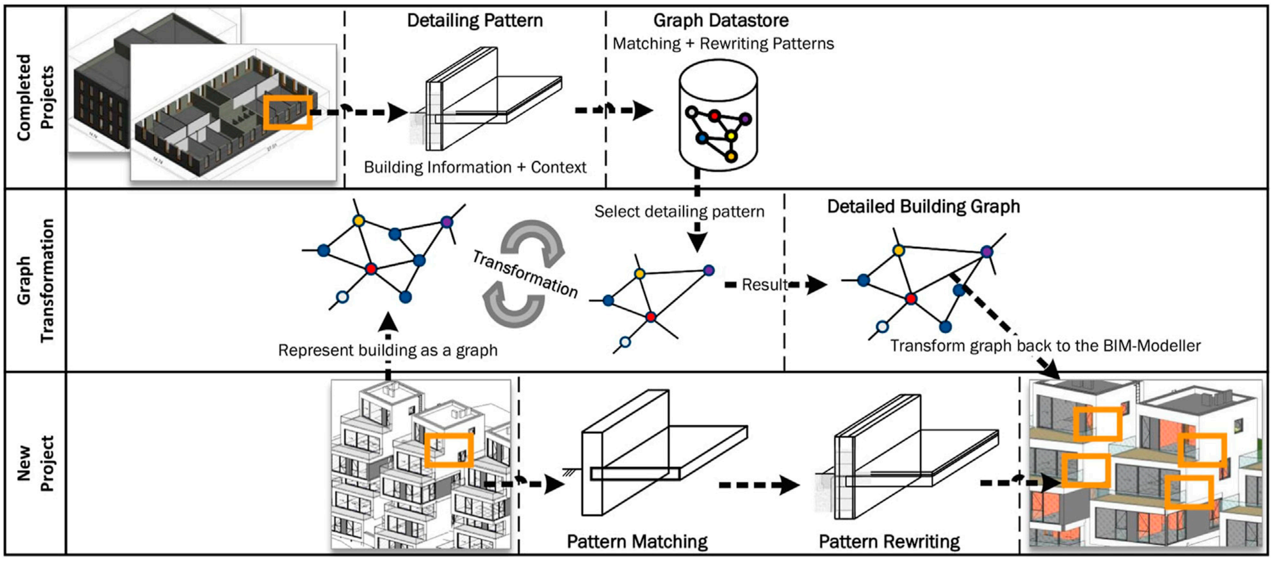


FIGURE 15 | Worfklow proposing a user-oriented interface to formalize architectural design patterns without knowledge of underlying representation and algorithms (Abualdenien and Borrmann, 2021).

to graphically capture architectural detailing patterns through the interfaces of common BIM software. The translation of these patterns to rewriting rules and the later application of rules is possible without knowledge of the underlying computational processes. A conceptual sketch of the workflow is shown in **Figure 15**.

Comparable in intention, Rossi (2021) developed a largely graphical interface to define vocabulary, rules, and a geometrically constrained search procedure for the assembly of segmented structures (Rossi and Tessmann, 2017b). Patow (2012) made the split grammars of Wonka et al. (2003) accessible in an interactive graph visualization and allowed users to refine a procedure by rewriting patterns. Equally, the interfaces of spatial grammar interpreters incrementally require less technical knowledge to define geometrically and semantically complex grammars. Hoisl and Shea (2011), Dy and Stouffs (2018) and Grasl (2021) developed largely graphical interfaces to their spatial grammar interpreters, requiring little knowledge of the underlying computational processes. These interfaces can be advantageous when learning a specific implementation of a rewriting formalism or for the rapid prototyping of design grammars departing from their geometrical frontend.

The interfaces discussed make it easier to define smaller sets of rules. From a certain amount of rules captured, a stricter organization or modularization of the rule sets is important. A promising approach to this end offers the research field of function-based design synthesis.

5.2 Function-Based Design Synthesis

Graphs have the ability to represent a system in any degree of abstraction. This property is exploited in model-based systems engineering (Haberfellner et al., 2019; Hick et al., 2021), also referred to as function-based design synthesis (Cagan et al., 2005; Chakrabarti et al., 2011). The fundamental idea is to begin the design with a set of requirements a system needs to fulfill and to convert respectively map them to a network of functions, the functions to subfunctions, and finally, the subfunctions to components, sometimes *structures*. Only the latter are assigned concrete geometry. Each of those layers can be represented as a network of objects, i.e. in a graph-based representation. The most common modeling conventions have been adopted by standard modeling languages like SysML (2021).

Graph rewriting can be employed within the scope of function-based design synthesis in two ways: first, for model-to-model transformations between the different abstraction layers. If a subfunction can be met by different components, different rules can be created to depict these possible transformations. By exploring the possible options, different concepts of an engineering system can be generated. This has been illustrated for mechanical engineering products by Bryant et al. (2006) or for the conceptual design of bridges by Slusarczyk and Strug (2017). A second use of graph rewriting techniques is of course on the structural level of abstraction, where rules can be embedded to determine the (optimal) configuration of function-derived elements. Tonhäuser and Rudolph (2017) show the entire process revisiting the well-known coffee maker example.

The abstraction involved in function-based design synthesis is a burden and a potential at the same time. Engineers naturally tend to details and visualizations, but thereby run the risk to get stuck in fixed ideas of design (Haberfellner et al., 2019). The abstraction of products by requirements and functions is a demanding and time-consuming process but may enable a better focus on the product essentials and trigger creativity. Further, it can simplify the definition and organization of rewriting rules (Helms and Shea, 2012). To understand the difference, the classification of engineering problems by Puppe (1990) can be referred to: deriving the structural components of a design from a model-based representation of functionality results in a combination of an *assignment* and a *configuration* problem. For further search and optimization tasks, this is easier to solve than the *planning* task most design grammars aim to solve. Nevertheless, it is also possible to use a set of rules to evolve a design through an optimized sequence of actions, traversing various incomplete, intermediary states. Techniques to guide such a generation process are discussed in the next chapter.

6 GENERATIVE USE OF REWRITING RULES

6.1 Classification of Approaches

A graph rewriting system represents a set of process steps, without specifying their logic of application. Thus, besides the rules, the process of applying them to one or a range of specific problems must be designed. This difference is essential, whereas often the misconception is encountered that a given grammar can simply "crank out" (Krishnamurti and Stouffs, 1993) design solutions. Likely, this misconception comes from the fact that many early grammars for design, e.g., the palladian grammar (Stiny and Mitchell, 1978), included a lot of domain knowledge. These rules were restricted to be applied in a relatively specific order and to relatively specific problems. Further, another set of grammars must be taken into consideration. These grammars have very generic rules, applicable to a variety of problems. Most grammars that aim to solve the truss optimization problem (Shea, 1997) belong to this type. These grammars heavily rely on a sound description of their *dynamic* application to a variable environment. To highlight the difference between these two types of grammars, Ruiz-Montiel et al. (2013) introduced the terms *expert grammars* and *naive grammars*. This classification is comprehensible, even though there can be observed some hybrid approaches combining naive and expert rules (Puentes et al., 2020).

The distinction of reasoning approaches according to the problem-specificity of rules is certainly comprehensible but is not the only one. Hou and Stouffs (2018) prefer to categorize grammars according to the generation logic involved, distinguishing an *object-oriented* and a *goal-oriented* type. The former studies the configuration of the desired design outcome and restricts the vocabulary to the subsystems which are finally present, e.g., the building elements of a built house (Mitchell, 1991). This principle, found in the set grammars discussed in **Section 4.2**, facilitates the definition of rules and the exploration of design alternatives. The goal-oriented type, instead, defines rules by reflecting the most concise design process, commonly leading to more abstract non-terminal vocabulary and intermediary design states. To prioritize among the many options of action that may lead to the desired goal state, either a global search with an evaluation of the entire design or a local reasoning mechanism is applied. This categorization into object-oriented and goal-oriented approaches is equally comprehensible and applicable to most works discussed in this review.

The presented criteria enable to reflect a given grammar by it's specificity to one application environment and by the characteristics of the vocabulary. However, there is no commonly agreed taxonomy that can subsume a wide range of grammar-based reasoning techniques or could even practically support engineers in the design of their rule application strategy. Of course, not every grammar needs to be utilized with a sophisticated process control. Especially for expert grammars and object-oriented grammars, engineers often can constrain the search to a few variable parameters and few necessary choices. If this is possible, the exploration of the solution space may be achieved manually, by combinatorial methods as the enumerative generation or black-box methods as *generate and test* optimization algorithms. These three methods are commonly understood and agreed (Cagan et al., 2005; Grasl and Economou, 2013; Ruiz-Montiel et al., 2013). Still, some research should be mentioned that attempts to get more fine-grained control of the generation process. Thereby, a body of research can be subsumed into the paradigm of agent-based modeling, e.g., Heckel (2006), Ruiz-Montiel et al. (2013), McComb et al. (2017), and Puentes et al. (2020), another one to the use of logical descriptions (Duarte, 2005; Stouffs, 2015; Hou and Stouffs, 2019). The cited literature is referred to for further reading, while the following sections are limited to the well-established approaches.

6.2 Search Strategies

6.2.1 Enumerative Generation

If the engineer is able to define a sequence of rule applications that likely lead to valid designs, it is possible to simply enumerate all possible outcomes. Thereby, a search tree may be used to depict the options of generation. This tree commonly has as root the initial design state, and as edges the applicable rewriting rules. The linked children nodes are derived designs (Campbell et al., 2009). Filtered for repetitive and invalid leaf nodes, the search tree can serve for an enumerative generation of designs. This is a relatively old idea (Stiny and Mitchell, 1978), but still popular, due to the good impression it gives about the strictness, respectively expressiveness of a given grammar. To remedy the computational load and complexity of filtering the entire solution space, some remedies exist. Lienhard (2017) proposed clustering methods for building designs that make the solution space more comprehensible to a user. Campbell et al. (2009) and Kumar et al. (2014) propose to only generate a small set of possible solutions from a search tree. Depending on the quality of the produced designs, the later generations are optimized by updating the probabilities of decisions that led to good designs.

This is essentially the idea of a set of optimization algorithms that can be summarized as *generate and test* approaches. Generate-and test approaches do not use search trees but optimize only the input parameters of a generation process based on the cost of the outcome. No formal model of the generation process is required, wherefore one may think of it as a black-box process. Iteratively, the configuration shall be improved until only one or a set of few good designs remain. The two main algorithmic ideas used are simulated annealing and genetic programming (Ruiz-Montiel et al., 2013).

6.2.2 Optimization Algorithms

Simulated annealing optimizes a single design configuration based on the analogy of the cooling of metal. A steadily decreasing "temperature" curve quantifies the willingness to accept deteriorations of cost by a rule application. An in-depth explanation of the algorithmic ideas is given by Cagan and Mitchell (1993), some recent applications include the optimization of piping systems of exhaust filters (Vogel, 2016) or rollerblade wheels (Zimmermann et al., 2018). For simulated annealing and a similar optimization algorithm, Königseder (2015) introduced methods to better understand the internals of grammars in a search process, e.g., the sensitivity and frequency of rules used. Remedying the restriction of simulated annealing to a single design solution, McComb et al. (2017) or Zimmermann et al. (2018) discuss the organization of multiple optimizations in parallel.

Genetic programming relies on a biological "survival of the fittest" principle. In every cycle, a set of designs is created where only the best ones are selected to go into the next phase or cycle. During every generation process, rules can be applied to conduct mutations of individuals. Further, the cross-over exchange of parameters is a desirable mechanism to create diversity within a population. The latter is desirable but very difficult as grammar-generated designs often do not share a common configuration of objects (van Diepen and Shea, 2019; Grzesiak-Kopeć et al., 2021). For hierarchical and well-structured representations, the works of Alber and Rudolph (2003), Talton et al. (2012), and Lienhard (2017) propose a possible solution: instead of processing the individual designs generated, it might be more promising to consider the rules the subject of a genetic optimization. Different variants are generated by different grammars in the first step. In further cycles, the grammars authoring the most successful individuals may be merged, by splitting and merging their vocabulary. This approach requires a deeper understanding of reasoning with grammars, but may contribute to make genetic optimization more powerful.

This concludes the overview of approaches to the reasoning for grammar-based design. In the last chapter, both potentials and shortcomings for further applications in the building sector are summarized.

7 CONCLUSION

7.1 Potentials

Graph models provide a powerful and flexible representation for many engineering products. For engineering design, the use of graph rewriting methods can enable the automation of complex design sequences. To this end, a variety of representation approaches can be distinguished, which can be classified according to the geometric meaning of the graph entities chosen. Low-level geometry representations give a high control and intuitiveness regarding geometric aspects, even though they require the introduction of higher-level textual or graphical interfaces. Set grammar approaches allow defining the design and design steps in a semantically more intuitive, object-oriented way. The extension of employed graph structures to entities of imperative logic or the combination of different programming paradigms may leverage the practical applicability of grammars in a broader context.

The development of graph rewriting systems for applications in engineering design receives increasingly more support. On the one hand, domain experts and learners with little knowledge about the underlying technology are encouraged by less technical and more graphical interfaces. In order to generate and optimize designs based on graph rewriting systems, established approaches can be relied on to perform an efficient search of vast solution spaces.

7.2 Shortcomings

To date, only a few industrial applications of graph rewriting methods have been known in engineering design. This may be owed to several challenges we discussed. One aspect is that the representation of a design problem by a graph model requires abstracting the system in a suitable manner. A variety of different approaches exists, with advantages and disadvantages. A key factor thereby is to represent and manipulate the geometry of engineering products properly. Approaches with a low-level representation of geometry often have the shortcoming of not enabling the definition of rules at a level of abstraction natural to engineers. Approaches with a high-level representation of geometry pose the challenge to efficiently store, transform, and interpret geometry. Despite a rich body of applied works, there is little theoretical discussion about the demanding task of defining a graph representation for a synthesis problem. Ideally, guidelines should be available to support engineers in the conceptual and technical design.

Given a meaningful representation, the efficient design of small sets of rules is a comparably resolved challenge. Still, the technical organization of grammars with larger rule sets to enable an efficient but variable generation of designs is a challenge. To this end, the use of function-based synthesis approaches seems promising, but yet has very few applications in the building sector. Further, the design of search methodologies in combination with a grammar is challenging. A large set of reasoning approaches have been described, differing by the way domain knowledge is formalized, the type of vocabulary, or the locality of evaluation criteria. However, a better uniform characterization and supportive guidelines could support engineers to better understand and design the functionalities of grammars for a generative design process.

AUTHOR CONTRIBUTIONS

LK: Main author, corresponding author SV: Writing sections, proofreading and contributing by prior research JA: Writing sections, proofreading and contributing by prior research AB: Writing sections, proofreading and contributing by prior research.

FUNDING

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

REFERENCES

Abualdenien, J., and Borrmann, A. (2021). "PBG: A Parametric Building Graph Capturing and Transferring Detailing Patterns of Building Models," in Proc. of the CIB W78 Conference 2021 (Luxembourg: International Council for Research and Innovation in Building and Construction).

Alber, R., and Rudolph, S. (2003). 43 - a Generic Approach for Engineering Design Grammars: Aaai Technical Report Ss-03-02

Aouat, A., Bendella, F., and Deba, E. a. (2012). "Tools of Model Transformation by Graph Transformation," in 2012 IEEE International Conference on Computer Science and Automation Engineering (IEEE), 425–428. doi:10.1109/icsess.2012.6269495

Bak, C. (2015). *GP 2: Efficient Implementation of a Graph Programming Language* (York, United Kingdom: University of York). Ph.D. thesis.

Borrmann, A., and Rank, E. (2009). Topological Analysis of 3D Building Models Using a Spatial Query Language. *Adv. Eng. Inform.* 23, 370–385. doi:10.1016/j.aei.2009.06.001

Braun, A., Tuttas, S., Borrmann, A., and Stilla, U. (2015). "Automated Progress Monitoring Based on Photogrammetric point Clouds and Precedence Relationship Graphs," in The 32nd International Symposium on Automation and Robotics in Construction and Mining (ISARC 2015) (Oulu, Finnland: The International Association for Automation and Robotics in Construction (IAARC)). doi:10.22260/isarc2015/0034

Bryant, C. R., McAdams, D. A., Stone, R. B., Kurtoglu, T., and Campbell, M. I. (2006). "A Validation Study of an Automated Concept Generator Design Tool," in Volume 4a: 18th International Conference on Design Theory and Methodology (Philadelphia, United States: ASME), 283–294. doi:10.1115/DETC2006-99489

Cagan, J., Campbell, M. I., Finger, S., and Tomiyama, T. (2005). A Framework for Computational Design Synthesis: Model and Applications. *J. Comput. Inf. Sci. Eng.* 5, 171–181. doi:10.1115/1.2013289

Cagan, J., and Mitchell, W. J. (1993). Optimally Directed Shape Generation by Shape Annealing. *Environ. Plann. B* 20, 5–12. doi:10.1068/b200005

Campbell, M. I., Rai, R., and Kurtoglu, T. (2009). "A Stochastic Graph Grammar Algorithm for Interactive Search," in Volume 8: 14th Design for Manufacturing and the Life Cycle Conference; 6th Symposium on International Design and Design Education; 21st International Conference on Design Theory and Methodology, Parts A and B (Las Vegas, United States: ASME), 829–840. doi:10.1115/DETC2009-86804

Chakrabarti, A., Shea, K., Stone, R., Cagan, J., Campbell, M., Hernandez, N. V., et al. (2011). Computer-based Design Synthesis Research: An Overview. *ASME J. Comput. Inf. Sci. Eng.* 11 (2), 021003. doi:10.1115/1.3593409

Chen, Y., Sareh, P., Yan, J., Fallah, A. S., and Feng, J. (2019). An Integrated Geometric-Graph-Theoretic Approach to Representing Origami Structures and Their Corresponding Truss Frameworks. *J. Mech. Des.* 141 (9), 091402. doi:10.1115/1.4042791

Chomsky, N. (1959). On Certain Formal Properties of Grammars. *Inf. Control.* 2, 137–167. doi:10.1016/s0019-9958(59)90362-6

Corradini, A., Montanari, U., Rossi, F., Ehrig, H., Heckel, R., and Löwe, M. (1997). "Algebraic Approaches to Graph Transformation - Part I: Basic Concepts and Double Pushout Approach," in *Handbook of Graph Grammars and Computing by Graph Transformation*. Editor G. Rozenberg (Pisa, Italy: World Scientific), 163–245. doi:10.1142/9789812384720_0003

Drewes, F., Hoffmann, B., and Plump, D. (2002). Hierarchical Graph Transformation. *J. Comput. Syst. Sci.* 64, 249–283. doi:10.1006/jcss.2001.1790

Duarte, J. P. (2005). A Discursive Grammar for Customizing Mass Housing: the Case of Siza's Houses at Malagueira. *Automation in Construction* 14, 265–275. doi:10.1016/j.autcon.2004.07.013

Dy, B., and Stouffs, R. (2018). Combining Geometries and Descriptions: A Shape Grammar Plug-In for Grasshopper. *eCAADe* 36 (2), 498–598.

Economou, A., and Grasl, T. (2018). "Paperless Grammars," in *Computational Studies on Cultural Variation and Heredity*. Editor J.-H. Lee (Singapore: Springer), 139–160. KAIST Research Series. doi:10.1007/978-981-10-8189-7_12

Ehrig, H. (2006). *Fundamentals of Algebraic Graph Transformation*. Berlin/Heidelberg: Springer-Verlag. doi:10.1007/3-540-31188-2

Ermel, C., Rudolf, M., and Taentzer, G. (1999). "The AGG Approach: Language and Environment," in *Handbook of Graph Grammars and Computing by Graph Transformation: Volume 2: Applications, Languages and Tools* (Singapore: World Scientific), 551–603. doi:10.1142/9789812815149_0014

Geiß, R., Batz, G. V., Grund, D., Hack, S., and Szalkowski, A. (2006). "GrGen: A Fast SPO-Based Graph Rewriting Tool," in *Graph Transformations*. Editors D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, et al. (Berlin, Heidelberg: Springer), 383–397. vol. 4178 of Lecture Notes in Computer Science. doi:10.1007/11841883_27

Geiß, R. (2008). *Graphersetzung mit Anwendungen im Übersetzerbau* (Karlsruhe, Germany: Karlsruhe Institute of Technology). Ph.D. thesis.

Ghamarian, A. H., de Mol, M., Rensink, A., Zambon, E., and Zimakova, M. (2012). Modelling and Analysis Using GROOVE. *Int. J. Softw. Tools Technol. Transfer* 14, 15–40. doi:10.1007/s10009-011-0186-x

Grabska, E., Łachwa, A., and Ślusarczyk, G. (2012). New Visual Languages Supporting Design of Multi-Storey Buildings. *Adv. Eng. Inform.* 26, 681–690. doi:10.1016/j.aei.2012.03.009

Grasl, T., and Economou, A. (2013). From Topologies to Shapes: Parametric Shape Grammars Implemented by Graphs. *Environ. Plann. B* 40, 905–922. doi:10.1068/b38156

Grasl, T. (2021). Grape Web Editor. Vienna, Austria: SWAP Architekten Zt GmbH. Available at: http://grape.swap-zt.com/App.

Grzesiak-Kopeć, K., Strug, B., and Ślusarczyk, G. (2021). Evolutionary Methods in House Floor Plan Design. *Appl. Sci.* 11, 8229. doi:10.3390/app11178229

Habel, A., Heckel, R., and Taentzer, G. (1996). Graph Grammars with Negative Application Conditions. *Fundamenta Informaticae* 26, 287–313. doi:10.3233/fi-1996-263404

Haberfellner, R., de Weck, O. L., Fricke, E., and Vössner, S. (2019). *Systems Engineering: Fundamentals and Applications*. Cham, Switzerland: Birkhäuser.

Havemann, S. (2005). *Generative Mesh Modeling* (Braunschweig: Technical University of Braunschweig). Ph.D. thesis. doi:10.24355/DBBS.084-200603150100-7

Heckel, R. (2006). Graph Transformation in a Nutshell. *Electron. Notes Theor. Comput. Sci.* 148, 187–198. doi:10.1016/j.entcs.2005.12.018

Heisserman, J. (1994). Generative Geometric Design. *IEEE Comput. Grap. Appl.* 14, 37–45. doi:10.1109/38.267469

Helms, B., and Shea, K. (2012). Computational Synthesis of Product Architectures Based on Object-Oriented Graph Grammars. *J. Mech. Des.* 134. doi:10.1115/1.4005592

H. Hick, K. Küpper, and H. Sorger (Editors) (2021). *Systems Engineering for Automotive Powertrain Development*. 1st ed (Cham: Springer International Publishing). Springer eBook Collection. doi:10.1007/978-3-319-99629-5

Hohmann, B., Havemann, S., Krispel, U., and Fellner, D. (2010). A GML Shape Grammar for Semantically Enriched 3d Building Models. *Comput. Graphics* 34, 322–334. doi:10.1016/j.cag.2010.05.007

Hoisl, F., and Shea, K. (2011). An Interactive, Visual Approach to Developing and Applying Parametric Three-Dimensional Spatial Grammars. *AIEDAM* 25, 333–356. doi:10.1017/s0890060411000205

Hoisl, F., and Shea, K. (2013). Three-dimensional Labels: A Unified Approach to Labels for a General Spatial Grammar Interpreter. *AIEDAM* 27, 359–375. doi:10.1017/S0890060413000188

Hooshmand, A., and Campbell, M. I. (2016). Truss Layout Design and Optimization Using a Generative Synthesis Approach. *Comput. Structures* 163, 1–28. doi:10.1016/j.compstruc.2015.09.010

Hou, D., and Stouffs, R. (2019). An Algorithmic Design Grammar Embedded with Heuristics. *Automation in Construction* 102, 308–331. doi:10.1016/j.autcon.2019.01.024

Hou, D., and Stouffs, R. (2018). An Algorithmic Design Grammar for Problem Solving. *Automation in Construction* 94, 417–437. doi:10.1016/j.autcon.2018.07.013

Jabi, W., Aish, R., Lannon, S., and Wardhana, N. (2018). "Topologic: A Toolkit for Spatial and Topological Modeling," in *Computing for a Better Tomorrow* (Poland: Lodz University of Technology).

Jakumeit, E., Blomer, J., and Geiß, R. (2021). GrGen Documentation. Karlsruhe, Germany: Institut für Programmstrukturen und Datenorganisation, Universität Karlsruhe. Available at: http://www.info.uni-karlsruhe.de/software/grgen/GrGenNET-Manual.pdf.

Jakumeit, E., Buchwald, S., and Kroll, M. (2010). Grgen.net. *Int. J. Softw. Tools Technol. Transfer* 12, 263–271. doi:10.1007/s10009-010-0148-8

Kahani, N., Bagherzadeh, M., Cordy, J. R., Dingel, J., and Varró, D. (2019). Survey and Classification of Model Transformation Tools. *Softw. Syst. Model.* 18, 2361–2397. doi:10.1007/s10270-018-0665-6

Kaveh, A., and Koohestani, K. (2008). Graph Products for Configuration Processing of Space Structures. *Comput. Structures* 86, 1219–1231. doi:10.1016/j.compstruc.2007.11.005

Kneidl, A., Borrmann, A., and Hartmann, D. (2012). Generation and Use of Sparse Navigation Graphs for Microscopic Pedestrian Simulation Models. *Adv. Eng. Inform.* 26, 669–680. EG-ICE 2011 + SI: Modern Concurrent Engineering. doi:10.1016/j.aei.2012.03.006

Knight, T. (2003). Computing with Emergence. *Environ. Plann. B Plann. Des.* 30, 125–155. doi:10.1068/b12914

Kolbeck, L., Auer, D., Fischer, O., Vilgertshofer, S., and Borrmann, A. (2021). "Modulare Brückenbauwerke aus carbonfaserbewehrtem Ultrahochleistungsbeton," in *Beton- und Stahlbetonbau Sonderheft "Schneller Bauen"* (Berlin, Germany: Ernst & Sohn), 116.

Königseder, C. (2015). *A Methodology for Supporting Design Grammar Development and Application in Computational Design Synthesis* (ETH Zurich). Ph.D. thesis. doi:10.3929/ethz-a-010567138

Krishnamurti, R., and Earl, C. F. (1992). Shape Recognition in Three Dimensions. *Environ. Plann. B* 19, 585–603. doi:10.1068/b190585

Krishnamurti, R., and Stouffs, R. (1993). "Spatial Grammars: Motivation, Comparison and New Results," in Proceedings of the 5th International Conference on Computer-Aided Architectural Design Futures (Pittsburgh, USA: Springer), 57–74.

Kumar, M., Campbell, M. I., Königseder, C., and Shea, K. (2014). "Rule Based Stochastic Tree Search," in *Design Computing and Cognition '12*. Editor J. S. Gero (Dordrecht: Springer Netherlands), 571–587. doi:10.1007/978-94-017-9112-0_31

Langenhan, C., Weber, M., Liwicki, M., Petzold, F., and Dengel, A. (2013). Graph-based Retrieval of Building Information Models for Supporting the Early Design Stages. *Adv. Eng. Inform.* 27, 413–426. doi:10.1016/j.aei.2013.04.005

Leblanc, T., Leblanc, J., and Poulin, P. (2011). "Component-based Modeling of Complete Buildings," in *Proceedings of Graphics Interface 2011* (Toronto, Ontario, Canada: Canadian Human-Computer Communications Society), 39, 87–100. GI 2011. doi:10.1177/009182961103900111

Lienhard, S. (2017). *Visualization, Adaptation, and Transformation of Procedural Grammars*. Ph.D. thesis. Lausanne: EPFL Lausanne. doi:10.5075/EPFL-THESIS-7627

Lindenmayer, A. (1968). Mathematical Models for Cellular Interactions in Development I. Filaments with One-Sided Inputs. *J. Theor. Biol.* 18, 280–299. doi:10.1016/0022-5193(68)90079-9

Lipp, M., Wonka, P., and Wimmer, M. (2008). Interactive Visual Editing of Grammars for Procedural Architecture. *ACM Trans. Graph.* 27, 1–10. doi:10.1145/1360612.1360701

Mc Neel and Associates (2021). Grasshopper Rhino. Seattle, United States: Mc Neel & Associates.

McComb, C., Cagan, J., and Kotovsky, K. (2017). Capturing Human Sequence-Learning Abilities in Configuration Design Tasks through Markov Chains. *J. Mech. Des.* 139 (9), 091101. doi:10.1115/1.4037185

McCormack, J., Cagan, J., and Antaki, J. (2008). "Aligning Shape Rule Creation with Modular Design: Minimizing the Cost of Using Shape Grammars," in Volume 4: 20th International Conference on Design Theory and Methodology; Second International Conference on Micro- and Nanosystems (ASMEDC), 107–118. doi:10.1115/DETC2008-49366

Mitchell, W. J. (1991). "Functional Grammars: An Introduction," in *Reality and Virtual Reality* (Newark, NJ: Association for Computer Aided Design in Architecture School of Architecture New Jersey Institute of Technology).

Nagl, M. (1979). *Graph-Grammatiken: Theorie Anwendungen Implementierung*. Wiesbaden: Vieweg. Studienbücher Informatik. doi:10.1007/978-3-663-01443-0

Nagl, M., Schürr, A., and Münch, M. (Editors) (2003). Applications of Graph Transformations with Industrial Relevance: International Workshop, AGTIVE'99, Kerkrade, The Netherlands, September 1-3, 1999 (Berlin: Springer-Verlag). Proceedings.

Nishida, G., Garcia-Dorado, I., Aliaga, D. G., Benes, B., and Bousseau, A. (2016). Interactive Sketching of Urban Procedural Models. *ACM Trans. Graph.* 35, 1–11. doi:10.1145/2897824.2925951

Oster, A., and McCormack, J. (2011). A Methodology for Creating Shape Rules during Product Design. *J. Mech. Des.* 133 (6), 061007. doi:10.1115/1.4004195

Patow, G. (2012). User-friendly Graph Editing for Procedural Modeling of Buildings. *IEEE Comput. Grap. Appl.* 32, 66–75. doi:10.1109/MCG.2010.104

Peysakhov, M., and Regli, W. C. (2003). Using Assembly Representations to Enable Evolutionary Design of Lego Structures. *AIEDAM* 17, 155–168. doi:10.1017/S0890060403172046

Puentes, L., Cagan, J., and McComb, C. (2020). Heuristic-guided Solution Search through a Two-Tiered Design Grammar. *J. Comput. Inf. Sci. Eng.* 20 (1), 011008. doi:10.1115/1.4044694

Puppe, F. (1990). *Problemlösungsmethoden in Expertensystemen*. Berlin, Heidelberg: Springer Berlin Heidelberg. Springer eBook Collection Computer Science and Engineering. doi:10.1007/978-3-642-76133-1

Rensink, A., and Taentzer, G. (2007). "Agtive 2007 Graph Transformation Tool Contest," in *International Symposium on Applications of Graph Transformations with Industrial Relevance* (Berlin: Springer-Verlag), 487–492.

Rensink, A. (2003). "The Groove Simulator: A Tool for State Space Generation," in *International Workshop on Applications of Graph Transformations with Industrial Relevance* (Berlin: Springer-Verlag), 479–485.

Riestenpatt, M., and Rudolph, S. (2019). "A Scientific Discourse on Creativity and Innovation in the Formal Context of Graph-Based Design Languages," in *Heron Island Conference on Computational and Cognitive Models of Creative Design*. Association for Computer Aided Design in Architecture. Herons Islands, Australia: Design Research Society (DRS).

Robinson, I., Webber, J., and Eifrem, E. (2015). *Graph Databases: New Opportunities for Connected Data*. second edition edn. Beijing and Boston and Farnham: O'Reilly.

Rossi, A., and Tessmann, O. (2017a). "Aggregated Structures: Approximating Topology Optimized Material Distribution with Discrete Building Blocks," in Proceedings of the IASS Annual Symposium 2017 Interfaces: Architecture, Engineering, Science, Hamburg, Germany. International Association of Shell & Spatial Structures.

Rossi, A., and Tessmann, O. (2017b). "Designing with Digital Materials: A Computational Framework for Discrete Assembly Design," in *Protocols, Flows and Glitches*. Editors P. Janssen, P. Loh, A. Raonic, and M. A. Schnabel (Hong Kong: CAADRIA).

Rossi, A. (2021). Wasp: Grasshopper Plugin for Discrete Assemblies.

Rozenberg, G. (1997). *Handbook of Graph Grammars and Computing by Graph Transformation*, Vol. 1. Singapore: World Scientific.

Rudolph, S. (2002). *Übertragung von Ähnlichkeitsbegriffen*. Stuttgart: Habilitation, Universität Stuttgart.

Ruiz-Montiel, M., Boned, J., Gavilanes, J., Jiménez, E., Mandow, L., and Pérez-de-la-Cruz, J.-L. (2013). Design with Shape Grammars and Reinforcement Learning. *Adv. Eng. Inform.* 27, 230–245. doi:10.1016/j.aei.2012.12.004

Runge, O., Ermel, C., and Taentzer, G. (2011). "AGG 2.0–new Features for Specifying and Analyzing Algebraic Graph Transformations," in *International Symposium on Applications of Graph Transformations with Industrial Relevance* (Berlin: Springer-Verlag), 81–88.

Schürr, A., Winter, A. J., and Zündorf, A. (1995). "Graph Grammar Engineering with PROGRES,". *Software Engineering - ESEC '95*. Editor W. Schäfer (Berlin: Springer-Verlag).

Shea, K. (1997). *Essays of Discrete Structures: Purposeful Design of Grammatical Structures by Directed Stochastic Search* (Pittsburgh: Carnegie Mellon University). Dissertation.

Silva, P. B., Müller, P., Bidarra, R., and Coelho, A. (2013). "Node-based Shape Grammar Representation and Editing," in Proceedings of the Workshop on Procedural Content Generation in Games PCG'13 (New York, United States: Association for Computing Machinery), 1–8.

Slusarczyk, E. G., and Strug, B. (2017). "A Graph-Based Generative Method for Supporting Bridge Design," in *24th EG-ICE International Workshop on Intelligent Computing in Engineering (EG-ICE 2017)*. Editors C. Koch, W. Tizani, and J. Ninić (Red Hook, NY: Curran Associates Inc).

Stiny, G. (1980). Introduction to Shape and Shape Grammars. *Environ. Plann. B* 7, 343–351. doi:10.1068/b070343

Stiny, G., and Mitchell, W. J. (1978). Counting Palladian Plans. *Environ. Plann. B Plann. Des.* 5, 189–198. doi:10.1068/b050189

Stouffs, R. (2015). "Description Grammars: An Overview," in *Emerging Experience in Past, Present and Future of Digital Architecture*. Editors Y. Ikeda, C. M. Herr,

S. Holzer, M. Kaijima, and M. Kim (Hong Kong: Association for Computer-Aided Architectural Design Research in Asia (CAADRIA)), 137–146.

Stouffs, R. (2019). "Predicates and Directives for a Parametric-Associative Matching Mechanism for Shapes and Shape Grammars," in *Blucher Design Proceedings* (São Paulo: Editora Blucher), 403–414. doi:10.5151/proceedings-ecaadesigradi2019_658

Strug, B., Ślusarczyk, G., Paszyńska, A., and Palacz, W. (2022). "A Survey of Different Graph Structures Used in Modeling Design, Engineering and Computer Science Problems," in *Graph-Based Modeling in Science, Technology and Art*. Editors S. Zawiślak and J. Rysiński (Cham: Springer International Publishing), 243–275. vol. 107 of Mechanisms and Machine Science. doi:10.1007/978-3-030-76787-7_12

Sysml (2021). SysML Open Source Project - what Is SysML? Who Created SysML. Available at: SysML.org.

Talton, J., Yang, L., Kumar, R., Lim, M., Goodman, N., and Měch, R. (2012). "Learning Design Patterns with Bayesian Grammar Induction," in *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology - UIST '12*. Editors R. Miller, H. Benko, and C. Latulipe (New York, New York, USA: ACM Press), 63. doi:10.1145/2380116.2380127

Tonhäuser, C., and Rudolph, S. (2017). "Individual Coffee Maker Design Using Graph-Based Design Languages," in *Design Computing and Cognition '16*. Editor J. S. Gero (Cham: Springer International Publishing), 513–533. doi:10.1007/978-3-319-44989-0_28

Tratt, L. (2010). "Theory and Practice of Model Transformations," in Third International Conference, ICMT 2010, Malaga, Spain, June 28-July 2. 2010. Proceedings. doi:10.1007/978-3-642-13688-7

van Diepen, M., and Shea, K. (2019). A Spatial Grammar Method for the Computational Design Synthesis of Virtual Soft Locomotion Robots. *J. Mech. Des.* 141 (10), 101402. doi:10.1115/1.4043314

Veit Batz, G., Kroll, M., and Geiß, R. (2008). "A First Experimental Evaluation of Search Plan Driven Graph Pattern Matching," in *Applications of Graph Transformations with Industrial Relevance*. Editors A. Schürr, M. Nagl, and A. Zündorf (Berlin, Heidelberg: Springer), 471–486. doi:10.1007/978-3-540-89020-1_32

Vestartas, P. (2021). *Design-to-Fabrication Workflow for Raw-Sawn-Timber Using Joinery Solver* (LausanneSwitzerland: EPFL). Ph.D. thesis. doi:10.5075/EPFL-THESIS-8928

Vilgertshofer, S., and Borrmann, A. (2017). Using Graph Rewriting Methods for the Semi-automatic Generation of Parametric Infrastructure Models. *Adv. Eng. Inform.* 33, 502–515. doi:10.1016/j.aei.2017.07.003

Vogel, S. (2016). *Über Ordnungsmechanismen im wissensbasierten Entwurf von SCR-Systemen* (Stuttgart: Universität Stuttgart). Dissertation. doi:10.18419/opus-8829

Whiting, M. E., Cagan, J., and LeDuc, P. (2018). Efficient Probabilistic Grammar Induction for Design. *AIEDAM* 32, 177–188. doi:10.1017/S0890060417000464

Wonka, P., Wimmer, M., Sillion, F., and Ribarsky, W. (2003). "Instant Architecture," in *ACM SIGGRAPH 2003 Papers on - SIGGRAPH '03*. Editor A. P. Rockwood (New York, USA: ACM Press), 669. doi:10.1145/1201775.882324

Wortmann, T. (2013). *Representing Shapes as Graphs : A Feasible Approach for the Computer Implementation of Parametric Visual Calculating* (USA: Massachusetts Institute of Technology). Diploma thesis.

Zimmermann, L., Chen, T., and Shea, K. (2018). A 3d, Performance-Driven Generative Design Framework: Automating the Link from a 3d Spatial Grammar Interpreter to Structural Finite Element Analysis and Stochastic Optimization. *AIEDAM* 32, 189–199. doi:10.1017/S0890060417000324

Conflict of Interest: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Publisher's Note: All claims expressed in this article are solely those of the authors and do not necessarily represent those of their affiliated organizations, or those of the publisher, the editors and the reviewers. Any product that may be evaluated in this article, or claim that may be made by its manufacturer, is not guaranteed or endorsed by the publisher.