Design Science

Improving design grammar development and application through network-based analysis of transition graphs

Corinna Königseder[1], Tino Stanković[1] and Kristina Shea[1]

1 *Engineering Design and Computing Laboratory, Department of Mechanical and Process Engineering, ETH Zurich, 8092 Zurich, Switzerland*

Abstract

Design grammars enable the formal representation of a vocabulary and rules that describe how designs can be synthesized just as the grammar rules of a spoken language define how to formulate valid, i.e., grammatically correct, sentences. Design grammars have been successfully applied in numerous engineering disciplines and enable the automated synthesis of designs within a defined design language. Design grammar development, however, is challenging and lacks methodological support. In this paper, a novel method is presented that supports the development and application of design grammars using transition graphs. In these, nodes represent generated designs and edges represent grammar rules that transform one design into another. Rather than using a tree structure to represent the possible application of rules, transition graphs are automatically generated and used to help designers better understand the developed grammar. The grammar designer is given feedback on (a) the rules, and (b) rule application sequences. This feedback can be used to (a) improve the grammar, and (b) apply it more efficiently. Two case studies, a gearbox synthesis task and a sliding tile puzzle, demonstrate the method. The results show the feasibility of the method to support design grammar development and application.

Key words: generative grammar, design grammar development, design synthesis, network analysis, transition graph

Received 5 November 2015
Revised 8 March 2016
Accepted 16 March 2016

Corresponding author
C. Königseder
ck@ethz.ch

Published by Cambridge University Press
© The Author(s) 2016
Distributed as Open Access under a CC-BY-NC-SA 4.0 license (http://creativecommons.org/licenses/by-nc-sa/4.0/)

Des. Sci., vol. 2, e5
journals.cambridge.org/dsj
DOI: 10.1017/dsj.2016.5

1. Introduction

Most people come across the term 'grammar rule' when learning a foreign language. Grammar rules allow them to combine their learnt vocabulary in a meaningful way such that the sentences they build can be understood by others familiar with the language. Linguistics is, however, only one area where grammars are used nowadays.

Since the seminal work by Chomsky (1957), various research fields emerged that utilize formal grammar concepts. Mullins & Rinderle (1991) define grammars as 'a structured way of describing the relationships between the entities of a

A shorter version of this manuscript has been published as 'Königseder, C., Stanković, T., and Shea, K., 2015. 'Improving Generative Grammar Development and Application Through Network Analysis Techniques', International Conference on Engineering Design (ICED), Milano, Italy.'

https://doi.org/10.1017/dsj.2016.5 Published online by Cambridge University Press

language, whether that language be English prose, a computer programming language, a shape language, or perhaps a language of mechanical function and form' (Mullins & Rinderle 1991). This definition suggests that grammars are used in a wide range of applications. Grammar rules are powerful representations to describe how from a finite set of elements (words, vocabulary) an infinite set of designs (sentences) can be developed.

Grammatical approaches have been successfully applied in numerous engineering design disciplines (Chakrabarti *et al.* 2011), e.g., in electrical engineering, architecture and mechanical engineering or in natural language processing, e.g., for automated language translation or speech recognition. In the area of compiler design, grammars are designed to automatically translate implemented source code from a higher programming language into machine code, and a whole research area of grammar engineering evolved dealing with the development of grammars. An elaborate set of methods to develop and analyze grammar rules exists in this area.

In architecture, grammars are successfully used to inspire designers' thinking when developing new designs, or to capture and merge architectural styles. In mechanical engineering, grammars are often used to develop product concepts in the early stages of the design process. Though their use has shown success in the past in generating both known and new designs, the development of grammar rules is challenging. Several researchers mention the need for more support in the development of engineering design grammars (Gips 1999; McKay *et al.* 2012*a*) and the lack of support for grammar design is still seen as one of the major drawbacks of grammatical design (Chakrabarti *et al.* 2011). Engineering design grammars are often developed for specific use cases and no commonly accepted criteria for 'good grammars' have been specified by the engineering design community. While in some areas, e.g., computer science in compiler design, different algorithms exist to analyze properties of the language that is described by a grammar, such analyses are usually not done when developing engineering design grammars. As an example, one can look at research on the automated synthesis of gearboxes, a commonly addressed engineering design task. Research on grammars for gearbox synthesis has been carried out by several researchers (Li & Schmidt 2004; Lin *et al.* 2010). Most researchers developed their own grammar for the problem instead of reusing previously developed ones and the grammar development process is usually not documented. Being presented only the final versions of different developed grammars and the synthesis results, it is not obvious to the designer which grammar or which particular rules are preferable for gearbox synthesis and why. Further, none of the publications investigated in depth how the grammar explores the design space. This makes it difficult to fully understand the importance and influence of single grammar rules on the synthesis process. The development of a new grammar for a particular design problem can therefore be seen as an art rather than a science. This makes the use of grammar-based methods for design synthesis challenging and might be one reason why they are still not wide-spread despite their potential for generating design alternatives and solution spaces.

The goal of the research presented in this paper is, therefore, to support human designers in developing grammar rules for grammar-based design synthesis methods. This is done by providing a method to systematically analyze developed grammar rules and their influence on the generated designs. The method is meant

https://doi.org/10.1017/dsj.2016.5 Published online by Cambridge University Press

to provide more support for (a) the development of the grammar rules due to a better understanding of how they explore the design space, and (b) the application of sequences of rules. Systematic analyses can help designers to understand new and existing grammars in depth and allow them to make more informed decisions on reusing or developing engineering design grammar rules.

The paper is structured as follows. In Section 2, an overview of Computational Design Synthesis (CDS) in general and using grammars is presented. The use of design grammars is illustrated on the example of gearbox synthesis. Then, transition graphs are introduced and the analogy to grammar-based CDS is given using the gearbox example. In Section 3, the Network-Based Rule Analysis Method (NBRAM) to support grammar development and application is presented. In Section 4, the computational implementation of the NBRAM is described. Section 5 demonstrates how the NBRAM can be used to analyze application conditions for rules in detail using a gearbox synthesis case study. Section 6 demonstrates how the method is used to analyze a grammar for a sliding tile puzzle. Beneficial rule sequences are identified on a small-scale and successfully applied to solve a larger-scale problem. The results show the feasibility of the method to support grammar development and application. The method and its generality are discussed in Section 7. Conclusions are given in Section 8.

2. Background

In this section, first a general CDS framework is presented. Then, grammar-based methods for CDS are presented and explained using a gearbox design synthesis case study. Background information is given on transition graphs and the analogy between transition graphs for compiler design and CDS is presented, which inspired the method presented in this paper.

2.1. Computational Design Synthesis (CDS)

CDS is a research area that develops methods and tools for supporting the generation of novel and creative, but also routine designs. Two major benefits of CDS methods are mentioned in Chakrabarti *et al.* (2011). First, the use of computer-based methods enables overcoming restrictions of human designers such as limited knowledge or design fixation. The computer is not biased per se and explores directions the human designer would probably not consider and thus computational methods have a chance to explore novel designs. Second, computer-based methods can support routine design by automating tedious tasks. In this paper, the terminology for the CDS process is used as defined by Cagan *et al.* (2005). In a first step, the designer formalizes the design problem at the required level of detail to allow for the synthesis of meaningful designs. After the representation is formalized, the CDS process consists of three repeated phases: generate, evaluate and guide. In the design generation phase, a design is selected and modified to represent a new design alternative that is then evaluated considering defined objectives and constraints. A decision is made in the search on how to proceed in the synthesis process, i.e., either to accept or reject the new alternative. The synthesis process is continued until either no further modifications are possible or it is stopped by a stopping criterion in the search method.

https://doi.org/10.1017/dsj.2016.5 Published online by Cambridge University Press

2.2. Grammar-based CDS

Stiny & Gips (1972) introduced shape grammars in 1972 and proposed the use of production systems (grammars) for design tasks. Parametric shape grammars were then developed and applied, e.g., to construct traditional Chinese ice-ray lattice designs (Stiny 1977) or to describe Palladio's style for villa ground plans (Stiny & Mitchell 1978). Since then, research on shape grammars has become an active research area especially in architecture and also in other areas like engineering design (Chakrabarti *et al.* 2011). Besides shape grammars, other types of grammars are used in engineering design. Examples are parallel grammars (Starling & Shea 2005), spatial grammars (Gmeiner & Shea 2013; Hoisl & Shea 2013; Hoisl 2012; McKay *et al.* 2012*b*) and graph grammars (Fu, Depennington & Saia 1993; Helms *et al.* 2009; Schmidt & Cagan 1997; Schmidt, Shetty & Chase 2000). An introduction to grammatical approaches for engineering design can be found in Mullins & Rinderle (1991), Rinderle (1991) and Cagan (2001).

In grammatical approaches to CDS, designers develop a grammar to represent a desired design language. It consists of a vocabulary, usually describing design elements and subsystems, as well as a set of grammar rules. These rules describe design transformations that are defined by a left-hand-side (LHS) and a right-hand-side (RHS), i.e., LHS → RHS. The LHS defines where the rule can be applied in a design and the RHS defines how the design transformation modifies the design.

Using graph grammars for design synthesis, each design is described using a graph representation that consists of nodes and edges. For example, nodes can represent components while edges describe functional or spatial relations between the nodes.

2.3. Example gearbox synthesis grammar

To demonstrate the use of grammars for CDS on an example, a gearbox synthesis grammar is presented in Figure 1. This rule set is further used in the case study in Section 5. The rule set consists of five topologic rules and is a subset of the rule set described by Königseder & Shea (2015). It was originally developed by Lin *et al.* (2010). Topologic rules change the topology of a graph, i.e., they add or remove nodes and edges. In Figure 1, the rule number and name are given (left) along with the graph grammar rule (middle) and a 3D representation of the rule (right).

An example rule application sequence consisting of two rule applications is visualized in Figure 2. The generated graphs are given on the left, a 3D representation of the respective designs on the right. The process is started from an initial design consisting of two shaft nodes. In iteration 1, rule 3 (*create a new gear pair*, see Figure 1) is applied. The LHS of the rule consists of two shafts. These are matched to the two shafts in the initial design. The RHS of rule 3 consists of the two shafts and adds two gears between the shafts. The edge between gear and shaft indicates that the gear sits on the shaft. The edge between the two gears indicates that the two gears are in mesh. In iteration 2, rule 1 (*create a new shaft*, see Figure 1) is applied. Again, the two shafts are matched and a new shaft is added connecting the two matched shafts via two added gear pairs. Rules to both add and remove graph elements are included in the rule set since it is anticipated that the grammar will be combined with a stochastic search algorithm to effectively search the solution space.

https://doi.org/10.1017/dsj.2016.5 Published online by Cambridge University Press

	Graph grammar rule	3D representation
1 - Create a new Shaft		
2 - Delete a Shaft		
3 - Create a new Gear Pair		
4 - Delete a Gear Pair		
5 - Replace a Gear Pair		
Legend	Shaft (graph) Gear (graph)	Shaft (3D representation) Gear (3D representation)

Figure 1. Grammar rule set for gearbox synthesis (adapted from Lin *et al.* (2010)).

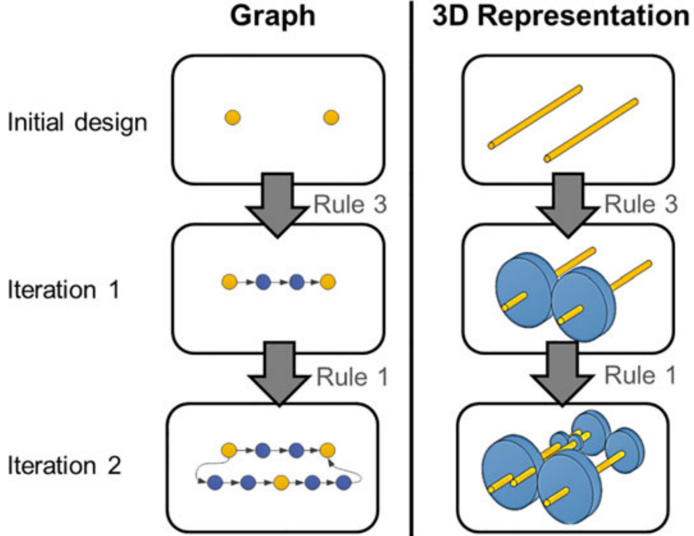


Figure 2. Example rule application for gearbox synthesis rule set.

Many researchers in grammar-based CDS view the synthesis of designs as search through a generative tree, e.g., Brown (1997). In this tree, each node represents a design and the edges represent rule applications. Tree-based search methods like Breadth-First Search (BFS), Depth-First Search (DFS) or more sophisticated tree-search methods (Kumar *et al.* 2012) can then be applied to search for desirable designs. In Figure 3, a representation of a generative search tree for the gearbox synthesis problem is given. The depth of the search tree is limited to two rule applications starting from the initial design.

https://doi.org/10.1017/dsj.2016.5 Published online by Cambridge University Press

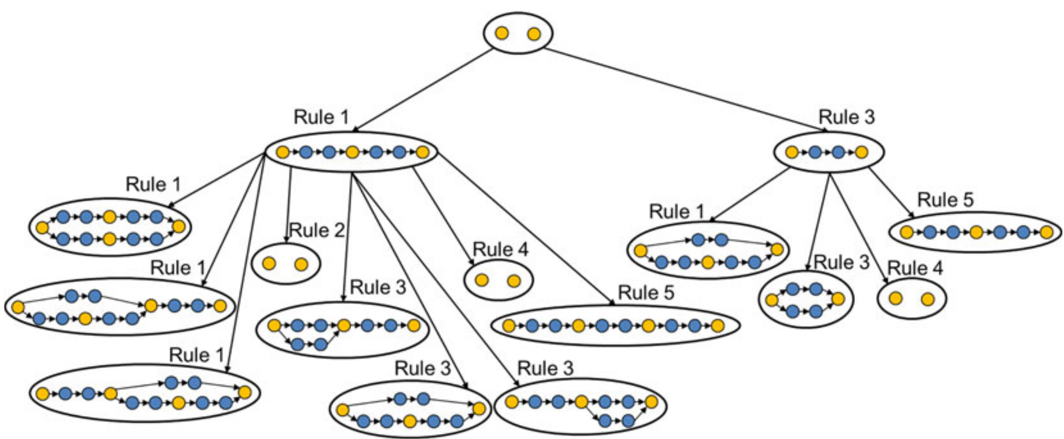


Figure 3. Generative search tree for gearbox synthesis problem.

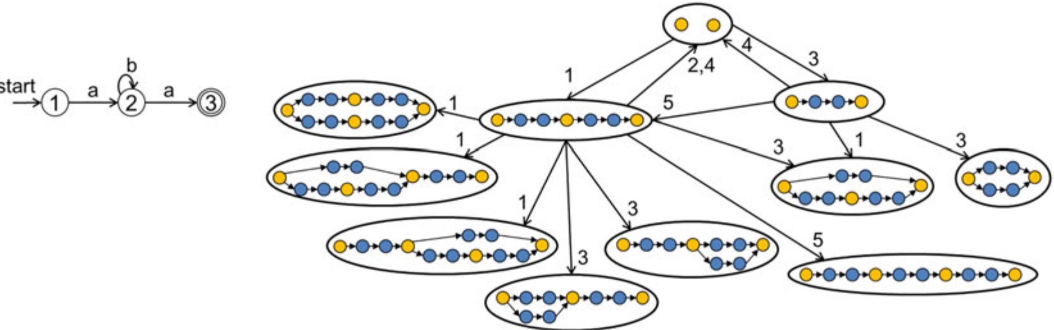


Figure 4. Analogy between transition graphs for a finite automaton (left) and to describe relations between designs generated during the CDS process (right).

2.4. Transition graphs and the analogy to CDS

Transition graphs as used, e.g., to describe finite automata in compiler design, inspired the presented research. They are introduced in the following and the analogy to CDS in engineering design is given.

Finite automata are recognizers that either accept or reject a given input string. Each finite automaton consists of a set of states including a start state and one or more final states, a set of input symbols (the input alphabet) and a transition function that defines the next states for each state and each symbol (Aho *et al.* 2006). Finite automata can be represented by 'transition graphs, where the nodes are states and the labeled edges represent the transition function. There is an edge labeled a from state s to state t if and only if t is one of the next states for state s and input a' (Aho *et al.* 2006).

An example for a transition graph for a finite automaton is given in Figure 4 (left) where the start state is 1, the final state is 3 and the input symbols are a and b.

This finite automaton accepts all strings of a's and b's that start and end with an a and have no or an arbitrary number of b's in between. With the

https://doi.org/10.1017/dsj.2016.5 Published online by Cambridge University Press

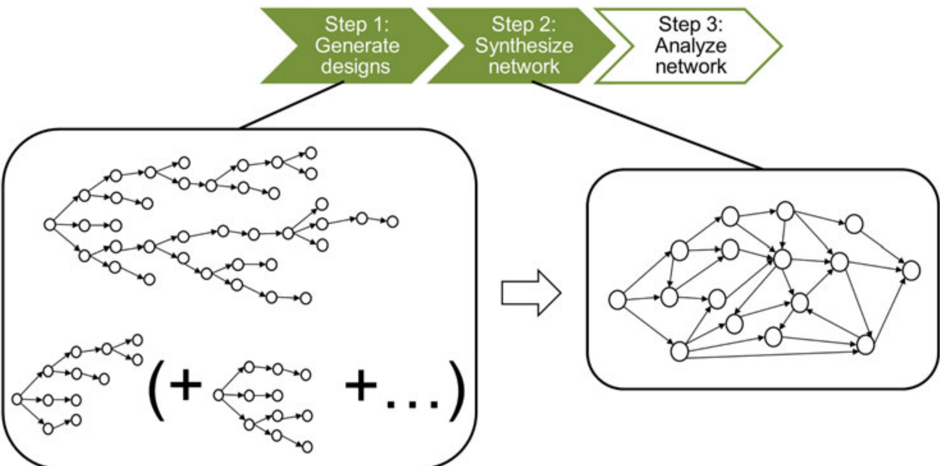


Figure 5. Steps 1 and 2: Network generation through generation of designs (left) and synthesis of designs to a network representation (right).

first input symbol a, the finite automaton transforms from the start state 1 to state 2. For every input symbol b, the finite automaton remains in state 2. The last input symbol a then transforms the finite automaton from state 2 into the final state 3. Examples of accepted strings are *aa, aba, abba* and *abbba.* Similar to an automaton that accepts or rejects a given sequence of input symbols, a transition graph, constructed for an engineering design grammar, could accept or reject a sequence of grammar rules applied to the initial design. In a transition graph for an engineering design grammar, each state represents a modified design and each input symbol represents a grammar rule transforming a state. An example transition graph for representing designs as states and rules as transitions between them is shown in Figure 4 (right). The designs presented here are those explored in Figure 3. The edges are labeled with the rule numbers that transform a design into another. Note that designs generated several times in the generative tree (Figure 3) are presented only once here leading to a network representation. Analyzing such a transition graph using techniques from data flow analysis, the grammar rules and influences of each rule application can be understood in detail.

Due to the identified need for more methodological support for design grammar development and inspired from research on transition graphs, a method is developed to support grammar development and application based on transition graphs.

3. Network-based rule analysis method (NBRAM)

The NBRAM analyzes the designs that are generated during CDS and the rules that are used to synthesize these. Figure 5 illustrates the three main steps.

In *Step 1*, designs are generated by searching through a generative tree. Starting from an initial design, i.e., the root of the generative tree, designs are generated through successive rule applications and each generated design is added to the generative tree as a child node of the previous design. Using tree-based search methods, such as DFS, i.e., expand first one rule sequence, or BFS, i.e., from one

https://doi.org/10.1017/dsj.2016.5 Published online by Cambridge University Press

design apply multiple different rules in parallel before moving to the next level, the design space can be explored. Other search methods are also possible to use in this step and the goal is not to find an optimal design, but to explore a portion of the design space that can be analyzed in the following steps. Figure 5 (left) shows example representations of explored design spaces when using tree-based search methods. Each node in the graph represents one design and each edge between the nodes represents the rule that was applied to transform a design into another one.

Using grammars and search algorithms, the same designs are often generated repeatedly and tree-based representations often represent these designs as multiple different nodes in the tree, each resulting from a different rule sequence. Many search algorithms store a list of already explored designs to avoid expanding on the same design more than once. The authors propose that representing these repeated designs as one unique design instead of multiple times in the search tree can help gain useful insights and decreases the search space size because it allows the generation of a state transition graph, where states are unique.

Therefore, in *Step 2* (Figure 5, right), all generated designs are analyzed to identify unique and repeated designs. Uniqueness is a problem dependent property and can, e.g., refer to a design's topology in a structural design problem or its parameter values for parametric problems. The designer developing the grammar defines what uniqueness means for the given problem. The tree-based representations are traversed gradually and every time a previously undiscovered design is found, it is given a new, unique ID (UID). Every time an already discovered design is found, its node in the tree-based representation is deleted and the edges, i.e., the rule with which the design was generated and the rules that were applied next, are connected to the already generated (unique) design. Doing this, the tree-based representations merge to one or more networks of design transitions. In these networks, called transition graphs, each node represents a unique design, or state, and each edge represents a transition from a source design to a target design. A simple example for this is already given in Figures 3 and 4 (right).

The transition graph can be analyzed to gain information about (a) the rules themselves, and (b) the rule application sequences. In *Step 3* of the NBRAM, different graph analyses are performed (see Figure 6). The designer is given two methods to access the information. First, automated network analyses are performed and results are presented to the human designer. Second, the network is represented visually for manual exploration of the generated designs and their relations. Implementation details of the respective graph searches are given in Section 4. Seven properties are identified to support the analysis of grammar rules and rule sequences. In general, more properties can be investigated. The seven properties presented below are selected to address topics that are (a) found most relevant to support grammar development and application by the authors, and are (b) previously discussed in grammar related research. The properties can be investigated with the NBRAM as shown in Figure 6.

Properties 1 to 3 are investigated automatically when using the NBRAM. In addition, the provided visualizations and interactive tools allow the designer to study the transition graph and investigate properties 4 to 7.

Property 1 (*Do–undo*): Do–undo rule pairs are identified, i.e., pairs of rules where one rule un-does what the other did. A simple example of such a rule pair

https://doi.org/10.1017/dsj.2016.5 Published online by Cambridge University Press

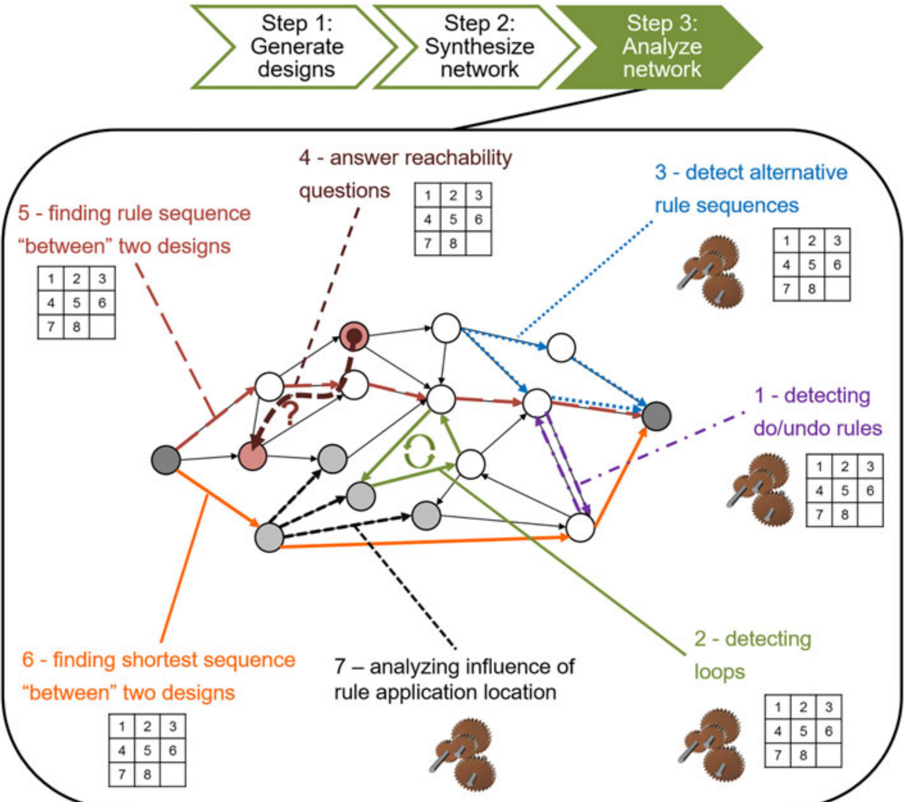


Figure 6. Step 3: Analyzing the transition graph to understand grammar rules and their application. The pictograms for gearbox and sliding tile puzzle indicate for which case study the respective analysis is conducted.

is two rules of which one adds a component and the other one deletes it. While rules that remove components are helpful in the generative process for back tracking, they can also cause difficulties. For the selection of an appropriate search algorithm, this information can be important because a repeated application of do–undo rules might result in generating the same designs over and over again. So, the designer can use this information on do–undo rules to either change the grammar, or select a search algorithm accordingly.

Property 2 (*Loops*): Loops in the transition graph can be identified. Similar to do–undo rules, a loop in the transition graph represents a sequence of rules that, when applied to a design, modify it but in the end recreate exactly the design the sequence started with. Avoiding such loops in the synthesis process, either by reformulating the grammar or through using this information in a search algorithm, can allow for faster design space exploration.

Property 3 (*Multiple paths*): Alternative rule sequences can be identified, i.e., sequences of rules that transform a given design s to a design t via different paths in the transition graph. Sometimes these alternative paths represent sequences of rules, where the order of the rule application does not matter.

https://doi.org/10.1017/dsj.2016.5 Published online by Cambridge University Press

For example, applying rule A first and then rule B leads to the same design as applying rule B first and then rule A. The intermediate designs, however, are different. In other cases, alternative paths can represent different rule application sequences where one sequence consists of more rule applications than the other. If multiple paths exist, the designer can reason about them and consider, e.g., if the rule set can be reduced by deleting or combining rules. Reducing the number of paths can speed up the synthesis process. Depending on the intermediate designs that are generated via the different paths, it can, however, also be useful to preserve the generation of multiple paths and keep the intermediate designs as starting points for further exploration.

Property 4 (*Reachability*): The designer of a grammar is often interested to know if the developed rules are able to generate a certain design or if there is a sequence of rule applications that transforms a given design into a desired design. Questions of reachability can be answered using the transition graph, e.g., is it possible to reach design t from design s? This allows, e.g., analyzing if design t can be synthesized starting the synthesis process from design s.

Property 5 (*Sequences*): Besides knowing if a design t can be reached from design s, designers are usually interested in knowing the required transformations. Which rules have to be applied to transform design s to design t? Understanding the application of rules in sequences depending on the design s on which the sequence is applied can help designers to reason about improving the grammar rules and to learn meaningful sequences.

Property 6 (*Shortest Sequence*): When more than one sequence of rule applications exists to transform a given design into a desired design, the designer either manually selects one or a search algorithm chooses which rules to apply. Sometimes it is worthwhile to explore rule sequences with many rule applications to generate promising intermediate designs (see also Property 3, *multiple paths*). Often, however, it is beneficial to know the shortest rule sequence to transform a design s into a design t. Learning shortest rule sequences to transform one design into another using the transition graph can then help to speed up the synthesis process.

Property 7 (*Location*): It is often the case that the LHS of a rule matches in several locations of a design. How does the rule application location influence the generated results? The effect of a rule's application location in the design can be analyzed by exploring the designs that are generated when applying the same rule to the same design but in different locations.

Analyses for the seven properties are conducted. The pictograms in Figure 6 for the gearbox synthesis (Section 5) and sliding tile puzzle (Section 6) indicate for which case studies the respective analysis results are presented.

4. Implementation

The transition graph is generated using GrGen.NET (GrGen), an open source graph rewriting tool (Geiß *et al.* 2006). It is visualized using OrganicVIZ (Cash, Stanković & Štorga 2014), a graph visualization tool capable of representing large graphs and supporting graph analyses as well as providing several filtering options. Using OrganicVIZ, the designer can manually study the transition graph to understand the search space. Nodes and edges can be changed in size and color

https://doi.org/10.1017/dsj.2016.5 Published online by Cambridge University Press

to provide an overview and emphasize certain designs in the search space. Highly connected designs, i.e., designs with several edges connecting it to other designs can, e.g., be represented with larger nodes. Additional information, e.g., on each design's attributes, can be displayed to the user for each graph node and edge.

For the automated graph analysis, graph grammar rules implemented in GrGen are used to detect do–undo rules (Property 1), loops (Property 2) and alternative sequences (Property 3). To find shortest paths between any two designs in the transition graph, a BFS with backtracking is implemented (Property 6). A console application is developed such that the designer can interactively search shortest paths between designs and determine reachability between designs (Property 4). To analyze arbitrary paths between designs (Property 5) and analyze the influence of the rule application location (Property 7), the visualization in OrganicVIZ can be used.

5. Case study I – gearbox synthesis

Gearbox design using generative grammars is a common CDS problem and research has been carried out by several scientists (Schmidt *et al.* 2000; Li & Schmidt 2004; Lin *et al.* 2010; Starling & Shea 2005; Starling 2004; Swantner & Campbell 2012; Pomrehn & Papalambros 1995). It is therefore considered as an appropriate test case for the NBRAM and demonstrates how the method can be used to address the properties *do–undo*, *loops*, *multiple paths* and *location*.

5.1. Introduction to the gearbox case study

The five topologic rules of the gearbox grammar described in Section 2.3 are used to generate gearbox designs. The search space is explored exhaustively until a predefined depth (d) of the search tree is reached. More details and the pseudo-code of the algorithm are presented in Appendix A. The NBRAM is then applied to synthesize a transition graph from the generated search tree and to analyze the grammar rules and their application in detail.

5.2. Results

Figure 7 shows the transition graph when the five topologic rules are applied exhaustively to an initial design (design with UID 1 in Figure 7). The initial design consists of input shaft, output shaft and a gear pair connecting the two. The maximum sequence length is set to two rules to generate the transition graph in Figure 7, i.e., the depth of the generative tree is two ($d = 2$).

The transition graph is generated automatically and is visualized using OrganicVIZ. Uniqueness is defined in this case study to distinguish between designs according to their topologies. Same colors of the graph nodes indicate that these nodes have the same number of forward and reverse speeds. The pictorial images are generated using yComp (Kroll *et al.* 2015) and are added manually. Node labels (UID) indicate the unique ID of each generated gearbox topology. This number is used to link the nodes in the OrganicVIZ visualization to the gearbox representations that are stored during the exhaustive generation and can be visualized, e.g., using yComp. The edge labels in Figure 7 indicate which rule is applied to transform a design into another one.

https://doi.org/10.1017/dsj.2016.5 Published online by Cambridge University Press

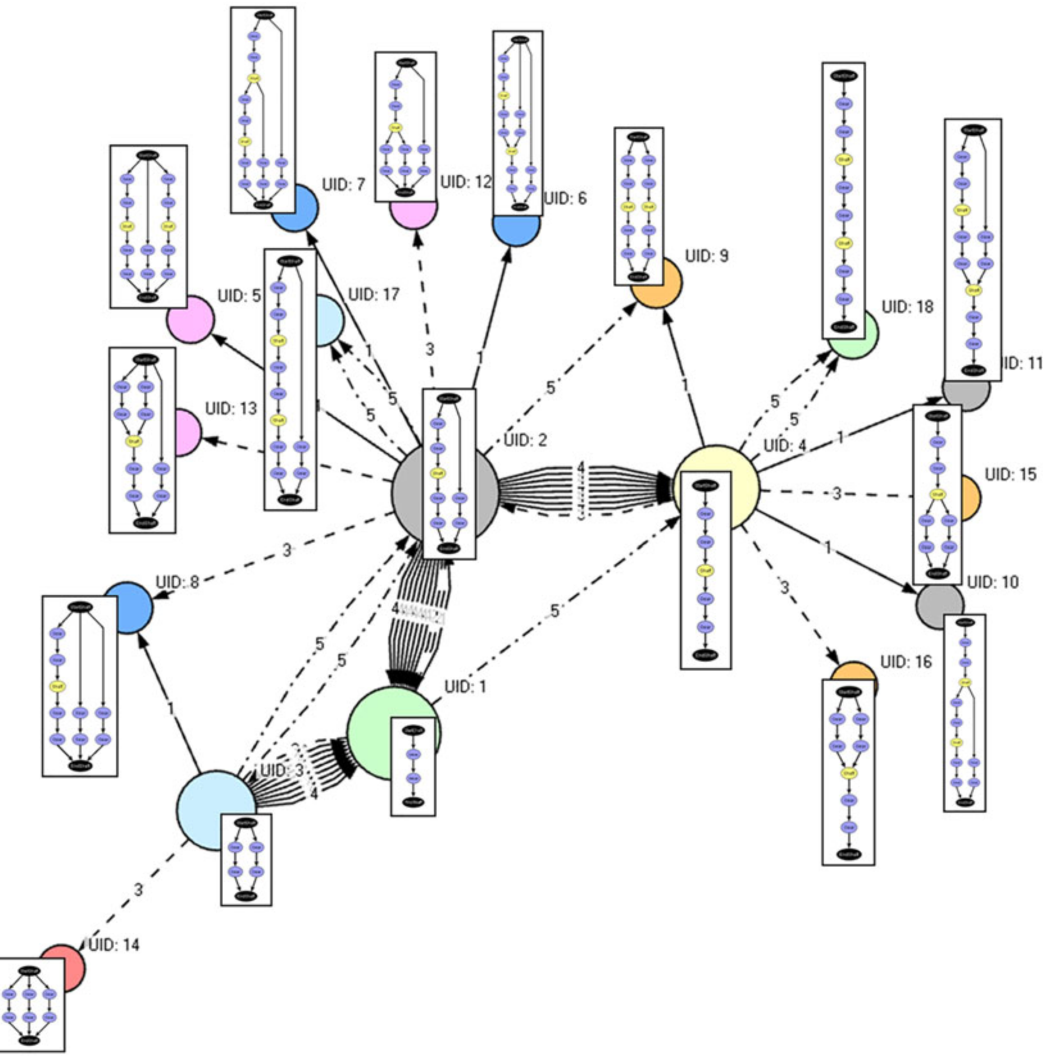


Figure 7. Transition graph for the gearbox case study. Rule sequences with two rules ($d = 2$) are explored exhaustively starting the synthesis from design with UID 1.

5.2.1. Analyzing LHS matches of rules (Property 7: location)

Analyzing all outgoing edges from one node, the human designer can explore the effect of LHS matches when the same rule is applied at different locations in a design. In Figure 7, rule 5 (*replace a gear pair*) is, e.g., applied at three different locations on the design with UID 2. Of the three rule applications two result in the same design (UID 17), while the third generates a different design (UID 9).

Between the designs with UID 1, UID 2, UID 3 and UID 4, rule 2 (*delete a shaft*) and rule 4 (*delete a gear pair*) are applied at various matches but result in the same designs. The human designer might find this interesting because they would expect a different behavior. Figure 8 represents a zoomed in view of the transition graph for the transitions between the designs with UID 1 and UID 2. Pictograms of the gearbox designs and example 3D visualizations are given. When rule 2 (*delete*

https://doi.org/10.1017/dsj.2016.5 Published online by Cambridge University Press

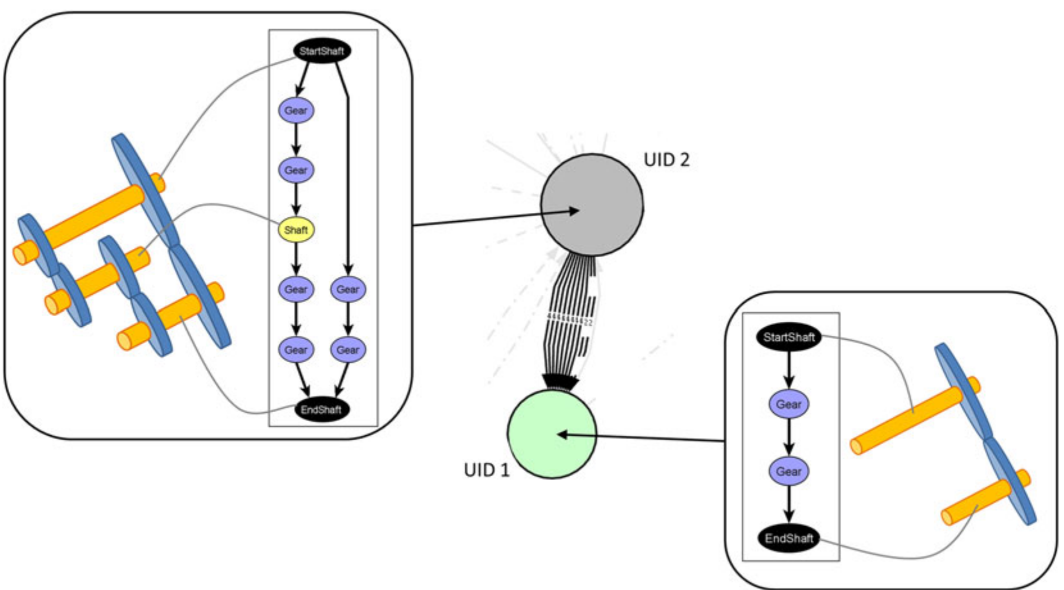


Figure 8. Zoomed in view on transitions for designs with UID 2 and UID 1. The graphs describing the gearbox designs are given along with an example 3D visualization.

a shaft) is applied to the design with UID 2, the human designer might expect that this rule is only matched once, since only one shaft can be removed between input and output shaft that have to remain in the design. This comparison between expected and actual matches of rules facilitates the identification of rules whose LHSs might be formulated too ambiguously or too constrained.

The transitions between the design with UID 2 and that with UID 1 are analyzed in more detail to understand how the formulation of the LHS leads to several matches. It is found that the LHS matching conditions for rules 2 and 4 lead to more matches than intended. Based on that knowledge, the rules are formulated differently to better represent the rule designer's intention of when a rule should be applied. In the original implementation, the rules are formulated using negative application conditions. These describe patterns in the LHS of a rule that define when the rule's application should be prohibited. When a rule is matched on a graph but also the pattern described by the negative application condition exists in this graph then the rule is not applied. Instead of using negative application conditions, i.e., describing patterns that should not exist in the graph, the LHSs of the rules are formulated to describe patterns that have to exist. These are termed positive application conditions and describe patterns that are required in the graph to allow a rule's application. In the following, the change from negative to positive application conditions is explained for rule 2. A more detailed explanation, also for rule 4, is given in Appendix B. In the original implementation, the LHS of rule 2 (*delete a shaft*) is looking for a shaft node and a gear node. The negative application condition describes a pattern such that the matched gear should not be in the same power flow path between input and output shaft as the matched shaft. This means an additional power flow path between input and output shaft has to exist to make sure that input and output shaft are still connected when the matched shaft is removed. Using this LHS rule formulation

https://doi.org/10.1017/dsj.2016.5 Published online by Cambridge University Press

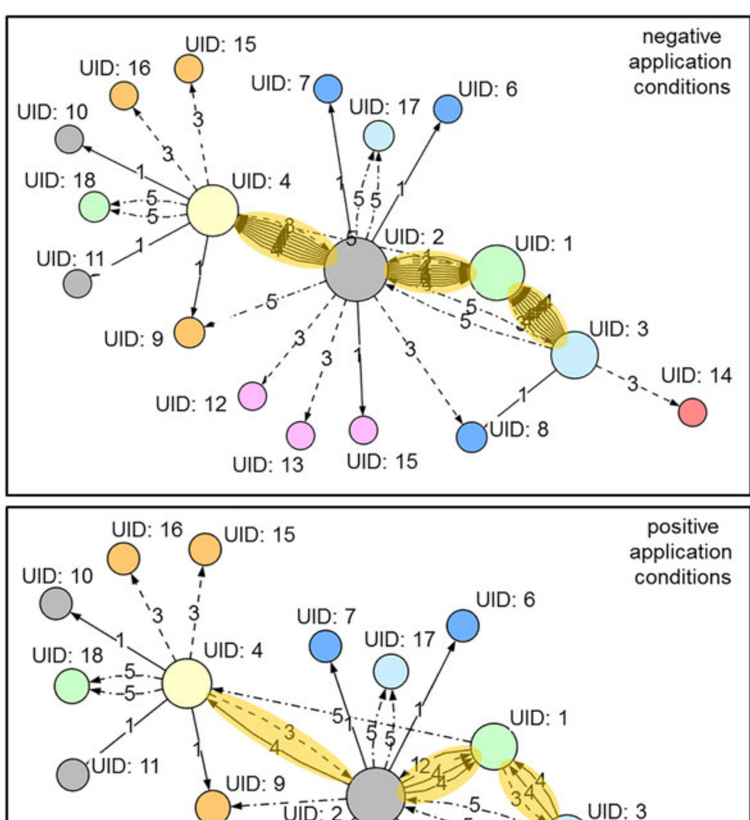


Figure 9. Difference when using old/negative (left) or new/positive (right) implementation of application conditions for rules 2 and 4 ($d = 2$).

with the negative application condition, the shaft in the middle of the left power flow path and either the upper or the lower gear in the right power flow path of the design with UID 2 (Figure 8, left) can be matched. This leads to two possible rule matches. Since the designer intends to have only one match for rule 2 on the design with UID 2, the rule is changed and formulated using positive application conditions. The LHS is changed to only look for a shaft and a positive application condition is formulated as a pattern that contains a gear pair that should not be in the same power flow path as the matched shaft. Changing rule 2 this way, it is only matched once on the design with UID 2.

Changing the formulation of the LHSs of rules 2 and 4 (see Appendix B for details) reduces the number of matches as intended. Figure 9 shows the transition graphs for rule sequences with two rules using negative application conditions (Figure 9, top) and positive application conditions (Figure 9, bottom) for the LHS matching. In both cases, 18 topologically different designs are generated,

https://doi.org/10.1017/dsj.2016.5 Published online by Cambridge University Press

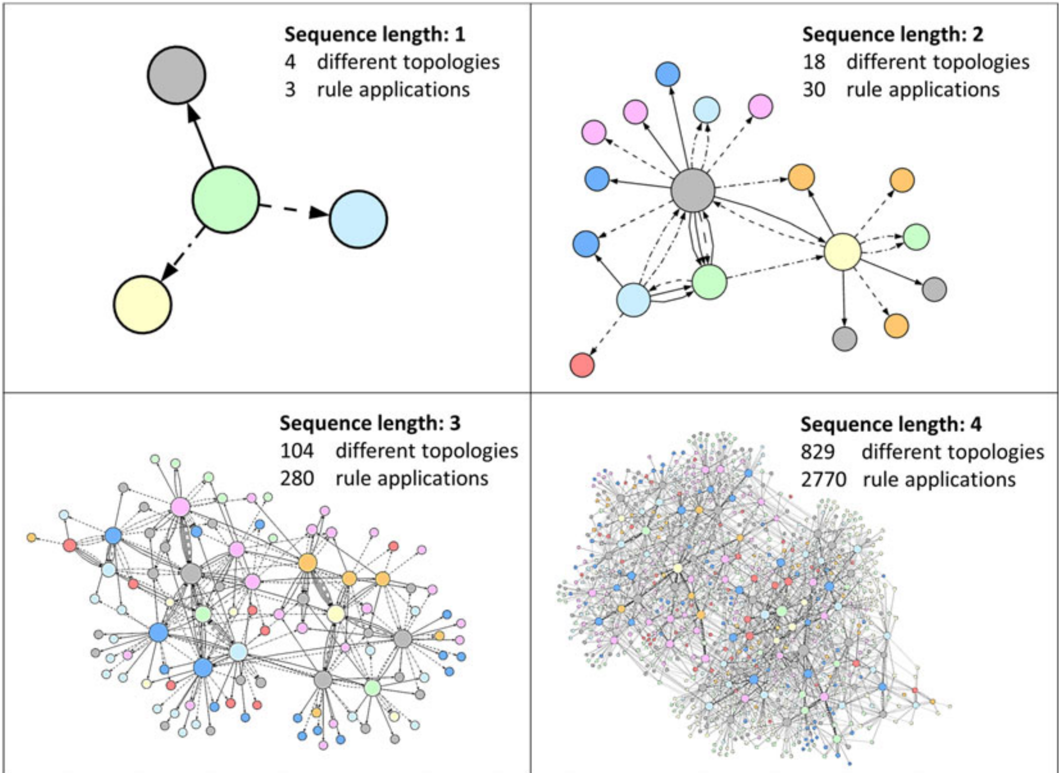


Figure 10. Transition graphs for rule sequences from 1 ($d = 1$) (top left) to 4 ($d = 4$) (bottom right) rule applications.

but the number of rule applications, i.e., edges in the transition graph, varies. It is reduced from 50 to 30 when the LHSs are changed. Regions where the number of edges changes are highlighted in Figure 9. This reduction of LHS matches does not reduce the number of designs that are generated. The speed of exploring the search space, however, changes significantly. For a sequence length of 4 rules, e.g., 829 designs are generated exhaustively within 25 minutes with the new LHS application conditions. When using the old LHS application conditions, by contrast, the process is stopped after approximately 1 hour due to lack of memory on the laptop that is used (MacBook Pro, Intel Core i7 CPU with 2.80 GHz, 8 GB RAM).

5.2.2. Understanding the search space

Figure 10 visualizes transition graphs for sequences with 1–4 rules. For the given rule set, the number of topologically different designs increases drastically and it can be observed that designs are highly connected to each other. This means the same designs can be generated via numerous, different rule sequences.

5.2.3. Detect do–undo rules (Property 1: do–undo)

Besides analyzing the LHSs of the rules, loops can also be detected automatically using the transition graph. Examples for do–undo rules are, e.g., rule 1 (*create a*

https://doi.org/10.1017/dsj.2016.5 Published online by Cambridge University Press

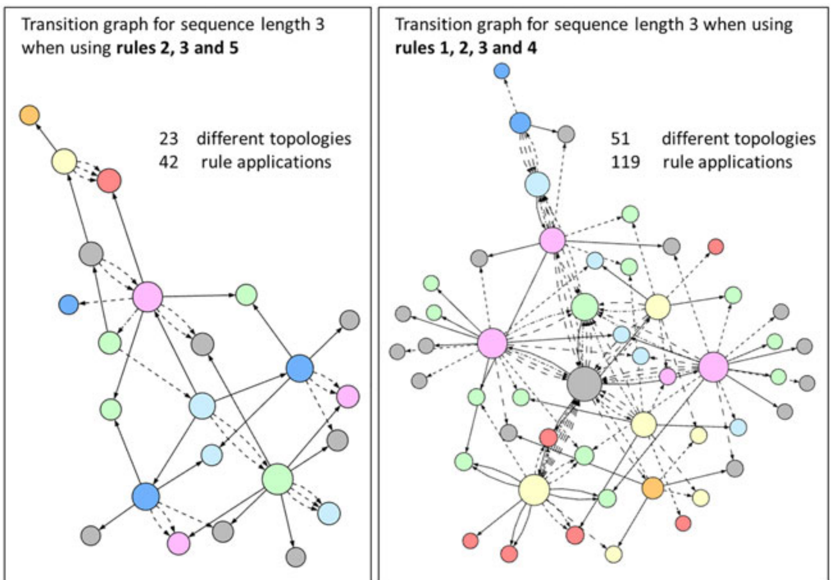


Figure 11. Transition graphs for rule sequences of 3 rule applications ($d = 3$) for different rule sets. Rules 1 and 4 are replaced by rule 5 (left) and rule 5 is replaced by rules 1 and 4 (right).

new shaft) $\longleftrightarrow$ rule 4 (*delete a shaft*), rule 3 (*create a gear pair*) $\longleftrightarrow$ rule 4 (*delete a gear pair*) and rule 1 (*create a new shaft*) $\longleftrightarrow$ rule 4 (*delete a gear pair*). Many loops are found, which is expected for highly connected transition graphs.

5.2.4. Analyze alternative paths (Property 3: multiple paths)

It is further found, that rule 5 (*replace a gear pair*) can be replaced by applying a sequence of rule 4 (*delete a gear pair*) and rule 1 (*create a new shaft*). The NBRAM is used to investigate whether rule 5 can be removed from the rule set. Transition graphs are generated for a sequence length of three rule applications ($d = 3$). Two rule sets are used, which are subsets of the original rule set. In the first rule set, rule 5 is kept but rules 1 and 4 are removed. In the second rule set, rule 5 is removed. Transition graphs for both rule sets are shown in Figure 11. The transition graph for the complete rule set, i.e., rules 1 to 5, and a sequence length of three ($d = 3$) is shown in the lower left corner of Figure 10. Removing rules 1 and 4 from the rule set results in a drastically reduced transition graph. Only 22% of the topologies are explored compared to the complete rule set. Removing rules 1 and 4 is therefore considered inappropriate. When rule 5 is removed from the rule set, 49% of the topologies are explored compared to the complete rule set. Besides the number of generated topologies, the connectivity of designs also varies depending on the used rule set. Using rule 5 results in a stronger connected transition graph, i.e., designs can be transformed into others requiring fewer rule applications. Further, when the exploration is limited to a certain number of rule applications, more designs are explored when rule 5 is included in the rule set.

For a sequence length of four rules, the effect of having rule 5 in the rule set is even stronger. Only 331 gearbox topologies are explored requiring 981 rule

https://doi.org/10.1017/dsj.2016.5 Published online by Cambridge University Press

applications when rule 5 is deleted from the rule set compared to 829 topologies and 2,770 rule applications when rule 5 is used.

5.3. Key findings

Using the NBRAM, different aspects about the gearbox synthesis task and the grammar are learnt. First, the LHSs of rules are analyzed and inefficient implementations of the LHS application conditions are discovered. Second, the highly interconnected nature of the search space and the exponential growth of the number of gearbox topologies when applying longer sequences of rules are understood from the transition graph visualizations. Third, do–undo rule pairs are identified and loops in the transition graph are analyzed automatically. This confirms the gained knowledge, that the gearbox designs are highly connected to each other. Fourth, it is discovered, that rule 5 can be replaced by the sequence rule 4 $\rightarrow$ rule 1. Using the NBRAM, it is found, however, that it is useful to have rule 5 in the rule set. This is in accordance with the decision by Lin *et al.* (2010) to develop this rule for gearbox synthesis.

6. Case study II – sliding tile puzzle

The sliding tile puzzle is used as a second case study to demonstrate how the NBRAM can be used to address properties 1 to 6. This problem has been used frequently since its invention in 1879 (Slocum & Sonneveld 2006) and it is still used in artificial intelligence research. It is easy to understand, however hard to solve both, for humans as well as the computer. Further, it is easily scalable, which is why it is used in this paper to show how beneficial rule sequences can be learnt on a small-scale problem and used to solve larger-scale problems.

6.1. Introduction to the sliding tile puzzle case study

In a sliding tile puzzle, a number of tiles are arranged in a rectangle with one tile missing. By sliding tiles one after another into the missing spot, the tiles have to be ordered in a defined way. Even though the principle is easy to understand, for larger puzzles, numerous possible states exist, leading to a vast number of designs that have to be explored when trying to solve the problem. In 1879 Johnson proved that for each n-tile puzzle where n is the number of tiles, there exist $(n + 1)!$ states with only half of them being solvable (Johnson & Story 1879). For the classical 8-tile puzzle, also called 3×3 puzzle, this leads, e.g., to $9!/2 = 181{,}440$ feasible states. The sliding tile puzzle is challenging not only to humans trying to solve it, but also for optimization algorithms due to the vast number of possible arrangements. When trying to understand the problem, human experts sometimes learn sophisticated relations between certain tile configurations and apply sequences to switch between known configurations quickly. As, in analogy to this, the goal in this paper is to support human designers in understanding relations between different designs and grammar rules that transform those designs, the sliding tile puzzle is a well-suited case study. It demonstrates that through analyzing transition graphs, human designers can gain deeper knowledge about designs and identify beneficial rule application sequences. The NBRAM is applied on a small-scale problem to increase the designer's understanding of

https://doi.org/10.1017/dsj.2016.5 Published online by Cambridge University Press

Table 1. Rule set for the sliding tile puzzle.

Rule	LHS (example) (active positions highlighted)	RHS (example) (active positions highlighted)	Description	Matching condition of LHS (positions of empty tile)
Up	a b d e c	a b c d e	Tile below the empty position is slid UP	
Down	a b c d e	a b d e c	Tile above the empty position is slid DOWN	
Right	a b c d e	a b c d e	Tile left of the empty position is slid RIGHT	
Left	a b c d e	a b c d e	Tile right of the empty position is slid LEFT	

the problem and identify beneficial sequences in a first step. Then the learnt rule sequences are applied automatically on a larger-scale problem.

A small version of the puzzle is used in the first part of the case study consisting of five tiles, numbered one to five, arranged on a 2 × 3 tile grid. A grammar with four rules is developed to modify any given puzzle. The four rules (*Up (U)*, *Down (D)*, *Right (R)* and *Left (L)*) are visualized in Table 1. The LHS of each rule shows an example design on which the rule can be matched and the RHS shows the design after the rule application. Involved tile positions are highlighted in gray. In the last column of Table 1, the possible matches for each rule are given for the 5-tile puzzle, indicated by the positions on which the empty tile can be positioned.

6.2. Part 1: Understanding the small-scale problem

All possible states of the 5-tile puzzle are explored exhaustively. The transition graph is generated consisting of all unique designs where each unique design represents a possible tile configuration as a vector, e.g., (1, 2, 3, 4, 5, 0) represents the target design. The NBRAM is applied and results for properties *do–undo*, *loops*, *multiple paths* and *reachability* are shown in the following sections.

6.2.1. Understanding the search space

As for the sliding tile puzzle, half of the states are not solvable, two transition graphs are generated for all possible permutations of the tiles with numbers (0, 1, 2, 3, 4, 5) with '0' denoting the empty tile. Both graphs are visualized in Figure 12 representing exactly what has been demonstrated mathematically in 1879, namely, that for a given puzzle, half of the states are solvable, whereas the other half are not, and that no solvable puzzle can be transformed into an unsolvable one and vice versa. Figure 13 shows a zoomed in view of the transition graph of solvable designs containing the final design. The node colors and sizes are visualized depending on the degree, i.e., the number of incoming and outgoing edges. Large red (dark)

https://doi.org/10.1017/dsj.2016.5 Published online by Cambridge University Press

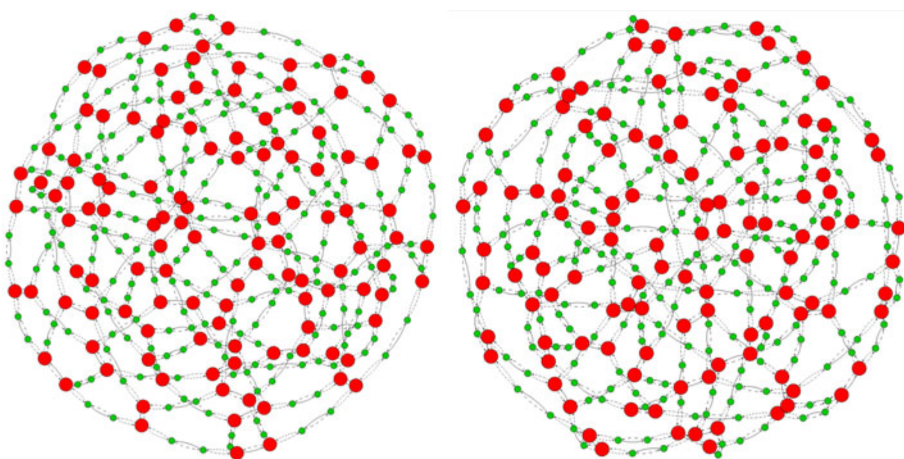


Figure 12. Transition graphs for solvable (left) and unsolvable (right) puzzles. Large, red nodes are of degree six, smaller, green nodes are of degree four.

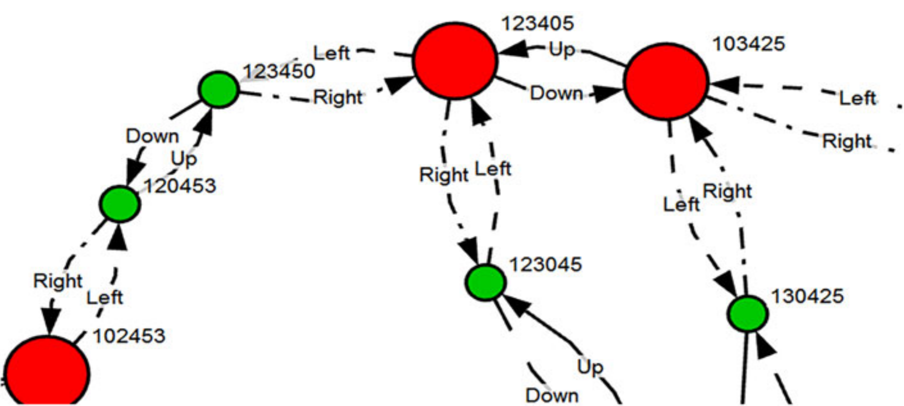


Figure 13. A zoomed in view of the transition graph showing designs and their transformations through rules. Large, red nodes are of degree six, smaller, green nodes are of degree four.

nodes are of degree six and represent designs where the empty tile is in the middle of the tile puzzle. The smaller green (light) nodes are of degree four and represent designs where the empty tile is in one of the corners of the tile puzzle. This shows that designs with higher degree, i.e., with the empty tile in the middle of the puzzle, are stronger connected to other designs because more rule applications are possible from this state than when the empty tile is in a corner.

6.2.2. *Answering questions of reachability (Property 4: reachability)*

For any design s, the question of whether or not it is solvable can be answered by analyzing the reachability of the final state, design t, from the given design s. Any design for which no path to the final state t can be found is unsolvable. This means that to distinguish between the solvable and unsolvable transition graph in Figure 12, one has to identify in which graph the final state is present (see Figure 13, upper left node). Having a closer look at an excerpt of the transition

https://doi.org/10.1017/dsj.2016.5 Published online by Cambridge University Press

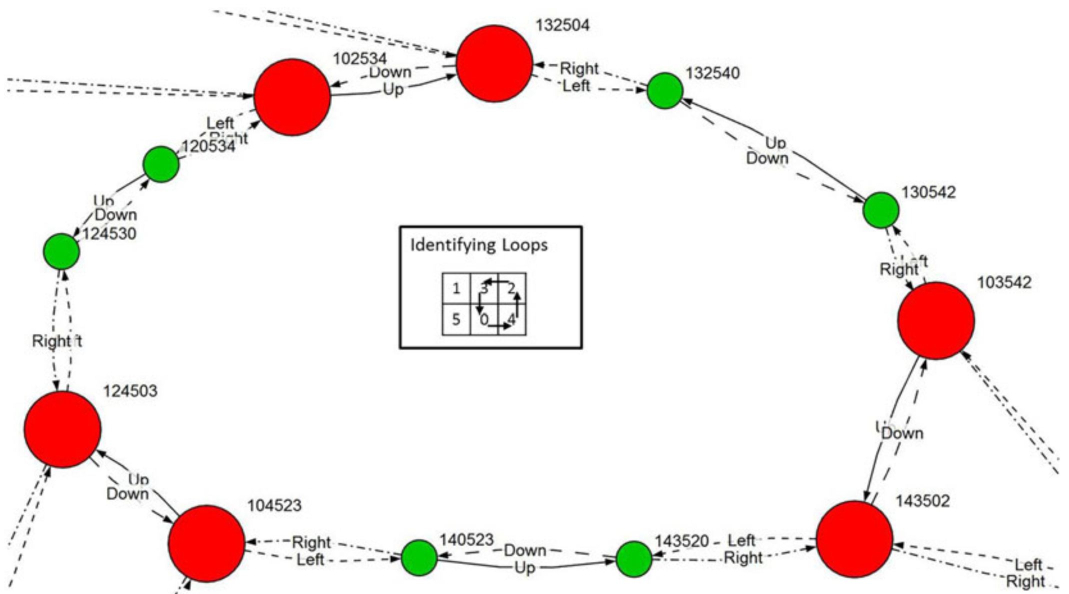


Figure 14. Loops are detected to reason about the rules and to avoid them during synthesis. Large, red nodes are of degree six and smaller, green nodes are of degree four.

graph (Figure 13) we can see relations between designs (states) expressed through rules (transitions) and identify designs that are reached via more rule applications than others. This is highlighted using the different colors and node sizes.

6.2.3. Detect do–undo rules (Property 1: do–undo)

Do–undo rules can easily be found, as for any rule to transform one design s into another design t in this case study there exists an undo rule to transform t to s. This lies in the nature of the problem, that for each tile that is slid into an empty position, the previous position of the tile becomes empty, allowing it to be slid back. As expected, two pairs of do–undo rules are identified, namely the pairs *Up* $\longleftrightarrow$ *Down* and *Left* $\longleftrightarrow$ *Right*.

6.2.4. Analyze alternative paths (Property 3: multiple paths) and loops (Property 2: loops)

Alternative paths, as well as loops can be detected (see Figure 14) representing what humans trying to solve the tile puzzle also easily experience, e.g., that sliding tiles in a squared region repeatedly will transform the puzzle to its initial state after a given number of moves. Figure 14 shows an example of a loop that rotates three tiles in counter clockwise direction. Understanding such loops can help to avoid the application of a sequence of rules that describes such a loop, as then the whole sequence is negligible. It can also visually be recognized that there might be a shorter and a longer sequence of rule applications to transform a design from one state to another.

Besides the manual interpretation of the transition graphs that represent the human's understanding of this simple problem well, the automated analyses are

https://doi.org/10.1017/dsj.2016.5 Published online by Cambridge University Press

1	2	5
4	8	7
3	0	6

Sequence of rule applications →

1	2	3
4	5	6
7	8	0

Figure 15. Example for the 8-tile puzzle.

also tested. Numerous alternative paths between designs are found and questions of reachability between any two designs are answered successfully. It is further demonstrated that the shortest paths can be found between any two designs.

6.3. Part 2: Applying knowledge on the large-scale problem

The 8-tile puzzle is used as a large-scale problem to analyze whether learning rule application sequences identified on a reduced problem, can help to solve larger-scale problems. An example puzzle is given in Figure 15.

The task is to find a sequence of rule applications transforming the given puzzle (left) to the desired puzzle (right) using the information learnt on the 5-tile puzzle. While the 5-tile puzzle with its $6!/2 = 360$ solvable states can be explored exhaustively, the 8-tile puzzle has a significantly larger search space with $9!/2 = 181{,}440$ solvable states making it harder, and for larger puzzles impossible, to explore exhaustively.

Humans often follow certain strategies to solve the tile puzzle. The most common is to start solving the puzzle from the top to the bottom. Doing this, the tiles for the first row are arranged first and as soon as each of them is in its correct position, they are not moved any more. Then the next row is considered. Having such strategies allows humans to focus on different aspects one after another and makes it easier to handle the complexity of a large-scale puzzle.

Similar to this, the 8-tile puzzle is decomposed into 5-tile puzzles and these are solved sequentially. Problem decomposition is a commonly used problem solving technique (Pólya 1957) and is frequently used in engineering design (Pahl & Beitz 1984). Various methods exist to decompose problems into sub-problems, e.g., target cascading for parametric problems (Kim *et al.* 2003), or agent-based approaches in optimization (Barbati, Bruno & Genovese 2012), where sub-problems are solved by different agents and then recombined. Problem decomposition is used in this research to demonstrate how beneficial rule sequences can be used to solve a design task.

6.3.1. Algorithm used to solve large-scale problem (includes Property 5: sequences and Property 6: shortest sequence)

The human-based strategy for solving the tile puzzle from the top is implemented in this case study using heuristics (see Appendix C for further details). Further, the 8-tile problem is subdivided into a series of 5-tile problems to solve. The smaller 5-tile puzzle is mapped into a region of the 8-tile puzzle and only tiles within the smaller 2×3 regions are changed. The remaining tiles stay untouched. Changing between 2×3 regions, e.g., first looking at the upper region, then at the lower region in the 3×3 puzzle, tiles can move through the whole 3×3 puzzle. A schematic overview of the algorithm used to solve the larger-scale problem is presented in Figure 16. The rationale behind this algorithm is that the

https://doi.org/10.1017/dsj.2016.5 Published online by Cambridge University Press

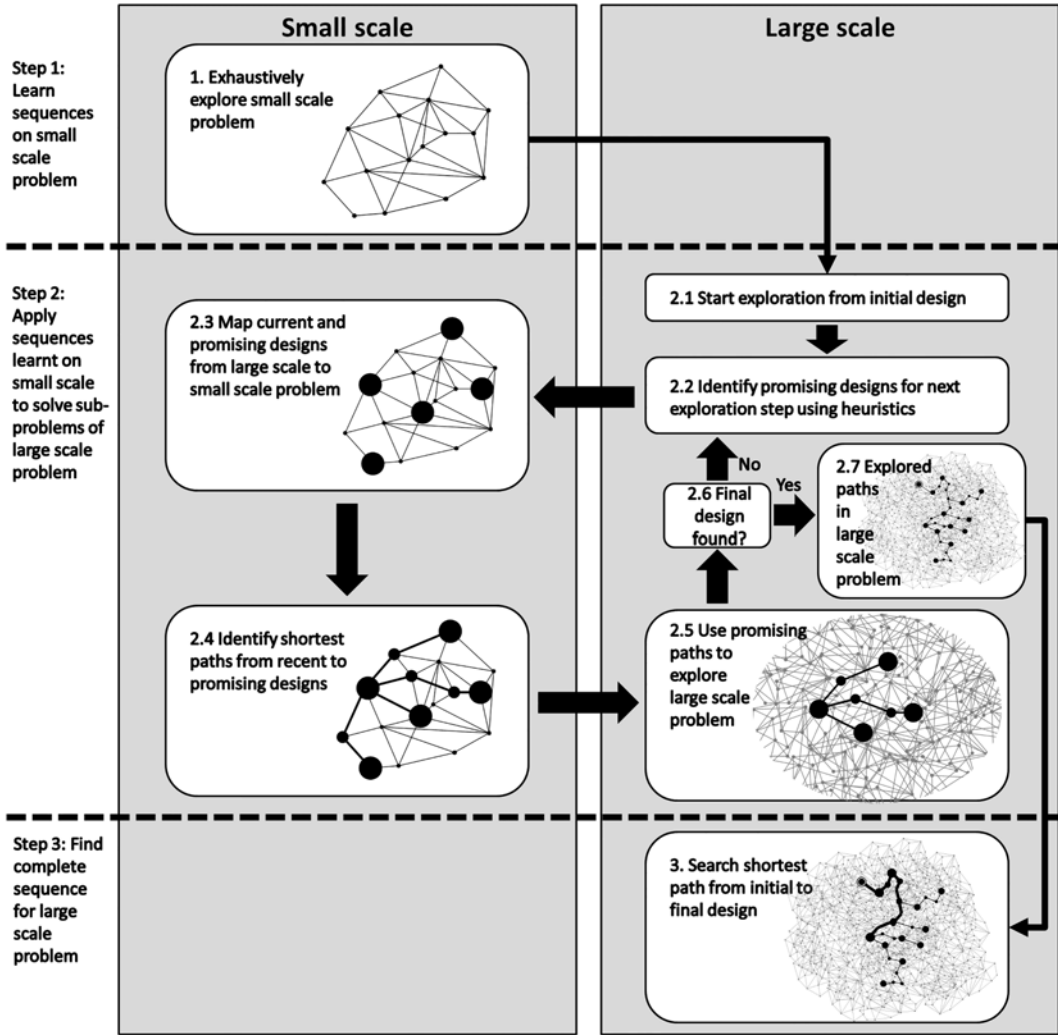


Figure 16. Schematic overview of the algorithm used to solve larger-scale problem based on beneficial rule sequences learnt on small scale.

information learnt on the small scale supports solving the large-scale problem by directing the search to explore only promising regions of the search space. Other algorithms can be used; however, the simple algorithm developed for this case study showed sufficient for demonstration purposes. It is described on a generic level in the following as it is generally applicable to decomposable problems for which heuristics are available to define promising designs. For the interested reader, the problem specific implementation for the sliding tile puzzle is given in Appendix D. It is based on two heuristics for changing between regions in the 8-tile puzzle and for identifying promising designs for the current iteration. Both heuristics are explained in more detail in Appendix C.

The algorithm is presented in Figure 16 and consists of three steps.

https://doi.org/10.1017/dsj.2016.5 Published online by Cambridge University Press

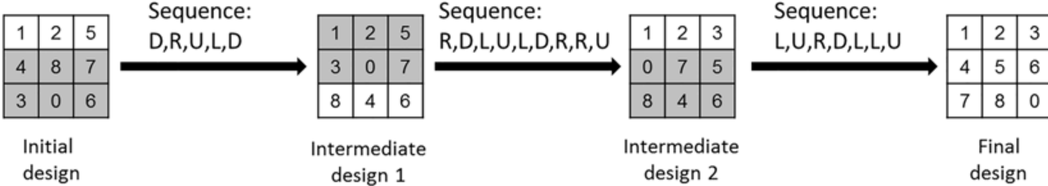


Figure 17. Sequence of rule applications to solve example 8-tile puzzle.

In *Step 1*, the search space of the small-scale problem is explored exhaustively. NBRAM is applied to the small-scale problem, i.e., the transition graph is generated and beneficial rule sequences are identified.

In *Step 2*, the search space of the large-scale problem is explored gradually. The learnt sequences from Step 1 are applied to explore parts of the search space of a larger-scale problem. The process is started from the initial design (Step 2.1). In Step 2.2, promising designs are identified. What is considered promising is problem dependent and heuristics, i.e., knowledge that is derived from experience, or rules of thumb can be used to determine if a design is promising. For the puzzle in Figure 15, e.g., a promising design would be one where tile '3' is in the first row such that the first row is '1', '2', '3', i.e., it fits with the final design. Once promising designs are identified, the currently explored design and the promising designs are mapped from the large-scale space to the small-scale space (Step 2.3). Through this mapping, the problem is reduced from the larger-scale to a smaller-scale problem for which rule sequences among designs are known. Using this information about the small-scale problem, sequences of rule applications, i.e., paths in the transition graph, can be found that reach the promising designs starting from the current design (Step 2.4). The found paths are then mapped back to the large-scale space, i.e., the rule sequences that successfully solve the small-scale problem are used to explore portions of the space of the large-scale problem (Step 2.5). This can be done for all of them or they can be filtered to consider only the most promising ones with respect to some criteria. Now that different paths to promising designs are explored, it is checked whether or not a final design is found (Step 2.6). If not, the search continues with Step 2.2 by selecting one of the explored designs and defining promising designs that should be reached from there. If a final design is found, this means that a sufficiently large portion of the large-scale space is explored (2.7) and the algorithm continues with Step 3.

In *Step 3*, the sequence of rule applications to solve the large-scale problem is found by searching the shortest path from the initial to the final design.

6.3.2. Results

The learnt rule sequences from the 5-tile puzzle are successfully applied to the 8-tile puzzle. For any solvable puzzle in the 8-tile space, a sequence of rule applications is found to transform the given puzzle into the final puzzle. For the initial design presented in Figure 15, for example, 21 moves are identified to transform the initial design into the final design. Figure 17 visualizes schematically, how the puzzle is solved. The first puzzle represents the initial design. The lower region is considered and therefore highlighted. Applying the rule sequence (D, R, U, L, D) generates the first intermediate design. Now the

https://doi.org/10.1017/dsj.2016.5 Published online by Cambridge University Press

upper region is considered and so forth until the final design is generated after 21 rule applications.

6.4. Key findings

NBRAM enables understanding the search space of the sliding tile puzzle visually, e.g., through the visualization of the two unconnected transition graphs of solvable and unsolvable designs. Further, it is shown that the understanding of the search space can be used to identify beneficial rule application sequences. In the second part of the case study, beneficial sequences are used to solve a larger-scale problem.

7. Discussion

The two case studies demonstrate how the NBRAM can be used for different problems. The NBRAM should preferably be used to analyze grammar rules before they are applied in the CDS process in combination with a search algorithm.

Both case studies use graph grammars. The presented method is, however, similarly applicable to other types of grammars, e.g., shape grammars. In the transition graph each state would represent a shape and each rule application would be represented as a transformation of the shape to generate a new shape. Emergent sub-shapes can be handled as different possible locations to apply a rule. One challenge, however, is defining what constitutes a unique design, i.e., shape similarity, if a set grammar approach to shape grammars is not used.

The gearbox case study shows how the transition graph can be used to analyze rule applications where the location of the rule application in the design influences the synthesis results. Results show how rules can be identified that are matched more often than intended, i.e., their LHSs are formulated too vaguely. Modifying the LHS formulation of the rules makes it possible to reduce the number of indifferent (from an engineering point of view) matches of these rules. While the generated designs remain the same, these changes can have a strong influence on the run time and required working memory when the rules are used in the CDS process. Therefore, these detailed analyses of the LHSs can not only help the human designer to debug and understand rule matches, but also to speed up the synthesis process when LHSs are formulated more specifically to the synthesis task.

In the sliding tile puzzle case study, the transition graph for a small-scale problem, the 5-tile puzzle, is generated and analyzed giving insights on the grammar as well as the problem itself. The 8-tile puzzle is used to demonstrate that rule sequences learnt for small-scale problems can be used to solve large-scale problems. For the given puzzle, a rule sequence of 21 rule applications was found, while more sophisticated algorithms might find a shorter sequence of rule applications. The difference can be explained with the sequential subdivision of the 8-tile puzzle in 5-tile regions that restrict rule applications to smaller regions. The aim of this part of the case study, however, is not to search for computer-competitive strategies to solve the sliding tile puzzle, but to demonstrate that beneficial rule sequences can be learnt and applied that is successfully shown also with a simple search algorithm. For the tile puzzle only one final state exists, but the presented algorithm can likewise be used for problems with several final states and would present the rule sequence to the first final state that is found.

https://doi.org/10.1017/dsj.2016.5 Published online by Cambridge University Press

For both case studies, the automated analysis of the synthesized transition graphs enables the designer to identify loops and alternative sequences of rules. The case studies demonstrate the potential of using visualizations and analysis of transition graphs to strengthen the human designers' understanding of developed grammar rules and the relations between designs and rule sequences. Depending on the problem at hand, it might be possible to explore the search space exhaustively for a smaller sub-problem, as in the tile puzzle case study, or for a limited number of rules in each applied rule sequence, as in the gearbox case study. For larger problems it is also possible to explore portions of the design space using stochastic search algorithms. Then, it is recommended to start the generation process repeatedly and from different initial designs to collect sufficient data and not let the choice of the initial design or the randomness of the search algorithm bias the results. The different designs are then combined to one or more transition graphs depending on whether or not the same designs are generated from different initial designs.

The algorithm that is proposed to solve the 8-tile puzzle is presented on a generic level in Figure 16. It is, however, only applicable to design problems, if they can be decomposed into smaller-scale problems. If this is not the case, beneficial rule sequences can still be learnt using NBRAM, but more sophisticated algorithms should be used to specify where to apply them.

For most real world engineering problems, the exact number of final designs is not known. Often, only some characteristics of final designs are known and an evaluation routine has to check whether these characteristics are present in an explored design and whether the design fulfills all constraints and can therefore be considered as a valid final design. The gearbox synthesis problem can be considered as such a real world problem. Finding beneficial rule application sequences can then be more complicated than the results of the sliding tile puzzle case study might suggest. This is especially the case for problems that cannot be decomposed into smaller-scale problems. Future research should address this issue and further investigate methods to automatically analyze the effect of the LHS application condition on the generated designs since in the current version of the method this is done through manual analysis of the visualization.

In this paper, seven properties for grammars are presented and it is shown how the NBRAM can be used to analyze these properties. Examples for further properties that can be considered in the future are the expressiveness of a grammar and the similarity of different grammars. The expressiveness property could indicate the breadth of ideas that can be described with a grammar. The similarity property could be used to identify similarities between different grammars based on the transition graphs they generate.

A modified sliding tile puzzle was used in research by Vale & Shea (2003) that is directed towards finding beneficial rule sequences. The method proposed in Vale & Shea (2003) learns performances of rule sequences from previous rule applications during the CDS process and uses machine learning (ML) techniques to decide on future rule applications. The focus is on accelerating the search process and learning during the CDS process. In this paper, by contrast, the focus is on supporting the human designer in developing grammars. Future research could, however, combine the presented approach with methods like the one presented in Vale & Shea (2003). Beneficial and counterproductive rule sequences could be learnt using the NBRAM and provided to the ML-based search as initial

https://doi.org/10.1017/dsj.2016.5 Published online by Cambridge University Press

information. Similarly, the ML-based search could provide visualizations of the learnt information to the human designer to further increase their understanding of the grammar rules and the designs they generate.

8. Conclusion

The method presented in this paper can support designers in developing and applying grammar rules. The major contribution of the presented method is that it represents a novel approach to analyze grammar rules by combining a graph representation (transition graph) of generated designs and rule applications with network analysis algorithms and interactive visualizations. Exhaustively exploring small portions of the search space to collect data and generate transition graphs to visualize relations between different designs and sequences of rule applications allows for a systematic rule analysis and can give human designers useful feedback about the grammar they developed. Designs, for which the same rule can lead to different designs depending on the rule's application location, can be identified and based on the resulting designs, the effect of the application location can be analyzed. This information can, e.g., be used to improve the LHS of a rule to increase or decrease the number of LHS matches. By manually analyzing the transition graph or computationally through graph search algorithms, loops in the transition graph can be identified. In addition, efficient rule application sequences can be identified through shortest path search in the transition graph. The two presented case studies demonstrate the potential of transition graph analysis to support the development and application of engineering design grammars.

Appendix A. Algorithm for exhaustive generation of designs

The pseudo-code for the algorithm for exhaustive generation of designs is given in Figure 18. The algorithm requires an initial design (*initialDesign*), the rule set (*ruleSet*) and a maximum number of rule applications (*maximumSequenceLength)* as input. Outputs are a list of explored designs (*graphs*) and a list (*transformations*) storing each applied rule, the design the rule is applied to, and the resulting design. Starting from an *initial design*, all rules in the rule set (*ruleSet*) are explored exhaustively until a given number of rule applications (*maximumSequenceLength)* from the *initial design*. The algorithm is implemented as a BFS with isomorphism check to not explore already found topologies repeatedly. It explores the design space in levels (*level*) until the maximum number of rule applications (line 2). For each *level*, the algorithm iterates through all rules in the rule set (line 3). The selected rule r is applied to all graphs from the previous level (*previous*, line 4). For each graph g, all matches m of the current rule r are identified (line 6) and for each match m a new graph (*newGraph*) is generated by applying rule r on graph g at match m (line 8). The generated graph (*newGraph*) is evaluated (line 9) and if it constitutes a topology that is not found yet in one of the previous levels, (*newTopology* returns *true* in line 10), then the design is stored in the list of designs to be expanded in the next level (*next*, line 11). Each generated graph (*newGraph*) is stored in the list *graphs* (line 13) and for each rule application, the rule and the previous and resulting graph are stored in *transformations* (line 14). Once all rules in the *ruleSet* are applied for the current *level*, the graphs from list *previous* are

https://doi.org/10.1017/dsj.2016.5 Published online by Cambridge University Press

```
Algorithm: Exhaustive Generation
Input: maximumSequenceLength, ruleSet, initialDesign
Output: graphs, transformations
1:  previous ← initialDesign;
2:  for level = 1 to maximumSequenceLength do
3:      for all rule r in ruleSet do
4:          for all graph g in previous do
5:              load(g);
6:              identify matches of r on g;
7:              for all match m in matches do
8:                  newGraph ← apply(r,g,m);
9:                  evaluate(newGraph);
10:                 if newTopology(newGraph) then
11:                     next ← add(newGraph);
12:                 end if
13:                 graphs ← add(nextGraph);
14:                 transformations ← add(g,r,nextGraph);
15:             end for
16:         end for
17:     end for
18:     previous ← next;
19:     clear(next);
20: end for
```

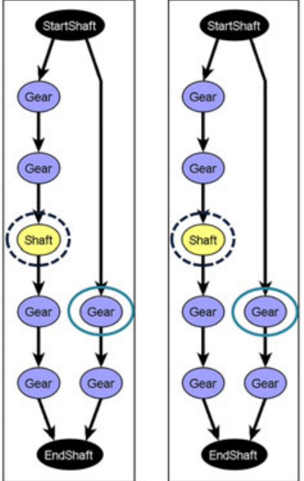
Figure 18. Pseudo-code for exhaustive generation.

replaced with those from list *next* (line 18), list *next* is cleared (line 19) and the algorithm continues with the next *level* (line 3).

Appendix B. Analysis of LHSs of rules 2 and 4 for the gearbox case study

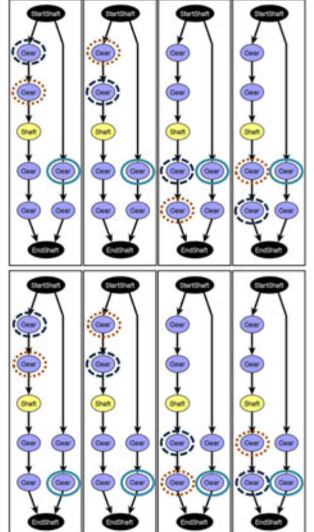
In Figure 19, all matches of rule 2 (*delete a shaft*) on the design with UID 2 (see Figure 8) are shown (left). The rule formulation in GrGen is provided on the right of Figure 19. LHS and RHS of the rule are indicated using curly braces. The LHS of the rule consists of a node of type *shaft* (*S0*) and a node of type *gear* (*G0*). In addition, a negative application condition is defined using the *negative* statement. Negative application conditions are used to define patterns that, when present in the graph, prohibit the rule's application. In case of rule 2, this is when the *gear G0* is reachable from *shaft S0*, or when *shaft S0* can be reached from *gear G0*. The rationale behind this condition is that when no *gear G0* exists that is not reachable by *shaft S0* or can be reached from *S0*, then there exist only power flow paths in the gearbox that involve *S0*. The rule is then not applied since removing *S0* would result in an invalid design, i.e., a design where input shaft and output shaft are not connected. The functions *reachableOutgoing*() and *reachableIncoming*() are used to identify the sets of nodes that can be reached or are reachable from *shaft S0*. The RHS of the rule is defined within the *modify* statement. First, the *shaft S0* is deleted. Then, two rules are applied that remove dangling nodes in the graph. The first is called *deleteUnusedGears* and removes all gears that were connected to *shaft S0*, i.e., those gears that are mounted on *shaft S0*. The second is called *deleteUnusedGearPairsAndShafts*. It searches through the remaining graph to identify gear pairs and shafts that are not required any more since they are not connected to both input and output shaft of the gearbox to form a valid power flow path.

https://doi.org/10.1017/dsj.2016.5 Published online by Cambridge University Press




```
rule DeleteAShaft
{
    S0:Shaft;
    G0:Gear;
    negative
    {
        if{(G0 in reachableOutgoing(S0,Edge,Element)) ||
        (G0 in reachableIncoming(S0,Edge,Element));}
    }
    modify
    {
        delete(S0);
        exec(DeleteUnusedGears);
        exec(DeleteUnusedGearPairsAndShafts);
    }
}
```

Figure 19. Rule 2 (*delete a shaft*) formulated using a negative application condition.



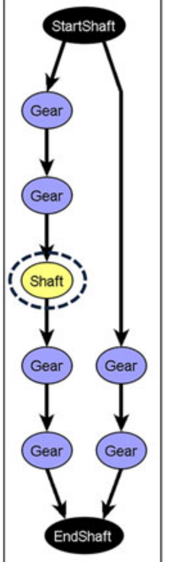
```
rule DeleteAGearPair
{
    G0:Gear ?--? G1:Gear;
    G2:Gear;
    negative
    {
        if{(G2 in reachableOutgoing(G1,Edge,Element)) ||
        (G2 in reachableIncoming(G1,Edge,Element));}
    }
    modify
    {
        delete(G0);
        delete(G1);
        exec(DeleteUnusedGearPairsAndShafts);
    }
}
```

Figure 20. Rule 4 (*delete a gear pair*) formulated using a negative application condition.

On the left in Figure 19, the two matches of rule 2 on the graph with UID 2 are shown. The *shaft S0* is matched to the same node, since it is the only node of type *shaft* in the graph. The two gears on the right power flow path through the gearbox can be matched to *gear G0* such that the negative application condition is not fulfilled and the rule can be applied to remove the *shaft S0* and the then dangling gear nodes to transform the design into that with UID 1.

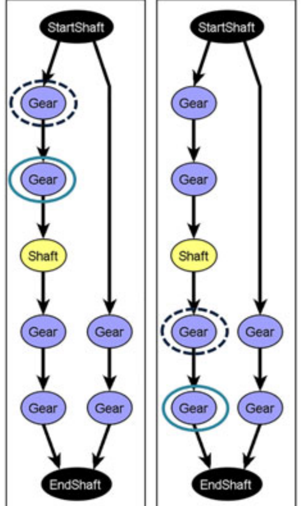
Comparing the graphs with UID 2 and UID 1 (see Figure 8), the rule designer might expect that two matches of rule 4 (*delete a gear pair*) are found to delete either the first or the second gear pair of the left power flow path. The transition graph, however, shows that rule 4 is applied to eight different matches on the graph with UID 2 to transform it to the graph with UID 1. Figure 20 shows the rule

https://doi.org/10.1017/dsj.2016.5 Published online by Cambridge University Press




```
rule DeleteAShaft
{
    S0:Shaft;
    independent
    {
        G0:Gear;
        if{!(G0 in reachableOutgoing(S0,Edge,Element)) &&
        !(G0 in reachableIncoming(S0,Edge,Element));}
    }
    modify
    {
        delete(S0);
        exec(DeleteUnusedGears);
        exec(DeleteUnusedGearPairsAndShafts);
    }
}
```

Figure 21. Rule 2 (*delete a shaft*) formulated using a positive application condition.



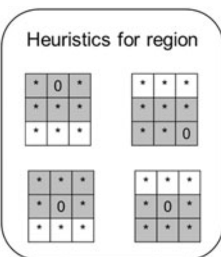

```
rule DeleteAGearPair
{
    G0:Gear ?-? G1:Gear;
    independent
    {
        G0;
        G1;
        G2:Gear;
        if{!(G2 in reachableOutgoing(G1,Edge,Element)) &&
        !(G2 in reachableIncoming(G1,Edge,Element));}
    }
    modify
    {
        delete(G0);
        delete(G1);
        exec(DeleteUnusedGearPairsAndShafts);
    }
}
```

Figure 22. Rule 4 (*delete a gear pair*) formulated using a positive application condition.

formulation in GrGen for rule 4. The expression '*G0:Gear ?–? G1:Gear*' indicates that two nodes of type *gear* have to exist and have to be connected by an edge. '?–?' indicates that the direction of the edge connecting the two nodes is not relevant. The remainder of rule 4 is structured similarly to rule 2 and therefore not described here in detail. On the left of Figure 20, the eight matches of rule 4 on the design with UID 2 are shown that transform it to the design with UID 1.

When the human designer understands that rules are not matched as intended, e.g., more often than necessary, the LHS of the rules can be formulated more carefully. Figures 21 and 22 show how this can be done for rules 2 and 4,

https://doi.org/10.1017/dsj.2016.5 Published online by Cambridge University Press



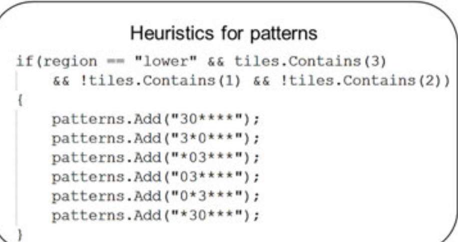


Figure 23. Heuristics for defining regions of sub-problems (left) and search patterns (right).

respectively. The LHSs are reduced to only contain the nodes that have to be removed. Instead of negative application conditions, positive application conditions are used (*independent* statement). The shaft (rule 2) or gear pair (rule 4) are removed from the graph when there is another gear that is neither reachable from the matched node(s), nor can it reach the matched node(s). In addition, the direction of the edge between gear *G0* and gear *G1* in rule 4 is specified to be directed from *G0* to *G1* ($G0 \rightarrow G1$) to reduce ambiguity.

Appendix C. Heuristics for the sliding tile puzzle

Two heuristics are used to steer the search space exploration for the sliding tile puzzle using the algorithm described in Appendix D. They are described in the following.

1. *Heuristic for region.*

Figure 23 (left) presents heuristics for which region to consider in the next iteration. The gray regions define, where the next modifications are performed. When the empty tile (indicated with '0' in Figure 23) is in the top or bottom row, the top or bottom region are selected. When it is in the middle row, further heuristics can be applied or one region can be selected randomly. Given, e.g., the initial puzzle in Figure 15, the lower region is selected based on the heuristic for the region.

2. *Heuristic for patterns.*

The common human strategy to solve the sliding tile puzzle from the top is implemented by defining search patterns. The patterns are described as six character strings and define desirable tile positions giving the tile numbers and '*' as a wildcard symbol. In Figure 23 (right) one example for a heuristic for patterns is given. It assumes that the lower region of the tile puzzle is considered and a tile with number '3' is present in this lower region but no tiles with numbers '1' or '2' are present. The heuristic in Figure 23 (right) defines patterns to identify all puzzles with tile '3' and the empty tile in the upper row of the selected region. This represents one part of the strategy commonly used by humans to move tiles that belong to the upper row (tiles '1', '2', '3') into the middle row and then, in a next step when the upper region is considered, into the upper row, e.g., to replace the '5' in the initial design in Figure 15.

https://doi.org/10.1017/dsj.2016.5 Published online by Cambridge University Press

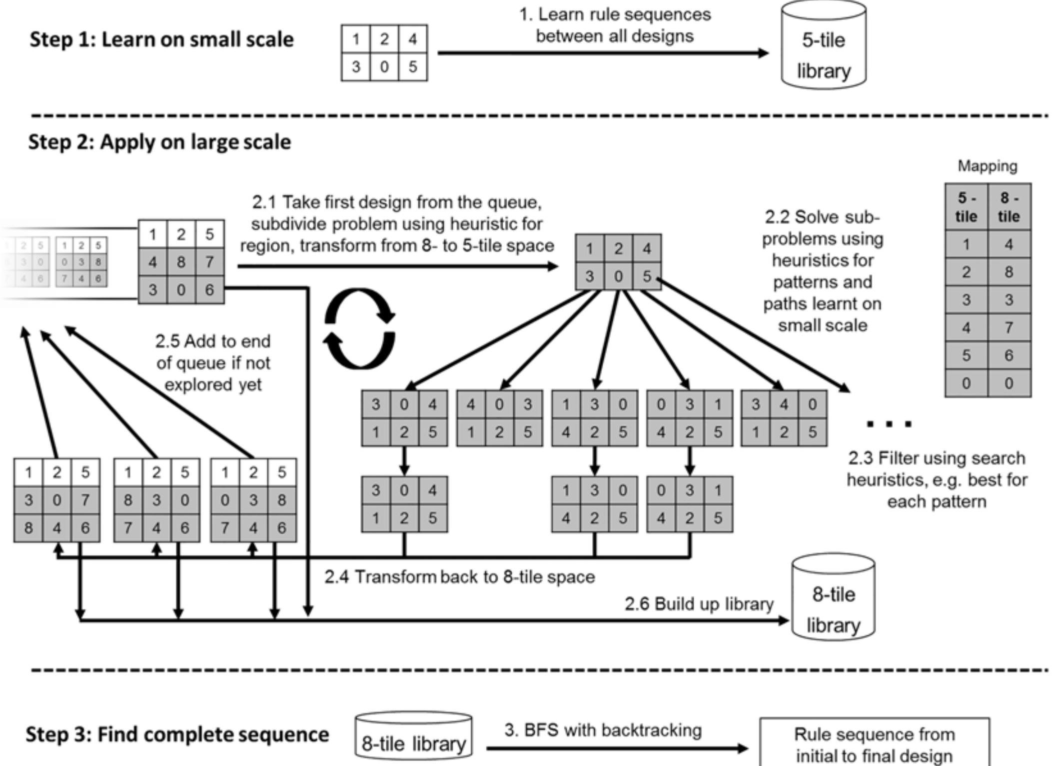


Figure 24. Process of gaining information on small scale (Step 1) and applying it to explore (Step 2) and solve larger-scale problems (Step 3).

Appendix D. Details on the algorithm used to solve the 8-tile puzzle based on rule sequences learnt on the 5-tile puzzle

The process to solve the 8-tile puzzle is presented in Figure 24. In Step 1, the transition graph is explored exhaustively for the small-scale puzzle and stored in the 5-tile library. This means all transformations between any 5-tile puzzles are known. In Step 2, the design space of the 8-tile puzzle is explored partially using a BFS approach. It is implemented using a queue. In each iteration the first element in the queue is considered as the current design (Step 2.1). The 8-tile puzzle is subdivided using the heuristic for the region (see Appendix C). The considered region is highlighted in gray in Figure 24. The tiles are then mapped from the 8-tile space to the 5-tile space. In this transformation, the tile numbers are mapped from the numbers {0, 1, 2, 3, 4, 5, 6, 7, 8} to the number {0, 1, 2, 3, 4, 5}. The empty tile (number 0) remains in both spaces, the remaining numbers have to be mapped such that the mapped design constitutes a solvable design in the 5-tile space. The mapping for the example iteration shown in Figure 24 is, e.g., (5-tile space – 8-tile space): (1–4), (2–8), (3–3), (4–7), (5–6), (0–0). The same mapping is applied to the search patterns for promising designs that are defined using the pattern heuristic. In Step 2.2, for each pattern, all designs that match it are identified in

https://doi.org/10.1017/dsj.2016.5 Published online by Cambridge University Press

the 5-tile library. In Figure 24, five example designs matching the heuristics for patterns (see Appendix C) are shown (30****, *03***, *30***, 03****, 3*0***), but many more are possible. The shortest paths, i.e., the rule sequences with the least rule applications, to these designs are identified using the 5-tile library. To avoid exploring an unnecessary large space of the 8-tile puzzle search space, for each pattern heuristic only the generated designs with the shortest rule sequences are considered further (Step 2.3). The considered designs are transformed back into the 8-tile space and are integrated into the larger-scale puzzle, i.e., the changed (gray) and unchanged (white) regions are combined (Step 2.4). For each explored design, a test is performed whether it was already found previously. If not, it is added to the end of the queue (Step 2.5). At the end of the iteration, the rule sequences leading from the first (currently considered) design in the queue to the designs that are now added to the queue are stored in the 8-tile library (Step 2.6). This means that the 8-tile library builds up information about the transition graph of the 8-tile space with the nodes representing puzzles and the edges representing rule sequences. This process is continued until the final design is found. As soon as it is found, the complete sequence to solve the 8-tile puzzle is generated (Step 3) by analyzing the 8-tile library, in which the generated designs in the 8-tile space and the sequences of rule applications to transform them are stored. As in Step 1, it is found by searching the shortest path using a BFS with backtracking with the only difference that instead of single rules, the transformations are sequences of rules.

References

Aho, A. V., **Lam, M. S.**, **Sethi, R.** & **Ullman, J. D.** 2006 *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc.

Barbati, M., **Bruno, G.** & **Genovese, A.** 2012 Applications of agent-based models for optimization problems: a literature review. *Expert Systems with Applications* **39**, 6020–6028.

Brown, K. N. 1997 Grammatical design. *IEEE Expert/Intelligent Systems and Their Applications* **12** (2), 27–33.

Cagan, J. 2001 Engineering shape grammars. In *Formal Engineering Design Synthesis* **(ed. E. K. Antonsson & J. Cagan)**. Cambridge University Press.

Cagan, J., **Campbell, M. I.**, **Finger, S.** & **Tomiyama, T.** 2005 A framework for computational design synthesis: model and applications. *Journal of Computing and Information Science in Engineering* **5**, 171–181.

Cash, P., **Stanković, T.** & **Štorga, M.** 2014 Using visual information analysis to explore complex patterns in the activity of designers. *Design Studies* **35**, 1–28.

Chakrabarti, A., **Shea, K.**, **Stone, R.**, **Cagan, J.**, **Campbell, M.**, **Hernandez, N. V.** & **Wood, K. L.** 2011 Computer-based design synthesis research: an overview. *Journal of Computing and Information Science in Engineering* **11**, 021003,1–10.

Chomsky, N. 1957 *Syntactic Structures*. Mouton de Gruyter.

Fu, Z., **Depennington, A.** & **Saia, A.** 1993 A graph grammar approach to feature representation and transformation. *International Journal of Computer Integrated Manufacturing* **6**, 137–151.

Geiß, R., **Batz, G.**, **Grund, D.**, **Hack, S.** & **Szalkowski, A.** 2006 GrGen: a fast spo-based graph rewriting tool. In *Graph Transformations* **(ed. A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro & G. Rozenberg)**. Springer.

Gips, J. 1999. Computer implementation of shape grammars, *NFS/MIT Workshop on Shape Computation*.

https://doi.org/10.1017/dsj.2016.5 Published online by Cambridge University Press

Gmeiner, T. & **Shea, K.** 2013 A spatial grammar for the computational design synthesis of vise jaws. In *International Design Engineering Technical Conferences and Computers and Information in Engineering Conference (ASME IDETC), Portland, OR, USA*.

Helms, B., **Eben, K.**, **Shea, K.** & **Lindemann, U.** 2009 Graph grammars – a formal method for dynamic structure transformation. In *11th International DSM Conference, Greenville, SC, USA*.

Hoisl, F. & **Shea, K.** 2013 Three-dimensional labels: a unified approach to labels for a general spatial grammar interpreter. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing* **27**, 359–375.

Hoisl, F. R. 2012. Visual, interactive 3D spatial grammars in CAD for computational design synthesis. Dissertation, Technische Universität München.

Johnson, W. W. & **Story, W. E.** 1879 Notes on the '15' puzzle. *American Journal of Mathematics* **2**, 397–404.

Kim, H. M., **Michelena, N. F.**, **Papalambros, P. Y.** & **Jiang, T.** 2003 Target cascading in optimal system design. *Journal of Mechanical Design* **125**, 474–480.

Königseder, C. & **Shea, K.** 2015 Comparing strategies for topologic and parametric rule application in automated computational design synthesis. *Journal of Mechanical Design* **138** (1), 011102-011101–011102-011112; doi:10.1115/1.4031714.

Kroll, M., **Beck, M.**, **Geiß, R.**, **Hack, S.** & **Leiß, P.** yComp [Online]. Available: http://www.info.uni-karlsruhe.de/software.php/id=6 (Accessed July 17, 2015).

Kumar, M., **Campbell, M. I.**, **Königseder, C.** & **Shea, K.** 2012 Rule based stochastic tree search. In *Design Computing and Cognition '12* **(ed. J. S. Gero)**. Springer.

Li, X. & **Schmidt, L.** 2004 Grammar-based designer assistance tool for epicyclic gear trains. *Journal of Mechanical Design* **126**, 895–902.

Lin, Y. S., **Shea, K.**, **Johnson, A.**, **Coultate, J.** & **Pears, J.** 2010 A method and software tool for automated gearbox synthesis. In *International Design Engineering Technical Conferences and Computers and Information in Engineering Conference (ASME IDETC), San Diego, CA, USA*.

McKay, A., **Chase, S.**, **Shea, K.** & **Chau, H. H.** 2012*a* Spatial grammar implementation: from theory to useable software. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing* **26**, 143–159.

McKay, A., **Chase, S.**, **Shea, K.** & **Chau, H. H.** 2012*b* Spatial grammar implementation: from theory to useable software. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing* **26**, 143–159.

Mullins, S. & **Rinderle, J.** 1991 Grammatical approaches to engineering design, part I: an introduction and commentary. *Research in Engineering Design* **2**, 121–135.

Pahl, G. & **Beitz, W.** 1984 *Engineering Design*. Springer.

Pólya, G. 1957 *How to Solve it a New Aspect of Mathematical Method*. Doubleday.

Pomrehn, L. P. & **Papalambros, P. Y.** 1995 Discrete optimal design formulations with-application to gear train design. *Journal of Mechanical Design* **117**, 419–424.

Rinderle, J. 1991 Grammatical approaches to engineering design, part II: melding configuration and parametric design using attribute grammars. *Research in Engineering Design* **2**, 137–146.

Schmidt, L. C. & **Cagan, J.** 1997 GGREADA: a graph grammar-based machine design algorithm. *Research in Engineering Design* **9**, 195–213.

Schmidt, L. C., **Shetty, H.** & **Chase, S. C.** 2000 A graph grammar approach for structure synthesis of mechanisms. *Journal of Mechanical Design* **122**, 371–376.

Slocum, J. & **Sonneveld, D.** 2006 *The 15 Puzzle: How It Drove the World Crazy. The Puzzle that Started the Craze of 1880. How America's Greatest Puzzle Designer, Sam Loyd, Fooled Everyone for 115 Years*. Slocum Puzzle Foundation.

Starling, A. C. 2004. Performance-based computational synthesis of parametric mechanical systems. Dissertation, University of Cambridge.

https://doi.org/10.1017/dsj.2016.5 Published online by Cambridge University Press

Starling, A. C. & **Shea, K.** 2005 A parallel grammar for simulation-driven mechanical design synthesis. In *International Design Engineering Technical Conferences and Computers and Information in Engineering Conference (ASME IDETC), Long Beach, CA, USA*.

Stiny, G. 1977 Ice-ray: a note on the generation of Chinese lattice designs. *Environment and Planning B: Planning and Design* **4**, 89–98.

Stiny, G. & **Gips, J.** 1972 Shape grammars and the generative specification of painting and sculpture. In *Proceedings of IFIP Congress 1971, 1972*. North Holland Publishing Co.

Stiny, G. & **Mitchell, W. J.** 1978 The Palladian grammar. *Environment and Planning* B **5**, 5–18.

Swantner, A. & **Campbell, M. I.** 2012 Topological and parametric optimization of gear trains. *Engineering Optimization* **44**, 1351–1368.

Vale, C. A. W. & **Shea, K.** 2003 A machine learning-based approach to accelerating computational design synthesis. In *International Conference on Engineering Design (ICED), Stockholm, Sweden*.

https://doi.org/10.1017/dsj.2016.5 Published online by Cambridge University Press