COMPUTER-AIDED DESIGN & APPLICATIONS, 2017
VOL. 14, NO. 4, 535–548
http://dx.doi.org/10.1080/16864360.2016.1257195

Inducing production rules to extend existing design grammars: The parse/derive method

Julian R. Eichhoff [a], Jens Schmidt [b] and Dieter Roller [a]

[a]University of Stuttgart, Germany; [b]IILS Ingenieurgesellschaft für Intelligente Lösungen und Systeme mbH, Germany

ABSTRACT
Graph-rewriting is a promising computation model for computer-aided design (CAD) applications that operate on graph-based design models. Graph-rewriting-based CAD systems rely on predefined production rules. These rules encode the set of possible actions that may be taken within the design process to make changes to the current design. This paper presents a method for automatically inducing new production rules from existing sample designs. The methods applicability is illustrated in context of conceptual spacecraft design. Results gained from experiments, where existing rules were deliberately left out and had to be rediscovered, show that the induced rules are often similar or identical to the original rules.

KEYWORDS
Graph-rewriting; design grammar; rule induction; rule inference

1. Introduction

Throughout product development, various design methods rely on graph-based models to represent different aspects of product designs. In Systematic Design [16] or the German engineering norm VDI 2221 [22], for instance, graphs are used to depict functionality, concepts for physical realization, modularity, and geometrical embodiment. In recent years there has been a growing interest in graph-based notations that may serve as a common representational framework over multiple phases of product design. Most prominent example for this is the Systems Modeling Language (SysML) [15], a dialect of the Unified Modeling Language (UML) [14], which is well known in software engineering. SysML particularly targets complex systems-of-systems engineering projects, where traceability of design decisions has to be ensured over subsequent phases of design maturation, over different engineering disciplines, and over the topology of integrated sub-systems.

Graph-rewriting (also known as graph-production) is a computation model that is philosophically rooted in the concept of model transformation. By means of so-called production rules, graph-rewriting systems transform a source graph into a (set of) target graph(s). Graph-rewriting has been used in several applications for computer-aided design (CAD) [1,6,11–13,17–21]. The design compiler "43" [1], for instance, is a comprehensive CAD environment that uses UML to represent all aspects of product design, i.e., from requirements, over functional modeling and principle design to geometry, manufacturing, and even product documentation. This is possible due to the very general applicability of UML on the one hand. On the other hand, graph-rewriting production rules allow to transform an existing model (e.g., requirements specification) into a new model, which is more detailed (e.g., functional decomposition) or captures different aspects (e.g., embodiment design).

However, handcrafting such rules can become a tremendous effort — a well-known problem in the field of expert systems, called the "knowledge engineering bottleneck". This paper, an extension of the CAD conference paper [7], demonstrates the application of the so-called parse/derive method with respect to a case from conceptual spacecraft design. This method is capable of automatically inducing production rules from given design graphs in context of an incomplete set of existing production rules. The latter is also a central aspect by which the parse/derive method is distinguished from most of the existing works in the field of rule induction for graph-rewriting. Previous approaches focused at inducing a complete rule set from scratch without considering the integration of an existing rule set, e.g., [2,3,10]. [5] in turn addressed a similar rule induction problem, where a rule had to be induced in context of a fixed rule sequence involving existing rules. In this paper, however, we are particularly interested in finding the most appropriate rule sequence for rule induction.

CONTACT Julian R. Eichhoff julian.eichhoff@informatik.uni-stuttgart.de

The remainder of the paper is structured as follows. First, section 2 draws analogies between product development and graph-rewriting to provide a common understanding on how graph-rewriting is able to support computer-aided design. Section 3 introduces the proposed rule induction method, whose application is then demonstrated with respect to conceptual spacecraft design in section 4. Concluding remarks and suggestions for future work are given in section 5.

2. The duality between product development and graph-rewriting

This section delineates how product development problems map to graph-rewriting problems. The motivating hypothesis behind this is stated in the following conjecture:

Conjecture 1: There is a duality between engineering design and graph-rewriting in such that every engineering design problem can be represented in terms of a graph-rewriting problem.

2.1. Product development as subsequent optimizations

We conceptualize product development as an optimization problem. More specifically, we consider it a "*search over a set of realizable technical artifacts for alternatives that show the highest compliance with a set of objectives*" [4]. Even for seemingly simple products, the set of alternative designs can become large. A common approach to reduce the complexity stemming from design methodology is to solve the overall problem in series of subsequent design phases. Each phase addresses a subset of the objectives given by a requirements specification or a design brief. We define an objective as follows:

Definition 1: *A tuple $o = (s_1, s_2)$ consisting of two states s_1 and s_2 is termed objective. It denotes that a system's current state s_1 should be changed to some desired state s_2.*

We consider different kinds of objectives:

Design-process-related objectives, or *process objectives* for short, refer to different states in the description of the artifact to be designed. What is desired is a change from a simplified representation to a more detailed one. In this sense, the overall design objective of product development is, for instance, the change from a given design brief (s_1) to a product documentation (s_2).

Requirements are product-related objectives, where s_2 represents the desired properties of the artifact to be designed, and s_1 is either empty (in the case of an original design) or s_1 represents properties of an existing artifact (in case of redesign, adaptive design, and variant design). A design method imposes process objectives to guide a designer in fulfilling requirements.

Definition 2: *A set of objectives $O = \{o_1, o_2, \ldots\}$ is called a problem if the transition from current states to desired states is not trivial and means for performing this transition are not readily available. If there are means available for solving the problem, we speak of a task.*

Definition 3: *In this work we consider means for problem solving to be equal with plans that describe a feasible sequence of actions, which can be performed in order to reach the desired state.*

Reasons for a plan being unavailable are:

- There are no plans known for solving the problem.
- The plan must be chosen from a vast set of possible plans.
- The objectives are incomplete and/or conflicting, such that adequate plans are not obvious.

Given this, we conceptualize product design as solving several consecutive optimization problems. Each design phase is concerned with finding a subset of realizable artifacts, whose properties minimize the distance to the desired states of the phase-specific objectives. [4] investigated the formulation of such design optimization problems with respect to different product phases and in the context of evolutionary design exploration. With each consecutive phase, different objectives are optimized narrowing down the set of design alternatives under consideration. Possibilities for moving back to previous phases are also given. In the end, the final subset constitutes the set of feasible design alternatives.

2.2. Graph-rewriting

This section repeats some essential definitions for graph rewriting-systems from [6]. For a complete introduction see [8,9].

Definition 4: *A label alphabet $A = (A_V, A_E)$ is a pair of sets of node labels and edge labels. A labeled graph over A is a system $G = (V_G, E_G, s_G, t_G, l_G, m_G)$ consisting of:*

- *A finite set of nodes V_G and a finite set of edges E_G*
- *A source function $s_G : V_G \to E_G$ and a target function $t_G : V_G \to E_G$ (for defining adjacencies and edge directions)*
- *A node labeling function $l_G : V_G \to A_V$ and an edge labeling function $m_G : E_G \to A_E$*

Definition 5: *A graph morphism $G \to H$ or morphism for short is a map from graph G to another graph H. It consists of two functions $g_V : V_G \to V_H$ and $g_E : E_G \to E_H$ that preserve sources, targets and labels. A bijective graph morphism is termed graph isomorphism and denoted by $\cong$.*

Definition 6: *A production rule $p = \langle L \leftarrow K \rightarrow R \rangle$ or rule for short is a pair of graph morphisms with a common domain K, called the interface. L is termed left-hand side and R right-hand side. Rules can be equipped with application conditions (see [9] for details) that further restrict their application beside their occurrence morphisms (see below).*

Definition 7: *The application of a rule p on graph G with respect to a certain occurrence $o : L \to G$, which results in graph H, is called a direct derivation and written $G \overset{p,o}{\Rightarrow} H$. It exists if and only if the double-pushout diagram of Fig. 1 can be constructed, where D is termed context. The graph morphism $q : R \to H$ is termed co-occurrence.*

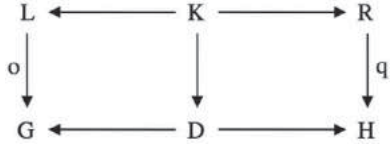

Figure 1. Double-pushout diagram.

Definition 8: *The symmetry of the double-pushout diagram suggests the construction of an inverse rule $\bar{p} = R \leftarrow K \rightarrow L$. Its corresponding direct derivation $H \overset{\bar{p},q}{\Rightarrow} G$ is also termed a direct parsing.*

Definition 9: *A derivation $G \overset{*}{\Rightarrow} H$ from graph G to graph H is a sequence of direct derivations $G \Rightarrow G_1 \Rightarrow G_2 \Rightarrow \cdots H$ over a set of rules $P = \{p_1, p_2, \ldots\}$. A derivation in the inverse direction $H \overset{*}{\Rightarrow} G$ is also called a parsing.*

Definition 10: *A graph-rewriting system is a pair (P, G) where P is a set of rules and G is an initial source graph used as starting point for derivations. The set $\mathcal{H}$ comprises all possible target graphs that may be produced by derivations over (P, G) and is called the language of the graph-rewriting system.*

2.3. Duality

Building on the above definitions, we suppose a duality between both fields, which is shown in Table 1. From this we are able to render product development tasks and problems in terms of graph-rewriting.

Table 1. Analogies between product development and graph-rewriting.

Product Development	Graph-Rewriting
State	Graph
Current State	Source Graph
Desired State	Target Graph
Realizable Technical Artifacts	Language of Graph-Rewriting System
Objective	Ordering Relation on Language
Action	Production Rule
Plan	Derivation

If the goal is to accomplish a design task, i.e., there are design objectives to be achieved and the designer is in knowledge of a plan for doing so, then the same plan can be encoded as a sequence of production rule applications. Performing the task then corresponds to computing derivations over this sequence.

If, in turn, a plan must be chosen from a vast set of possible plans, or if the objectives are incomplete and/or conflicting, such that adequate plans are not obvious, then this corresponds to determining a rule sequence for reaching an optimal target graph. In graph-rewriting (or computer science in general) this problem is known as the reachability problem.

Definition 11: *Given a finite set of production rules P, a source graph G_0 and a target graph H, the reachability problem is defined as follows: does $G_0 \Rightarrow_P H$ hold?*

Adapting this problem to the setting of design optimization, we are in search for an appropriate rule sequence that, when applied in derivation, produces (a set of) graph(s) which minimize a design-specific objective function.

Definition 12: *Given a finite set of production rules P, a source graph G_0 and an objective function $f(H \in \mathcal{H})$, which provides an ordering relation on the language $\mathcal{H}$ of graph-rewriting system (P, G_0), the graph-rewriting design optimization problem is defined as follows: What is an appropriate rule sequence $s^* \in P^+$ such that $s^* = \arg\min_{s \in P^+} f(H)$ with $G_0 \overset{s}{\Rightarrow} H$ does hold?*

In this paper, 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 [19]. 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 [6].

Now, in the last design problem mentioned, there are no plans known for solving the problem. When translating this to graph-rewriting, we yield another kind of reachability problem, where the production rules are (partly) unknown. To tackle this problem, we aim at inducing new rules from existing designs. Therefore, we assume there is at least one pair of source graph and corresponding target graph available, which represents the input and output design states of an existing (positive) sample design.

Definition 13: *Given the set of all possible production rules $\mathcal{P}$, a source graph G_0 and target graph H, the production rule induction problem is defined as follows: what is a sufficient finite set of production rules $P \subseteq \mathcal{P}$ such that $G_0 \Rightarrow_P H' \cong H$ does hold?*

The main contribution of this paper is a method for solving this problem under the restriction that there is a set of existing production rules, which is insufficient for reaching the target graphs, but should be reused whenever possible. Further, we restrict the problem herein to the induction of a single rule. This is actually valid and sufficient for solving the above problem, as the induced rule can be seen as a composite form of multiple (missing) rules. In graph-rewriting theory this is called amalgamation [9]. It refers to the idea that multiple rules can be joined to form a single rule, and vice versa, that one rule can be separated into a sequence of rules.

3. Solving the production rule induction problem

The idea of this approach can be paraphrased visually (cf. Fig. 2): Imagine graphs G_0 and H as two cities being separated by a river. To be able to go from G_0 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_0 and parsing sequences from H (reverse application of rules). The pair of derivation and parsing sequences achieving the highest similarity of produced graphs defines the place for "constructing the bridge". To ease this search, we employ a simplified representation of graphs and production rules, which is based on label frequencies. With this representation we are able to simulate different combinations of derivations and parsings over existing rules in order to estimate which rules need to be applied in what quantities. Later, the actual rule sequences for parsing/derivation are determined by searching over possible orderings with the estimated rule quantities. We are now going to introduce a running example from the domain of conceptual spacecraft design [17]. For this section we will use an excerpt of this case to demonstrate each of the method's stages. In section 4 we will elaborate on the complete case.

Example 1: Figure 3 shows the source graph of this case (top), together with the first three production rules, and the target graph (bottom) that results from applying these rules on the source graph. The sequence of rule application for reaching this target graph is (p_1, p_2, p_3, p_3). Figure 4 gives an overview on the vocabulary used for node and edge labels and shows the taxonomies defined over these. The source graph represents an initial situation in the design of a spacecraft (SC) propulsion system (PROPSYS). Here, a special kind of propulsion system, a cold-gas system (CGS), is preselected for being implemented. To determine how this system is implemented, a graph-rewriting system is employed. Specifically, it is used to determine feasible topologies for mechanical components by means of systematic functional decomposition [16].

The first rule starts this process by adding a node, which represents the installation space used for the CGS propulsion system (FUELA). With reference to Fig. 4 the label FUELA (fuel area) denotes a subclass of A (area), whereas CGS is a subclass of PROPSYS. The used graph-rewriting engine automatically resolves these taxonomical relations during rule application such that the PROPSYS node of rule 1 can actually be matched with the CGS node of the source graph. This inheritance concept, which is derived from object-oriented programming, allows specifying rules that are more generally applicable. The same applies to rule 2 with respect to A and FUELA. Rule 2 adds the tasks which should be fulfilled by the components installed in the FUELA space (STORE, MANAGE, THRUST) and imposes an order on the sequence for executing these tasks. Rule 3 is then used to further specify how THRUST is generated with thruster clusters (THRSTRCL). An additional link to the engine node (ENG) is drawn establishing traceability

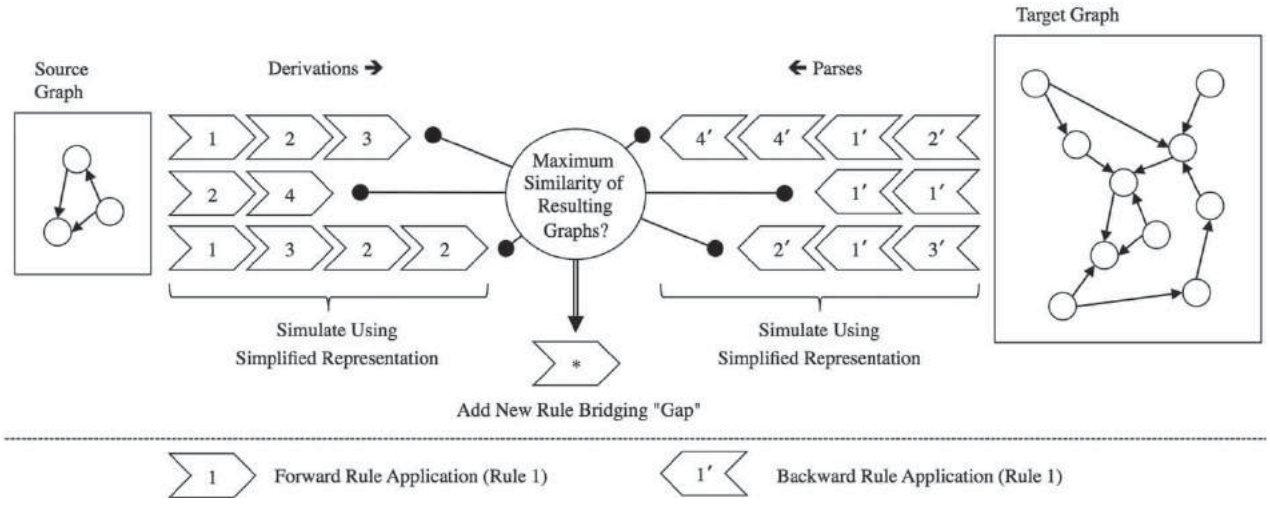

Figure 2. Illustration of parse/derive method.

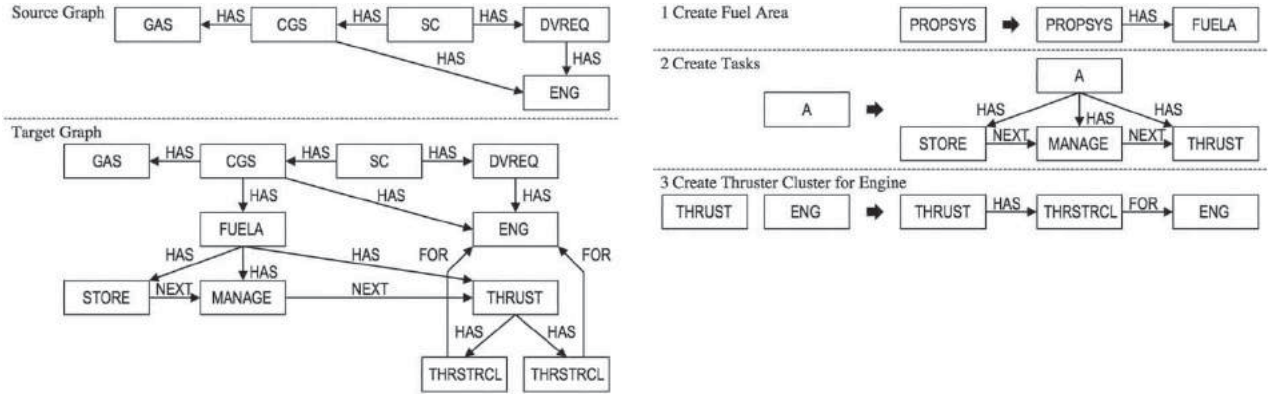

Figure 3. Simplified graph-rewriting example.

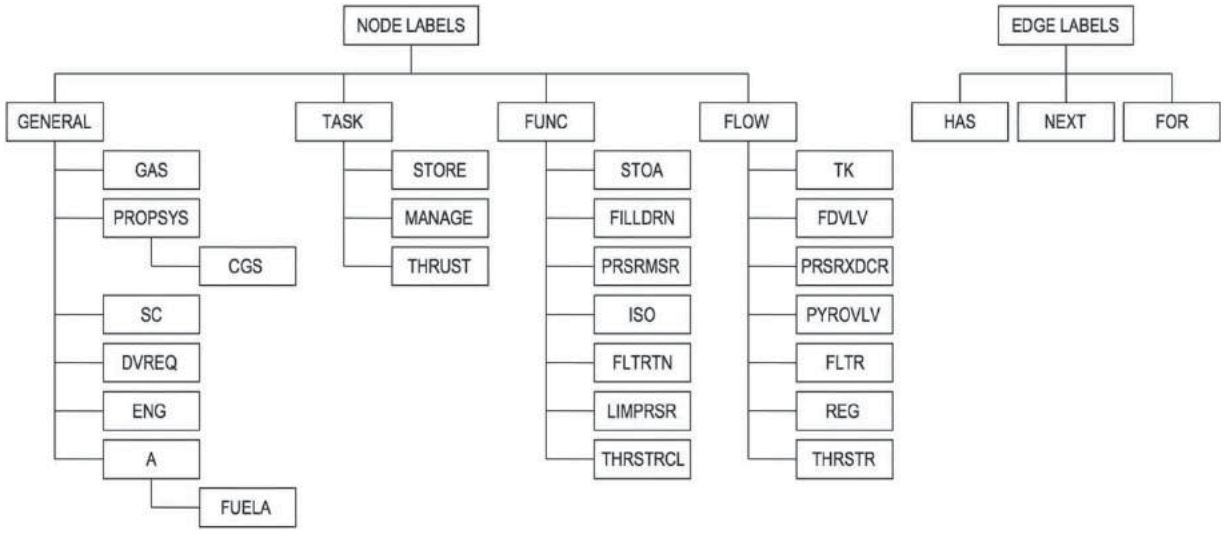

Figure 4. Node and edge label taxonomies.

with associated delta-v requirements (DVREQ). If a rule can be applied multiple times at the same position within the current graph, it is called self-independent (or self-dependent in the other case). Rule 3 is self-independent and can be applied multiple times with respect to the existing THRUST and ENG nodes. How many times a rule is actually applied depends on the chosen rule application sequence, which is in this case (p1, p2, p3, p3). From a semantical perspective, applying rule 3 twice denotes that two redundant thruster clusters are used to generate the required thrust. As shown in section 4, the derivation continues detailing the other tasks. For now, we stop with rule 3 and the target graph resulting from sequence (p_1, p_2, p_3, p_3).

3.1. Grounding rules

Table 2 demonstrates how variables are used for implementing the behavior of rules 1, 2 and 3 by means of small procedural programs, which are invoked by the graph-rewriting engine during derivation. Applying a rule means running through all operations within its program from top to bottom. Each operation signalizes whether it was applicable or not. If one operation is not applicable, then the whole rule application fails. Variables for nodes, edges and labels take account of the different possibilities for applying a rule. The label inheritance feature described above, for instance, is implemented using label-variables in conjunction with an operation relatedLabels testing for subclass relations. Each operation within the program makes use of already set variables (e.g., addEdge links two previously specified nodes) and/or assigns new values to variables (e.g., getNode selects a node from the host graph and assigns its unique identifier to a node-variable and the node's label to a label-variable).

Obviously, there may be multiple options for and hence combinations of variable assignments, for instance, with respect to the getNode operation. The used graph-rewriting engine is designed to test different variable assignments with each invocation of a rule's program. This can be done until there are no more untested combinations for variable assignment left.

The parse/derive method makes use of these variables in an inverse manner: 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 (see section 3.1.2) by means of critical pair analysis [8]. 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.

Second, given the grounded versions of a rule, frequency tables over the node/edge labels being affected by the rule can be computed (see section 3.1.3). Label frequencies taken before and after the application of a grounded rule are used to determine label changes caused by that rule. These differences, denoted by $\vec{d}$, are a simplified, vectorized representation of the rule's graph transformations. In the sense of this simplification, a derivation corresponds to the summation of label frequency differences over all applied rules. Adding this sum to the label frequencies of the host graph results in the frequencies of the final graph. The procedure is the same for parsing, except that the differences of rules are negated.

To limit the combinatorial complexity associated with grounding rules, we first determine which node and edge labels are feasible for grounding. Feasible labels are those appearing in source and target graphs, as well as those appearing in left-hand sides of rules. All other labels are irrelevant for deriving the target graph from the source graph, as they cannot be removed by any rule within the rule set.

Table 2. Rules 1, 2 and 3 implemented as procedural programs. Variables are printed in italics.

Rule 1	Rule 2	Rule 3
getNode(*node-1*, *label-1*)	getNode(*node-1*, *label-1*)	getNode(*node-1*, THRUST)
relatedLabels(*label-1*, PROPSYS)	relatedLabels(*label-1*, A)	getNode(*node-2*, ENG)
addNode(*node-2*, FUELA)	addNode(*node-2*, STORE)	addNode(*node-3*, THRSTRCL)
addEdge(*node-1*, *node-2*, HAS)	addNode(*node-3*, MANAGE)	addEdge(*node-1*, *node-3*, HAS)
	addNode(*node-4*, THRUST)	addEdge(*node-3*, *node-2*, FOR)
	addEdge(*node-1*, *node-2*, HAS)	
	addEdge(*node-1*, *node-3*, HAS)	
	addEdge(*node-1*, *node-4*, HAS)	
	addEdge(*node-2*, *node-3*, NEXT)	
	addEdge(*node-3*, *node-4*, NEXT)	

Example 2: The relevant vocabulary identified with respect to example 1 is: A, CGS, DVREQ, ENG, FOR, FUELA, GAS, HAS, MANAGE, NEXT, PROPSYS, SC, STORE, THRSTRCL, THRUST

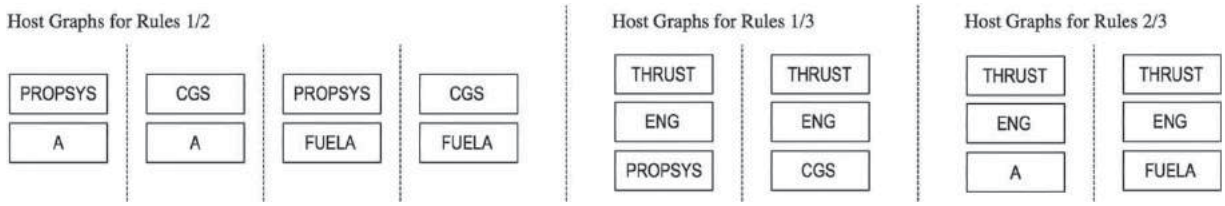

Figure 5. Possible host graphs for testing dependencies among rules 1, 2 and 3.

3.2. Extended critical pair analysis

This analysis provides means to determine dependencies among rules statically, i.e., upfront to actual derivation. Therefore it produces prototypical situations for applying a pair of rules and checks whether the direct derivations over these rules stand in conflict. To produce these situations all possible overlaps of both rules' occurrences of minimal size are computed. For rules involving variables for node/edge labels these variables are grounded using the previously determined feasible label set. For further details on the theoretical foundations of critical pair analysis see [8].

In order to compute all possible grounded rules we do not limit the set of host graphs to those where left-hand sides overlap. Rather all possible common host graphs of minimal size are computed. Therefore we term this phase *extended* critical pair analysis.

Example 3: Figure 5 shows all possible host graphs for the rules of example 1.

A pair of rules $\{p_i, p_j\}$ is then applied on the found host graphs in sequence (p_i, p_j) and in reverse sequence (p_j, p_i). With each host graph there may be several possibilities for applying a rule (if there are multiple left-hand side occurrences), where some of these possibilities can result from changes introduced by the preceding application of the other rule within the pair. Critical pair analysis classifies each derivation over rules p_i and p_j into one of the following kinds:

- Independent: Both rules can be applied no matter in what sequence.
- Parallel dependent: When rule p_i is applied first then p_j cannot be applied anymore and vice versa.
- Sequentially dependent: Rule p_i introduces elements that are required by p_j. Hence, p_i must be applied first, before p_j can be applied.
- Sequentially dependent because of negative application conditions: Rule p_i introduces a condition that causes p_j to become inapplicable.

Example 4: The derivations of rule pairs on host graphs of Fig. 5 are classified as follows:

- Rules 1/2: For each of the four host graphs there is one independent derivation and one sequentially dependent derivation with applicable sequence (p_1, p_2).
- Rules 1/3: For both host graphs there is one independent derivation.
- Rules 2/3: For both host graphs there is one independent derivation and one sequentially dependent derivation with applicable sequence (p_2, p_3).

In addition to analyzing the applicability of rules with respect to each other, we also determine the applicability of rules with respect to source and host graphs. In case of the latter we actually test the applicability of the inverse rule.

Example 5: For example 1 we determine the applicability of rules on source and target graphs:

- Applicability on source graph: Rule 1 unlimited, rules 2 and 3 are not applicable.
- Applicability on target graph: Rule 3 is applicable two times, rules 1 and 2 are not applicable.

3.3. Label frequency approximation

One central aspect of the proposed method is to simulate derivations from the source graph and parses from the target graphs using an approximate representation of graphs and production rules. This allows for fast exploration of possible solutions to the rule induction problem. The computationally more expensive verification of the reduced set of approximate solutions is then carried out afterwards using actual graph-rewriting.

As approximate representation we employ frequency tables that capture the appearance of node and edge labels within graphs. In order to represent production rules we use the difference of label frequencies resulting from the rules application. Though this ignores any topology defined on the graphs it provides a strong indicator in context of the given application scenario, where different labels are used to denote engineering concepts or product components. Further, to simulate the reverse application of rules, signs of label frequency differences just need to be negated.

Example 6: The label frequencies of source and target graphs of example 1 are:

- Source graph: 1 × CGS, 1 × DVREQ, 1 × ENG, 1 × GAS, 4 × HAS, 1 × SC
- Target graph: 1 × CGS, 1 × DVREQ, 1 × ENG, 2 × FOR, 1 × FUELA, 1 × GAS, 10 × HAS, 1 × MANAGE, 2 × NEXT, 1 × SC, 1 × STORE, 2 × THRSTRCL, 1 × THRUST

Example 7: The differences in label frequencies caused by rule application are:

- Rule 1: +1×HAS, +1×FUELA
- Rule 2: +3×HAS, +1×MANAGE, +2×NEXT, +1×STORE, +1×THRUST
- Rule 3: +1×HAS, +1×FOR, +1×THRSTRCL

3.4. Rule induction

From the label frequency vectorization of the parse and derivation processes, a mixed-integer quadratic program (MIQP) 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 parsing label frequencies is minimal? We denote this using two vectors $\vec{x}$ and $\vec{y}$, 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 $\{\vec{x}^*, \vec{y}^*\}$ that minimizes the distance between the resulting parsing/derivation frequency vectors.

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 condition 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 condition of Eq. 2).

Further, we have to distinguish between self-independent and self-dependent rules when formulating these constraints (Eq. 3). Recall that rules which may be applicable multiple times to the same occurrence are considered self-independent. Otherwise, if one application prevents any further applications of the rule on the same occurrence, it is considered self-dependent. To reflect this within the optimization constraints, auxiliary variables a and b are introduced. These are either equal to the actual number of rule applications in the case of self-dependent rules, or turn into binary variables in the case of self-independent variables indicating the presence of one or more applications of that rule.

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

$$\text{s.t. for each } k: \begin{cases} a_k \le numApp(G,k) \\ \quad + \sum_{j \in J} x_j & \text{if } J \neq \emptyset \\ a_k \le numApp(G,k) & \text{else.} \end{cases}$$

$$\times \text{ where } J = \{j | seqDep(j,k) = true\}$$

$$\text{and for each } v: \begin{cases} b_v \le numApp(H,v) \\ \quad + \sum_{u \in U} y_u & \text{if } U \neq \emptyset \\ b_v \le numApp(H,v) & \text{else.} \end{cases} \tag{2}$$

$$\times \text{ where } U = \{u | seqDep(u,v) = true\}$$

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

$$b_v = \begin{cases} y_v & \text{if rule indexed by v} \\ & \quad \text{is self - dependent} \\ 1 \leftrightarrow y_v \ge 1 & \text{else.} \end{cases} \tag{3}$$

Example 8: Eq. 4 exemplifies the composition of the optimization function of Eq. 1 with respect to the edge label HAS. The source and target graph frequencies are taken from example 6 and the frequency differences for rule 2 and 3 correspond to those of example 7.

$$(\vec{x}^*, \vec{y}^*) = \arg\min_{\vec{x},\vec{y}} \left[\ldots + \left(\left(4 + \begin{pmatrix} 3 \\ 1 \end{pmatrix}^T \vec{x} \right) - \left(10 - \begin{pmatrix} 3 \\ 1 \end{pmatrix}^T \vec{y} \right) \right)^2 + \cdots \right] \tag{4}$$

Example 9: The sequential dependency constraints for the running example are shown in Eq. 5. The used indices

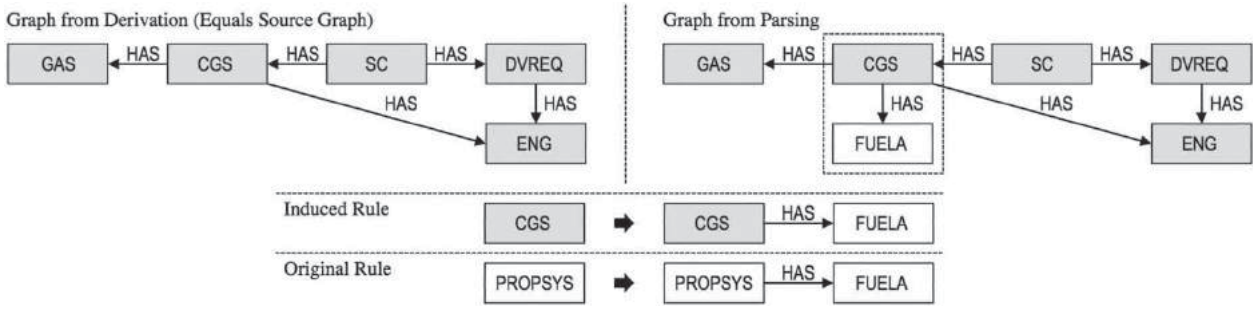

Figure 6. Induced rule corresponding to original rule 1. Nodes mapped by largest subgraph isomorphism are shaded.

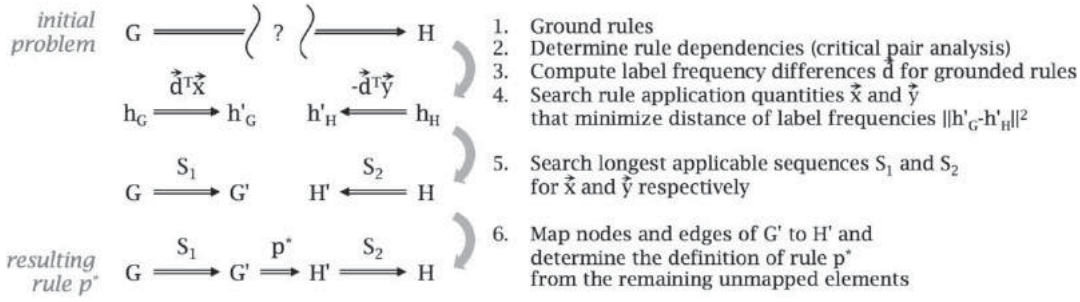

Figure 7. The parse/derive method.

denote correspondence to rule 2 or rule 3.

$$\begin{aligned} a_2 \leq 0 \quad & a_2 = 1 \leftrightarrow x_2 \geq 1 \\ b_2 \leq y_3 \quad & b_2 = 1 \leftrightarrow y_2 \geq 1 \\ a_3 \leq x_2 \quad & a_3 = 1 \leftrightarrow x_3 \geq 1 \\ b_3 \leq 2 \quad & b_3 = 1 \leftrightarrow y_3 \geq 1 \end{aligned} \tag{5}$$

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 label frequency simplification used earlier. The longest pair of applicable sequences is considered optimal. From this pair all derived and parsed graphs are gathered.

Example 10: The optimal rule quantities identified by the MIQP are:

- Derivation from source graph $\vec{x}^*$: $0 \times P_2$ and $0 \times P_3$
- Parsing from target graph $\vec{y}^*$: $1 \times P_2$ and $2 \times P_3$

It is easy for the GA to determine an appropriate sequence for parsing, i.e., (p_3, p_3, p_2).

Finally, the most similar pair of derivation/parse graphs is chosen to obtain the resulting rule. Therefore, the pair of graphs which share the largest subgraph isomorphism is determined in a series of tests for isomorphism. A mapping between both graphs is established for every common node or edge. Elements that cannot be mapped will then be subject to the new rule's graph transformations.

Example 11: Figure 6 shows the rediscovery of rule 1 under presence of existing rules 2 and 3.

All steps of the parse/derive method are summarized in Fig. 7.

4. An illustrative example

The design graph grammar, which has been used as a baseline for experimentation, was originally developed by [17] to automatically generate propulsion system topologies for spacecrafts. The original grammar is capable of producing topologies for three different engine types, i.e., cold-gas systems, mono-propellant systems and bi-propellant systems. It has been deployed for "43" [1], a graph-rewriting-based design automation software. The integration with 43 allows designers to easily explore different topologies, which result from varying requirements. It also facilitates the calculation of associated parametrics, e.g., to determine masses associated with fuel and components.

In this work we focus solely on the topology generation for cold-gas systems. Figure 8 shows the flow schematic of a cold-gas system with two thrusters (left)

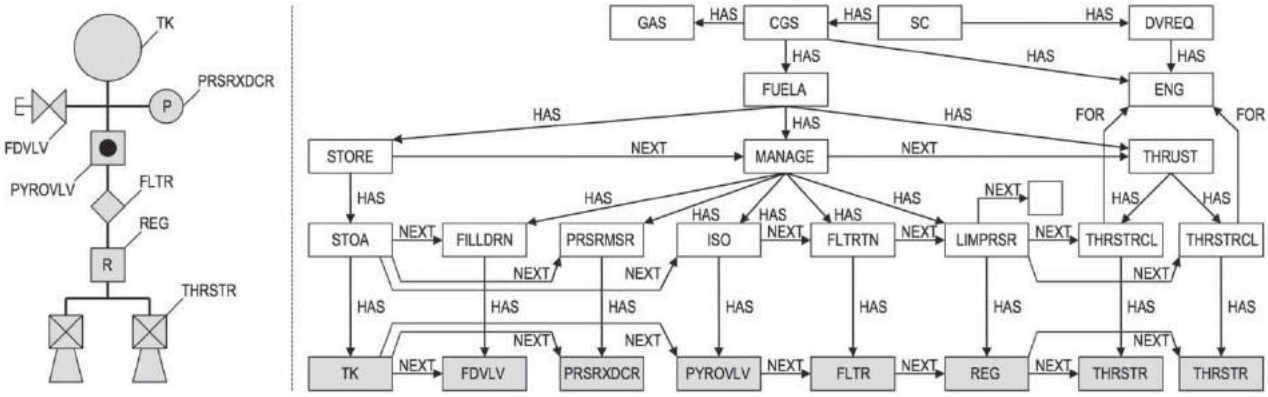

Figure 8. Left: typical flow schematics used for conceptual propulsion system design. Right: corresponding design graph representation, which results from graph-rewriting. Common elements within both representations are shaded. Used abbreviations are explained in text.

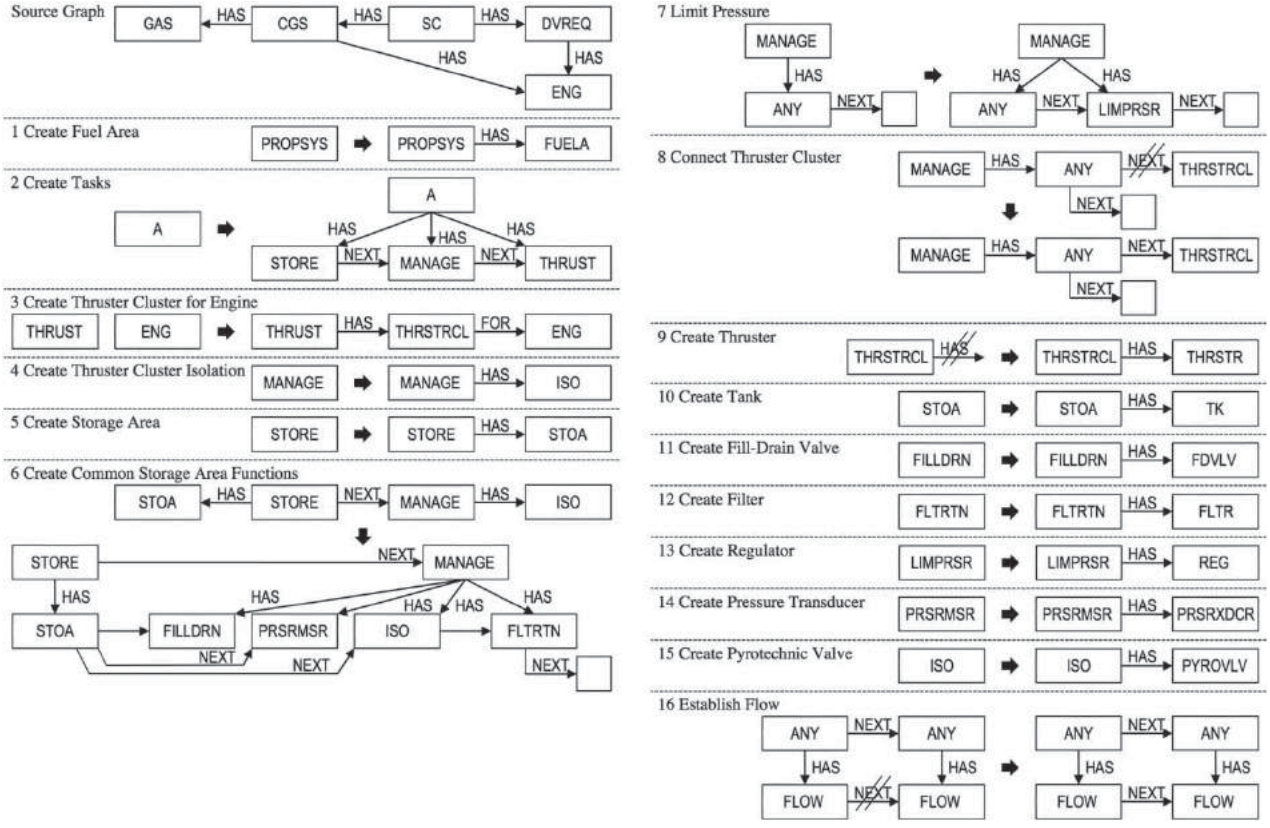

Figure 9. Source graph and complete rule set used for experimentation. Used abbreviations are explained in text.

and its corresponding graph representation (right). This sample design has been used as target graph for rule induction. Figure 9 shows the source graph (top left), which holds information about the basic requirements. In the original grammar there are several requirements associated with these nodes. They are part of equation systems, which are processed by solvers in parallel to derivation. We do not consider these parametrics for now and focus on topology generation. Figure 9 also shows the relevant subset of rules for deriving the target graph from the source graph using rule sequence $(p_1, p_2, p_3, p_3, p_4, p_5, p_6, p_7, p_8, p_8, p_9, p_9, p_{10}, p_{11}, p_{12}, p_{13}, p_{14}, p_{15}, p_{16}, p_{16}, p_{16}, p_{16}, p_{16}, p_{16}, p_{16})$. The first three rules of this set have already been explained in example 1. We now continue the derivation at rule 4 (for rules 1 to 3 see example 1).

Rule 4 begins with specifying functions for managing the propulsion system (MANAGE), i.e., it adds the function of isolating the thruster clusters from the fuel storage (ISO). With rule 5 the task of storing fuel is associated with a corresponding function, i.e., the provision of a storage area (STOA). Building on this initial layer

Table 3. Sequential dependencies. Application of 2nd rule is dependent on 1st rule's application.

	2nd Rule															
1st Rule	p1	p2	p3	p4	p5	p6	p7	p8	p9	p10	p11	p12	p13	p14	p15	p16
p1		•														
p2			•	•	•											
p3								•	•							
p4						•									•	
p5						•				•						
p6							•	•			•	•		•		
p7								•					•			
p8																
p9																
p10																•
p11																•
p12																•
p13																•
p14																•
p15																•
p16																

Table 4. Label frequency changes caused by application of rules.

Label	p1	p2	p3	p4	p5	p6	p7	p8	p9	p10	p11	p12	p13	p14	p15	p16
HAS	+1	+3	+1	+1	+1	+3	+1		+1							
NEXT		+2				+5	+1	+1								+1
FOR			+1													
FUELA	+1															
STORE		+1														
MANAGE		+1														
THRUST		+1														
THRSTRCL			+1													
ISO				+1												
STOA					+1											
FILLDRN						+1										
PRSRMSR						+1										
FLTRTN						+1										
LIMPRSR							+1									
THRSTR									+1							
TK										+1						
FDVLV											+1					
FLTR												+1				
REG													+1			
PRSRXDCR														+1		
PYROVLV															+1	

of functions, rule 6 continues functional decomposition by adding and connecting functions fill drain (FILLDRN), pressure measurement (PRSRMSR), and filtration (FLTRTN). It connects these functions with directed edges representing the direction of fuel flow within the system (NEXT). Rule 6 also adds an unlabeled node to the (current) end of the fuel flow signalizing following rules where to resume the extension of the flow. This actually happens with rule 7. It searches for Some node (ANY) that shows the mentioned pattern and attaches a function for limiting pressure (LIMPRSR). Rule 8 acts somewhat similar; it also looks for the last function within the fuel flow and connects the already added thruster clusters to this function. At this point, the layer of functions is complete. Now, derivation continues with detailing what flow components (FLOW) can be used for physically realizing the functions, i.e., in the terms of Systematic Design [16] we are going to derive a working structure (or principal solution) from the defined function structure. This happens with the remaining rules within the sequence. Rules 9 to 15 select a corresponding flow component for each function. Specifically, components tank (TK), fill drain valve (FDVLV), filter (FLTR), regulator (REG), pressure transducer (PRSRXDCR), and pyro valve (PYROVLV) are used for implementation. Rule 16 fulfills the special role of transferring the topology that has been defined on the function structure on to the flow components. By referring to a whole category of labels (FLOW), rule 16 actually encodes 49 basic rules resulting from the combination of the 7 labels in the category FLOW (see taxonomy in Fig. 4). This special property of rule 16 is considered in the following experiments.

The sequential dependencies among rules discovered within the extended critical pair analysis are shown in Table 3. Moreover, the direct applicability of rules on the source graph and target graph are as follows:

Table 5. Compilation of results for leave-one-rule-out experiments without rule 16. Additional rewiring operations, which do not correspond to an existing rule, are denoted by X.

Rule ID	Number of MIQP Optima	Matches with Original Rules	Time CP (ms)	Time MIQP (ms)	Time GA (ms)
1	1	1	1076	208	1934
2	1	2	1088	185	50186
3	29	3	1101	171	83999
4	4	4	1085	172	12122
5	4	5	1126	168	11709
6	1	6	1064	189	6740
7	5	7 + X	1061	194	13182
8	1	8	1033	164	3203
9	32	9	1093	232	55613
10	2	10	1091	206	1613
11	3	11	1103	232	5694
12	3	12	1107	147	5648
13	2	13 + 7	1102	130	7781
14	3	14	1122	190	5925
15	2	15	1084	233	2431

Table 6. Compilation of results for leave-one-rule-out experiments including rule 16. Additional rewiring operations, which do not correspond to an existing rule, are denoted by X.

Rule ID	Number of MIQP Optima	Matches with Original Rules	Time CP (ms)	Time MIQP (ms)	Time GA (ms)
1	12	1 + 2+3 + 8	18762	4081	548016
2	5	2 + 3+4 + 5+6 + 7+8 + 12 + 13	18165	4619	149215
3	21	3 + 8	17534	4646	1334476
4	1	4 + 6 + 7	18556	4109	106462
5	1	5	18990	4424	122475
6	10	6	13688	2591	764908
7	2	5 + 7 + X	17187	4250	166564
8	3	8	17589	308	131339
9	25	9	17834	2636	1348416
10	2	10	18619	5833	65652
11	1	11	19526	4705	67347
12	1	12	18567	4272	57568
13	1	13 + 7	18588	4810	79373
14	1	14	18845	3980	66459
15	6	15 + 4	19211	4313	178328
16	13	16 + 10 + 13 + X	1144	187	17608

- Applicability on host graph (derivation): p_1 unlimited, all other rules are not applicable
- Reverse-applicability on target graph (parsing): p_8 applicable 2 times, p_{16} applicable 7 times, all other rules are not applicable

From the grounding of rules, changes of label frequencies for each rule were computed. These are shown in Table 4.

The following subsections describe the results of experiments, where one rule was left out from the mentioned rule set, and the rule induction method was given the task to determine a proper replacement, such that the derivation of the given target graph from the given source graph is maintained. Each experiment was conducted twice, with and without consideration of rule 16. Hence, two different target graphs were used for experimentation. The one including the changes of rule 16 is shown in Fig. 8. The other target graph is identical to the latter except for the NEXT-edges among the shaded flow elements. Mixed integer quadratic programs were produced for each experiment using the data shown in Tables 3 and 4, where the data for the rule to be searched is excluded.

Table 5 summarizes the findings for the set of experiments associated with rule sequence ($p_1, p_2, p_3, p_3, p_4, p_5, p_6, p_7, p_8, p_8, p_9, p_9, p_{10}, p_{11}, p_{12}, p_{13}, p_{14}, p_{15}$). In most cases the rule definition that was originally used to derive the target graph could be rediscovered by the induction algorithm. The time needed for rule induction mainly depends on the number of optima found by MIQP optimization, and hence the number of GA invocations. The latter amounts the most computation time in comparison to the extended critical pair analysis (CP) and MIQP.

In case of rule 13 the induced rules also incorporate graph transformations that are actually part of rule 7, which is already present in the rule set. The induction of rule 7 represents another interesting situation, where the induced graph transformations basically correspond to those of the original definition, but a different

strategy for deriving the target graph is chosen. Rule 7 then has to perform some additional operations to change existing adjacencies. The results for rule sequence $(p_1, p_2, p_3, p_3, p_4, p_5, p_6, p_7, p_8, p_8, p_9, p_9, p_{10}, p_{11}, p_{12}, p_{13}, p_{14}, p_{15}, p_{16}, p_{16}, p_{16}, p_{16}, p_{16}, p_{16}, p_{16})$ are summarized in Table 6. Due to the increased complexity, the effect of "cannibalizing" existing rules is more prevalent.

We observed two recurring patterns where rules did not correspond to the original ones:

First, since the MIQP is just an approximation of the actual graph-rewriting behavior, the optimization problem may be under constrained in terms of sequential dependency. This yields derivation and/or parsing targets that do work in terms of label frequencies, but the actual rule application in derivation/parse does fail.

Second, the rules may be applicable in a different sequence compared to the original derivation while the label frequencies remain the same. In result, the graph differs in terms of topology. In this case a rule is learned that performs the necessary changes in order to re-establish the original derivation.

5. Conclusion

A method for inducing production rules in context of an existing but incomplete rule set has been proposed. This method is supposed to become an enabling technology to the use of graph-rewriting for implementing computer-aided design software. Supporting this vision, the methods applicability has been tested with respect to an example from the field of conceptual spacecraft design. The results gathered support the practical feasibility of the approach.

Nonetheless, further extensions are required to yield a comprehensive framework for automatically maintaining graph-rewriting systems in context of engineering design applications. Therefore, we suggest further research in to the refinement of induced rules by means of amalgamation (combination of rules) and generalization.

ORCID

Julian R. Eichhoff http://orcid.org/0000-0002-8748-8904
Jens Schmidt http://orcid.org/0000-0003-4049-1046
Dieter Roller http://orcid.org/0000-0002-2438-5676

References

[1] Alber, R.; Rudolph, S.: '43'—A Generic Approach for Engineering Design Grammars, Computational Synthesis: From Basic Building Blocks to High Level Functionality, Papers from the 2003 AAAI Spring Symposium, Vol. SS-03-02, 2003, 11–17.

[2] Ates, K.; Zhang, K.: Constructing VEGGIE: Machine Learning for Context-Sensitive Graph Grammars, 19th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2007), 2007, 456–463. http://dx.doi.org/10.1109/ICTAI.2007.59

[3] Costa, F.; Sorescu, D.: The Constructive Learning Problem: an efficient approach for hypergraphs, Workshop on Constructive Machine Learning (CML) at the 2013 Conference on Neural Information Processing Systems (NIPS 2013), 2013, 1–5.

[4] Eichhoff, J.R.; Roller, D.: A survey on automating configuration and parameterization in evolutionary design exploration, Artif. Intell. Eng. Des. Anal. Manuf., 29(4), 2015, 333–350. http://dx.doi.org/10.1017/S0890060415000372

[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, 1–8.

[6] 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, 011006-011006-10. http://dx.doi.org/10.1115/1.4032576

[7] Eichhoff, J.R.; Baumann, F.; Roller, D.: Reconstructing Design Processes by Machine Learning of Graph-Rewriting Production Rules, Proc. CAD'16, 2016, 157–161. http://dx.doi.org/cadconfP.2016.157-161.

[8] 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, 35–63. http://dx.doi.org/10.3233/FI-2015-1282

[9] Ehrig, H.; Golas, U.; Habel, A.; Lambers, L.; Orejas, F.: M-Adhesive Transformation Systems with Nested Application Conditions. Part 1: Parallelism, Concurrency and Amalgamation, Math. Struct. Comput. Sci., 24(04), 2014, 1–48. http://dx.doi.org/10.1017/S0960129512000357

[10] Fürst, L.; Mernik, M.; Mahnič, V.: Graph Grammar Induction as a Parser-Controlled Heuristic Search Process, Applications of Graph Transformations with Industrial Relevance (AGTIVE 2011), 4th International Symposium, 2011, 121–136. http://dx.doi.org/10.1007/978-3-642-34176-2_12

[11] Helms, B.; Shea, K.: Computational Synthesis of Product Architectures Based on Object-Oriented Graph Grammars, J. Mech. Des., 134(2), 2012, 021008. http://dx.doi.org/10.1115/1.4005592

[12] Jin, Y.; Li, W.: Design Concept Generation: A Hierarchical Coevolutionary Approach, J. Mech. Des., 129(10), 2007, 1012–1022. http://dx.doi.org/10.1115/1.2757190

[13] Kurtoglu, T.; Swantner, A.; Campbell, M. I.: Automating the Conceptual Design Process: 'From Black Box to Component Selection', Artif. Intell. Eng. Des. Anal. Manuf., 24(01), 2010, 49–62. http://dx.doi.org/10.1017/S0890060409990163

[14] Object Management Group: OMG Unified Modeling Language (OMG UML), 2015, http://www.omg.org/spec/UML/2.5/

[15] Object ManagementGroup: OMG Systems Modeling Language (OMG SysML), 2015, http://www.omg.org/spec/SysML/1.4/

[16] Pahl, G.; Beitz, W.; Feldhusen, J.; Grote, K.-H.: Engineering Design: A Systematic Approach, 3rd ed., Springer, London, 2007.

[17] Schmidt, J.; Rudolph, S.: Gaining System Design Knowledge by Systematic Design Space Exploration with Graph Based Design Languages, International Conference of Computational Methods in Sciences and Engineering (ICCMSE 2014), AIP Conf. Proc. 1618, 2014, 390–393. http://dx.doi.org/10.1063/1.4897755

[18] Schmidt, L. C.; Cagan, J.: Recursive Annealing: A Computational Model for Machine Design, Res. Eng. Des., 7(2), 1995, 102–125. http://dx.doi.org/10.1007/BF01606905

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

[20] Schmidt, L. C.; Shetty, H.; Chase, S. C.: A Graph Grammar Approach for Structure Synthesis of Mechanisms, J. Mech. Des., 122(4), 2000, 371–376. http://dx.doi.org/10.1115/1.1315299

[21] Siddique, Z.; Rosen, D. W.: Product Platform Design: A Graph Grammar Approach, ASME Paper No. DETC99/DTM-8762, 1999.

[22] Verein Deutscher Ingenieure: Methodik zum Entwickeln und Konstruieren technischer Systeme und Produkte (VDI 2221), Beuth, Berlin, 1993.