# IMPROVING GENERATIVE GRAMMAR DEVELOPMENT AND APPLICATION THROUGH NETWORK ANALYSIS TECHNIQUES

**Königseder, Corinna; Stanković, Tino; Shea, Kristina**
ETH Zurich, Switzerland

## Abstract

Design grammars have been successfully applied in numerous engineering disciplines, however, the lack of support for grammar development is seen as one of the major drawbacks of grammatical approaches in Computational Design Synthesis (CDS). In this paper, a method is presented that supports the development and application of design grammars. Concepts from compiler design and transition graphs are used to help designers understand developed grammars in depth. The grammar designer is given feedback on a) the rules, e.g., if there are redundant or do-undo rules, and b) rule application sequences, e.g., which sequences should be preferred or avoided. This feedback can be used to a) improve the grammar, and b) apply it more efficiently. The case study demonstrates how the method is used to analyze a grammar for a sliding tile puzzle. Knowledge learnt on small scale was then successfully applied to solve a larger scale problem. The results show the feasibility of the method and its generality is discussed.

**Keywords**: Computational design synthesis, Conceptual design, Generative Grammar, Network analysis, Visualisation

**Contact**:
Corinna Königseder
Engineering Design and Computing Laboratory, ETH Zurich
Department for Manufacturing and Process Engineering
Switzerland
ck@ethz.ch

Please cite this paper as:
Surnames, Initials: *Title of paper*. In: Proceedings of the 20th International Conference on Engineering Design (ICED15), Vol. nn: Title of Volume, Milan, Italy, 27.-30.07.2015

# 1 INTRODUCTION

Grammatical approaches have been successfully applied in numerous engineering design disciplines (Chakrabarti et al., 2011), e.g. in electrical engineering, architecture and mechanical engineering as well as in natural language processing, e.g. for automated language translation or speech recognition. These different areas, all emerging from the formal study of grammars, evolved in different directions. In the area of compiler design, grammars are designed formally to automatically translate implemented source code from a higher programming language into machine code, and a whole research area of grammar engineering has evolved dealing with the development of grammars. In architecture, VLSI design and engineering, grammars are often used to develop product concepts in the early stages of the design process and are often formulated less mathematically. In some areas, grammars are mainly used as pencil and paper grammars, i.e. they do not require computational implementations. Engineering design grammars are often developed for very specific use cases and no commonly accepted criteria for "good grammars" have been specified by the engineering design community.

While with formal grammars, as in compiler design, many 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 popular Computational Design Synthesis (CDS) task. Research has been carried out by several researchers (Li and Schmidt, 2004, Lin et al., 2010) and different grammars as well as algorithms to guide the synthesis process exist. Most researchers develop 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 which grammar or which particular rules are preferable for gearbox synthesis and why. Further, none of the publications investigates in depth how the grammar explores the space. This makes it difficult to fully understand the importance and influence of single grammar rules on the synthesis process. Several researchers mention the need for more support in the development of engineering design grammars (Gips, 1999, McKay et al., 2012) and the lack of support for grammar design is still seen as one of the major drawbacks of grammatical design (Chakrabarti et al., 2011).

In a recent approach a systematic grammar rule analysis method (GRAM) was developed to support the development of grammars for CDS (Königseder and Shea, 2014). GRAM supports the analysis of single rules, however, extensive information on how the rules change individual designs is lacking as well as detailed information on how sequences of rule applications explore the design space. Providing more support to 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, the authors expect major improvements in the quality of solutions produced from design grammars. Such analyses can help designers understand new and existing grammars in depth and allows them to make more informed decisions on reusing or developing engineering design grammar rules.

The authors support grammar development with a novel approach that combines concepts developed for compiler design with formal grammars used for CDS to combine the advantages of both, i.e. the representation of states and transitions from compiler design with the captured engineering knowledge used in engineering design grammars. The presented research analyzes the potential of using concepts from compiler design to support grammar development and application through giving feedback on how a developed grammar explores the design space. There are two aspects to this, which both are addressed. First, the grammar designer is given feedback on the rules, e.g. if there are redundant or do-undo rules. Second, the application of the grammar can be analyzed in more detail to identify preferable rule application sequences or patterns in rule sequences that should be preferred or avoided. With this information, engineering designers are given a means to more efficiently design and apply grammar rules for CDS. In this paper, first the method is presented. Next, the case study focuses on the second aspect, i.e. on understanding rule application sequences through analyzing small scale problems and then applying the learnt knowledge to solve larger scale problems.

The paper is structured as follows. In Section 2, an overview of CDS using grammars is presented, as well as basics on compiler design. Section 3 describes the method used in this paper to analyze engineering design grammars using transition graphs followed by the implementation details (Section 4). In Section 5 the case study, a sliding tile puzzle, is presented. Results on analyzing the tile puzzle

and applying the learnt knowledge to a larger scale tile puzzle are then presented (Section 6) and discussed (Section 7). The paper concludes with summarizing the key achievements and discussing future research directions to further improve engineering design grammar development.

## 2 BACKGROUND

In this paper, the terminology for the CDS process is used as defined in Cagan et al. (2005). In the 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 grammar rule is selected and applied to the current design transforming it into 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, either to accept or reject the new alternative. The synthesis process is continued until either no further rule applications are possible or it is stopped by a stopping criteria in the search method. In grammatical approaches to CDS, designers develop a grammar to represent a desired design language. It consists of a vocabulary, usually describing design components or subsystems, as well as a set of grammar rules. These rules describe design transformations, i.e. LHS → RHS, that are defined by a left-hand-side (LHS), i.e. where the rule can be applied in a design, and a right-hand-side (RHS) defining the design transformation. Using graph grammars for design synthesis, each design can be described using a graph representation consisting of nodes and edges. For example, nodes can represent components while edges describe functional or spatial relations between the nodes. (Gips and Stiny, 1980)

The success of CDS methods is dependent on various aspects such as an appropriate representation of the design problem, meaningful and quantifiable design evaluation, a suitable algorithm to guide the search and reasonable strategies to select rules to generate design alternatives. Several approaches to identify preferable search strategies and to learn meaningful sequences of rules (Vale and Shea, 2003) have been developed. What is unique about the presented research is the way it supports multiple phases of the CDS process, namely the representation and the guidance step, in one method and that unlike existing methods, it collects knowledge about the problem through rule analysis before the actual CDS search process is started. The rule development is supported through an increased understanding of how each individual rule transforms a given design and the synthesis process is supported through prior knowledge about meaningful rule sequences to solve sub-problems. Problem decomposition is a commonly used problem solving technique in engineering design (Pahl and 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 et al., 2012), where sub-problems are solved by different agents and then recombined. The case study shows that when it is possible to decompose a larger scale problem into sub-problems, and learn knowledge about sub-problems exhaustively, then the presented method can be used to solve larger scale problems using the knowledge learnt on small scale.

Two concepts from compiler design are adapted in the presented research: a) state representations as used to describe finite automata (FA) and transition graphs, and b) data flow analysis techniques as used in intermediate code optimization through analyzing data flows in graph representations. FA are recognizers that either accept or reject a given input string. Each FA 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) FAs 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)

Similar to an automaton that accepts 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 on the initial design. Each modified design would then represent a state and each grammar rule would represent a symbol *a* transforming state *s* to state *t*. Analyzing such a transition graph using techniques from data flow analysis, the grammar and influences of each rule application can be understood in detail.

## 3 NETWORK GENERATION AND ANALYSIS METHOD

The method is based on analyzing the designs that are generated during CDS and the rules that are used to do so. Figure 1 illustrates the three main steps. In the first step, 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 Depth-First Search (DFS), i.e. expand first one rule sequence, or Breadth-First Search (BFS), i.e. from one 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 in this step 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 1 (left) represents example representations of explored design spaces when using tree-based search methods. Each node in the graphs represents one design and each edge between the nodes represents the rule that was applied to transform a design into another one.
In most algorithms, the same designs can be 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 decrease the search space size. To do so, in the second step of the method (Figure 1, right), all generated designs are analyzed to identify unique and repeated designs. Uniqueness is a property that is problem dependent and can, e.g., be unique topologies in a structural design problem or designs with the same 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. 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 rule that was 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, each node represents a unique design, or state, and each edge represents a transition from a source design to a target design.

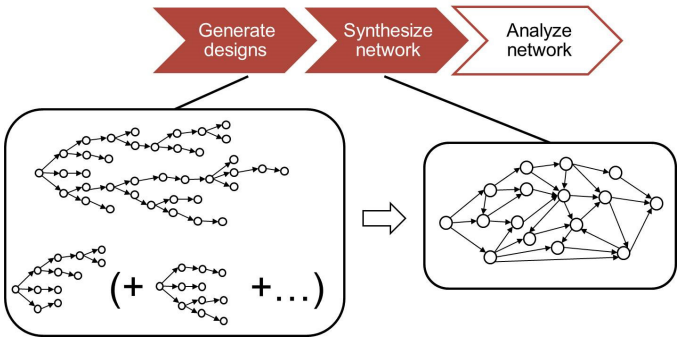


*Figure 1. Steps 1 and 2: Network generation through generation of designs (left) and synthesis of designs to one network representation (right).*

The transition graph can be analyzed to gain knowledge about a) the rules themselves, and b) the rule application sequences. In the third step of the method (see Figure 2), different graph analyses are performed. The designer is additionally given two methods to access the information. The network is represented a) visually for manual exploration of the generated designs and their relations, and b) results from automated network analyses investigate the following aspects:

1. Do-undo rule pairs are identified, i.e. pairs of rules where each rule un-does what the other did. A simple example of such a rule pair is two rules of which one adds a component and the other one deletes it. For the selection of an appropriate search algorithm, this information can be important because a repeated application of do-undo rules might get the synthesis process stuck 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 guidance algorithm accordingly.
2. Loops in the transition graph can be identified. Similar to do-undo rules, a loop of rules is a sequence of rules that, when applied to a given design, generate exactly this start design.

Avoiding such loops in the synthesis process, either by reformulating the grammar or through using this information in a search algorithm, can allow for a faster design space exploration.

3. Alternative rule sequences are identified, i.e. sequences of rules that transform a given design *s* to a design *t* via different paths in the transformation network. If such alternative paths exist, the designer can reason about the alternative paths and consider, e.g. if the rule set can be reduced by deleting or combining rules.

Additionally, interactive tools to study the transition graph allow to answer the following questions:

4. Is it possible to reach design *t* from design *s*? This allows, e.g., to analyze if design *t* can be synthesized starting the synthesis process from design *s*.
5. 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 reason about improving the grammar rules and help learn meaningful sequences.
6. What is the shortest rule sequence to transform a design *s* into a design *t*? Learning shortest rule sequences to transform one design into another can help to speed up the synthesis process.

In the remainder of this paper the authors give details on the implementation and show how the method can be used to learn rule application strategies from the transition graph on a small scale problem and use this learnt knowledge to tackle larger scale problems.

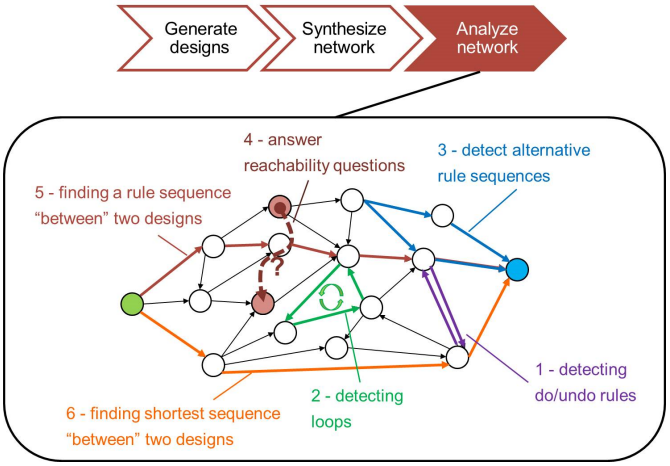


*Figure 2. Step 3: Analyzing the graph to understand the grammar rules and their application.*

## 4 IMPLEMENTATION

The transition graph is generated using GrGen, an open source graph rewriting tool (Geiß et al., 2006). It is visualized using OrganicVIZ (Cash et al., 2014), a graph visualization tool capable of representing large graphs and supporting graph analyses as well as providing several filtering options. Using OrganicVIZ, the user can manually study the transition graph to understand the search space. Nodes and edges can be changed in size and color 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 as well as loops and alternative sequences. This information is stored in separate files for further consideration by the human designer. To find shortest-paths between any two designs in the transition graph, a BFS with backtracking is implemented in C#. A console application is developed such that the designer can interactively search shortest paths between designs and determine reachability between designs.

## 5 CASE STUDY: TILE PUZZLE

The sliding tile puzzle is used as a case study for the presented research. This problem has been used frequently since its invention in 1879 (Slocum and Sonneveld, 2006) and it is still used in artificial intelligence research. In this puzzle, a number of tiles are arranged in a square 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 and Story, 1879). For the classical 8-tile puzzle (also called 3x3 puzzle), this leads, e.g., to 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 states and apply sequences to switch between known states quickly. As, in analogy to this, the goal in this paper is to make human designers capable of 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 useful rule application sequences for design transformations. The method is applied on a small scale problem to gain knowledge in a first step. Then the learnt knowledge is applied on a larger scale problem.

## 5.1 Understanding the small scale problem

A small version of the puzzle is used in this case study consisting of five tiles, numbered one to five, arranged on 2x3 tile grid. A grammar with four rules is developed to modify any given design puzzle. The four rules ('Up', 'Down', 'Right' and 'Left') are visualized in Figure 3. The LHS of the 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 grey. In the last column 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. 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 results from the manual and the computer-supported analysis of the transition graph are presented in Section 6.

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

*Figure 3. Rule set for the sliding tile puzzle.*

## 5.2 Applying knowledge on the large scale problem

The 8-tile puzzle is used as a large scale problem to analyze whether learning strategies, in particular rule application sequences identified on a reduced problem, can help to solve larger scale problems. An example puzzle is given in Figure 4. The task is to find a sequence of rule applications transforming the given puzzle (left) to the desired puzzle (right) using the knowledge 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. To apply the learnt knowledge, the smaller 5-tile puzzle is mapped into the 8-tile puzzle and only tiles within the smaller 2x3 regions are changed. The remaining tiles stay untouched. Changing between 2x3 regions, e.g. first looking at the upper region, then at the lower region in the 3x3 puzzle, tiles can move through the whole 3x3 puzzle. Dividing the problem into smaller sub-problems, humans can more easily define heuristics or

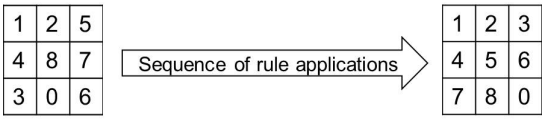


*Figure 4. Example for the 8-tile puzzle.*

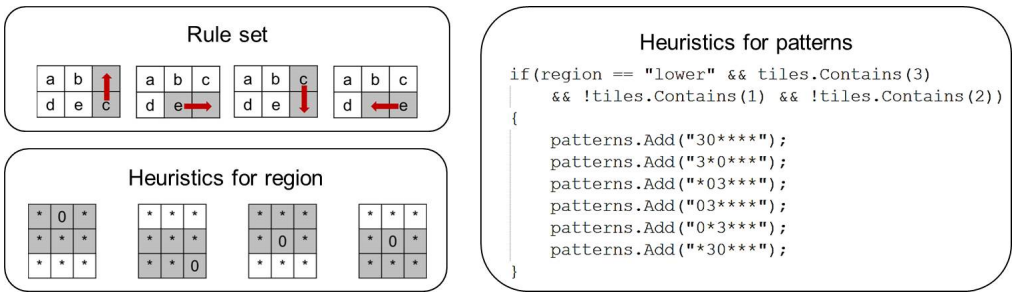


*Figure 5. Rule set (top left) and heuristics for defining regions of sub-problems (bottom left) and search patterns (right) defined by the human designer.*

strategies to solve problems like the sliding tile puzzle. Figure 5 presents such heuristics for the larger scale puzzle that have been adapted from human strategies. The rule set defined for the 5-tile puzzle can be kept. To define regions for sub-problems, the human can analyze the large-scale puzzle and select the region to modify in the next step, e.g. based on the position of the empty tile. In Figure 5 the grey regions define the region in which the next modifications are performed. When the empty tile (indicated with "0" in Figure 5) 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 the initial puzzle in Figure 4, the lower region is selected, however, it is still not clear how to change the tiles in the region.

General heuristics, e.g. trying to finish the puzzle from the top, can be implemented by defining search patterns. Following this general strategy, one can define search patterns as given in Figure 5 (right). The patterns are described as six character strings and define desirable tile positions giving the tile numbers and "*" as a wildcard symbol. In Figure 5 (right) one example is given to identify all designs 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, into the upper row, e.g. to replace the "5" in the initial design. The rule set, heuristics for the region, and heuristics for patterns depending on the region and the current tile configuration are implemented.

The process to solve the 8-tile puzzle is presented in Figure 6. In a first step exhaustive knowledge, i.e. the transition graph, is generated for the small scale puzzle and stored in the 5-tile library. In the second step, partial knowledge about the 8-tile puzzle is generated using a BFS approach. Implemented using a queue, in each iteration the first element in the queue is expanded. The 8-tile puzzle is subdivided using the heuristic for the region. The considered region is highlighted in grey in

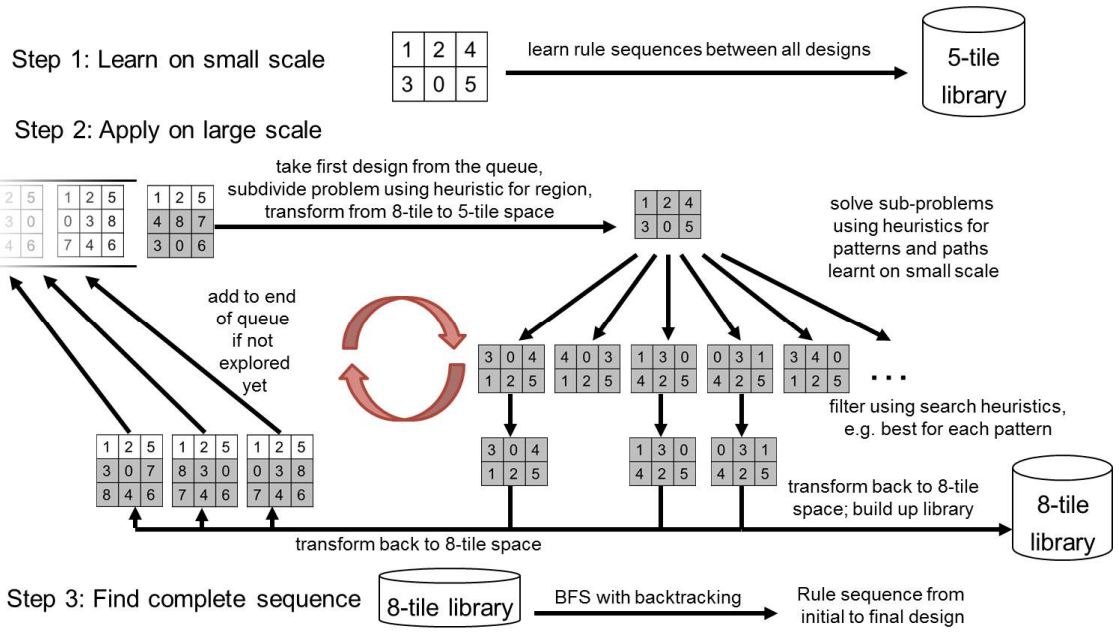


*Figure 6. Process of gaining knowledge on small scale (Step 1) and applying it to explore (Step 2) and solve larger scale problems (Step3).*

Figure 6. The tiles are 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 iteration shown in Figure 6 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 patterns defined in the pattern heuristic and for each pattern all designs that match it are identified in the 5-tile library. In Figure 6 five example designs matching the heuristics for patterns (30****, 03****, 3*0***, 0*3***, *30***, *03***) are shown, 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 one with the shortest rule sequence is considered further. The rule sequences and their respective generated designs (transformed back into 8-tile space) are added to the queue and stored in the 8-tile library. This means that the 8-tile library stores a 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 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.

## 6 RESULTS

As for the 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 7 (left) 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 vise-versa. 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 7, one has to identify in which graph the final state is present. Having a closer look at an excerpt of the transition graph (Figure 7, right) one 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 different colors and node sizes. 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. Alternative paths, as well as loops can be detected (see Figure 8) 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 8 shows an example of a loop that rotates four slides in counter clockwise direction. Understanding such loops can help to avoid the application of a sequence 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, automated

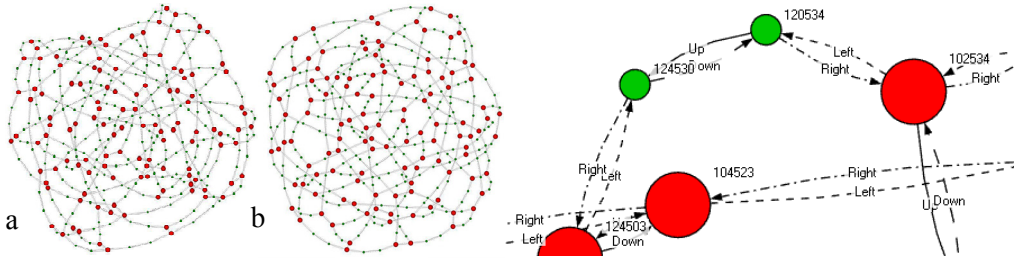


*Figure 7. Transition graphs for solvable (a) and unsolvable (b) puzzles (left) and a zoomed in view of the transition graph showing designs and their transformation through rules (right).*

analysis is also tested. As expected, two pairs of do-undo rules were identified, namely the pairs "Up" ←→"Down" and "Left"←→"Right". Numerous alternative paths between designs were found and questions of reachability between any two designs were answered successfully. It has further been shown that the shortest paths have been found between any two designs. In the second part of the case study, the learnt knowledge from the 5-tile puzzle was successfully applied to the 8-tile puzzle. For any solvable puzzle in the 8-tile space, a sequence of rule applications was found to transform the given puzzle into the desired one. For the initial design presented in Figure 4, for example, 21 moves were identified to transform the initial puzzle (1,2,5,4,8,7,3,0,6) into the final configuration (1,2,3,4,5,6,7,8,0): (1,2,5,4,8,7,3,0,6) → {Sequence: D, R, U, L, D} → (1,2,5,3,0,7,8,4,6) → {Sequence: R, D, L, U, L, D, R, R, U} → (1,2,3,0,7,5,8,4,6) → {Sequence: L, U, R, D, L, L, U} → (1,2,3,4,5,6,7,8,0).

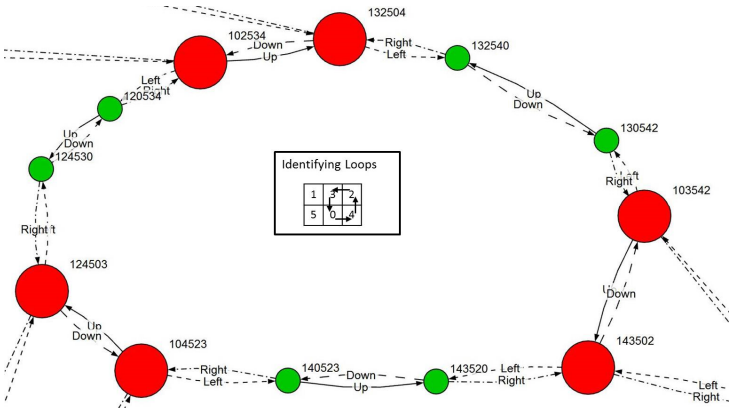


*Figure 8. Loops are detected to reason about the rules and avoid them during synthesis.*

## 7 DISCUSSION

As described above, the transition graph for a small scale problem, the 5-tile puzzle, was generated and analyzed giving insights into the grammar as well as the problem itself. Analyzing the transition graph allowed to find meaningful and meaningless sequences and approaches to how to replace longer sequences with shorter ones. Using the shortest-path algorithm allowed to identify the rule sequence with the least rule applications between any two designs in the search space. In the case study, the design space has been explored exhaustively for the small scale problem. For larger problems it is 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 algorithm bias the results. Depending on the problem at hand, it might be possible to explore the search space exhaustively for a smaller sub-problem, as in the case study, or for a limited number of rules in each applied rule sequence. Even though generating search spaces and deriving shortest sequences is not feasible for large problems, the knowledge gained by examining portions of the search space can be exploited to solve larger scale problems, as has been shown with the 8-tile puzzle. The 8-tile puzzle was used to demonstrate that strategies learnt for small scales can be used to help human designers to solve large scale problems. For a 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 the study, however, was not to search for computer-competitive strategies, but to enable human designers to reason about the search space in a more systematic way. The authors, therefore, consider the solution to the 8-tile puzzle to demonstrate the potential of using visualizations and analysis of transition graphs to strengthen human designers' understanding of developed grammars and the relations between designs and rule sequences. For the tile puzzle only one final state exists, but the presented method can likewise be used for problems with several final states and would present the rule sequence to the first final state that is found. Additional research is required in testing the developed method on an engineering design case study, e.g. an automated gearbox synthesis problem, where different gearbox topologies would, e.g., relate to different tile configurations in the tile puzzle and each gearbox with the desired number of speeds

would represent a final design. Future research includes problems with an unknown numbers of states, as well as rule applications where the resulting state is dependent on the LHS match in the design that is selected, i.e. the location of the rule application in the design influences the synthesis results.

## 8 CONCLUSION

The research presented in this paper analyzes the potential of using concepts from compiler design, in particular transition graphs, to support human designers in the context of Computational Design Synthesis using engineering design grammars. It has been shown that exhaustively searching small portions of the search space to collect data and generate transition graphs to visualize relations between different designs and sequences of rule applications can give human designers useful feedback about the grammar they developed. Through both manual analysis of the transition graph or computationally through graph search algorithms, loops in rule applications can be identified. Additionally, efficient rule application sequences can be identified through shortest path searches in the transition graph. The analyzed properties of the generated transition graph showed helpful in the case study, where the knowledge gained on a small scale problem is used to solve a larger scale problem, thus underlining the potential of formal transition graph analysis to support the development and use of engineering design grammars.

## REFERENCES


AHO, A. V., LAM, M. S., SETHI, R. & ULLMAN, J. D. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition)*, 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.

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., STANKOVIC, T. & STORGA, 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 - 021003-10.

GEIß, R., BATZ, G., GRUND, D., HACK, S. & SZALKOWSKI, A. 2006. GrGen: A Fast SPO-Based Graph Rewriting Tool. *In:* CORRADINI, A., EHRIG, H., MONTANARI, U., RIBEIRO, L. & ROZENBERG, G. (eds.) *Graph Transformations.* Berlin Heidelberg: Springer

GIPS, J. Computer Implementation of Shape Grammars. Workshop on Shape Computation MIT, 1999, 1999.

GIPS, J. & STINY, G. 1980. Production Systems and Grammars - a Uniform Characterization. *Environment and Planning B-Planning & Design,* 7**,** 399-408.

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. 2014. Systematic Rule Analysis of Generative Design Grammars. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing,* 28.

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. *ASME 2009 Int. Design Engineering Technical Conf. & Computers and Information in Engineering Conf.* San Diego, CA.

MCKAY, A., CHASE, S., SHEA, K. & CHAU, H. H. 2012. Spatial grammar implementation: From theory to useable software. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing,* 26**,** 143-159.

PAHL, G. & BEITZ, W. 1984. *Engineering Design,* Berlin, Springer.

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.

VALE, C. A. W. & SHEA, K. 2003. A Machine Learning-Based Approach To Accelerating Computational Design Synthesis. *Int. Conf. Engineering Design, ICED '03.* Stockholm: The Design Society.