TAF Journal of Advances in Technology and Engineering Studies JATER
2017, **3**(4): 150-169

PRIMARY RESEARCH

In search of missing design rules: Using rule induction to extend existing rule bases

Julian R. Eichhoff [1*], Felix W. Baumann [2], Dieter Roller [3]

[1, 2, 3] University of Stuttgart, Stuttgart, Germany

Index Terms
Learning Problems
Induction Rules
Conceptual Design

Received: 24 February 2017
Accepted: 12 June 2017
Published: 21 August 2017

***Abstract*—** This paper focuses on a machine learning problem that is especially relevant for the automation of graph-based conceptual design. In this field, graph-rewriting systems can be used to facilitate design automation. The present work deals with the automatic induction of production rules for graph-rewriting systems from example design graphs. An approach to this is presented and empirically evaluated in the context of prototypical applications for functional design. The approach suffices the requirement that the learned rules have to fit an existing rule set, but the inspection of existing rule definitions is prohibited. The results suggest that the proposed approach is able to achieve reasonable, but sometimes unexpected learning results.

I. INTRODUCTION

RAPH-BASED models play an important role in product design. Particularly in conceptual design, graphs are used as abstract representation, e.g., to depict functionality, topology and physical relations among product components. Unified Modeling Language (UML) and its derivate for systems engineering, Systems Modeling Language (SysML) [1], for instance, are well known modeling frameworks that heavily rely on graph-based representations. Graph-rewriting is an expressive (Habel and Plump [2] showed that double-pushout graph-rewriting approaches supporting sequential composition and iteration are Touring complete) computation model that uses graphs for its data structure, and thus becomes a natural choice for implementing computed-aided conceptual design software [3, 4, 5, 6, 7, 8, 9, 10, 11]. The design compiler 43 [7], for instance, is a compre-hensive software using graph-rewriting to automatically generate geometry from abstract UML-based product descriptions. It provides interfaces to various Computer-Aided Design (CAD) applications, simulation software, databases (e.g., for storing data about stock components), and general-purpose mathematical computation software. Recently, Schmidt and Rudolph [12] used 43 to automatically create flow schematics for spacecraft propulsion systems.

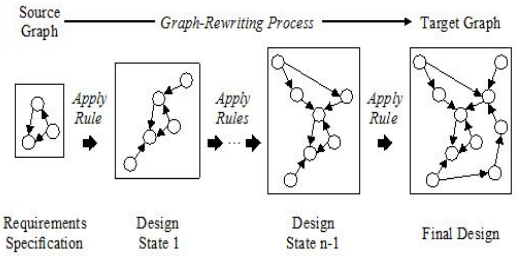

Fig. 1 . Simplified illustration of using graph-rewriting for CAD

Different propulsion system topologies are derived by simply changing initial parameters defined in the source graph. The graph-rewriting engine then incrementally builds up the topology. Graph-rewriting systems rely on a set of so-called production rules for deriving graphs. These rules are iteratively applied to generate a set of output graphs from an input graph. Speaking in the words of our domain: Production rules are encoded design actions,

*Corresponding author: Julian R. Eichhoff
†Email: eichhoff@informatik.uni-stuttgart.de

which are used to produce some target design models from a given design model in a series of transformations (see Figure 1).

This claim is supported by examining the duality between graph-rewriting and design. Design states are represented as graphs, where nodes typically denote the elements of a design (e.g., compo-nents and (sub) assemblies, but also functions, principles, modules, and requirements). Relations among these elements are denoted by edges (e.g., component-to-assembly, function-to-component, function-to-function, requirements tracing, etc.). Attributes of design elements or relations (used to define mass, price, geometric properties, etc.) are represented in terms of labels, which are assigned to nodes or edges respectively. A design decision is the choice of one design state from a set of possible states (design alternatives) based on information which is provided at the time of decision making. In graph-rewriting-based design systems, a design decision corresponds to the choice of a subsequent graph from a set of possible graphs based on information that is encoded in another, existing "host graph".

Such a move from one graph to another graph is called a direct derivation. The term derivation is used to denote a chain of multiple subsequent direct derivations, hence a sequence of design decisions. In graph-rewriting the reason for any direct derivation is the application of a production rule. Such a rule captures the changes that are made from one graph to another. It also defines constraints on possible host graphs, which must be fulfilled before a rule can be applied. What rules are considered for application at a certain point in derivation is either controlled by prewritten program or by a search framework that tests different sequences of rule applications. In graph-rewriting-based design automation this search is guided towards a sequence of rule applications that produces a design graph maximizing compliance with some given design goals, i.e., a form of design optimization. In this work we focus on one aspect thereof, namely the definition of production rules.

Hand-crafting production rules can be a tremendous effort, which is a well known problem in the field of expert systems, called the "knowledge engineering bottleneck". Consider functional decomposition [13] as an example: Here, the input model is a black box description of the product and the output models are possible decompositions, i.e. function structures, of that black box. Kurtoglu and Campbell [14] showed how to yield a rule set for functional decomposition by manually inspecting the commonalities among a set of function structures. Later, Kurtoglu et al. [10] extended their work in terms of a graph grammar for functional decomposition and component-to-function assignment. In order to reduce the effort on hand-crafting rules, our general aim is to automize the generation of production rule sets. Therefore methods from machine learning for rule induction are utilized. As we will shortly see, this field of research is not new. Nonetheless, most works consider the learning problem as a "blank page" and induce a completely new rule set from a set of given examples.

This paper, in turn, focuses on the extension of an existing rule set: How to automatically induce a set of rules that is capable of deriving a set of given output graphs from a given input graph while reusing as many rules as possible of an existing, yet incomplete initial rule set? We propose having such an adaptive machine-learning strategy facilitates a user's understanding of the rewrite-system's behavior. Especially because engineers are allowed to incorporate their own hand-crafted rules, and automatically induced rules are only added where needed. The present work proposes two related approaches for this task and compares their performance with respect to computational efficiency and quality of learning results. At the current stage we are interested in learning a single missing rule for two reasons: First, learning a single missing rule in context of others is the basis for inducing a set of rules. In section 6.1 ways are discussed on how to use the proposed single rule induction algorithms to induce multiple missing rules. Second, by learning a single missing rule one may actually learn a composite form of multiple rules.

In graph-rewriting theory this is called amalgamation [15]. It refers to the idea that multiple production rules can be joined to form a single rule, and vice versa, that one rule can be separated into a sequence of rules. The remainder of this paper is structured as follows: First, we provide fundamental definitions for labeled graphs and graph-rewriting systems. Then, we introduce an example application of graph-rewriting for functional decomposition, which is later used as benchmark for experimentation. What follows is a review of related work from the fields of grammar and rule induction and an outline of their limitations. After a formal discussion of the machine learning problem, an approach for solving it is proposed and its implementation using Genetic Programming (GP) is explained. The approach is empirically tested in a series of computer experiments and analyzed with respect to its learning performance and computational efficiency. The paper closes with concluding remarks on the results and future work.

ISSN: 2414-4592
DOI: 10.20474/jater-3.4.5

II. GRAPH-REWRITING PRELIMINARIES

Next is a series of definitions for labeled graphs and graph-rewriting systems from Eichhoff and Roller [16], which are mainly based on Ehrig et al. [15]:

Definition 1. 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 $s_G : V_G \rightarrow E_G$ and a target function $t_G : V_G \rightarrow E_G$ (for defining adjacencies and edge directions).
A node labeling function $l_G : V_G \rightarrow A_V$ and an edge labeling function $m_G : E_G \rightarrow A_E$.

Definition 2. A graph morphism *G→H* or morhpism for short is a map from graph G to another graph H. It consists of two functions $g_V : V_G \rightarrow V_H$ and $g_E : E_G \rightarrow E_H$ that preserve sources, targets and labels. A bijective graph morphism is termed graph isomorphism and denoted by ≅.

Definition 3. 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. A rule *p* can be equipped with an additional application condition $ac_L o$ over *L* in order to further restrict the application of the rule beside its occurrence morphism (see below).

Such a rule is written as $p = \langle ac_L, L \leftarrow K \rightarrow R \rangle$ (without loss of generality we suppose rules are defined with left-hand side application conditions only. Ehrig et al. [15] calls this "rules with left application condition"). Application conditions are tree-structured logical formulas over graphs.

Every formula within such a tree must be satisfied by a morphism before a rule with application condition can be applied. An application condition ac_Lover graph *L* and the satisfaction of $ac_L$by a morphism $o : L \rightarrow G$ are defined inductively following Ehrig et al. [15].

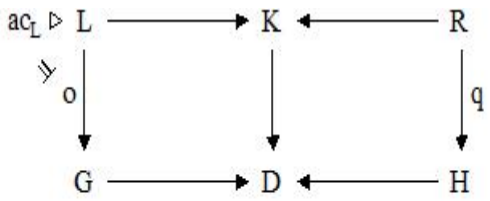

Fig. 2 . Double-pushout diagram. Arrows depict graph morphisms

Nesting: ∃ (a, $ac_{,G}$) is an application condition over L for every morphism $a : L \rightarrow G$ and every application condition ac_G over G'. Here, ac_G is a nested application condition. It introduces additional restrictions, which are themselves dependent on the outer application condition. A morphism $o : L \rightarrow G$ satisfies ∃(a, ac'_G), denoted $o \vDash \exists(a, ac_G)$ if there exists a morphism $b : G' \rightarrow G$ such that $b \circ a = o$ and b⊨ ac'_G.
Simple application condition: TRUE is an application condition over *L*. Every morphism satisfies TRUE. This is needed at the point where no further nesting of application conditions is required, i.e. ∃(a,TRUE).
Negation: For an application condition ac_L over *L*, $\neg ac_L$ is an application condition over *L*. A morphism $o : L \rightarrow G$ satisfies $\neg ac_L$ if o does not satisfy ac_L. Conjunction: For application conditions ac_L, i over *L* with i ∈ I (for all index sets I), $\wedge_{i \in I} ac_{Li}$ is an application condition over L. A morphism $o : L \rightarrow G$ satisfies $\wedge_{i \in I} ac_{Li}$ if it satisfies each ac_L, i.

Definition 4. The application of a rule p on graph G with respect to a certain occurrence $o : L \rightarrow G$, which results in graph *H*, is called a direct derivation and written $G \overset{p,q}{\Longrightarrow}$can be constructed, where *H*. It exists if and only if the double-pushout diagram of Figure. 2 *D* is termed context. In the case of p being equipped with ac_L over *L* then o⊨ ac_L must also hold for any corresponding direct derivation. The graph morphism $q : R \rightarrow H$ is termed co-occurrence. It is the process of replacing an occurrence of *L* in graph *G* with *R* leading to a new graph *H*. Thereby *K* and *D* contain those nodes and edges of the rule respectively the graph which are preserved during rule application.

Definition 5. A derivation $G \Rightarrow_p$ Hfrom graph G to graph H is a sequence of direct derivations $G \Rightarrow G_1 \Rightarrow G_2 \Rightarrow \ldots \Rightarrow H$ over a set of rules *P*= $\{p_1, p_2, \ldots\}$ In this work we suppose that graph-rewriting systems are terminating, i.e., there are no infinite sequences of direct derivations.

Definition 6. A graph-rewriting system is a pair *(P,G)* where *P* is a set of rules and G is an initial host graph used as starting point for derivations. The set *H* comprises all graphs that may be produced by derivations over *(P,G)* and is called the language of the graph-rewriting system.

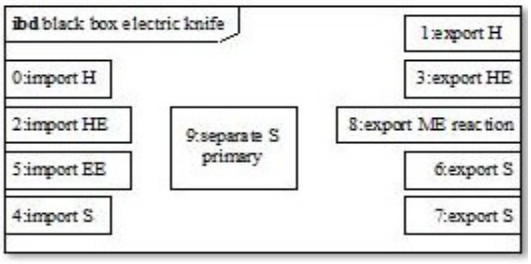

Fig. 3 . Black box model of the electric knife example. Depicted as Sys ML Internal Block Diagram. H = human, HE = human energy, EE = electrical energy, ME = mechanical energy, S = solid

ISSN: 2414-4592
DOI: 10.20474/jater-3.4.5

III. GRAPH-REWRITING FOR FUNCTIONAL DECOMPOSITION

The herein proposed rule induction algorithms were tested in context of a well-documented application of graph-rewriting for functional decomposition. Sridharan and Campbell [8] used graph-rewriting to define the functional model of an electric knife. The functional model is established in a process that mimics the functional decomposition procedure suggested by Pahl et al. [13]: From a simplified black box model that describes the knife's primary function (see Figure. 3), i.e., to "separate solids", a more complex model is derived. The latter is called function structure; it depicts what additional sub-functions are needed and how they must be related to each other in order to attain the desired primary function. Within the graphical notation, directed edges represent flows of signals, materials and forms of energy. The incident nodes represent functions, which consume, produce or otherwise affect the flows.

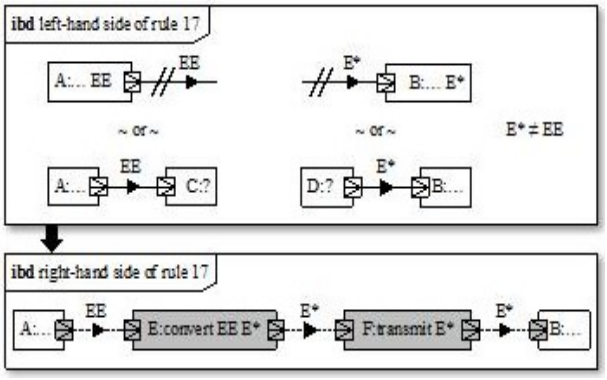

Fig. 4 . Rule 17. Left-hand side and right-hand side depicted as SysML internal block Diagrams. EE = electrical energy, E∗ = arbitrary kind

Figure. 3. Black box model of the electric knife example. Depicted as SysML Internal Block Diagram. H = human, HE = human energy, EE = electrical energy, ME = mechanical energy, S = solid. Sridharan and Campbell [8] defined a graph grammar to automate functional decomposition. To a great extent, the rules defined for this grammar were reused within the following machine learning experiments. Therefore the rules were re-implemented to match the prototype graph-rewriting system used for experimentation. The main difference with the rules' original formulation is the use of so-called elementary graph operations:

Definition 7. Let X=$X_V \cup X_E \cup X_A$ be the union of finite sets of variables, where $X_V = \{x_V | x_V \in V_G\}$and $X_E = \{x_E | x_E \in E_G\}$ are variables over the nodes and edges of some graph *G*. $X_A = \{x_A | x_A \in A\}$re variables over the label alphabet A. An elementary graph operation is a relation $f : \langle G, X\rangle \mapsto [\langle G', X'\rangle]$ that maps an input graph *G* under consideration of variables X to an output set of new pairs of graphs and variables $[\langle G', X'\rangle]$. Every pair in $[\langle G', X'\rangle]$ stands for an instance where the operation could be applied with respect to ⟨G,X⟩.

If the output set is empty, the operation could not be applied at all. Further, by writing $f : [\langle G, X\rangle] \mapsto [\langle G', X'\rangle]$ we denote that the elementary graph operation is applied multiple times on a set of input pairs. In consequence, $[\langle G', X1'\rangle]$ contains all sets of output pairs taken together as a union set.

Theorem 1. Let $f_p : G \mapsto [\langle G'_n, X'_n\rangle]$ be a relation that maps an input graph G to an output set of pairs of graphs and variables $[\langle G'_n, X'_n\rangle]$ by recursively applying a set of elementary graph operations $\{f_i | i = 1, ..., n\}$ in the order of increasing i:

$$f_p\,(\mathrm{G}) = \begin{cases} (f_i(\langle G_i = G, X_i\rangle) = [\langle G'_i, X'_i\rangle] : i = 1 \\ f_i([\langle G_{i-1'}, X_{i-1'} \cup X_i\rangle]) = [\langle G'_i, X'_i\rangle] : 1 < i \leq n \end{cases}$$

Further, let [H] be the answer set generated by taking together each H from direct derivations $G \overset{p,o}{\Longrightarrow} H$ over all possible occurrences *o* of rule *p*. Then f_p is an equivalent representation of rule p with respect to *G* if and only if the set of graphs $[G'_n]$ taken from $[\langle G'_n, X'_n\rangle]$ equals [H].

Using elementary graph operations for rule definition provides a unified representation for the left-hand side morphisms, right-hand side morphisms and application conditions. For the problem at hand, 10 different elementary graph operations were defined:

getNode: $\langle G, X \supseteq [x_A]\rangle \mapsto [\langle G' = G, X' = X \cup x_v \cup [x'_A]\rangle]$ shorthand notation: getNode $(x_V, x_(A, 1), x_(A, 2), ...)$. Selects a node x_V of G that matches some given labels $[x_A] \subseteq [x'_A]$ where $[x'_A]$is the complete set of labels for x_V.

noOutEdge: $\langle G, X \supseteq x_V, x_A\rangle \mapsto [\langle G' = G, X' = X\rangle]$ shorthand notation: noOutEdge(x_V, x_A). Requires a selected node x_V to have no outgoing edge with some given label x_A. noInEdge works the same except that it checks for incoming edges.

hasLabel: $\langle G, X \supseteq x_V, x_A\rangle \mapsto [\langle G' = G, X' = X\rangle]$; shorthand notation: noOutEdge(x_V, x_A). Requires a selected node x_V to have the given label x_A.

relatedLabels: $\langle G, X \supseteq \{x_{A,1}, x_{A,2}\}\rangle \mapsto [\langle G' = G, X' = X\rangle]$; shorthand notation: relatedLabels$(x_{A,1}, x_{A,2}$. Requires two given labels $x_{A,1}$ and $x_{A,2}$ to be taxonomically

ISSN: 2414-4592
DOI: 10.20474/jater-3.4.5

related, e.g., by means of the functional basis.

not Equal: $\begin{cases} \langle G, X \supseteq \{x_{A,1}, x_{A,2}\}\rangle \mapsto [\langle G' = G, X' = X\rangle] \\ \langle G, X \supseteq \{x_{V,1}, x_{V,2}\}\rangle \mapsto [\langle G' = G, X' = X\rangle] \end{cases}$

short hand notation: related Labels$(x_{A,1}, x_{A,2})$, related Labels$(x_{V,1}, x_{V,2})$. Requires two given labels $x_{A,1}, x_{A,2}$ or two given nodes $x_{V,1}, x_{V,2}$ to be different, s.t. $x_{A,1} \neq x_{A,2}$ or $x_{V,1} \neq x_{V,2}$ respectively.

useOutNT: $\langle G, X \supseteq \{x_A\}\rangle \mapsto [\langle G', X' = X \cup \{x_{V,1}, x_{V,2}\}\rangle]$; short hand notation: useOutNT $(x_{V,1}, x_{V,2}, x_A)$. Selects a node $x_{V,1}$ of G that is adjacent to a non-terminal (NT) node $x_{V,2}$ via an outgoing edge with the given label x_A. G' is a copy of G where the non-terminal node and the edge are removed. useInNT works the same except that the edge must directing towards $x_{V,1}$.

addNode: $\langle G, X \supseteq [x_A]\rangle \mapsto [\langle G', X' = X \cup \{x_v\}\rangle]$; short hand notation: get Node$(x_V, x_{A,1}, x_{A,2}, ...)$. Adds a node x_V with the given labels $[x_A]$. The resulting graph is G'.

addEdge: $\langle G, X \supseteq x_{V,1}, x_{V,2}, x_A\rangle \mapsto [\langle G', X' = X\rangle]$; short hand notation: addEdge $(x_{V,1}, x_{V,2}, x_A)$. Adds an edge of the given label directing from $x_{V,1}$ to $x_{V,2}$. The resulting graph is G'.

addLabel: $\langle G, X \supseteq \{x_V, x_A\}\rangle \mapsto [\langle G', X' = X\rangle]$; short hand notation: addLabel (x_V, x_A). Adds an additional label x_A to a selected node x_V. The resulting graph is G'.

addOutNT: $\langle G, X \supseteq x_{V,1}, x_A\rangle \mapsto [\langle G', X' = X \cup \{x_{V,1}\}\rangle]$; short hand notation: addOutNT $(x_{V,1}, x_{V,2}, x_A)$. Adds a non-terminal node $x_{V,2}$ and links it with a selected node $x_{V,1}$ via a new edge from $x_{V,1}$ to $x_{V,2}$ with the given label x_A. The edge directs towards the non-terminal and the resulting graph is G'. addInNT works the same except that the edge will be directing towards $x_{V,11}$.

In order to further ease the formulation, additional so-called choice operators were introduced. These operators can be used to form groups of elementary graph operations denoting alternative left-hand sides for a rule. This syntactic aid allows formulating slight variations of a rule combinedly. The operations that can be grouped are those addressing a rule's left-hand side, i.e. getNode, noOutEdge/noInEdge, hasLabel, relatedLabels, notEqual, and useOutNT/useInNT. Rule 17 is a more complex rule within the mentioned grammar. A graphical representation of the rule is shown in Figure. 4. Semantically, the rule provides function B with some required form of energy E* by converting Electrical Energy (EE) taken from function A into E* and transmitting it to node B. For instance, imagine an electro motor that converts EE into mechanical energy which is then transmitted to some other place within the product by means of a gear assembly. There are two optional ways in which nodes A and B are selected from the graph:

1. The primary flow of A (B) is EE (E*), but there is no outgoing EE-flow (incoming E*-flow) connected to A (B).
2. A (B) is an arbitrary node that is linked to a non-terminal via an outgoing EE-flow (incoming E*-flow).

The first selection pattern can be regarded as "push"-strategy where the rule actively searches for underdefined parts within the function model, whereas in the latter pattern the rule application is requested (or "pulled") by a non-terminal. The former pattern involves the use of an application condition to ensure that only nodes are selected which are not connected to some kind of edge. Another application condition of rule 17 is the constraint that for E* any kind of energy can be used except EE. Formulating the same rule in terms of elementary graph operations yields:

Choose: $\begin{cases} 1.1\Big(& getNode(x_{v,1}, "EE") \\ & noOutEdge(x_{v.i}, "EE") \\ 1.2\Big(& useOutNT(x_{V,1}, x_{V,98}, ""EE"") \end{cases}$

Choose: $\begin{cases} 2.1\Big(& getNode(x_{v,4}, x_a) \\ & noinEdge(x_{v,4}, x_a) \\ & useInNT(x_{V,4}, x_{V,99}, x_A) \\ 2.2\Big(& notEqual(x_A, "EE") \\ & relatedLabels(x_A, "E^*") \end{cases}$

$addNode(x_{v,s}"convert", "EE", X_a)$
$addNode(x_{v,3}"transmit", X_a)$
$addEdge(x_{v,1}, x_{v,2}, "EE")$
$addEdge(x_{v,2}, x_{v,3}, X_A)$
$addEdge(x_{v,3}, x_{v,4}, X_A)$

In the same manner, the rules mentioned in Sridharan and Campbell [8] having the identifiers 3, 4, 5, 6, 17, 20, 24, 25, 26, 27, 29, 33 were formalized. Starting from the black box shown in Figure. 3 and applying these rules in the sequence 26, 4, 29, 24, 27, 5, 25, 20, 3, 6, 17, 33 yields five valid function structures. The variation is due to the different possibilities for applying each rule with respect to the current host graph. Every graph produced throughout the derivation process is considered a valid function structure, if it complies the following requirements:

1. There are no non-terminal nodes left.
2. Each "import" function must have an outgoing flow.
3. Each "export" function must have an incoming flow.

ISSN: 2414-4592
DOI: 10.20474/jater-3.4.5

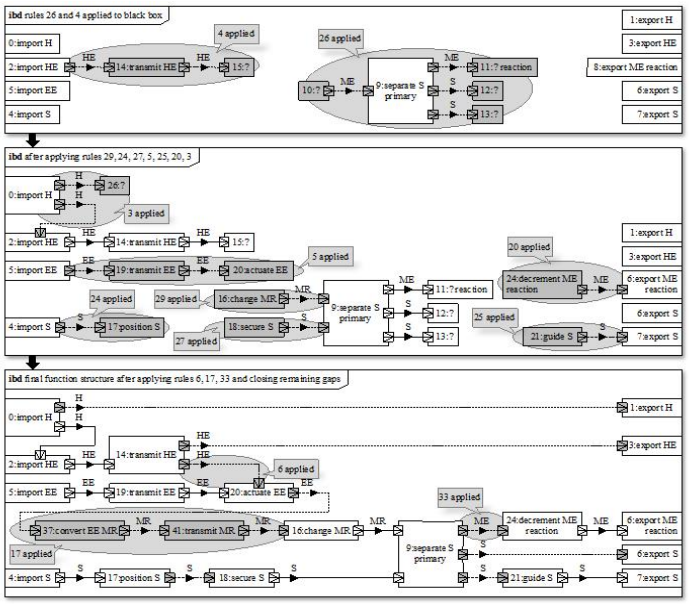

Fig. 5 . Derivation of one possible functional decomposition for the electric knife example. Depicted as a series of SysML Internal Block Diagrams. Starting point is the black box shown in Figure. 3. H = human, HE = human energy, EE = electrical energy, ME = mechanical energy, MR = rotational mechanical energy, MT = translational mechanical energy, S = solid.

Figure 5 shows the derivation process for the function structure that is also mentioned in Sridharan and Campbell [15]. Rules 3-6 are making the product an electrically operated device, which is turned on and off by its (human) user. Rules 26, 17, 29, 20 and 33 determine how the electrical energy enables the primary function: by converting electrical energy to mechanical energy, which is then used for cutting. Rules 24, 25 and 27 are used for handling the material that needs to be cut.

After the application of rule sequence 26, 4, 29, 24, 27, 5, 25, 20, 3, 6, 17, 33 follows a post-processing step. It uses a special “closing-gaps” rule (ID 16) that iteratively tries to add edges in order to make the graph a valid function structure.

IV. RELATED WORK

The literature surveys of [17] and Pappa and Freitas [18] point out that there has been a continuing interest in the problems of grammar and rule induction (also called inference) over the last 30 years. Though various contributions address the induction of context-free string grammars there has been little research on learning context-sensitive graph-grammars. Remarkably the early work of [19] was one of the few attempts to learn a context-sensitive grammar.

However, the formation of context on the left-hand sides is very restrictive. It is limited to certain graph templates (star or chain structures). The induction algorithm starts with trying to reproduce the first example with a minimum amount of rules. For that reason rules with right-hand sides of maximum size are preferred. In the following iterations the algorithm tries to reuse the current rule set and only adds further rules if necessary to reproduce the next example in queue. The induction algorithms of Jeltsch and Kreowski [20] and [21] start by producing a highly specialized initial grammar that covers all given training examples.

Therefore the initial rules' right-hand sides correspond to the training examples. In order to yield a more general grammar the existing rule set is iteratively decomposed towards rules that can be applied multiple times. Jeltsch and Kreowski [20] decompose rules according to edge-disjoint coverings in the right-hand sides. The applicability of rules can be further extended by renaming nonterminals. [21] decompose rules by searching right-hand sides for subgraphs that suffice a given rule template. The resulting rules are then reverse-applied changing the right-hand side of the original rule. Beside that the right-hand side of a rule can be reduced if it is unifyable with the right-hand side of another rule. Both works focus on hyperedge replacement systems, where Jeltsch and Kreowski [20] induce a context-free hyperedge grammar and [21] search for context-sensitive rules. However, the context of the latter is again strictly limited to predefined rule templates.

ISSN: 2414-4592
DOI: 10.20474/jater-3.4.5

The Subdue method [22] iteratively adds one rule at a time. At each iteration it searches for subgraphs that frequently appear in the training examples. From these subgraphs one is chosen as the right-hand side of the new rule. All appearances of the subgraph are then replaced by a non-terminal, which in turn is used as the left-hand side 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.

VEGGIE [23] is a Subdue derivate that adds rudimentary support for context-sensitive rules. Here, context is identified from overlapping subgraph instances. Given the part where instances overlap, new non-terminals are introduced that similarly connected. These form the context-sensitive left-hand side of two new rules. One adds the non-overlapping part in its right-hand side, the other turns the non-terminals back to terminals. Though rooted in different fields, [24] and Costa and Sorescu [25] share the idea of inducing rules by assessing frequencies of subgraph pairs in training examples and both induce context-sensitive rules. AGM [24] builds on association rule mining. Their induction algorithm starts with counting the instances of all possible subgraphs and computes their probability of appearance, called support. All subgraphs sufficing a certain support threshold are considered for rule generation. The conditional probability of G_2 appearing given that G_1 is present in the graph then determines the confidence that there is a rule with left-hand side G_1 and right-hand side G_2. Costa and Sorescu [25] also count frequencies of co-occuring subgraphs. However, the subgraphs which form a rule are further required to stand in a special core-interface-relation. A core subgraph consists of all nodes and edges that lie at a certain distance to some seed node. An interface subgraph in turn consists of nodes and edges that shield the core from the rest of the graph. Now a core can be replaced with another core if their corresponding interfaces match. Hence, the left-hand side and right-hand side both consist of core-interface-pairs with matching interfaces. None of these approaches to learning context-sensitive graph grammars is readily applicable under the mentioned restrictions as they are bound to limitations:

1. Left-hand sides were limited to rule templates [19], [21] or left-hand sides and right-hand sides were limited to single components [24], [25]. Hence, existing approaches are not capable of accepting multiple isolated subgraphs in their left-hand sides (e.g., to link these components with new edges).

2. All grammars have been induced from scratch. Completing a set of given (unknown) rules has not been considered. This complicates rule induction as existing rules may produce or require nodes, edges, or application conditions that cannot be inferred from the training examples. Consider two existing rules for instance, one producing a non-terminal the other consuming a different non-terminal. The rule to be learned may be required to "bridge the gap" between both rules' application by transforming one non-terminal into the other. However, this information cannot be directly read from the example graphs as they do not contain any non-terminals. In order to address these limitations machine learning can be conceptualized as a search over a space of possible hypotheses [26]. In this case hypotheses are candidates for rule p and the search space is defined by the vocabulary and syntax used for formulating rules. The goal of the learning algorithm is then to search for rule candidates that best describe the training data.

A. Problem Statment

Goal state

Let $(\hat{P}, G_0)$ be a graph-rewriting system that produces language $\hat{H}$ and let $[\hat{H}] \subseteq \hat{H}$ be a set of target graphs within that language, which is obtained by derivation over rule sequence $S = (p_1, p_2, ..., p_n)$. In terms of our domain, G_0 is, for instance, the black box model and $[\hat{H}]$ is the set of desired function structures.

Given State: Let $(\check{P}, G_0)$ be a graph-rewriting system that produces language $\check{H}$, where $\check{P} \subset \hat{P}$ and $[\check{H}] \cap \hat{H} = \theta$. Rule set $\hat{P}$ is unknown. Further, all rules of $\check{P}$ are confidential, i.e., for every $p \in P$ we are not in knowledge of the graphs $\{L, K, R\}$ and application condition ac_L used for defining $p = \langle ac_L, L \xleftarrow{\square} R \xrightarrow{\square} \rangle$. However, we are allowed to query derivation results, i.e., produced graphs, from $(\check{P}, G_0)$.

Learning Problem: In order to reach the goal state from the given state, additional rules P need to be added to yield an enhanced graph-rewriting system $(\check{P} \cup P, G_0)$ which is able to produce a language H such that $[\hat{H}] \subseteq$H. Hence, the learning problem is: What is a proper P for $(\check{P} \cup P, G_0)$?

A. *Decidability and Restrictions*

The above problem is equal to determining the reachability for each $H \in [\hat{H}]$given $\check{P} \cup P$as rule set. Follow-

ISSN: 2414-4592
DOI: 10.20474/jater-3.4.5

ing [27] we define the reachability problem:

Definition 8. Given a finite set of rules $\check{P}\cup pP$, an initial graph G_0 and a final graph H, the reachability problem is defined as follows: does $G_0 \xrightarrow{\square}_{\check{P}\cup P})H$ hold? If we can confirm that every $H\in[\hat{H}]$is reachable from G_0 using $\check{P}\cup P$ then P is a proper solution for our learning problem. For general graph-rewriting the reachability problem is undecidable [28].

However, in the case of finite-state graph-rewriting systems the reachability problem is decidable. Here, the set of possible graphs which can be derived from G_0 up to isomorphism is finite. In theory, H can be tested for isomorphism with every graph within that set to determine reachability.

Nonetheless, computing the complete set of derivable graphs can be become an issue of complexity. Reachability is also decidable for infinite-state graph-rewriting systems with special restrictions. At this point the interested reader is referred to [27], who investigated the decidability of the reachability problem with respect to different types of graph-rewriting systems. In this paper we impose the following restriction on the learning problem:

Restriction 1: $(\check{P}\cup P, G_0)$ is a finite-state graph-rewriting system.

Defining a finite-state graph-rewriting system is in itself an interesting problem, which is subject to current research. Recently Bisztray and Heckel [29] proposed an approach that combines previous approaches instead of using a single termination criterion. This is supposed to grant greater flexibility in the definition of a graph rewriting system. A review of previous approaches can also be found in Bisztray and Heckel [29]. However, in the following we will restrict the learning problem to a fixed application order to yield a finite-state graph-rewriting system, specifically:

Restriction 2: We suppose the sequence $S = (p_1, p_2, ..., p_n)$ of the goal state to be given, such that the application order of rules is known to the learner, but the definition of rules remains unknown. The rules to be learned *P* are one or more rules within that sequence. So far we are aware that the reachability problem is decidable for a certain finite set of rules $\check{P}\cup P$.

However, *P* is not fixed in context of the learning problem: Each rule $p \in p$ is chosen from a possibly infinite set of possible rule formulations *P*. Hence, the decidability of the learning problem is also dependent on characteristics of P.

Restriction 3: In order to maintain decidability of the learning problem, we require *P* to be finite. We now show how *P* is restricted to a finite set with respect to our domain specific language for functional decomposition:

1. Every left-hand side operation can only refer to elements that are part of a graph $G_i\in[G_i]$.
2. Every right-hand side operation that produces changes which cannot be undone by a later operation, can only refer to elements that are part of $\hat{H}_k$ but are not part of $G_i\in[G_i]$. Such persistent elements are termed monotonic elements. With respect to our functional design grammar all elements are monotonic except non-terminals and edges that link non-terminals to other nodes.
3. Non-monotonic elements, i.e., elements that are added temporarily and maybe removed in derivation, are not completely random.

Within the functional design grammar non-terminals function as placeholders, which are later replaced by edges. Hence, the set of possible non-terminals to be added is limited to the edges of $\hat{H}_k$ that are not part of $G_i\in[G_i]$.

VI. OPTIMIZATION-BASED APPROACH TO RULE INDUCTION

Given the above restrictions the learning problem can be formulated in terms of a combinatorial optimization problem over the language of possible rule sets. Therefore, let P^+ be the positive closure of P, which denotes the space of all possible rule sets to be considered, i.e., $P\in P^+$ Further, let $d \cong (\cdot,\cdot)$ be a similarity distance function that measures the difference of two sets of graphs, then the optimization problem can then be stated as follows:

$$P* = \frac{argmin}{P\in P^+} d \cong ([H], [\hat{H}])$$
$$[H]\subseteq H where(\check{P}\cup P, G_0) \mapsto H$$

In other words, we are searching a set of rules P^* that is, when added to the existing graph-rewriting system $(\check{P}, G_0)$, able to produce a set of graphs [H] that show the highest similarity with the target graphs $[\hat{H}]$. In the optimal case reachability is established when $\forall_{\hat{H}\in[\hat{H}]}(H \cong \hat{H}|H\in[H])$. This optimization problem is now subject to various re-formulations in order to yield a problem that can be readily implemented as computer program.

A. *Learning Multiple Rules*

First, let us consider the case where a single rule p_i needs to be added, i.e., $P^* = \{p_i\}$: As we are aware of the sequence of rule application $S = (p_1, p_2, ..., p_n)$, it is known at what point in derivation $i\in(1, 2, ..., n)$ the new rule p_i is supposed to be applied. Thus, we can pre-compute the set

ISSN: 2414-4592
DOI: 10.20474/jater-3.4.5

of potential host graphs $[G_i]$on which p_i is applied on by deriving over the subsequence from p_1 up to p_{i-1}:

$$P* = \frac{argmin}{P_i \in P} d \cong ([H], [\hat{H}])$$
$$[H] \subseteq H = \begin{cases} (\check{P}U\{p_i\}, G_0) \mapsto H : i = 1 \\ (\check{P}U\{p_i\}, [G_i]) \mapsto H : i > 0 \end{cases}$$

From this a scheme for learning multiple rules can be established: Case 1 - Subsequent rules: Now, suppose two or more rules of S are missing and these rules are directly following in sequence to each other, e.g., (p_{i-1}, p_i, p_{i+1})should be learned. This case is equal to learning a single rule that replaces the complete subsequence. The resulting rule is a composition of all rules in the subsequence, a so-called E-concurrent or parallel rule [15].

Case 2 - Two separate rules: Suppose that two rules of S are missing, and these rules are not directly following in sequence to each other, e.g., $\{p_{i-1}, p_{i+1}\}$ should be learned and p_i is an existing rule that should be reused. Solving the above optimization problem would lead to a composition of rules (p_{i-1}, p_i, p_{i+1}) as discussed within the first case. However, p_i is not supposed to be replaced by the learned rule. Yet, by simply adding the second rule to the optimization problem's search space, we are able to determine both rules in parallel while keeping any existing rules in between. The preparatory selection of host graphs for the second rule must select graphs from the preceding derivation results, e.g., the answer set resulting from the application of $(p_1, p_2, ..., p_{i-1}, p_i)$. This must be performed repeatedly, as the first rule is allowed to change as well.
Case 3 - Multi-rule-learning: Suppose there are more than two rules of S missing, e.g., $\{p_{i-2}, p_i, p_{i+2}\}$ are missing while $\{p_{i-1}, p_{i+1}\}$ are existing rules. Here, we repeatedly apply the processes of cases 1 and 2: First, p_{i-2} and a combined rule for (p_i, p_{i+1}, p_{i+2}) are learned. With a second invocation of the learner, the latter is refined by learning two separate rules for p_i and p_{i+2}. Repeating this process lends a scheme to iteratively learn multiple missing rules under consideration of existing rules.

B. *Learning Process*

Let there be m target graphs $H_k \in [\hat{H}]$ where $k \in (1, 2, ..., m)$. We obtain a simplified version of the above problem by considering one target graph at a time.

$$P^*_{i,k} = \frac{argmin}{P_{i,k} \in P} d \cong ([H], [\hat{H}])$$
$$H_k \in H = \begin{cases} (\check{P}U\{p_i\}, G_0) \mapsto H : i = 1 \\ (\check{P}U\{p_i\}, [G_i]) \mapsto H : i > 0 \end{cases}$$

In the worst case this simplification leads to the generation of a different rule for every $\hat{H}_k$, whereas in the best case the rule learned for one target graph is also capable of deriving the other target graphs as well. Using this we yield an algorithm for solving the learning problem.

C. *Reachability Learner*

Input: The graph-rewriting system $(\check{P}, G_0)$ that shall be enhanced; the application sequence of rules S; the position i of the rule that is to be learned; and the set of target graphs [Hˆ] that shall be reproduced.

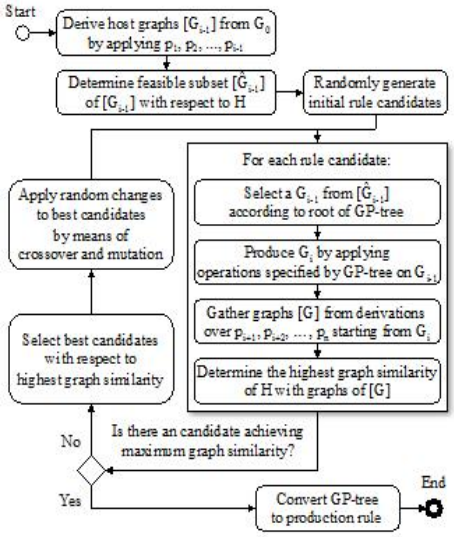

Fig. 6 . Suggested genetic programming implementation

Output: A set of rules P^*capable of reproducing [Hˆ] when used in $(\check{P} \cup P^*, G_0)$Let Let $K = \{k | \hat{H}_k \in [\hat{H}]\}$ While k∈K do Find a suitable rule $p^*_{i,k} for H_k$:

$$P^*_{i,k} = \frac{argmin}{P_{i,k} \in P} d \cong (H_k, \hat{H}_k)$$
$$H_k \in H = \begin{cases} (\check{P}U\{p_i\}, G_0) \mapsto H : i = 1 \\ (\check{P}U\{p_i\}, [G_i]) \mapsto H : i > 0 \end{cases}$$

Add $p^*_{i,k} to P^*$, and remove k from K End while.

VII. GENETIC PROGRAMMING IMPLEMENTATION

Genetic Programming [30] is a well known metaheuristic that builds on evolutionary principles for solving combinatorial optimization problems. It has proven suitable for large search spaces and is robust against local op-

ISSN: 2414-4592
DOI: 10.20474/jater-3.4.5

tima. In comparison to traditional genetic algorithms it uses a variable-sized tree representation of individuals. At this point the interested reader is referred to Wong and Leung [31] for a thorough introduction to GP and evolutionary search heuristics in general. The method's key concepts are its tree representation of possible solutions and the sampling of solutions, which is based on evolutionary principles. Following Eichhoff and Roller [32] we use GP to implement the reachability learner. Figure 6 summarizes all aspects of the genetic programming process.

A. *Representation of Rules*

The GP-tree is used to store two kinds of information:

1. On which host graph will the rule be applied on?
2. What elementary operations will be applied on the selected host graph?

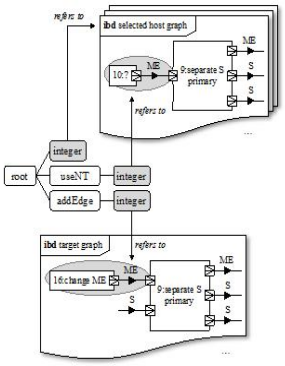

Fig. 7 . Example GP-tree of a rule candidate for rule 29 including references to host and target graphs. Host and target graphs are depicted as SysML Internal Block Diagrams. ME = mechanical energy, S = solid

The first question is addressed by the GP-Tree's root node. Using an integer number that is associated with the root node, a host graph for applying the rule candidate is chosen from a set of possible host graphs. Since the order of rule applications $S = (p_1, p_2, ..., p_{i-1}, p_i, p_{i+1}, ..., p_{n-1}, p_n)$ is known a priori, the set of possible host graphs is a subset of G_{i-1}, which is obtained by derivation $G_0 \Rightarrow S_1, G_{i-1}$ over subsequence $S_1 = (p_1, p_2, ..., p_{i-1})$. However, not all graphs in G_{i-1} are feasible host graphs for deriving target graphs [H]. If $G_{i-1} \in [G_{i-1}]$contains nodes or edges that cannot be removed in the following derivation, i.e., over subsequence $S_2 = (p_i, p_{i+1}, ..., p_{n-1}, p_n)$, and $H \in [H]$does not contain these nodes or edges, then H is not reachable from G_{i-1}. Hence, the GP-trees root node is only allowed to choose from $[\hat{G}_{i-1}]$, which is a filtered subset of possible host graphs where infeasible host graphs are excluded. The criteria for exclusion are determined by the language used for formulating rules. In our sample case function nodes and flow edges cannot be deleted from a graph once added (they are monotonic). Only non-terminal nodes are allowed to be removed. After having selected a $\hat{G}_{i-1}, G_i$ is initialized as a copy of $\hat{G}_{i-1}$. The remaining nodes of the GP-tree are then used to answer the second question. Each of these GP-nodes represents a single elementary operation of the kinds addNode, addEdge, addNT and useNT. Every such node has a single integer number associated, which is used to determine what node/edge is added/removed from G_i. This is done by selecting nodes and edges either from the selected host graph $\hat{G}_{i-1}$ or from the target graph H.

addNode: For all terminal nodes in *H* that are not yet present in G_i, add the one specified by the index node to G_i. The operation is not applicable if the set of addable nodes is empty. addEdge: For all edges in H that are not yet present in G_i, add the one specified by th index node to G_i. Incident nodes that are not present in $G_i$are added as well. The operation is not applicable if the set of addable edges is empty. addNT: For all edges in H that are not yet present in G_i, but where exactly one incident terminal node already exists in G_i, select the edge specified by the index node. Add a NT and the selected edge to G_i, where one end of the edge marks the existing terminal node, and the other is the newly added NT. The operation is not applicable if the set of edges to be considered is empty.

useNT: For all NTs in G_i, remove the one specified by the index node. The operation is not applicable if the set of NTs is empty. Now, in order to obtain the final G_i, i.e., the graph that results from applying the candidate rule, the initial copy of G_{i-1} is propagated through the GP-tree, where at each node the corresponding elementary operation is applied. Figure 7 shows an example GP-tree.Now, in order to obtain the final G_i, i.e., the graph that results from applying the candidate rule, the initial copy of G_{i-1} is propagated through the GP-tree, where at each node the corresponding elementary operation is applied. Figure 7 shows an example GP-tree.

B. *Evolutionary Sampling of Rule Candidates*

In GP a sample of solutions is called population. With each iteration of the GP a new generation of this population is produced by means of selection, recombination and mutation operators.

ISSN: 2414-4592
DOI: 10.20474/jater-3.4.5

Selection determines which solutions, or individuals, of the current generation should be used to determine the next generation. This choice is based on the value of the goal function an individual achieves. In the context of GP we also speak of the fitness of an individual referring the to idea of "survival of the fittest". Mutation and crossover operators are then applied on the fittest solutions, a process called breeding: Mutation randomly modifies parts of an existing solution to form new individuals for the next generation. Recombination, or chromosome crossover, randomly joins parts of two solutions (the parents) to create a new individual.

C. *Fitness Evaluation*

The selection of individuals is based on a single fitness evaluation criterion, i.e., a measure of similarity between derived graphs and target graphs. It ranges in the interval [0,1] where 0 denotes the highest similarity. The implementation of this measure is a crucial aspect for the learner's efficiency. A stepwise approach to measure the similarity of graphs has been implemented. As long as the minimal distance 0 has not been reached, the following steps are repeated for every graph G derived from G_i: First, we check if G is isomorphic to H for the reachability approach, or if H contains G (subgraph isomorphism) for the coverability approach, respectively. If this is true, the state of highest similarity 0 is reached. Otherwise, G and H are compared with respect to their label frequency tables. In case of different frequency tables, their Quadratic-Chi distance is used to give a similarity value ranging within [0.75,1]. If the frequency tables are identical, a graph similarity algorithm is used to compare both graphs with respect to their labels and their topology. Here, we used the algorithm of Nikolić [33, 34, 35] to produce a value within [0.5,0.75].

Graph Similarity

Input: G_i resulting from GP-tree-propagation; remaining sub-sequence S_2.

Output: Value of the minimum distance between target graph H and the set of graphs resulting from derivation Initialize similarity value a=1 Derive remaining subsequence S_2 starting from G_i and gather all produced graphs [G]

For $G \in [G] do$

If $H \cong G do$

Return 0

Else Compute frequency tables h(G) and h(H) over the graphs' node and edge labels

Let $b = d_2(h(G), h(H))$ be the Quadratic-Chi distance of both frequency tables If b≠0

$a = min(a, 0.75 + \frac{b}{4})$

Else

Let c be the distance value resulting from applying the graph similarity algorithm of Nikolić [33] on G and H

$a = min(a, 0.5 + \frac{c}{4})$

End if

End if

Return a

End while

D. *GP-Tree to Rule Conversion*

In the follow-up to GP, the best found individual is converted from GP-tree representation to a production rule. The conversion of GP-tree nodes to elementary operations is straight forward: Any node of the GP-tree (except the root) either refers to elements of host graph G_{i-1}, elements of target graph H, or elements of both graphs. In order to use elements of G_{i-1}, e.g., to get a reference on existing nodes or to remove non-terminals, corresponding left-hand side operations are added. Conversely, if a GP-node refers to elements of H, any corresponding right-hand side operation is added.

E. *Multi-Rule-Learning*

The implementation of the multi-rule-learning scenario as described in the problem statement requires a few adaptions to the process: First, with each invocation of the GP-learner two rules are learned in parallel. This corresponds to case 2 of learning multiple rules as described in the problem specification. Specifically the learner determines definitions for rules p_i and p_j of sequence $S = (p_i, p_{i+1}, p_{i+2}, ..., p_{j-1}, p_j)$, where rules $(p_{i+1}, p_{i+2}, ..., p_{j-1})$ are existing rules. Therefore two GP-trees of the mentioned kind are evolved in parallel, one for p_i and one for p_j. Further, if the sequence of existing rules $(p_{i+1}, p_{i+2}, ..., p_{j-1})$ is unknown, an additional, simple GP-tree is added to select what existing rules are applied and in what sequence. Second, after a valid solution is found being capable to reach the target graph, the second induced rule p_j is refined with another invocation of the GP-learner. This follows from case 3 of learning multiple rules. However, instead of using the original source graph, at this time the result from deriving over $(p_i, p_{i+1}, p_{i+2}, ..., p_{j-1}, p_j)$, i.e., the host graph

ISSN: 2414-4592
DOI: 10.20474/jater-3.4.5

of p_j, is used as new source graph. If a valid solution can be found for this new induction problem as well, then the initial result for p_j is discarded and the two new induced rules together with the next sequence of existing rules is used in replace. The process can be stopped when no more existing rules can be reused. Figure 8 illustrates this process. Third, two extra optimization goal are added. One of these goals aims at maximizing the applicability of existing rules, such that one candidate solution is preferable over the other if it enables more existing rules to be applied. However, this applicability score is governed by the similarity with the target graph, which yields the following formula:

$$(1 - \frac{1}{|S|}) \cdot d \cong (H, \hat{H})$$

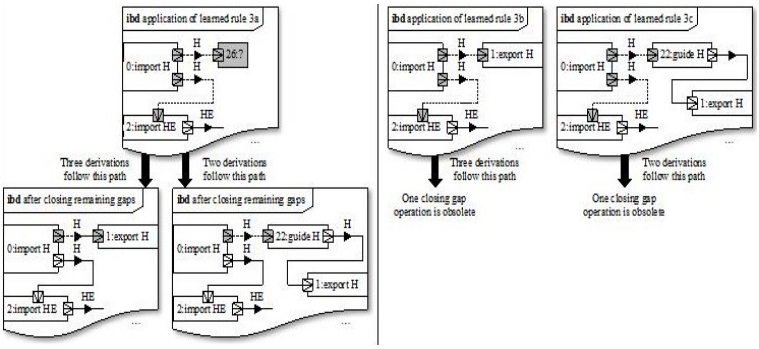

Fig. 8 . Differences in the generality of rules and the effect of cannibalization exemplified with respect to varying definitions of rule 3. Application of original, general variant (3a) is depicted on the left. Applications of two learned variants (3b, 3c) are shown on the right. The latter are more specific and need to be used in combination to derive all target graphs. Modified graphs are depicted as SysML Internal Block Diagrams. H = human, HE = human energy.

Where |S|≥1 denotes the number of rules of sequence S which were actually applicable (note that at least the first rule is applicable, since it is being induced). The third optimization goal is a parsimony score. It prefers candidate solutions with smaller GP-trees over solutions with larger ones. Here, the three GP-trees are not treated equally. Specifically, the GP-tree defining the first rule p_i is affected from this score, since we are interested in inducing minimal rules while reusing existing rules whenever possible (see cannibalization effect in results section). Nonetheless, overly large rule sequences are also penalized by this score in order to avoid unnecessary rule applications. In comparison with p_i, however, the parsimony pressure on the rule sequence is relatively small (factor 1:100). The second induced rule p_j is not subject of the parsimony score, as this rule will be refined in the next iteration of the GP-learner.

VIII. SINGLE-RULE-LEARNING EXPERIMENTS

The graph grammar of Sridharan and Campbell [8] was used as benchmark for an initial batch of computer (Intel(R) Core(TM) i7-5930k CPU @ 3.50 GHz, 64 GB RAM @ 2133 MHz, Windows 10) experiments. In each experiment one rule of this grammar was excluded, and the learners were given the task to find definitions for the missing rule such that a set of five given function structures can be derived from the black box model. Every experiment was repeated 30 times. The mean number of generations and elementary op-erations computed within each run serve as measure for the computational effort. To quantify the similarity with the original rule formulation, the left-hand side occur-rences and right-hand side co-occurrences were com-pared by means of the same graph-similarity algorithm described earlier. Next, we discuss the quality of learning results, whereas Table 1 provides a summary of the computational efforts.

A. *Generality of Rules*

We quantify the generality of a rule by the number of target function structures the rewrite system is capable to derive using the rule. With respect to the present experiments a rule shows maximum generality if it supports the derivation of all five target function structures. The original rules always have this property as they were used to generate the set of target function structures. However, rules found by the GP learners differed in terms of generality. Rule 25, for instance, is supposed to attach a guide-M^* node in front of an export-M^* node, where M^* is some arbitrary kind of material. This applies to both, the export-H

ISSN: 2414-4592
DOI: 10.20474/jater-3.4.5

and export-S nodes. In the original formulation a label-variable in combination with a relatedLabels operation is used to realize this behavior. The GP-learner, in contrast, identifies two specific rules: one for the export-H case and the other for export-S.

B. *Cannibalization*

It may happen that the learner discovers a rule which fulfills its intended purpose but additionally incorporates elements that are meant to be added/removed by another rule. Consider rule 3 for example (see Figure. 9): The task of rule 3 is to add an edge from import-H to import-HE and to ensure that there will be a second outgoing H-flow from import-H. In the original formulation this is done by adding an outgoing NT to import-H signalizing following rules to add the edge. In most cases this rule definition was learned (see rule 3a in Figure. 9). An alternative strategy found by the GP learner was to include the latter task as well. Here, two different rule formulations are needed (see rules 3b and 3c in Figure. 9): One rule establishes an H flow from import-H to export-H. It can be used to derive three out of the five function structures. The remaining two function structures have a guide-H put right in front of the export-H. Hence, a second rule is learned that links import H with guide H.

C. *Swapping of Duties*

We further observed an effect we term the swapping of duties. The effect appears if there is a rule succeeding the rule to be learned and this existing rule is able to perform some of the operations which are actually supposed to be part of the learned rule. In this situation the existing rule acts as partial substitute. However, since we consider the rule sequence to be fixed, the original task of this substituting rule will not be accomplished anymore. This is where the learner sets in and determines a new rule which compensates for the original rule's missed duties. This effect is prevalent for rule 5 for instance. The purpose of rule 5 is to add transmit-EE and actuate-EE nodes. Further, in its original formulation, it is supposed establish a directed path from import-EE via transmit-EE to actuate-EE. From all learned definitions of rule 5, this was the frequent variant. Yet, several other variants were learned that differ with respect to the added edges. Since the close gaps rule is capable of adding the required edges as well, rule 5 just has to add any two edges which are needed in the target graph. Specifically, the found rules added edges between secure-S/separate-S, separate-S/export-S, position-S/secure-S, position-S/export-S, transmit-HE/export-HE. Here the duties are swapped with the very general close-gaps rule: While the learned rule accounts for some of the links that need to be established to yield the final function structure, the close-gaps rule takes care of completing the mentioned EE chain at the end of derivation.

TABLE 1
RESULTS OF LEAVE-ONE-RULE-OUT EXPERIMENTS

	Generations		Elementary Operations		Similarity with Original Rule	
Rule ID	Mean	Std.Dev.	Mean	Std.Dev	Mean	Std.Dev
256 Rule Candidates per Generation						
3	3.18	2.53	21784.3	12136.57	0.8	0.14
4	2.57	1.57	39349.03	30926.80	0.53	0.03
5	4.48	2.04	82132.41	42074.77	0.5	0.34
24	1.5	0.77	31551.52	16787.68	0.8	0.23
26	26.97	26.38	174748.13	88550.17	1	0
27	1.44	0.66	32934.56	14705.61	0.81	0.23
29	5.45	5.71	90898.52	58852.95	0.98	0.09
32 Rule Candidates per Generation						
6	6.26	6.33	5047.49	3470.81	0.97	0.11
17	4.13	2.41	1505.99	798.26	0.41	0.16
20	4.21	3.56	4553.37	2544.57	0.67	0.4
25	5.03	4.69	5165.8	3118.38	0.75	0.27
33	1.43	0.88	525.45	195.99	0.3	0.27
n=30						

ISSN: 2414-4592
DOI: 10.20474/jater-3.4.5

IX. MULTI-RULE-LEARNING EXPERIMENTS

We are now going to switch to a design case from conceptual spacecraft design to illustrate the methods capabilities in a more complex and practical setting. Schmidt and Rudolph [12] used the graph-rewriting-based design compiler 43 [7] to create a parametric design model for spacecraft propulsion systems. Here, graph-rewriting rules are used to incrementally build up a configurational design starting from an initial requirements definition. With a set of 62 rules in total, the graph-rewriting system is capable of deriving various variants of cold-gas (CGS), mono-propellant (MONO) and bi-propellant (BI) type propulsion systems.

Example flow schematics of such systems are shown in Figure. 10. The graph-rewriting system starts from a graph thats is used to represent the mission re-quirements for the propulsion system. It then follows the principles of functional decomposition to identify suitable component configurations that fit these requirements. This graph-rewriting system has been used as gold standard for rule induction experiments with the pro-posed methods. Therefore we assume the situation of an engineer who has already prepared a graph-rewriting system for deriving cold-gas propulsion systems, and now wants to extend this system to derive (more com-plex) mono-propellant and bi-propellant systems as well. This situation is reflected by the following rule induction problem. There is an existing graph-rewriting systems consisting exactly of those rules which are necessary for deriving cold-gas propulsion systems.

Example design graphs for mono-propellant and bi-propellant propulsion system are available, however, the current rule set is insufficient for deriving these kinds of propulsion systems. The rule induction problem is now to extend the current rule set, such that the derivation of the mono-propellant and bi-propellant examples becomes possible. The source graph for these new derivations consists of a single SC node representing the space craft. The cold-gas system is the result of the derivation process illustrated by Figures. 11 and 12. What follows is a description of the rules applied therein: CG1 Create Fuel Area: starts transforming the source graph by adding a node, which represents the installation space used for the propulsion system (FUELA).

CG2 Create Tasks: adds the tasks which should be fulfilled by the components installed in the installation space (STORE, MANAGE, THRUST) and imposes an order on the sequence for executing these tasks.

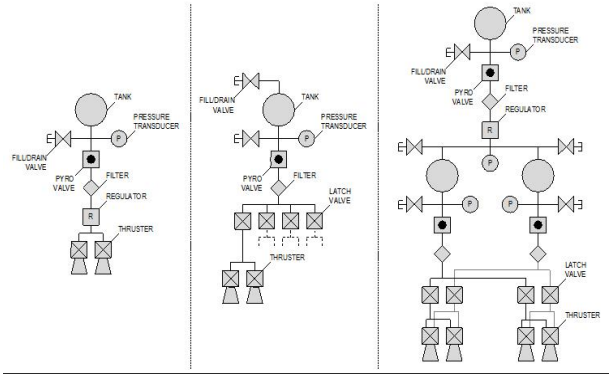

Fig. 9 . Flow schematics of a cold-gas (left), mono propellant (center) and bi propellant (right) type propulsion system

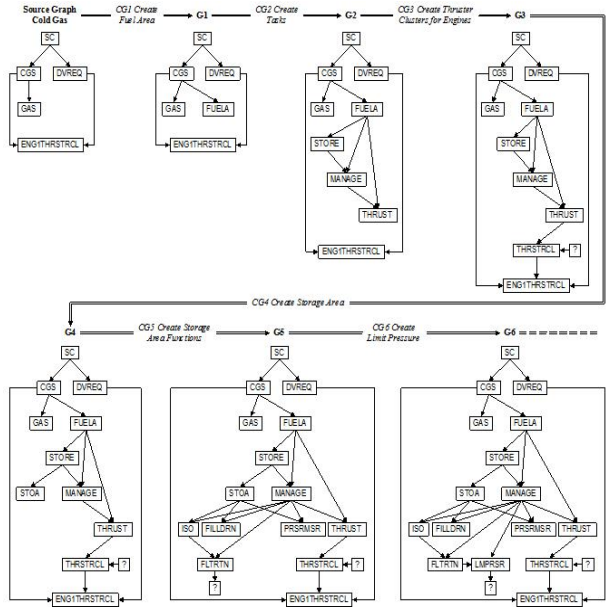

Fig. 10 . Derivation of a cold-gas propulsion system. Edge labels omitted

CG3 Create Thruster Clusters for Engines: is then used to further specify how THRUST is generated. In this case, the rule is applied once in order to generate one thruster cluster (THRSTRCL). An additional link to the engine node (ENG) is drawn establishing traceability with the associated delta-v requirement (DVREQ). CG4 Create Storage Area: associates the task of storing fuel with a corresponding function, i.e., the provision of a storage area (STOA). CG5 Create Storage Area Functions: continues func-tional decomposition by adding and connecting functions fill/drain fuel (FILLDRN), pressure measurement (PRSRMSR), isolation (ISO) and filtration (FLTRTN). It connects these functions with directed edges representing the direction of fuel flow within the system.

ISSN: 2414-4592
DOI: 10.20474/jater-3.4.5

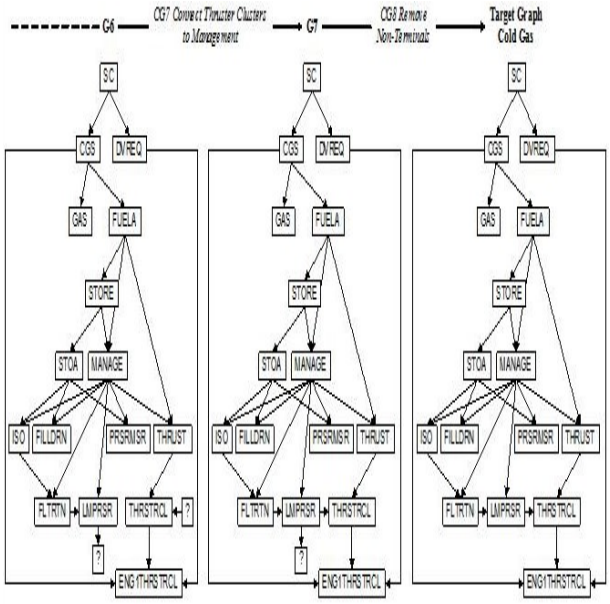

Fig. 11 . Derivation of a cold-gas propulsion system (continued). Edge labels omitted

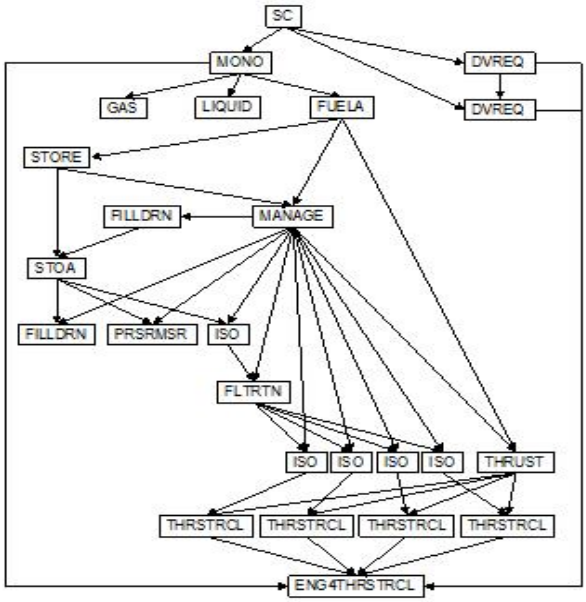

Fig. 12 . Target graph of the mono-propellant propulsion system. Edge labels omitted

CG6 Create Limit Pressure: attaches an additional function to limit pressure (LIMPRSR) after filtration. CG7 Connect Thruster Clusters to Management: selects the yet unconnected ends of the fuel flow and establishes the missing link.

In this case, the limit pressure function and the single thruster cluster are recognized as ends of the fuel path which are to be connected. CG8 Remove Non-Terminals: deletes the now obsolete non-terminals (?), which were used for denoting the ends of the flow path. In a final step, the generated topology of sub-functions is directly transformed into a component configuration. A set of additional rules is responsible for this transformation and chooses a realizing component for each sub-function. In the case of the cold-gas propulsion system (see Figure. 10 left) the following function-to-component assignment is used: ISO ↦ pyro valve; FILLDRN ↦ fill/drain valve; PRSRMSR ↦ pressure transducer; FLTRTN ↦ filter; LIMPRSR ↦ regulator; THRSTRCL ↦ an assembly of thrusters. However, the latter component assignment step was not part of the experiments conducted for the present paper. The rule induction tasks were limited to the preceding functional decomposition part, such that the learner had to extend the rule set CG1, CG2, ... CG8 with additional rules, which make it possible to reach the target graphs shown in Figure. 13 and Figure. 15.

A. *Extending Rule Set to Derive Mono-Propellant Systems*

In the first experiment, the goal was to determine a graph-rewriting system that is capable of deriving the mono-propellant propulsion system represented by the graph shown in Figure. 13. This target graph had been generated using rules CG1, CG2, CG4, CG5, CG7 and CG8 of the cold-gas derivation together with additional rules which are specifically used for mono-propellant type systems. The rule sequence used for derivation is (CG1, CG2, MONO1, MONO2 (×4), CG4, CG5, MONO3, CG7 (×4), CG8). Given the target graph and rules CG1, CG2, ... CG8, the algorithm for learning missing rules without knowing the actual rule sequence has been applied.

Specifically, the algorithm was given the limit to use at maximum two iterations of the GP procedure to induce suitable missing rules. Furthermore, GP was parameterized to use a population of 256 individuals and the maximum number of generations was set to 1000. The aggregated results over 30 repetitions of this ex-periment are shown in table 2. Thus, the procedure GP has been invoked 60 times in total. In 11 cases GP exceeded the maximum number of generations and did not come up with a valid solution. However, all of these cases occurred with the second GP iteration, where the first iteration already came up with a solution. Due to the randomization operations which are used by GP, a high variance in the running times and the number of reused existing rules can be observed. Taking

ISSN: 2414-4592
DOI: 10.20474/jater-3.4.5

both GP iterations together and computing the mean over 30 repetitions, one rule induction run required 661 GP generations, 8.2M direct derivations and took 35 minutes. The mean number of existing rules reused is 4. Figure 14 depicts the distribution of reused rules. The largest set of reused rules is CG1, CG2, CG4, CG5, CG8. This is also the most frequently appearing induction result (30% of the experiment runs). What follows is an illustration of one example from this group of results: The derivation starts from the source graph, which consist of a single SC node representing the space craft that is to be equipped with a propulsion system. The first induced rule is applied on this source graph adding and connecting the mono-propellant propulsion system (MONO).

TABLE 2
RESULTS OF MULTI-RULE-LEARN EXPERIMENT FOR MONO-PROPELLANT DESIGN CASE

	Generations		Duration (s)		Elementary Operations		Reused Rules	
Iteration	Mean	Std.Dev.	Mean	Std.Dev.	Mean	Std.Dev	Mean	Std.Dev
1st	235.70	126.10	67.11	37.58	881,039.80	612,292.21	2.73	1.39
2nd	425.20	446.58	2,021.71	3,026.38	7,320,093.53	8,708,223.69	0.87	1.01
Both n=30	660.90	494.34	2,088.82	3,047.99	8,201,133.33	8,962,664.10	3.60	1.33

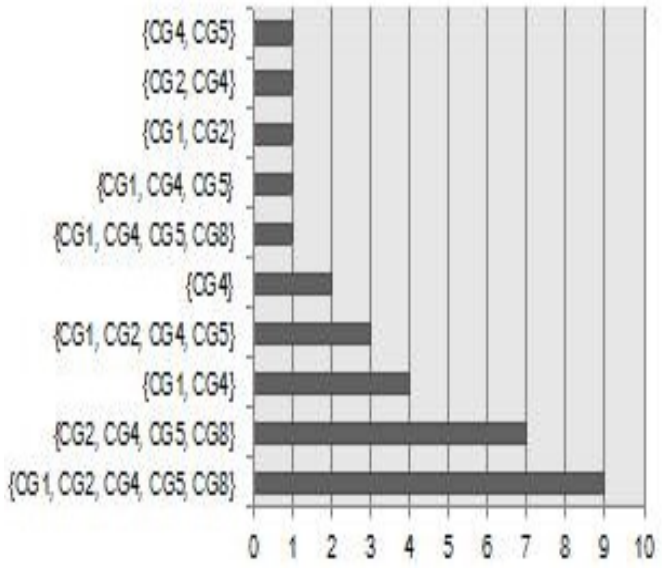

Fig. 13 . Frequencies of reused rule combinations for mono-propellant design case (n=30)

On the resulting graph the existing rules 1, 2 and 5 are reused to add the fuel area node (FUELA), to do the initial functional decomposition into tasks STORE, MANAGE, THRUST, and to add the storage area (STOA) function respectively. After these steps, the next induced rule is applied:

$$\frac{getNode(x_{V,1}, "SC")}{\begin{array}{l} addNode(x_{V,2}, "MONO") \\ addEdge(x_{V,1}, x_{V,2}, "PROPSYS") \end{array}}$$

$$\frac{\begin{array}{l} getNode(x_{V,1}, "MANAGE") \\ getNode(x_{V,2}, "THRUST") \end{array}}{\begin{array}{l} addNode(x_{V,3}, "ISO") \\ addOutNT(x_{V,1}, x_{V,98}, "FUE") \\ addInNT(x_{V,3}, x_{V,99}, "NXTFUE") \\ addNode(x_{V,4}, "THRSTRCL") \\ addNode(x_{V,5}, "ISO") \\ addEdge(x_{V,1}, x_{V,5}, "FUE") \\ addEdge(x_(V, 2), x_(V, 4), ""FUE") \\ addEdge(x_{V,1}, x_{V,3}, "FUE") \\ addNode(x_{V,6}, "THRSTRCL") \\ addEdge(x_{V,3}, x_{V,6}, "NXTFUE") \end{array}}$$

Then, existing rule 6 is applied to add the common storage area functions (FILLDRN, PRSRMSR, ISO, FLTRTN), and rule 50 removes the LASTFUE non-terminal which was added by rule 6. All remaining elements to complete the target graph (cf. Figure. 13) are added by the last induced

$$\text{rule: } \frac{\begin{array}{l} useOutNT(x_{V,1}, x_{V,98}, "FUE") \\ getNode(x_{V,2}, "FLTRTN") \\ getNode(x_{V,3}, "ISO") \\ getNode(x_{V,4}, "MONO") \\ getNode(x_{V,4}, "ISO") \\ getNode(x_{V,6}, "SC") \\ getNode(x_{V,7}, "THRUST") \\ getNode(x_{V,8}, "THRSTRCL) \\ getNode(x_{V,9}, "STOA") \\ useInNT(x_{V,3}, x_{V,99}, "NXTFUE") \\ getNode(x_{V,10}, "THRSTRCL") \end{array}}{\begin{array}{l} addNode(x_{V,11}, "ISO") \\ addEdge(x_{V,1}, x_{V,11}, "FUE") \end{array}}$$

ISSN: 2414-4592
DOI: 10.20474/jater-3.4.5

$addEdge(x_{V,2}, x_{V,11}, "NXTFUE")$
$addNode(x_{V,12}, "THRSTRCL")$
$addNode(x_{V,13}, "ISO")$
$addEdge(x_{V,2}, x_{V,3}, "NXTFUE")$
$addEdge(x_{V,1}, x_{V,13}, "FUE")$
$addEdge(x_{V,2}, x_{V,13}, "NXTFUE")$
$addNode(x, V, 14"GAS")$
$addEdge(x_{V,4}, x_{V,14}, "PRSRNT")$
$addNode(x_{V,15}, "DVREQ")$
$addEdge(x_{V,2}, x_{V,5}, "NXTFUE")$
$addEdge(x_{V,6}, x_{V,15}, "START")$
$addNode(x_{V,16}, "FILLDRN")$
$addEdge(x_{V,7}, x_{V,8}, "FUE")$
$addNode(x_{V,17}, "THRSTRCL")$
$addEdge(x_{V,13}, x_{V,12}, "NXTFUE")$
$addNode(x_{V,18}, "LIQUID")$
$addEdge(x_{V,11}, x_{V,17}, "NXTFUE")$
$addNode(x_{V,19}, "DVREQ")$
$addEdge(x_{V,7}, x_{V,12}, "FUE")$
$addEdge(x_{V,1}, x_{V,16}, "FUE")$
$addEdge(x_{V,16}, x_{V,9}, "NXTFUE")$
$addNode(x_{V,20}, "ENG4THRSTRCL")$
$addEdge(x_{V,17}, x_{V,20}, "ENG")$
$addEdge(x_{V,15}, x_{V,19}, "NXTDVREQ")$
$addEdge(x_{V,6}, x_{V,19}, "DVREQ")$
$addEdge(x_{V,5}, x_{V,10}, "NXTFUE")$
$addEdge(x_{V,19}, x_{V,20}, "ENG")$
$addEdge(x_{V,7}, x_{V,17}, "FUE")$
$addEdge(x_{V,8}, x_{V,20}, "ENG")$
$addEdge(x_{V,15}, x_{V,20}, "ENG")$
$addEdge(x_{V,4}, x_{V,18}, "FUEL")$
$addEdge(x_{V,10}, x_{V,20}, "ENG")$
$addEdge(x_{V,12}, x_{V,20}, ""ENG")$

B. *Extending Rule Set to Derive Bi-Propellant Systems*

The second experiment set the goal towards deriving the bi-propellant propulsion system shown in Figure. 15. The rule sequence that was used for derivation is (CG1, BI1, BI2, BI3, BI4, CG2 (×3), BI5, BI6, BI7 (×2), BI8 (×2), CG4 (×3), CG5 (×3), BI9, BI10, BI11, BI12, CG7 (×4), CG8 (×2)). Comparing Figure. 13 and Figure. 15, it becomes obvious that the differences between the original, cold-gas system and the bi-propellant system are greater than those between the mono-propellant system and the bi-propellant system.

Hence, the rule induction task in this experiment is expected to be more complex. This circumstance has been considered in the maximum number of GP generations, which is set to 2000 for this experiment. The population size remained at 256 individuals. The aggregated results over 30 experiment runs are shown in table 3. In this experiment the maximum number of generations has been exceeded only 2 times and again only in the second iteration.

The results gained in this experiment also show high variance. Taking both GP iterations together and computing the mean over 30 repetitions, one rule induction run required 1442 GP generations, 16.7M direct derivations and took 192 minutes. The mean number of existing rules reused is 4. Figure 16 depicts the distribution of reused rules. In this case the most frequent sets of reused rules are CG1, CG2, CG5 and CG2 (×2), CG5 (×2) (both 17 of the experiment runs). The case where the most rules were reused oc-curred only once and comprises the rules CG1 (×3), CG1 (×3), CG1 (×3), CG7, CG7.

TABLE 3
RESULTS OF MULTI-RULE-LEARN EXPERIMENT FOR MONO-PROPELLANT DESIGN CASE

	Generations		Duration (s)		Elementary Operations		Reused Rules	
Iteration	Mean	Std.Dev.	Mean	Std.Dev.	Mean	Std.Dev	Mean	Std.Dev
1st	711.70	233.23	2,375.54	4,682.45	4,507,243.30	4,675,690.31	2.50	1.11
2nd	729.83	530.37	9,138.32	12,825.31	12,167,631.67	16,151,192.51	1.77	1.57
Both n=30	1,441.53	664.41	11,513.86	14,931.26	16,674,874.97	17,220,399.52	4.27	1.93

ISSN: 2414-4592
DOI: 10.20474/jater-3.4.5

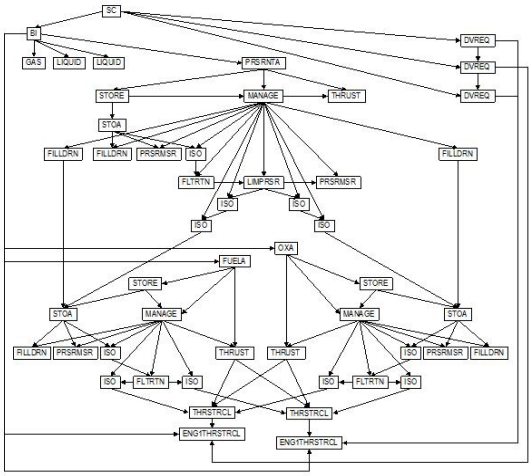

Fig. 14 . Target graph of the bi-propellant propulsion system. Edge labels omitted.

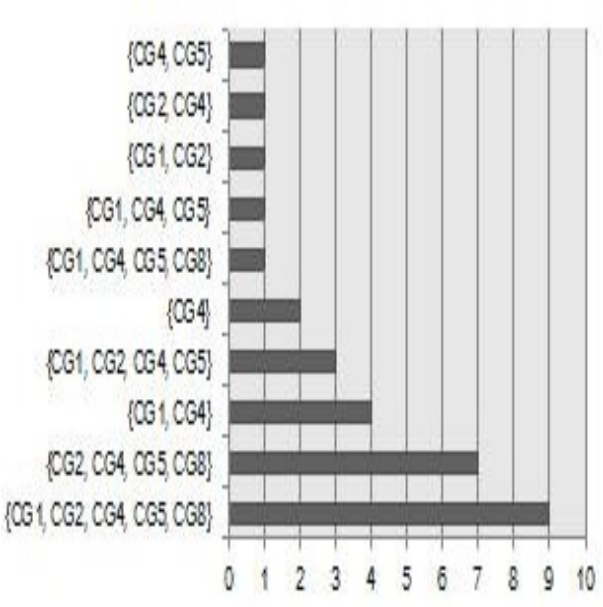

Fig. 15 . Frequencies of reused rule combinations for bi-propellant design case (n=30).

X. CONCLUSION

A method for optimization-based induction of graph-rewriting production rules was presented. The approach suffices the requirements that induced rules should cooperate with an existing rule set, but the learner is not allowed to inspect the definitions of existing rules. Computer experiments showed the practical applicability of the method to extend existing rule bases to derive new design examples using existing rules together with newly induced rules. Comparisons with the manually created rules of the gold standard showed that the induced rules may not necessarily correspond to human intuition. However, they are functional for reaching the desired target design. Hence, these induced rules may serve as starting point for further manual inspection and refinement. Future work should target the implementation of such a "solution polishing" method. Further, the method should be extended by a generalization mechanism that aggregates specific rules to yield a rule definition which allows to derive a greater span of target graphs. Lastly, additional goal functions that consider the semantics of the represented concepts need to be added in order to provide further guidance to the learners and improve the similarity with man-made rule definitions.

REFERENCES

[1] Object Management Group, "OMG systems modeling lan-guage", 2015 [Online]. Available: goo.gl/3gRMtr

[2] A. Habel and D. Plump, "Computational completeness of programming languages based on graph transformation," *in Foundations of Software Science and Computation Structures, 4th International Conference*, Berlin, Germany, 2013.

[3] L. C. Schmidt and J. Cagan, "Recursive annealing: A computational model for machine design," *Research in Engineering Design*, vol. 7, no. 2, pp. 102–125, 1995. **DOI:** 10.1007/bf01606905

[4] L. C. Schmidt and J. Cagan, "GGREADA: A graph grammar-based machine design algorithm," *Research in Engineering Design*, vol. 9, no. 4, pp. 195–213, 1997. **DOI:** 10.1007/bf01589682

[5] Z. Siddique and D. W. Rosen, "Product platform design: A graph grammar approach," *in Proceeding of the ASME Design Engineering Technical Conference*, Lass Vegas, NY, 1999.

[6] L. C. Schmidt, H. Shetty and S. C. Chase, "A graph grammar approach for structure synthesis of mechanisms," *Journal of Mechanical Design*, vol. 122, no. 4, pp. 371–376, 2000. **DOI:** 10.1115/1.1315299

[7] R. Alber and S. Rudolph, "43 a generic approach for engineering design grammars," *in proceedings of AAAI Spring Symposium - Computational Synthesis: From Basic Building Blocks to High Level Functionality*, California, CA, 2003.

[8] P. Sridharan and M. I. Campbell, "A grammar for function structures," *in Proceeding of the ASME Intlernational Design Engineering Technical Conference and Computers*

ISSN: 2414-4592
DOI: 10.20474/jater-3.4.5

and Information in Engineering Conference, Lass vegas, NY, 2004.

[9] Y. Jin and W. Li, "Design concept generation: A hierarchical coevolutionary approach," *Journal of Mechanical Design*, vol. 129, no. 10, pp. 1012–1022, 2007. **DOI:** 10.1115/1.2757190

[10] T. Kurtoglu, A. Swantner and M. I. Campbell, "Automating the conceptual design process: From black box to component selection," *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, vol. 24, no. 01, pp. 49–62, 2010. **DOI:** 10.1007/978-1-4020-8728-8_29

[11] B. Helms and K. Shea, "Computational synthesis of product architectures based on object-oriented graph grammars," *Journal of Mechanical Design*, vol. 134, no. 2, pp. 1–14, 2012. **DOI:** 10.1115/1.4005592

[12] J. Schmidt and S. Rudolph, "Gaining system design knowledge by systematic design space exploration with graph based design languages," *in Proceeding of the International Conference of Computational Methods in Sciences and Engineering*, Athens, Greece, 2014.

[13] G. Pahl, W. Beitz, J. Feldhusen and K. H. Grote, "*Engineering design: A systematic approach*," London, UK: Springer, 2007.

[14] T. Kurtoglu and M. I. Campbell, "Automated synthesis of electromechanical design configurations from empirical analysis of function to form mapping," *Journal of Engineering Design*, vol. 20, no. 1, pp. 83–104, 2009. **DOI:** 10.1080/09544820701546165

[15] H. Ehrig, U. Golas, A. Habel, L. Lambers and F. Orejas, "M-adhesive transformation systems with nested application conditions part 1: Parallelism, concurrency and amalgamation," *Mathematical Structures in Computer Science*, vol. 24, no. 04, pp. 1–48, 2014. **DOI:** 10.1017/s0960129512000357

[16] J. R. Eichhoff and D. Roller, "Designing the same but in different ways: Determinism in graph-rewriting systems for function-based design synthesis," *Journal of Computing and Information Science in Engineering*, vol. 16, no. 1, pp. 011 006, 2016. **DOI:** /10.1115/1.4032576

[17] C. De la Higuera, "A bibliographical study of grammatical inference," *Pattern Recognition*, vol. 38, no. 9, pp. 1332–1348, 2005. **DOI:** 10.1016/j.patcog.2005.01.003

[18] G. L. Pappa and A. A. Freitas, "Towards a genetic program-ming algorithm for automatically evolving rule induction algorithms," *in Proceeding of the Workshop on Advances in Inductive Rule Learning, Workshop at the 15th European Conference on Machine Learning and the 8th European Conference on Principles and Practice of Knowledge Discovery in Databases*, Berlin, Germany, 2004.

[19] B. Bartsch-Spörl, "Grammatical inference of graph grammars for syntactic pattern recognition," *in Proceedings of 2nd International Workshop on Graph Gramars and Their Application to Computer Science*, Berlin, Germany, 1983.

[20] E. Jeltsch and H. J. Kreowski, "Grammatical inference based on hyperedge replacement," *in proceedings 4th International Workshop on Graph Gramars and Their Application to Computer Science*, Berlin, Germany, 1991.

[21] L. Fürst, M. Mernik, and V. Mahnič, "Graph grammar induction as a parser-controlled heuristic search process," *in proceedings of 4th Intertnational Symposium, Applications of Graph Transformations with Industrial Relevance,* Berlin, Germany, 2012.

[22] D. J. Cook and L. B. Holder, "Substructure discovery using minimum description length and background knowledge," *Journal of Artificial Intelligence Research*, vol. 1, no. 1, pp. 231–255, 1994. **DOI:** 10.5220/0003637901720178

[23] K. Ates and K. Zhang, "Constructing veggie: Machine learning for context-sensitive Graph Grammars," *in Proceedings of the 19th IEEE International Conference on Tools with Artificial Intelligence*, Patras, Greece, 2007.

[24] Bahrudin, H. S. Alam and T. Haiyunnisa "Computational fluid dynamic simulation of pipeline irrigation system based on ansys," *International Journal of Technology and Engineering Studies*, vol. 2, no. 6, pp. 189-193, 2016. **DOI:** 10.20469/ijtes.2.40005-6

[25] E. B. Nejad and R. A. Poorsabzevari, "A new method of winner determination for economic resource allocation in cloud computing systems," *Journal of Advances in Technology and Engineering Research*, vol. 2, no. 2, pp. 12-17, 2016. **DOI:** 10.20474/-jater2.1.3

[26] A. Inokuchi, T. Washio and H. Motoda, "An apriori-based algorithm for mining frequent substructures from graph data," *in proceedings of 4th European Conference, Principles of Data Mining and Knowledge Discovery,* , Berlin, Germany, 2000.

[27] F. Costa and D. Sorescu, "The constructive learning problem: An efcient approach for hypergraphs," *in proceedings of Constructive Machine Learning, Workshop at the 2013 Conference on Neural Information Processing Systems, Workshop at the 2013 Conference on Neural Information Processing Systems*, Lake Tahoe, CA, 2013.

ISSN: 2414-4592
DOI: 10.20474/jater-3.4.5

[28] C. Sammut, "Learning as search," *in Encyclopedia of Machine Learning*, C. Sammut and G. I. Webb, Eds. New York, NY: Springer, 2010, pp. 572–576.

[29] N. Bertrand, G. Delzanno, B. König, A. Sangnier and J. Stückrath, "On the decidability status of reachability and coverability in graph transformation systems," *in Proceedings of the 23rd International Conference on Rewriting Techniques and Applications, Schloss Dagstuhl: Leibniz-Zentrum für Informatik*, Nagoya, JA, 2012.

[30] D. Plump, "Termination of graph rewriting is undecidable," *Fundamenta Informaticae*, vol. 33, no. 2, pp. 201–209, 1998. **DOI:** 10.1006/jsco.1995.1037

[31] D. Bisztray and R. Heckel, "Combining termination proofs in model transformation systems," *Mathematical Structures in Computer Science*, vol. 24, no. 04, pp. 1–30, 2014. **DOI:** 10.1017/s0960129512000369

[32] J. R. Koza, "*Genetic programming: On the programming of computers by means of natural selection*," Cambridge, MA: MIT Press, 1992.

[33] M. L. Wong and K. S. Leung, "*Data mining using grammar based genetic programming and applications*," New York, NY: Kluwer Academic Publishers, 2002.

[34] J. R. Eichhoff and D. Roller, "Genetic programming for design grammar rule induction," *in Proceedings of the 9th International Web Rule Symposium, Rule ML 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* , Aachen, Germany, 2015.

[35] M. Nikolić, "Measuring similarity of graph nodes by neighbor matching," *Intelligent Data Analysis*, vol. 16, no. 6, pp. 865–878, 2012.

— This article does not have any appendix. —

ISSN: 2414-4592
DOI: 10.20474/jater-3.4.5