Title:
Reconstructing Design Processes by Machine Learning of Graph-Rewriting Production Rules

Authors:
Julian R. Eichhoff, julian.eichhoff@informatik.uni-stuttgart.de, University of Stuttgart
Felix, Baumann, felix.baumann@informatik.uni-stuttgart.de, University of Stuttgart
Dieter Roller, dieter.roller@informatik.uni-stuttgart.de, University of Stuttgart

Keywords:
Design Automation, Machine Learning, Graph-Rewriting, Functional Decomposition, Rule Induction

DOI: 10.14733/cadconfP.2016.157-161

Introduction:

Graph-based models play an important role in product design. Particularly in conceptual design, graphs are used for abstract representation of functionality, topology and physical relations among product components. Graph-rewriting is an expressive computation model operating on graphs, and thus becomes a natural choice for implementing computed-aided conceptual design.

Graph-rewriting systems rely on a set of so-called production rules for deriving graphs. However, handcrafting such rules can become a tremendous effort — a well-known problem in the field of expert systems, called the “knowledge engineering bottleneck”. In this paper we discuss approaches to the automatic induction of production rules from given design graphs using machine learning. Four approaches were compared with respect to an application in conceptual design, specifically functional decomposition. The parse/derive method is an original contribution of this paper.

Main Idea:

Sridharan and Campbell [9] proposed a graph grammar for functional decomposition. Their set of production rules is capable of deriving complex function structures, i.e., a directed graph labeled with functions and flows, from a simple black box description of a product’s overall function. In [9] this grammar is used to determine the functional decomposition of an electric knife (see Fig. 1 for a simplified version). For the present work, these rules are used as benchmark for machine learning.

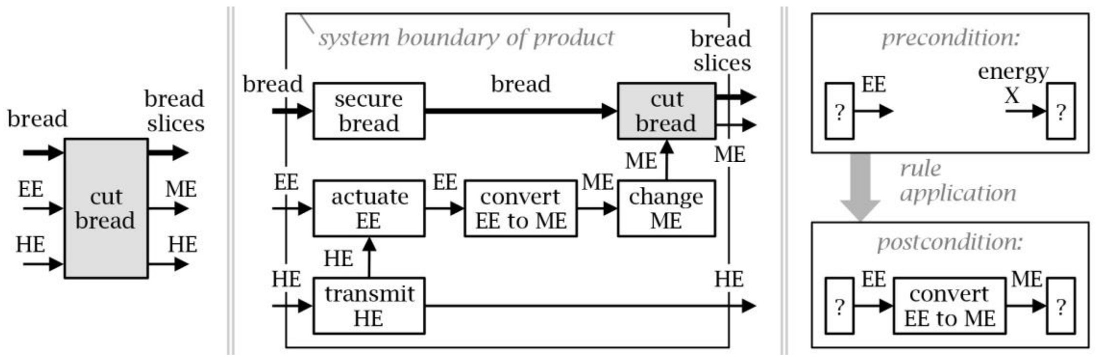

Fig. 1: Black box (left), function structure (center), example of a production rule that adds an energy conversion function (right). EE/HE/ME refer to electrical/human/mechanical energy, respectively.

There are various ways for implementing graph-rewriting systems. Herein, we suppose that production rules consist of a set of elementary graph transformations, which are applied on a host graph in order to yield an output graph (e.g. adding/removing nodes/edges/labels). The subsequent application of multiple rules, called derivation, leads to a series of produced graphs. Now, suppose graph G is a black box function model, and graph H is a unknown, fully evolved function structure, then H can be derived from G by applying a sequence of production rules p_1, p_2, …, p_n.

Now, consider the case where G and H are given, but the production rules and/or their sequence of application are not known. This resembles a situation where the requirements for a design task and its final results are documented, yet the way in which the results were achieved is unclear. This is the context where machine learning methods may be applied to reconstruct the design process. We distinguish four different situations thereof:

- A) No production rules are known.
- B) All production rules are known, but their sequence of application is unknown.
- C) Only a subset of the needed production rules is known, and nothing is known about the application sequence.
- D) Only a subset of the needed production rules is known, yet the sequence in which existing and missing rules should be applied is known.

The remainder of this paper deals with the implementation of machine learning strategies for each of these cases.

Case A: Literature surveys point out that there has been a continuing interest in the problems of grammar and rule induction (also called grammar/rule inference) over the last 30 years [3,7]. A well-known approach within that field is called Subdue [2]. Subdue induces a complete set of production rules from pairs of input/output graphs. It iteratively adds one rule at a time. At each iteration Subdue searches for subgraphs that frequently appear in the training examples. From these subgraphs one is chosen as the post-condition of the new rule. All appearances of the subgraph are then replaced by a non-terminal, which in turn is used as the pre-condition of the new rule. After this compression step the next rule is searched. From the candidate set, that rule is chosen which achieves the best compression ratio. Candidates are generated by "growing" subgraphs. The process starts from single nodes and successively adds a neighboring edge or an edge and a node.

Case B: The problem addressed here is a special form of the reachability problem in graph-rewriting [1]: Given a finite set of rules P, an initial graph G and a final graph H, the reachability problem is defined as follows: is there a derivation from G to H using P? If we are dealing with finite-state graph rewriting systems, reachability is decidable, whereas infinite graph-rewriting systems have to comply certain requirements (see [1] for details). Herein, we suppose that the space of possible rule sequences is finite (e.g. by defining limits for repeating rules). Hence, the set of graphs that can be produced, i.e. the graph-rewriting system's language, is finite as well.

In order to determine reachability, a search algorithm is needed that searches the combinatorial space of possible rule sequences. The search succeeds if a sequence is found that is able to reproduce the target graph H. In order to implement the search, any discrete search algorithm is applicable. However, the search space factorially increases with the rule sequence length, so a complete search most often becomes infeasible. Hence, some works in this field used meta-heuristics like simulated annealing [8]. Another important factor influencing efficiency is the independence of rules. If the rule set contains rules being independent from each other, different rule sequences may lead to the same result — an effect called confluence. Different techniques for handling confluence leading to a more efficient exploration of the search space are discussed in [4].

Case C: One of the methods described in [4], the critical-pair analysis, also provides a basis for the following parse/derive method, a method for dealing with case C and the main contribution of this paper. The idea of this approach can be paraphrased visually: Imagine G and H as two cities being separated by a river. To be able to go from G to H (derivation) the river is supposed to be bridged by a rule that must be learned. The parse/derive method approaches the river from both sides and searches for narrows along the river: I.e., it simultaneously searches for derivation sequences from G and parse sequences from H, where parsing denotes the reverse application of rules. The pair of derivation and parse sequences achieving the highest similarity of produced graphs defines the place for "constructing the bridge". The method involves multiple stages:

Initially, all existing rules are grounded, i.e., any variable used for specifying node, edges and labels are fixed to constants. The set of grounded rules is obtained from the permutation of possible variable instantiations. Grounding rules serves two purposes within the parse/derive method:

First, they are used to determine sequential dependencies among rules by means of critical pair analysis. This method looks for prototypical situations, where two rules stand in conflict (parallel dependence), or one rule requires the prior application of the other (sequential dependence). Candidates for such situations are found by inspecting the possible overlaps of grounded rules. See [6] for further details.

Second, given the grounded versions of a rule, frequency tables over the node/edge labels being affected by the rule can be computed. Label frequency tables taken before and after the application of a grounded rule are used to determine the differences in label frequencies, denoted by d. These differences are a simplified, vectorized representation of the rule's graph transformations. In the sense of this simplification, a derivation corresponds to the summation of frequency table differences over all applied rules. Adding this sum to the frequency table of the host graph results in the frequency table of the final graph. The procedure is the same for parsing, except that the frequency table differences of rules are negated.

From the vectorization of the parse and derivation processes, a quadratic linear integer optimization problem can be formulated (Eq. 1). The optimization problem targets the question: What rules need to be applied in what quantity, such that the difference of derivation and parse frequency tables is minimal? We denote this using two vectors x and y, for derivation and parsing respectively. The length of both vectors corresponds to the size of the set of grounded rules, and each row represents the times a rule is being applied. The goal is to find a pair x*, y* that minimizes the distance between the resulting parsing/derivation frequency tables. The problem is constrained by the identified sequential dependencies among rules (Eq. 2). Every rule is either applied on the elements of the initial graph, or on the elements added by a previously applied rule. Hence, if a rule is not sequentially dependent on others, it can only be applied on the initial graph (see second conditions of Eq. 2). In this case the upper bound for applications of a rule must be lower or equal to the number of possible applications on the initial graph numApp(G,k) or numApp(H,v), where k and v are indices for derivation rules and parsing rules respectively. If sequentially dependent rules exist, this upper bound is raised by the number of sequentially dependent rule applications (see first conditions of Eq. 2).

$$\left(\vec{x}^*,\vec{y}^*\right)=\underset{\vec{x},\vec{y}}{\arg\min}\sum_{i=1}^{m}\left(\left(h_{G,i}+\vec{d}_i^T\vec{x}\right)-\left(h_{H,i}-\vec{d}_i^T\vec{y}\right)\right)^2 \tag{1}$$

$$\begin{aligned}&\text{s.t. for each } k:\begin{cases} a_k \le numApp(G,k)+\sum_{j\in J} x_j & \text{if } J\neq\emptyset \\ a_k \le numApp(G,k) & \text{else.}\end{cases} \quad \text{where } J=\{j \mid seqDep(j,k)=true\}\\ &\text{and for each } v:\begin{cases} b_v \le numApp(H,v)+\sum_{u\in U} y_u & \text{if } U\neq\emptyset \\ b_v \le numApp(H,v) & \text{else.}\end{cases} \quad \text{where } U=\{u \mid seqDep(u,v)=true\}\end{aligned} \tag{2}$$

$$a_k=\begin{cases} x_k & \text{if rule indexed by k is self-dependent}\\ 1\leftrightarrow x_k\ge 1 & \text{else.}\end{cases} \qquad b_v=\begin{cases} y_k & \text{if rule indexed by k is self-dependent}\\ 1\leftrightarrow y_k\ge 1 & \text{else.}\end{cases} \tag{3}$$

Having identified promising quantities for the number of rule applications, the next step targets the question: In what sequences do the found rules have to be applied? This is answered using a genetic algorithm (GA) that searches over the now reduced space of possible rule sequences for derivation and parsing. Using the given rule application quantities, the GA tries to actually apply the rules stepping aside from the frequency table simplification used earlier. The longest pair of applicable sequences is considered optimal. From this pair all derived and parsed graphs are gathered.

Finally, the most similar pair of derivation/parse graphs is chosen to obtain the resulting rule. A mapping between both graphs is established for every common node or edge. Elements that cannot be mapped will be subject to the new rules graph transformations. Fig. 2 provides a summary of the method.

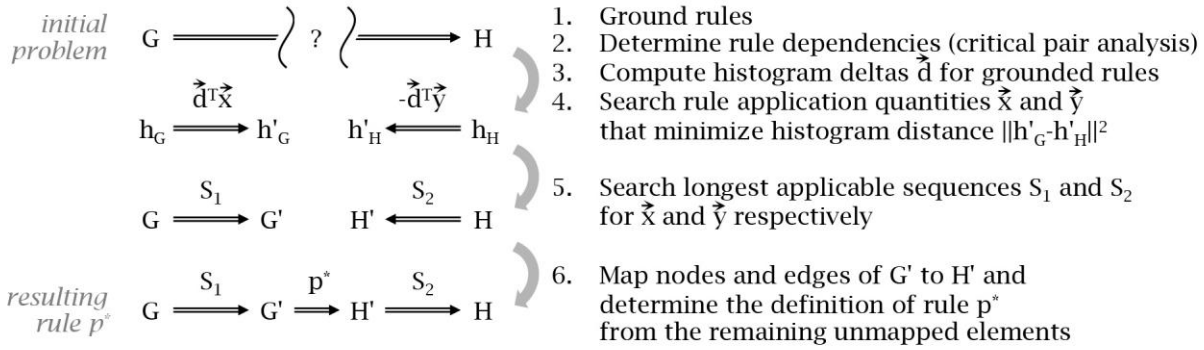
initial problem
G
?
H
$\vec{d}^T\vec{x}$
$-\vec{d}^T\vec{y}$
h_G
h'_G
h'_H
h_H
S_1
S_2
G
G'
H'
H
resulting rule p^*
G
S_1
G'
p^*
H'
S_2
H
1. Ground rules
2. Determine rule dependencies (critical pair analysis)
3. Compute histogram deltas $\vec{d}$ for grounded rules
4. Search rule application quantities $\vec{x}$ and $\vec{y}$ that minimize histogram distance $||h'_G-h'_H||^2$
5. Search longest applicable sequences S_1 and S_2 for $\vec{x}$ and $\vec{y}$ respectively
6. Map nodes and edges of G' to H' and determine the definition of rule p^* from the remaining unmapped elements

Fig. 2: Process steps of the parse/derive method.

Case D: If access to the definition of rules is provided and the reverse application of rules is available, then case D is a trivial application of the parse/derive method. However, in practical applications, definitions of existing rules are likely to be considered confidential, since they encode valuable design rationale. Further, rewrite-systems often are not readily designed to deal with the reverse application of rules for parsing.

A method capable of learning rules under these restrictions is the genetic programming approach of [5]. Genetic programming (GP) is a meta-heuristic that builds on evolutionary principles for solving combinatorial optimization problems. The method's key concepts are its tree representation of possible solutions and the sampling of solutions, which is based on evolutionary principles. In this case GP-trees are used to store two kinds of information: On which graph will the rule be applied? And, what graph transformation operations will the rule apply on the graph? The first question is addressed by a GP-tree's root node. Using an integer number associated with the root, a host graph for applying the rule candidate is chosen from a set of possible host graphs. Since the order of rule applications is known a priori, the set of possible host graphs can be directly obtained by means of derivation. The remaining nodes of the GP-tree are used to answer the second question. Each of these nodes represents a single graph transformation operation. Every such node has a single integer number associated, which is used to determine what node/edge is added/removed. The rule candidate is then applied by propagating a host graph through the GP-tree, where at each node the corresponding transformation is applied.

With each iteration of the GP, a new population of GP-trees is produced by means of the evolutionary principles of selection, recombination and mutation. Selection determines which solutions, or individuals, of the current iteration should be used for generating the individuals of the next iteration. This choice is based on the “fitness” of each individual. A rule candidate’s fitness is determined by deriving the rest of the rule sequence starting from the newly generated host graph. Each derived graph is then compared with the target graph. A rule candidate that facilitates the derivation of a graph being most similar to the target graph is considered optimal. Until optimality is reached, mutation and crossover are used to produce new individuals: Mutation randomly modifies parts of an existing solution to form new individuals for the next iteration. Recombination randomly joins parts of two solutions to create a new individual.

Results: All four approaches were applied within in computer experiments using the graph grammar of [9] as benchmark. During the experiments, rules were removed from the rule set and/or the sequence of rule applications was left open. Then the methods for A to D were applied to determine the missing rule(s) or the application sequence. Considering the page limitation of this article, only a summary is given. Supplementary material is provided online at http://ouky.de/accompanying-materials/cad-2016.

Due to the high complexity of the tasks of C and D, the computational affordances are high. On a conventional computer (PC with 2.5 GHz 4-core CPU and 16 GB RAM) the evaluation of a single leave-one-rule-out experiment ranges from several minutes up to a few hours depending on the complexity of the rule and its position within the rule sequence. Approaches A and B in turn operate within seconds up to a few minutes. The approaches of C and D are capable of producing rules that are highly similar to the original rule formulations. In contrast to this, Subdue induces a complete rule set from scratch, where the new rules reflect the frequencies of recurring graph patterns within the training graphs. If this was not the purpose of the original rules, correspondence with learned rules cannot be guaranteed for case A. Besides this, effects can be observed with C and D, which we term "cannibalization" and "the swapping of duties". The first effect is that the rule to be learned also replaces existing rules by performing their operations as well. The second effect appears if there is a rule succeeding the rule to be learned, and this existing rule is able to take over some of missing rule's operations. However, then the substituting rule's original task will not be accomplished anymore. This is where the learner sets in and determines a new rule compensating the original rule's missed duties.

Conclusions:

This article addressed a crucial issue in the context of graph-based conceptual design automation, namely the configuration of graph-rewriting systems used for producing design graphs. It has been shown that various techniques exist to determine the production rules and rule application sequences, both essential parts of a graph-rewriting system. With the proposed parse/derive method, this toolbox is further extended. We suppose that with the ongoing advancement of high-performance computing such methods will facilitate the practical implementation of graph-based expert systems for CAD.

References:

[1] Bertrand, N.; Delzanno, G.; König, B.; Sangnier, A.; Stückrath, J.: On the Decidability Status of Reachability and Coverability in Graph Transformation Systems, 23rd Intl. Conf. on Rewriting, 2012, pp. 101–116.

[2] Cook, D.J.; Holder, L.B.: Substructure Discovery Using Minimum Description Length and Background Knowledge. J. Artif. Intell. Res., 1(1), 1994, pp. 231–255. http://dx.doi.org/10.1613/jair.43

[3] De la Higuera, C.: A Bibliographical Study of Grammatical Inference, Pattern Recogn., 38(9), 2005, pp. 1332–1348. http://dx.doi.org/10.1016/j.patcog.2005.01.003

[4] Eichhoff, J.R.; Roller, D.: Designing the Same, but in Different Ways: Determinism in Graph-Rewriting Systems for Function-Based Design Synthesis, J. Comput. Inf. Sci. Eng., 16(1), 2016, pp. 011006-011006-10. http://dx.doi.org/10.1115/1.4032576

[5] Eichhoff, J.R.; Roller, D.: Genetic Programming for Design Grammar Rule Induction, RuleML 2015 Challenge, the Special Track on Rule-based Recommender Systems for the Web of Data, the Special Industry Track and the RuleML 2015 Doctoral Consortium hosted by the 9th Intl. Web Rule Symposium, 2015, pp. 1–8.

[6] Ehrig, H.; Golas, U.; Habel, A.; Lambers, L.; Orejas, F.: M-Adhesive Transformation Systems with Nested Application Conditions. Part 2: Embedding, Critical Pairs and Local Confluence, Fund. Inform., 118(1-2), 2012, pp. 35–63. http://dx.doi.org/ 10.3233/FI-2015-1282

[7] Pappa, G.L.; Freitas, A.A.: Towards a Genetic Programming Algorithm for Automatically Evolving Rule Induction Algorithms, Advances in Inductive Rule Learning, Workshop at the 15th European Conf. on Machine Learning and the 8th European Conf. on Principles and Practice of Knowledge Discovery in Databases, 2004, pp. 93–108.

[8] Schmidt, L.C.; Cagan, J.: GGREADA: A Graph Grammar-Based Machine Design Algorithm, Res. Eng. Des., 9(4), 1997, pp. 195–213. http://dx.doi.org/10.1007/BF01589682

[9] Sridharan, P.; Campbell, M.I.: A Grammar for Function Structures, ASME 2004 Intl. Design Engineering Technical Conf. and Computers and Information in Engineering Conf., 2004, pp. 41–55. http://dx.doi.org/10.1115/detc2004-57130