A Graph Grammar Methodology for Generative Systems

Matthew Campbell
University of Texas at Austin[1]

Abstract

This paper puts forth a view of graph transformation systems as a useful way to organize the construction activities involved in design – be it the design of engineering artifacts or of any creative endeavor. While the concept of graph grammars has existed for nearly 40 years in an esoteric corner of artificial intelligence research [1], researchers in design automation have realized their worth in encapsulating knowledge and heuristics of a particular problem domain. In this paper, the fundamental challenges for graph transformations are studied especially in the context of design. In particular, the activities of recognizing, choosing, and applying rules are studied and two engineering examples are provided to illustrate the power of this approach in design automation.

1   Introduction

Graph transformation systems, or graph grammars, reside in graph theory research as a way to rigorously define mathematical operations such as addition and intersection of graphs. Mathematicians who have developed the foundations of this research often claim a possible application in adaptable systems, reconfigurable networks [2-4]. Recently, engineering design researchers have appropriated the concept of graph transformations and graph grammars to formalize the creation of complex engineering systems [5]. Electric circuits, truss structures, and chemical process plants are just a few of the artifacts of engineering design activities that are easily represented by graphs.

Obviously such artifacts are difficult to design, and yet in retrospect, we can consider an artifact's development from its inception as mere specifications through to its final configuration as a series of additions and modifications. The initial specifications can be represented as a simple graph in which the desired inputs and outputs are cast as arcs and nodes of the "to be designed" artifact or possibly as a single node which represents the foundation of the design. From this initial specification, the design process can be viewed as a progression of graph transformations that lead to the final configuration.

Readers familiar with only engineering design research or only graph transformation research may be puzzled at why these two very different fields of research should be combined. Clearly, any real-world engineering design problem offers complexities beyond what one can encapsulate within a

---

[1] Author's contact info: Department of Mechanical Engineering; 1 University Station; Mail Code C2200; Austin, TX 78712-0292; 512-232-9122; mc1@mail.utexas.edu

graph. However, simpler mathematical methods – numerical optimization, for example – have utterly transformed engineering design. In optimization, one simply has a vector (fixed array) of decision variables that signify the real parameters of a design. With some metric of a design's worth, one is able to run an optimization process to find the best values for the set of decision variables. Depending on the accuracy of the metric, the decision variables can be manifest as actual values in the artifact. Generating new candidates in optimization is as simple as placing different values within the vector of the decision variables. It is proposed that with a graph representation of a design as opposed to this simple vector representation, a richer, more varied set of candidates can be created within a computational search for an optimal design.

When viewing the artifact as a graph constructed from an initial simpler graph, one needs to develop a set of rules to capture the valid transformations that can occur. Through the application of each graph grammar rule, the design is brought closer to completion. These rules need to be established prior to the design process as an encapsulation of heuristics for designing. While the majority of research on graph grammars has dealt with only purely mathematical constructs, this view of a grammar as applied to creative systems has several distinguishing traits.

First, there is a distinct three step process: recognize, choose, and apply (as illustrated in Figure 1). While others researchers have indicated this process [6], an automated design system would benefit from systematically executing these three steps. At any stage in the creation of a system, we would like to know all possible changes that can be made. All possible rules should be checked with the graph to determine whether their application conditions allow the rule to be applied. Some rules may not be applicable; some may; and others may be applicable at more than one location in the graph. In Section 3, the details of a generic recognition process for generative systems are described. From this recognition process, a list of choices is concatenated. The decision now rests on a designer or some computational agent to choose the best rule and location from the list of choices. The creative agent must be knowledgeable enough to predict how the design will be affected and judge the benefit in invoking one option over another (see Section 4). Finally, the rule is applied after the choice is made. This application of a rule may introduce new nodes and arcs to the graph, remove nodes and arcs, or simply change qualities within existing nodes and arcs. A discussion of rule application specific to generative systems is shown in Section 5.

In addition to this quality of distinct *recognize*, *choose* and *apply* operations, generative systems also
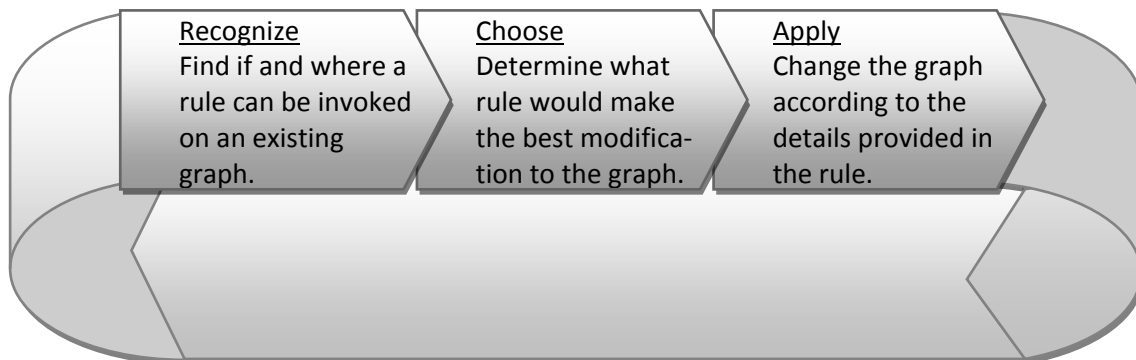

Figure 1: For Generative systems, we consider three distinct steps of 1) recognizing what rules are applicable, 2) choosing one of these rules to apply, and 3) the application of the rule which involves a graph transformation of the host to a newly synthesized state.

function on a time-free scale where no parallel transformations are required. The emphasis in these grammars is clearly on the *choice* step. There appears to be no advantage in invoking several rules at once to transform a graph from one state to another in design. If confluence [7] exists amongst several rule choices, and thus the order of application of such rules has no affect on the design, then it is acceptable and recommended to simply apply them in series with the agent kept informed of each graph transformation. This is not to say that time is not important in design, but rather to emphasize the importance of *choice* in generative systems. Given efficient rule recognition and application processes as discussed in the subsequent sections, it is likely that the bulk of time will be spent in deciding which rule to invoke. The creative process in engineering or other disciplines can be viewed as separate from the actual construction of the designed artifact. A plan for a new building is completed through careful analysis and drawings prior to a construction for that building; a construction which may in fact take advantage of parallel and series operations, but the decisions made in the generative process are best executed one at a time, and hence there is no need to determine if rules can be invoked simultaneously in design.

2   Graph Grammar Fundamentals

In graph grammars, a set of rules are devised to capture the complexities of a particular design problem. These rules contain a left hand side of application conditions, L, and a right hand side of
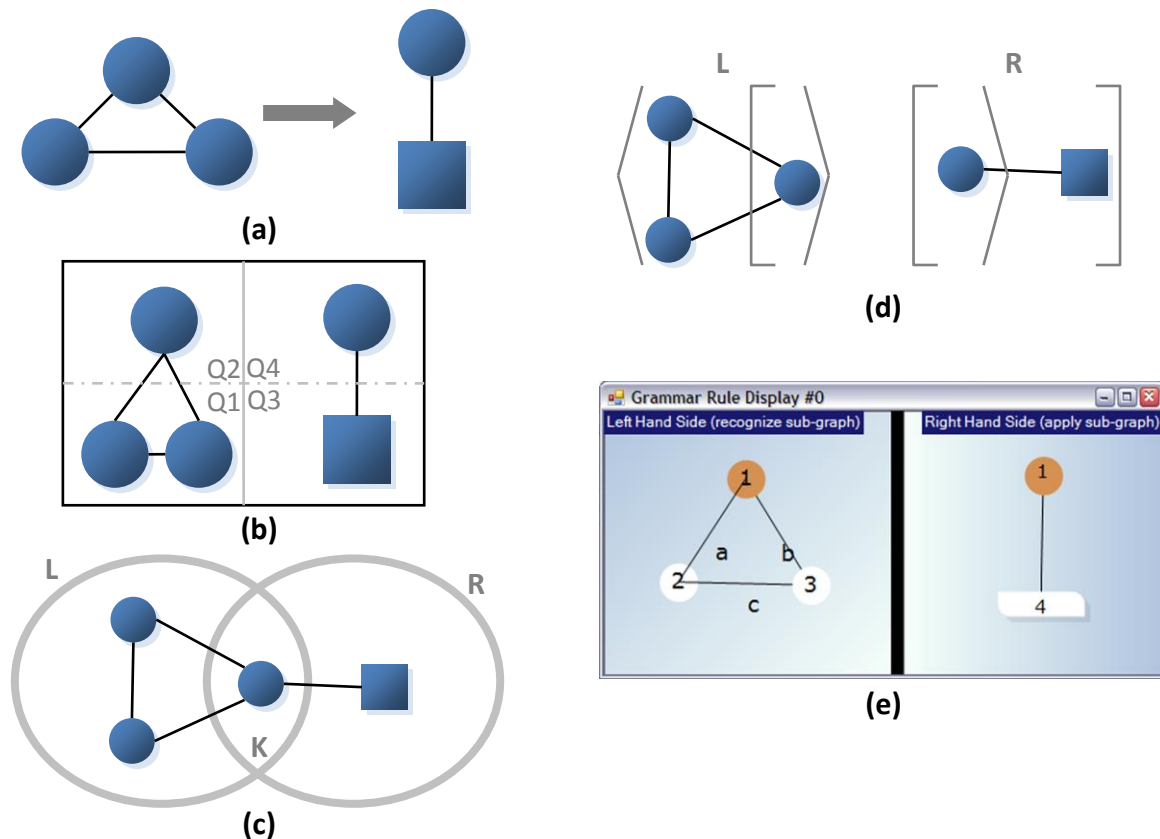

Figure 2: An example graph grammar rule is shown five different ways. There is a single node which is common to both the left-hand-side (L) and a right-hand-side (R). The final two representations have been adopted in this research.

3

resulting graph transformations, R. In general, L and R share some common elements; nodes and arcs which provide the context for a rule application. This intersection of L and R is referred to as K and it is crucial to indicate the overlap in the representation of grammar rules. Graphically, as in Figure 2a, it may not be clear what elements L and R share. Spatially, one can move the common elements to a particular region of a diagram, and delineate the K elements via the cut-and-paste-diagram approach [8, 9] (Figure 2b) or a Venn diagram (Figure 2c). Additionally, Figure 2d shows a representation that is often preferred in this research where the left-hand-side of the rule is delineated by the inequality symbols, "<" and ">" and the right-hand-side by square brackets, "[" and "]." In this way, L and R can be shown separately with the symbol "[" dividing out the K elements in L and the symbol ">" dividing those elements in R. Finally, in the implementation of this research, it has proven easier to simply draw separate L and R graphs and indicate the common elements by a predefined color (the orange elements in Figure 2e are common to both L and R).

As discussed in the introduction, many engineering systems use graph-based representations in both the design and analysis of complex systems. In Figure 3, five example engineering graphs are shown. One of the interesting aspects shared by some of these graphs is the violation that the arcs of the graphs reference exactly two nodes. The use of dangling or unconnected arcs is often used; however, this is not supported in traditional graph transformation systems. The development of a generic graph transformation system described here is the first to support these dangling arcs.
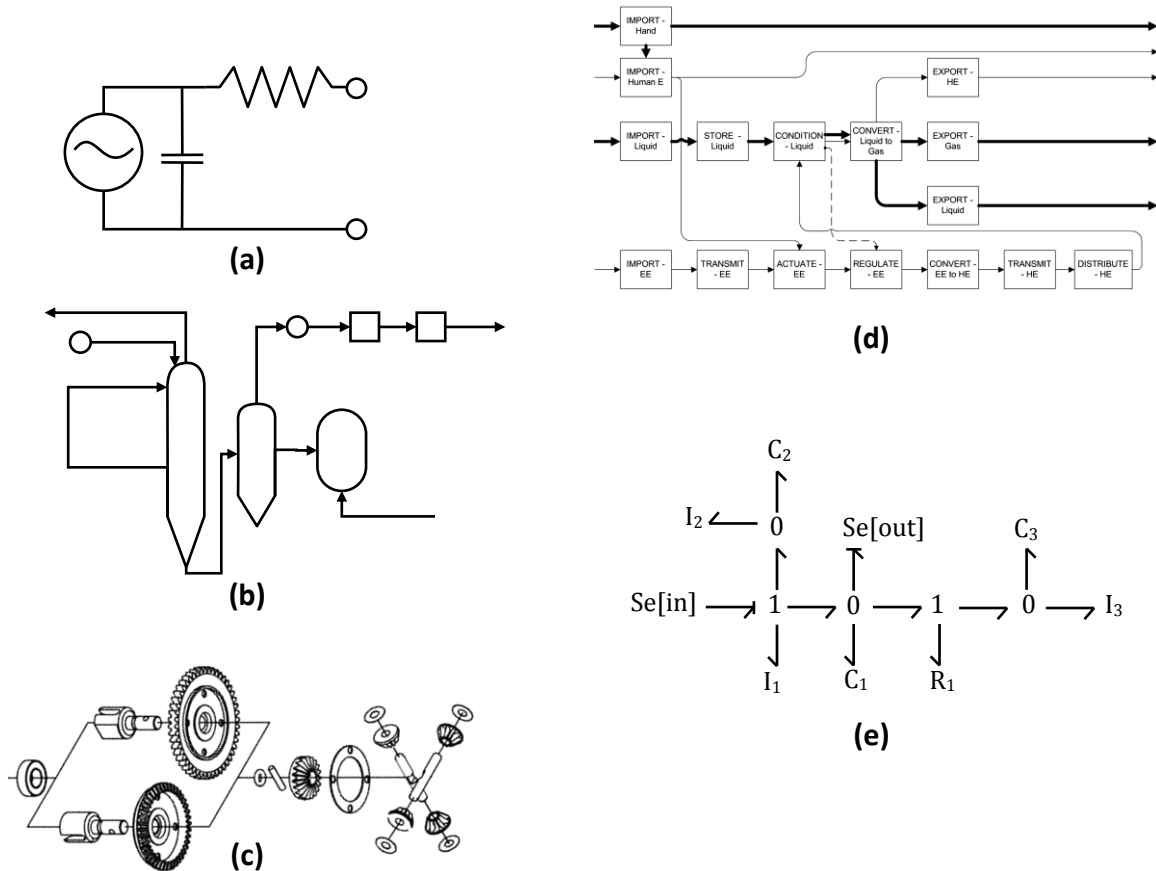

Figure 3: Examples of graphs used in engineering design and analysis: a) an electrical circuit, b) a chemical processing diagram, c) an exploded view, d) a function structure, and e) a bond graph.
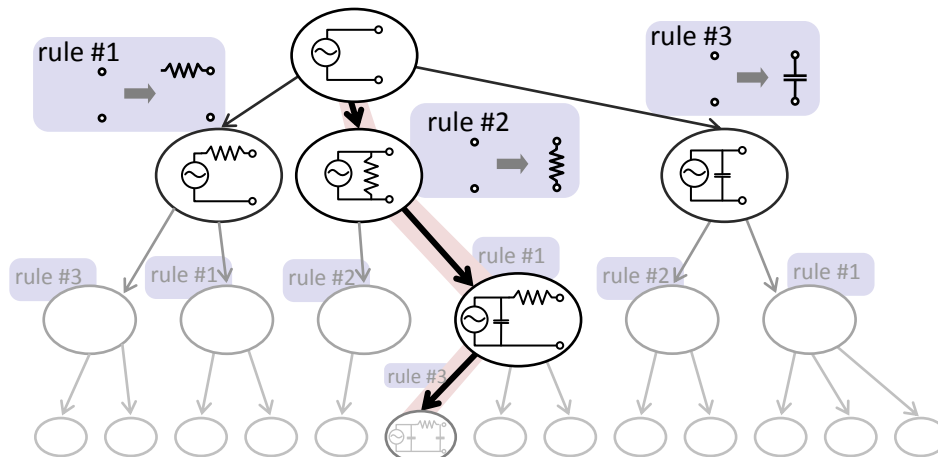
Figure 4: The search process tree typically studied in artificial intelligence and problem solving can be used to represent the design process specified by the graph grammar rules.

In addition, to the use of graphs to represent the artifacts of engineering, we can also use a specialized graph to represent the design process. As is often done in computational search and problem solving, the search tree can be used to capture the space of the design process. At the top of the tree, the seed graph of the problem is specified. The graph grammar's recognize, choose, and apply cycle is used to fulfill the successor or transforming functions of the search process. Figure 4 shows an illustration of this design process for the creation of an electric circuit. The three steps of the grammar system of Figure 1 implicitly provide the representation for this tree. The *recognition* function determines all the possible options that can be invoked from a state in the tree, the *choice* commits to invoking one of the options, and the *apply* provides the instructions to modify the state into a new state. In this way, graph grammars can be seen as a methodology for the transition functions in state-space search [10]. Clearly, one of the benefits of state-space search is that it is independent of the representation of states and their transitions, but since so many problems are representable by graphs, a fundamental approach to their transformations is welcome.

The view of grammar rules has proven to have uses in some domains purely as a way of organizing human thought and without taking advantage of computational development. Stiny [11] has adopted the grammar concept as a way to encapsulate the design of architectural structures. This work has spawned an enthusiastic community of researchers in architecture and engineering that create shape grammars for a variety of creative systems such as the design of traditional Turkish houses [12], Chinese lattices [13], and coffee makers [14]. Research by Krishnamurthy [15] have provided fundamental computational research for implementing the *recognition* and *application* steps on two- and three-dimensional shapes represented by building on the vertices and edges of spatial graphs. However, this research has yet to result in a comprehensive shape grammar implementation. An important classification scheme offered by Chase [16] defines grammars by their level of implementation. As is shown in Figure 5 (adapted from Chase), an additional task is offered to the *recognize*, *choose*, and *apply* steps of the grammar, which is the actual development of the grammar rules. Of these four tasks, we may have a designer developing a set of rules to guide or restrict their own creative process (Scenario 1), a separate human developer and designer (Scenario 2), or some combination of human developer, human designer, and computational agent. Many of the works that classify specific architectural styles or engineering systems rarely pass the
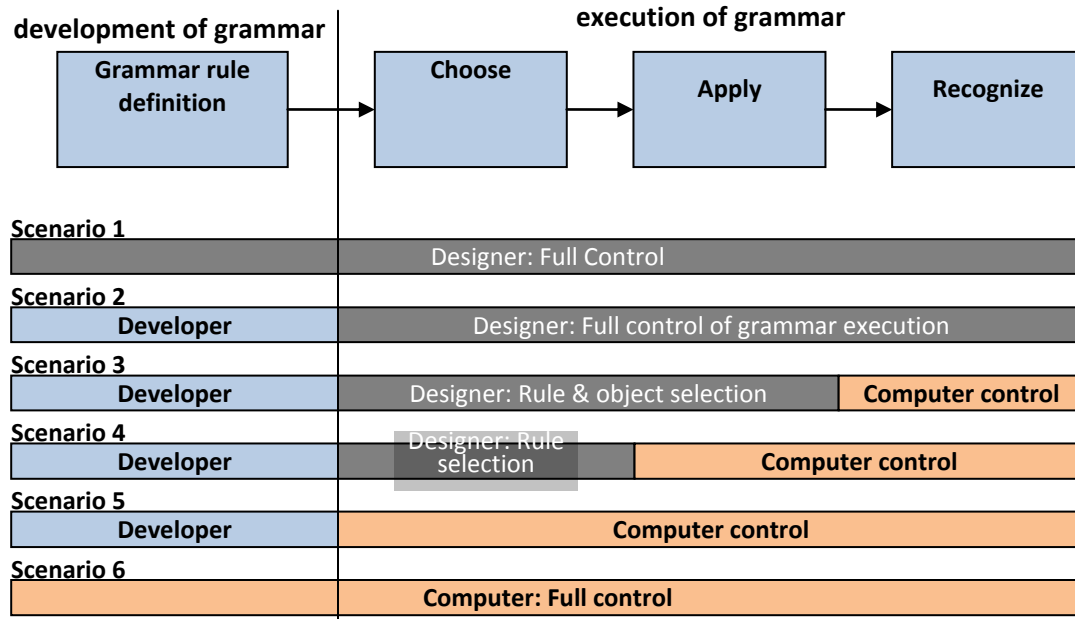
development of grammar   **execution of grammar**

| Grammar rule definition | Choose | Apply | Recognize |

Scenario 1

| Designer: Full Control |

Scenario 2

| Developer | Designer: Full control of grammar execution |

Scenario 3

| Developer | Designer: Rule & object selection | Computer control |

Scenario 4

| Developer | Designer: Rule selection | Computer control |

Scenario 5

| Developer | Computer control |

Scenario 6

| Computer: Full control |

Figure 5: Chase's six scenarios for classifying grammars (Chase, 1998).

second scenario as the research is focused primarily on the construction of rules. Indeed, these rules are so cleverly constructed that one could "play designer" and essentially create a Hepplewhite chair [17] or Frank Lloyd Wright prairie house [18], by simply *choosing* to *apply* successive rules that are *recognized* on previous states. This is to essentially traverse down the search tree of solutions in a controlled way such that the final result is purely the creative act of an uninformed designer, but is also indistinguishable from the original master!

The results from these human-driven grammars provides a strong motivation for implementing a fundamental graph grammar (and a shape grammar) tool to allow for the easy development of rules, and the automatic recognition, application, and perhaps even choosing of rules to launch a new era of design automation. In the remaining sections of this paper, we discuss the foundations of a graph grammar framework and show examples of rules that can be created.

3   Subgraph Recognition

The act of recognizing when a rule can be applied is to identify a subgraph within the host that matches with the graph depicted as the left hand side of the rule (as shown in Figure 6). In the larger mathematical community, this is referred to as subgraph isomorphism. Additionally, we adopt the view that the act of *matching* a node or arc to one in the host is that the labels must be a subset. When more labeling is used to qualify the elements of L, the number of possible subgraphs that are recognized reduces drastically. The labels within both the graph L and the graph, Host, in Figure 6 constrain the recognition to a single subgraph (node 2 and arc 10). This subgraph is recognized even though the matching location contains more labels than L. If all of the labels are removed in L, then there will be 16 possible subgraphs that match to L (two for each arc).
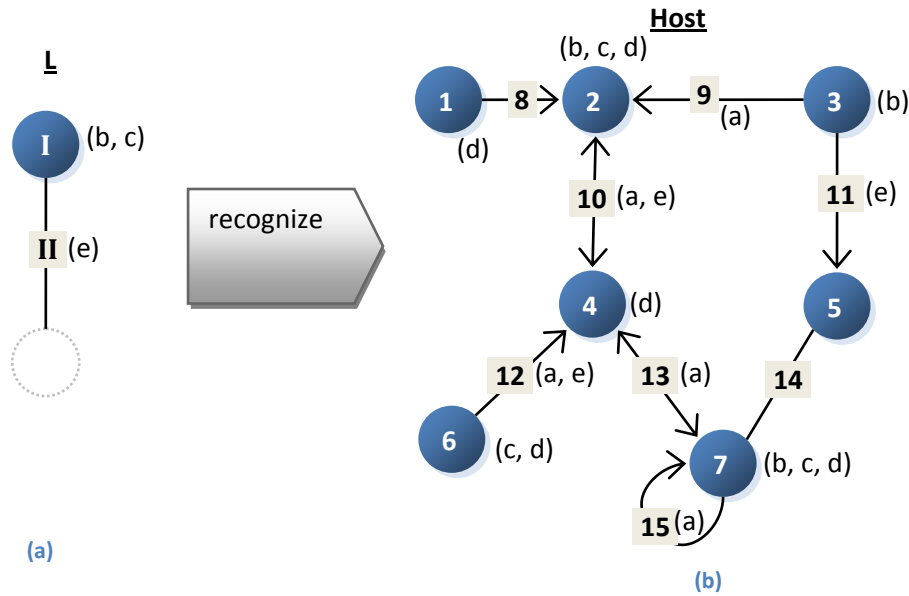
Figure 6: The recognition requires that we find the subgraph of the left hand side (a) of the rule within the host (b). In this example, nodes and arcs are referred to by the corresponding number, and labels for nodes and arcs are shown in parentheses.

3.1 Subgraph Booleans

For generative systems, rules are likely constructed to model or contain particular heuristics or design guidelines. Label rich recognition can be interpreted as, "if a state of the design contains all the characteristics of a particular rule, then that rule is applicable." However, there are cases when rules may not be easily represented by subsets of labels. It is for this reason that we additionally equip each node and arc within the left hand side graphs with a Boolean, ***containsAllLocalLabels***. If this Boolean is set to true then the set of labels is enforced to be identical in both rule and host. In this case, the rule shown in Figure 6 will not be recognized within the host at all. If all the labels were removed in the L graph and this Boolean is set to true, then we would find only one location (node 5 and arc 14) as opposed to the 16 discussed above.

It is clear from this example that by setting the label matching as "equal to" as opposed to subset, the number of recognized locations may be reduced significantly. In addition to this restriction, there are a number of additional Booleans that can be implemented to better capture the intended rule recognitions. For each node, one can limit the number of arcs that are connected to it. In our implementation, a Boolean called ***strictDegreeMatch*** is created. This essentially means that a node in a rule's L will match only with a node in the host that has the same number of arcs connecting to it. In the preceding example of Figure 6 with the labels removed from L and the ***strictDegreeMatch*** set to true, there are only two locations that are properly matched (node 1 and arc 8, and node 6 and arc 12) as opposed to the 16.

For the arcs, we can provide an additional criterion on the direction of the arc with the Boolean, ***directionIsEqual***. When this is set to false, we assume that undirected arcs can be matched with

Table 1: A summary of the subgraph Booleans present in Section 0

Subgraph Boolean	Applies to what aspect of rule	when TRUE…
spanning	graph	host contains an equivalent set of nodes as L
induced	graph	host contains an equivalent set of arcs as L for the recognized set of nodes
containsAllGlobalLabels	graph	host contains an equivalent set of global labels as L
strictDegreeMatch	node	host node contains an equivalent set of arcs connected to it as L node (nodes have same degree)
containsAllLocalLabels	node	host node contains an equivalent set of labels as L node
containsAllLocalLabels	arc	host arc contains an equivalent set of labels as L arc
directionIsEqual	arc	host arc has equivalent direction characteristics as L arc
danglingMatch	arc	host arc must be equivalently unconnected at ends as the L arc

directed arcs or doubly-directed arcs in the host. When set to true, the Boolean limits matching undirected only with undirected, directed only with other directed with the same sense, and doubly-directed with doubly-directed. Furthermore, an additional Boolean related to arcs handles the dangling nature of arcs. The L in Figure 6 only includes a single node and a single arc; however, many approaches to graph theory disallow arcs from having an unspecified or dangling end. In the development of the methodology and implementation presented here, this dangling nature is addressed. In the examples that have played out from Figure 6, the missing node at the bottom of L is treated as a wildcard that matches with any node regardless of its characteristics. If one creates a rule that seeks to match an arc to similar dangling arcs in the host, then this would prevent the rule from recognizing any legitimate subgraphs within the host. In the following implementation a Boolean, called ***danglingMatch***, is created that prevents the rule from matching with arcs that are defined between two nodes.

Finally, the ambiguity in defining the true nature of subgraph can result in the graph-wide Booleans, ***induced*** and ***spanning***. A spanning subgraph is one that contains all the nodes of the host graph but not necessarily all the arcs. Clearly, this results in no recognition in the previous example since the host contains more nodes. When the ***induced*** Boolean is set to true the subgraph is required to be an induced subgraph within the host. This means that L must contain all arcs that exist between the recognized nodes in the host. These subgraph Booleans are commonly accepted as fundamental to graph theory [19, 20]. Graphs can also be assigned labels similar to those in the nodes and arcs. As a result, we create a final Boolean that relates to the global labels being a proper subset or equivalent to those in the host. A summary of these subgraph Booleans are shown in Table 1. While some of these have existed before in past graph theory literature, we have expanded the set to include all possible ambiguities in what is meant by a subgraph. Since a graph can be defined as an interacting set of sets (e.g. set of labels, sets of arcs, and sets of nodes) the Booleans allow us to independently define how the subsets are determined for each of these components.

3.2   Recognition Procedure

As part of employing graph grammars in generative systems, a basic recognition function is developed based on the actions that we, as humans, easily but tediously perform in the prior exercises of Figure 6. On the most general level this recognition takes two inputs: the L of a rule,

and the host graph. What is expected to result is a list of subgraphs where L is found in the host. For a particular rule and host, there may be no subgraph, one subgraph, or many subgraphs. These resulting subgraphs are often referred to as partial graph morphisms of L or how L is *mapped* into the host. It has been shown that performing such subgraph matching (also known as subgraph isomorphism in mathematical literature) is computationally intensive for large unlabelled graphs. In the past, it has erroneously been labeled an NP-complete problem [21] but realistically, a system with a rich set of labels and conditions as discussed above, will lead to a very quick recognition process of matching subgraphs. Tests have been performed within the implemented environment that show the recognition process does take a long time when the graphs are unlabelled, undirected, and have 100 or more nodes.

The approach developed in our implementation is a recursive search starting with a single node in L. The first node in L is checked to see if it *matches* (in the sense of the subset descriptions discussed in Section 3.1) with each of the nodes in the host. For those in which it is a successful match, a modified depth-first-search (DFS) is invoked. Through the DFS, valid subgraph matches are found at the bottom of the search tree where matching elements (both nodes and arcs) from the host are uniquely mapped to elements of L. There are several important but subtle facts to draw from this recursive implementation. First, while the matches that the first L node makes with the host are the starting points, some of these might not result in a valid mappings while others may in fact lead to more than one unique mapping. Second, the fact that all nodes in the L graph match with a unique node in the host is an assumption explicitly stated in some literature as the *identification criteria* [22]. In Figure 7a, a rule's L specifies that two nodes and one arc are to be found within the host (recognized locations exist at <2, 10, 4> and <7, 13, 4>). If the identification criterion is relaxed the rule can be matched with <node 7, arc 15, and node 7> of the host in Figure 6b as well. This is likely not the intention of the rule and the implementation is limited to enforcing the identification criteria to increase recognition efficiency. Third, creators of the grammar rules can affect the efficiency of this process by altering the order in which the nodes are presented. A computational
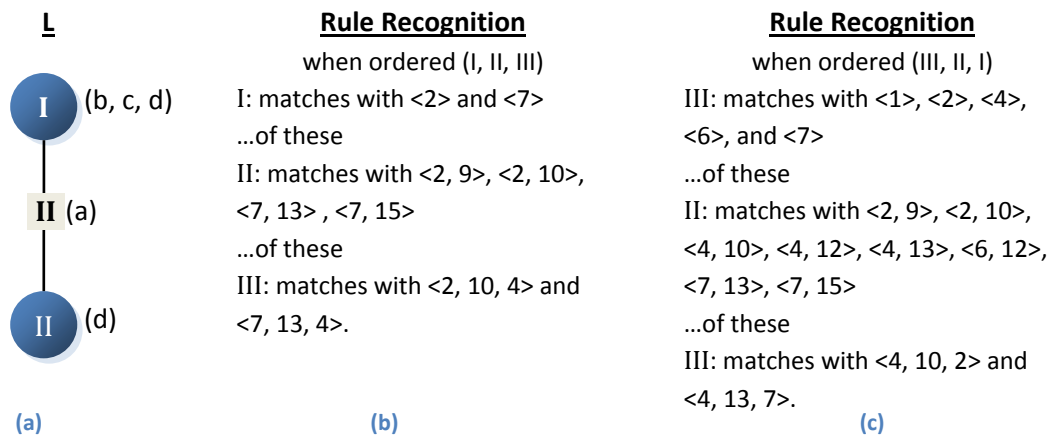
L	**Rule Recognition** when ordered (I, II, III)	**Rule Recognition** when ordered (III, II, I)
I (b, c, d)	I: matches with <2> and <7> ...of these	III: matches with <1>, <2>, <4>, <6>, and <7> ...of these
II (a)	II: matches with <2, 9>, <2, 10>, <7, 13> , <7, 15> ...of these	II: matches with <2, 9>, <2, 10>, <4, 10>, <4, 12>, <4, 13>, <6, 12>, <7, 13>, <7, 15> ...of these
II (d)	III: matches with <2, 10, 4> and <7, 13, 4>.	III: matches with <4, 10, 2> and <4, 13, 7>.
(a)	(b)	(c)

Figure 7: An example for the procedure of recognition: a) an example rule, b) nodes ordered from most restrictive to least aremore efficient as there are fewer opportunities to store and expand at each level of the depth first search, c) a poor ordering of nodes leads to twice as many candidate subgraphs being expanded before the same two are found (note: that the ordering is different in b to correspond to the order that they are presented).
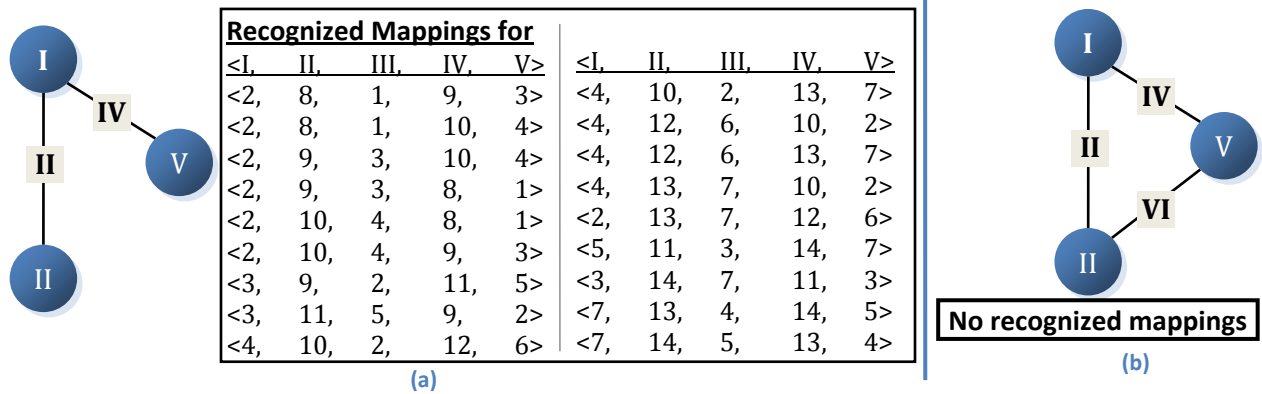
Recognized Mappings for									
<I,	II,	III,	IV,	V>	<I,	II,	III,	IV,	V>
<2,	8,	1,	9,	3>	<4,	10,	2,	13,	7>
<2,	8,	1,	10,	4>	<4,	12,	6,	10,	2>
<2,	9,	3,	10,	4>	<4,	12,	6,	13,	7>
<2,	9,	3,	8,	1>	<4,	13,	7,	10,	2>
<2,	10,	4,	8,	1>	<2,	13,	7,	12,	6>
<2,	10,	4,	9,	3>	<5,	11,	3,	14,	7>
<3,	9,	2,	11,	5>	<3,	14,	7,	11,	3>
<3,	11,	5,	9,	2>	<7,	13,	4,	14,	5>
<4,	10,	2,	12,	6>	<7,	14,	5,	13,	4>

(a)

No recognized mappings

(b)

Figure 8: Two simple rules require similar computational effort; however, the one shown as (a) is recognized 16 times within the host shown in Figure 6b, while the rule shown as (b) is not recognized at all.

structure for a graph will necessarily store the set of nodes as a list of elements. In graph theory the order is of no consequence as the relationships provided by the arcs is the true defining quality of the nodes. However, by placing the most restrictive node (the node that is hardest to match) first in the list of nodes one can eliminate wasted search that would occur if a less restricted node is presented first. An example of this is shown in Figure 7. The node I in L is more restrictive than node III and therefore, the amount of search required to arrive at the same mappings is affected. The order of expansion is shown in Figure 7b and c wherein the order shown in Figure 7c requires three times the number of expansions.

The depth-first-search proves to be a more efficient approach to recognition both in computational speed and memory. A more brute force method is to first find all permutations that the set of nodes within L match with the set of nodes in the host, and then remove those that are not connected in a similar manner. The DFS approach is efficient but offers variable recognition times depending on the constraints (labels and Booleans) and the size of L.

As a final recognition example, two rules are provided in Figure 8 which differ only by the addition of a single arc. As we follow their recognition in the host shown in Figure 6b, we start with node I being recognized with all nodes (<1>, <2>… <7>) in the host. This expands to 15 potential locations at the second level with the matching of arc II (it is important to note that DFS does these one at a time, but for explanation purposes, we consider the levels of each search expanding all at once, which is more akin to breadth-first search). As the search progresses to elements III, IV and V, the number of possible locations changes to 14, 16, and 16 respectively. For the rule shown in Figure 8a, these are the 16 mappings that are returned, but for rule shown in Figure 8b, the "bottom" of the search tree includes a final arc, arc VI, which is unmatched in any of the 16 previous mappings. Thus, the computation spent finding a match is fruitless.

An additional item to note in these mappings is that locations including the same nodes and arcs are identified multiple times. For example, the first and fourth locations in the example (Figure 8a) represent the same subgraph in the host, but the L elements are mapped to different elements (<2,8,1,9,3> and <2,9,3,8,1>) in the host. This is because the overall recognize-choose-apply requires that we uniquely match each node and arc to one in the host. This becomes more apparent
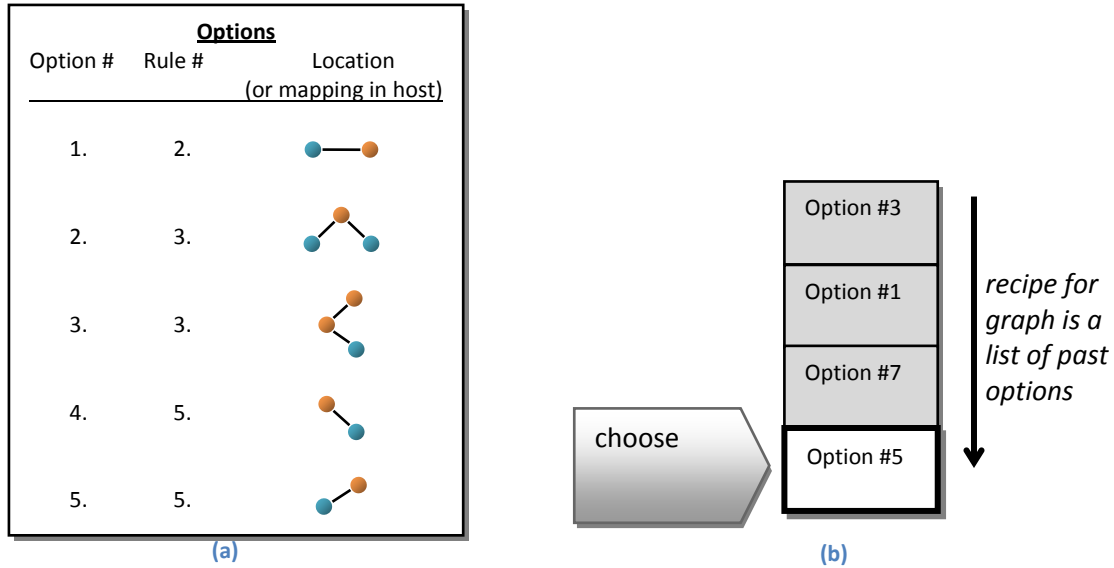
Figure 9: The act of choosing is simply to make a choice from a list generated through recognition.

in the discussion of rule application (Section 5) since the rule essentially includes individual instructions for each of the elements matched to the L graph.

4   Choosing: The true creative task

From subgraph recognition, we can enumerate all possible rules that can be invoked on a particular host graph. This enumeration is not simply a list of which rules from a set of rules are recognized but a three-tuple list that includes a unique identifier known as the option number, the rule, and the rule's recognized location with the host. The resulting options are presented to the creative agent as input for the choice activity as shown in Figure 9. The output of *choose* is simply a selection from this list of options.

The creative decisions in the design process are reduced to this simplicity, a matter of decision-making. In this regard the intricacies of this process no longer become the purview of graph theory but rather decision sciences, psychology, or engineering design. Representation of the options and the choice, which can be viewed as a simple integer identifying the preferred option number, are straightforward. Furthermore, the list of choice integers (Figure 9b) that results from *choose* can be seen as a recipe for creating the system. One can view this list of integers as the path to follow through the search tree (Figure 4) to obtain a particular graph. This helpful reduction of a graph to a meaningful list of integers may prove useful in combining this representation with nonlinear integer programming [23] to generate candidate decision integer recipes. In such a case, the creative agent may be encapsulated in an optimization process.

5   Application

Over the past thirty years, two separate fundamental approaches to application have been developed. This section describes how the two approaches are combined to handle all possible graph changes. The two approaches have been distinguished from one another as: algebraic versus

algorithmic, gluing versus connecting, or context-sensitive versus context-free. Figure 10 shows an overall flowchart for the application procedure adopted here. At first glance, the procedure appears overly complex for what would seem to be a straight-forward graph operation, but this flowchart captures in a rigorous way all graph transformations that may occur; addition, subtraction, or modification to nodes, arcs, and labels. The thick dotted line in the figure shows the algorithmic path of the application procedure while the long 'U' shaped lines indicate subgraphs or partial graph morphisms, where $A \subset B$ indicates that A is a subgraph of B. The input to application includes the three items on the left of Figure 10. The rule and the L-mapping (or partial graph morphisms, or locations) are the result of the previous *choose* function. These along with the host graph, G, represent the first elements of what is commonly referred to as the Double-Pushout method [24], which is discussed in detail in the following subsection. Following the pushout that removes elements from the host (Step 1 in Figure 10); a second pushout adds new elements to the graph (Step 2 in Figure 10). This concludes the traditional algebraic approach to graph transformation; however, for completeness, a third step is implemented to accomplish Free-Arc Embedding of possible dangling arcs. This third step (Step 3 in Figure 10) is based on a previous approach referred to as edge-directed Neighborhood Controlled Embedding [25] and is further illuminated in Section 5.2.

5.1   The Algebraic Double Pushout Method

The transformation of a graph from an initial state G (in Figure 10) to a new state H is accomplished by the Double Pushout Method. Fundamentally, the approach first removes any unwanted elements, followed by a connection of new elements. In Section 2, a description of grammar rules is provided that describes how the L and R of a rule may have some overlap. This overlap is referred to as the context graph, K, and indicates what parts of the L-mapping are to be kept through the rule application. Any elements (nodes, arcs, or labels therein) that are to be deleted in this rule transformation are therefore stored in L but not in K. One could view the DPO method as a function:
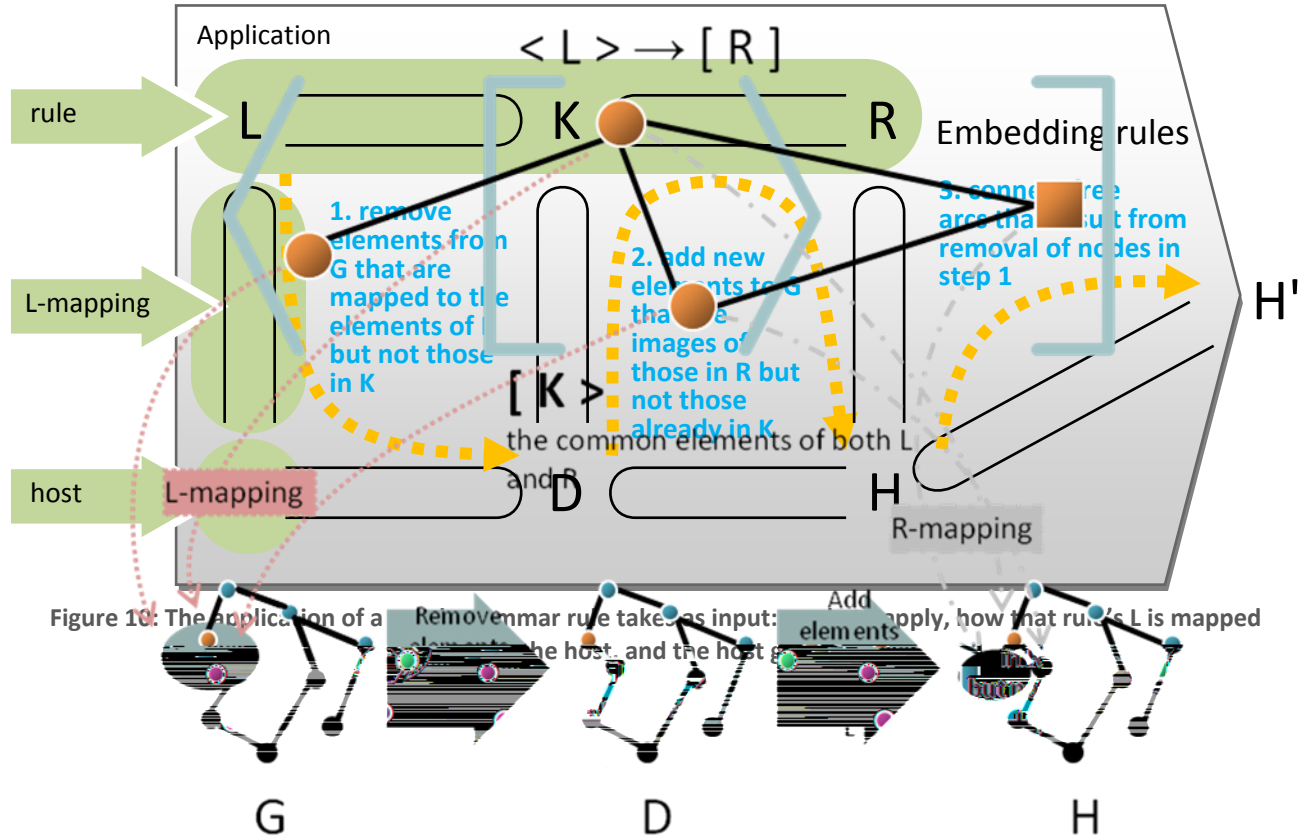
Figure 10: The Application of a grammar rule takes as input: Removed elements... Add elements... apply, how that rule's L is mapped to the host, and the host g...

Figure 11: An example of a DPO rule application. First elements are deleted followed by the addition of new elements.

$$H = G - (L - R) + (R - L) \tag{1}$$

where the parts of L that are not in R are first deleted (L-R), followed by the addition of elements in R but not in L (R-L). To clarify the graph subtraction functions, K should be represented either explicitly or implicitly. Of course, the rule itself does not change but rather it provides the change through the morphism, L-mapping. Figure 11 illustrates the Double Pushout approach through an example.

A smaller graph, D, is created through the deletion of elements in G. Just as K is both a subgraph of L and R, the graph D, is a subgraph of both G and H. Furthermore, K is a subgraph of D which is important because K is needed to provide the context for how the new elements are to be added. In the second part of Equation 1, the elements that are in R but not in L are added to the graph. This prevents us from re-adding the elements of K but allows us to attach the new elements in a general way to a host (see example in Figure 11). This method of adding elements via graph transformation is sometimes referred to as the 'gluing' method or the context-sensitive method. One final change that can be made in this part of application is to modify the labels in the K elements. This modification happens in a similar way to the overall rule operation but is applied on a set as opposed to a graph. Within each element in K, we identify the labels that exist when in L that are no longer stated when in R and delete them. This is then followed by adding the new labels that are in R but not in L; the common labels are left alone. In terms of implementation, we represent L and R as independent graphs. As such there is no explicit representation of the intersection graph K that

results. What is done to represent the K elements is to define the arcs and nodes with identical 'names' in L and R as the same element (this is shown in Figure 2e), when the user constructs a rule. It is important to note that this name is separate from labels which are used to represent important qualities in the graph. Name is an implementation fix to storing K elements in two separate instances, which is essential to capture the aforementioned modification of K labels.

5.2   The Algorithmic Free Arc Embedding

The second approach to graph application is often referred to as the "algorithmic approach" or the "connecting approach". In some instances this approach and the DPO approach are able to perform the same graph modifications, and in general the DPO method is preferred for its straightforward representation. The algorithmic methods [26] is an earlier graph transformation method and is so termed because it is easier to implement than the algebraic approach. Furthermore, approaches like this are 'context-free' which on a low-level relates to the lack of a common or context graph, K, but on a higher-level means that the rules can be applied in more generic way.

Consider the example in Figure 12. The three nodes of the left hand side, L, are removed leaving two dangling arcs. These elements are replaced by two new nodes and a single connecting arc. The two dangling arcs are then connected following a condition that informs the transformation process of where these arcs are to be connected. This approach is similar to the edge-directed Neighborhood Controlled Embedding or edNCE approach [27]. In edNCE, nodes that are connected to the nodes that are removed in the deletion of L are referred to as a neighborhood. These neighborhood nodes are then checked to see if they match the labels of the edNCE production rules. If a match is made then a new arc is added between the matching neighborhood node and one of the new nodes added by R. These productions repair or connect the new nodes of the graph with the nodes in the host remaining after the deletion of L (graph D). The productions in edNCE are represented in the form:

$$(\mu, d;\, \delta, d') \tag{2}$$

where μ is a label that must exist in the neighbor node for a correct match, δ is the name of the node added by R, and d and d' are the old and new directions respectively of the arc that attaches the host to the new R nodes. Even though the number of nodes that exist in the neighborhood of the rule transformation is unlimited, a limited number of rules such as the single one shown in Figure 12 may produce any number of new arcs, thus allowing for more generic graph transformations.

The new approach developed here is referred to as Free Arc Embedding and takes advantage of our ability to handle dangling arcs. In traditional DPO, a condition is enforced that prevents dangling arcs from resulting in a rule (this condition along with the identification condition mentioned earlier are typically enforced in DPO but not in the simpler Single Pushout Method [28]). Following the algebraic approach (see Figure 10), we gather the dangling arcs that were connected to the nodes deleted in the first part of DPO and check them with specific embedding conditions of the form:

if free arc ((contains label, μ)   (3)
     AND
     (was previously connected to the deleted node matched with the L node, λ)
     AND
     (is currently attached to a neighbor node with label, η))
     AND
     (is in the direction of d))
then (connect to node, δ in the direction of d').

This expands the representation of the embedding rules from that shown in Equation 2 to

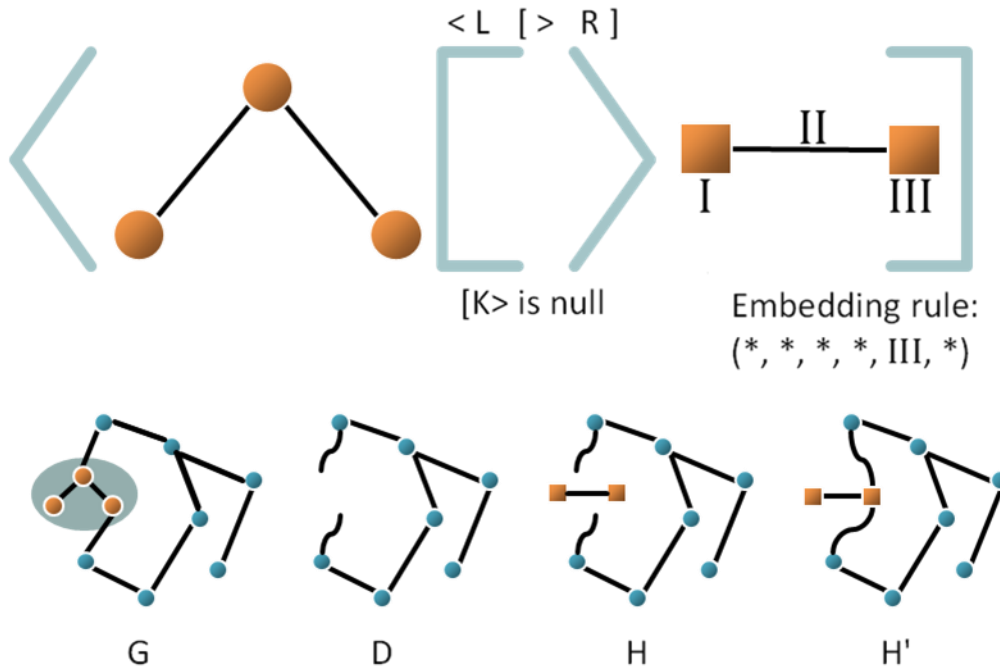$(\mu, \lambda, \eta, d; \delta, d')$ (4)

Figure 12: An example of Free Arc Embedding. With no context elements (K is null) to define how to connect the new elements with the remaining parts of the host, an embedding rule is used to connect the dangling arcs. In the example a single rule simply specifies that any dangling arcs are to connect to the new node, III.

15

and provides more control and possibilities for connecting the host to R. Any of the first four elements of this rule representation may be left blank to specify that the rule is not concerned about a particular quality. In fact, if the second and third elements are left blank, we recapture the edNCE model shown in Equation 2. If none of the first four qualities of the rule are specified, then the rule will be a valid transformation for all free arcs that exist. This is seen in the example of Figure 12 where all dangling arcs are connected to node, III. Any free arcs that meet none of the conditional rules are deleted as a final step in creating the new host, H'.

This embedding system differs by reconnecting the free arcs as opposed to creating new arcs as is done in edNCE – another benefit in allowing dangling arcs to exist in a graph. At first, this approach appears to lack the generality of the earlier edNCE system that can create any number of new arcs in the neighborhood of the L deletion. However, the addition of a seventh quality, *D* gives the new approach the ability to create more arcs than what is originally contained in the host by creating duplicate free arcs to connect to the new elements added by the rule.

It is important to note that in some instances the order of the free arc rules is important. Any combination of embedding rules can be used to identify a free arc. If two rules are "recognized" with the same free arc only the first one will modify it, as the arc will no longer be 'free' after the rule is applied; however, in cases when *D* is set to true, a copy of the free arc will remain for the other rules to operate on it. As a result, when a rule duplicates an arc in this way, it does not prevent further rules from being recognizing the free arc.

Essentially, this approach to applying the embedding rules only on free arcs is equivalent to the approach of applying embedding rules on the neighborhood nodes, since the neighborhood nodes are defined by any nodes that had been attached to deleted nodes through such arcs. There is discussion in the literature to extend the edNCE method to capture more global affects. This would allow new arcs to be established between the new R nodes and others within the host that are not neighbors. For generative systems, we are doubtful that any transformations would require this global effect, thus this application approach (as summarized in Figure 10) is limited to establishing only local connections with each rule application. Of course, one could envision complex engineering systems as having far-reaching or system-wide phenomena such as heat transfer, or radiation that might suggest the connecting of new arcs in such a way. But in the creation of some schematic, network, exploded view, or manufacturing process, all arcs simply provide connectivity between elements and not a description of far-reaching physical behaviors. Furthermore, the developer of engineering design rules is unlikely to intend a single rule to reach out to elements outside of the immediate connection of the newly added elements. Examples of such engineering design grammars are provided in the next section.

6 Example Grammar Systems

To date, the aforementioned methodology has been used to create over 300 grammar rules. The storage of graphs and rules is accomplished through XML where properties within the node and arc objects are easily managed. Rules are simply stored as two separate graphs, a left-hand side and a right-hand side, and a number of Booleans as is discussed in Section 3.10. Two applications

implemented thus far are shown below. In addition to these, a sheet metal grammar [29], a function structure grammar [30], and several test problems [31] have been implemented.

6.1 Neural Networks

Neural Networks have existed now for two decades as an approach to model data or learn appropriate actions for a fixed set of inputs [32]. The typical network contains at least the nodes or neurons that represent the distinct inputs and outputs, and any number of hidden nodes arranged in layers. The arcs or dendrites between the neurons must not create any loops in the graphs (feed-forward graph), and should provide at least one connection to each of the inputs and the outputs. The typical approach to creating a neural network is to define a fixed topology and use an optimization method to tune the weights and biases within the network. There has been some published attempts to determine the topology as well [33, 34]. These approaches essentially and unwittingly define a grammar for defining such graphs like that shown here.

Figure 15 shows the four grammar rules defined for neural networks. The first rule (Figure 15a) simply adds arcs to ensure that no inputs or outputs are left unconnected. In the left hand side of the rule, the ***induced*** Boolean is set to true preventing its recognition when the two nodes are already connected by an arc. The remaining rules are used to add additional hidden nodes and arcs to the network. In Figure 14, two example neural networks are shown that have been created by randomly invoking rules and transitioning from an initial seed of input and output nodes. The choosing can be done randomly, through a dialog with the user, or through a search process. In Figure 13, a dialog is shown that provides a user with the list of recognized rules and their locations. Notice that rule #2 is shown nine times where each is a unique morphism in the host graph to the right. In the future, the grammar rules can be chosen via an optimization method to

(a) Rule #1 and #2

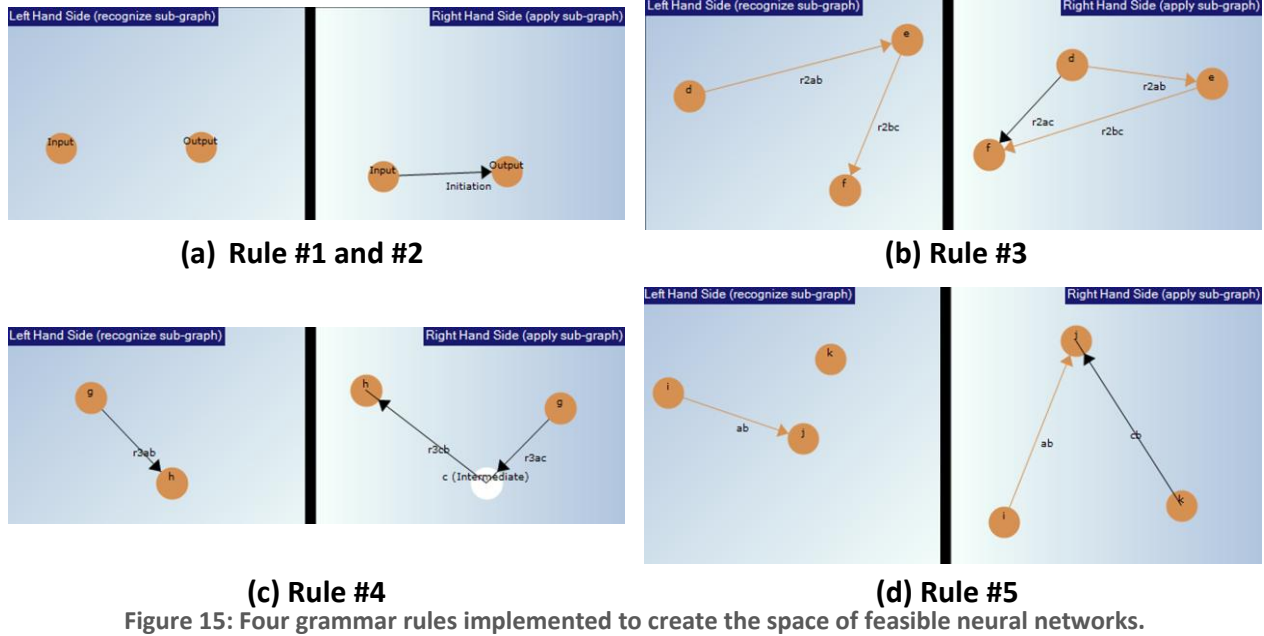**(b) Rule #3**

(c) Rule #4

(d) Rule #5

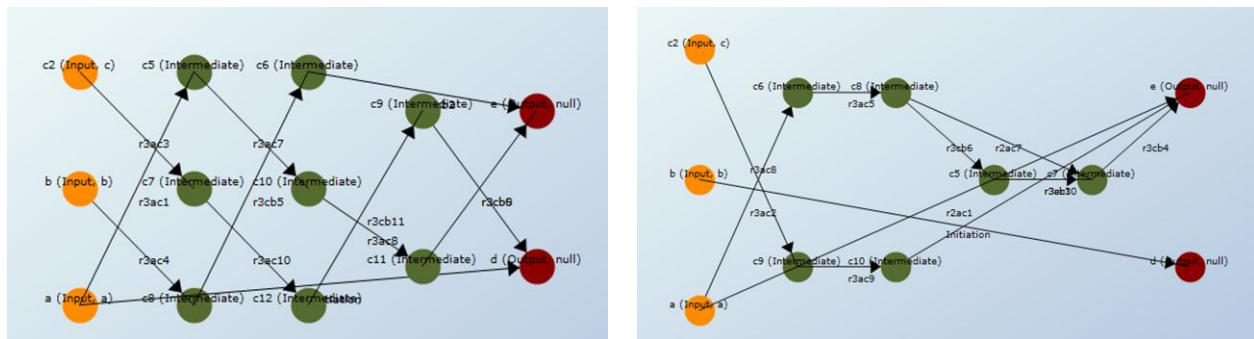Figure 15: Four grammar rules implemented to create the space of feasible neural networks.

Figure 14: Two example neural networks created by invoking the rules from Figure 13 randomly.
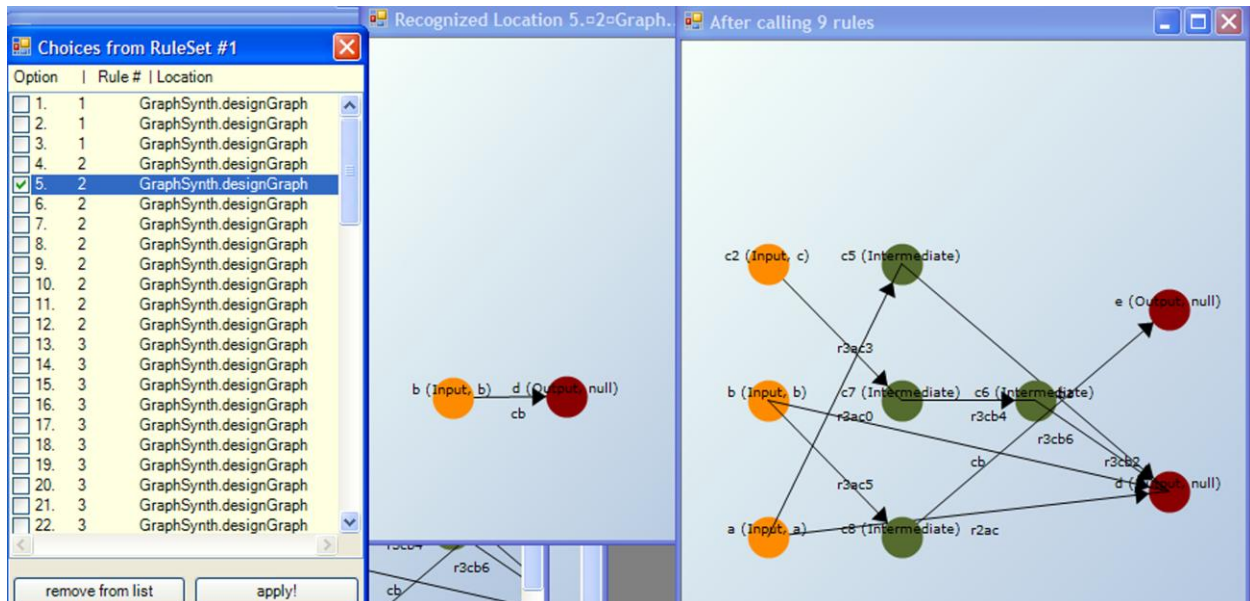

Figure 13: On the left, a list of options is presented to the user to choose which rule and its location to apply to the host shown to the right. In the center, the location of option 5 is shown for arc, cb.

test and synthesize topologies for a specific set of data. Using a best firs search combined with a hybrid optimization algorithm, we are able to find optimal topologies for various meta-modeling problems [35]. Results shown therein are more promising than previous attempts to automate the creation of neural network topologies.

6.2 Mechanical Concept Generator

In this second example, researchers at University of Texas at Austin and at University of Missouri-Rolla are capturing the ways components are used in small consumer products to fulfill particular functions [36, 37]. To date, the researchers have input data from over 23 devices and derived 189 grammar rules for changing a function structure into a configuration of components. In Figure 16, an example seed graph is shown for this problem. The graph, known as a function structure, is common in conceptual mechanical design [38] as it shows the interactions and dependencies of functions required in the product. In Figure 18, four grammar rules from the set of 189 are shown that transform the function structure into a configuration by replacing the function blocks with one or more components.

These rules are defined based on the dissection of real devices and hence the relationship between them is unpredictable but they lead to feasible (and meaningful, in most cases) solutions. From a particular function structure, we have systematically invoked grammar rules in a breadth first search to create over fifty-thousand topologies from a single function structure. One of these configurations is shown in Figure 17. Note in this example that the size of the graph does not necessarily increase from seed to final topology. This graph grammar simply transforms the graph by removing as many nodes as it adds. However, in generative systems such transformations usually results in a more complex graph.
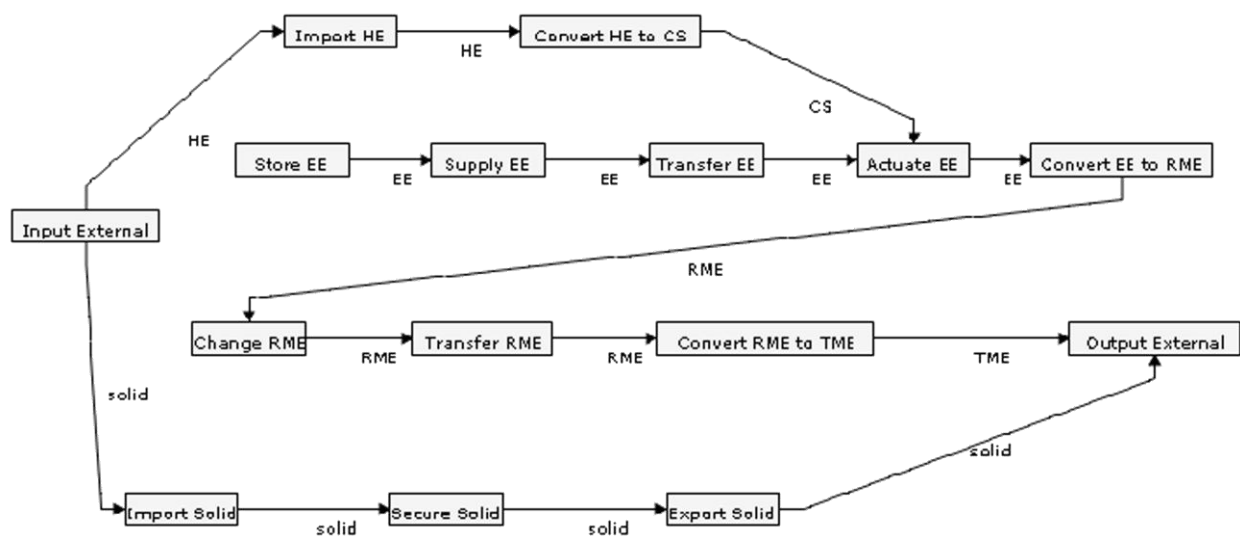

Figure 16: The starting point for the mechanical concept generator grammar is a function structure which is a conceptual design tool comprised of functional blocks and connecting energy, material or signal flows.
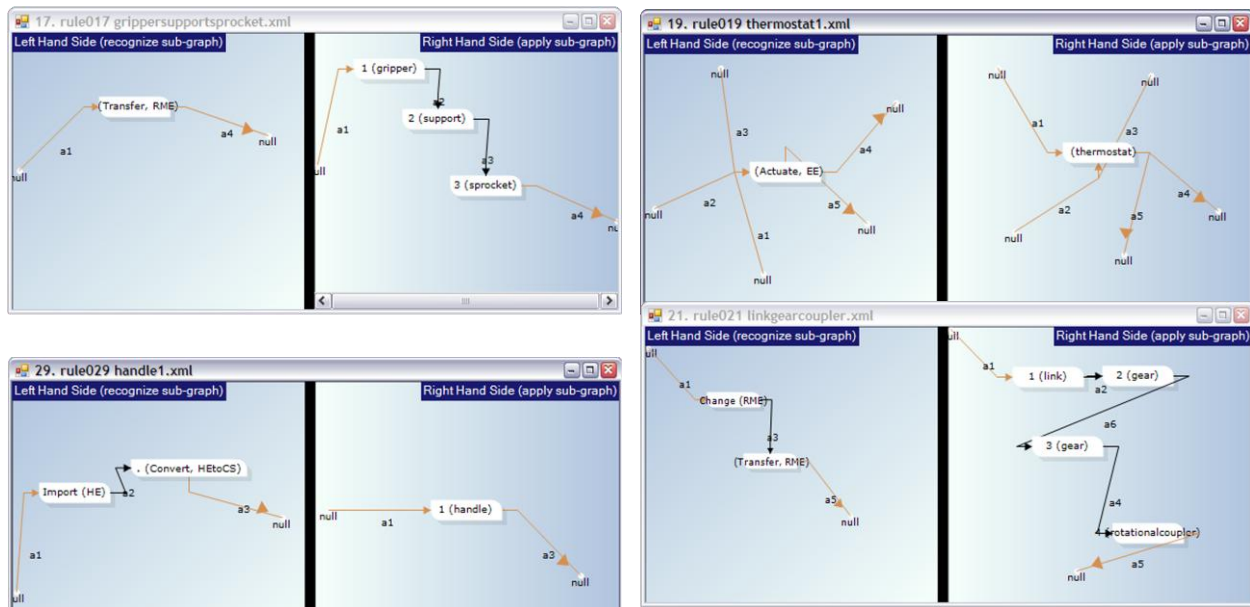
Figure 18: Four rules from the *mechanical concept generator grammar.* The rules replace functions with components that have been observed to have solved those functions in existing devices.
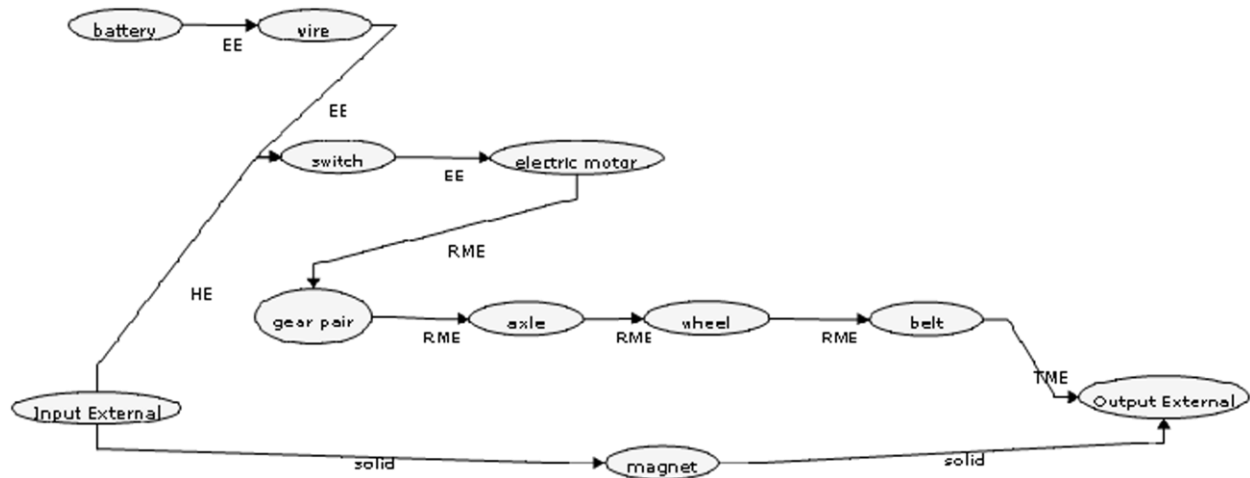

Figure 17: A final configuration of components created for the function structure shown in Figure 16 from the rules within the *mechanical concept generator grammar.*

7 Validation of Approach

As discussed above, the graph grammar approach is ideal for problems where the solution is not known in advance. Unlike a path-planning algorithm where the start and end are known in advance, design is all about finding an end, ideally an optimal end or solution to the problem at hand. As has been noted [39], rule-based systems naturally work in a forward-chaining manner building on previous states. Backward chaining is essentially impossible in design since nothing is known of the final artifact in advance. Rules are constructed to define a tree of possible solutions that is either infinite (in the case of the neural network example) or prohibitively large. Proof for this approach's
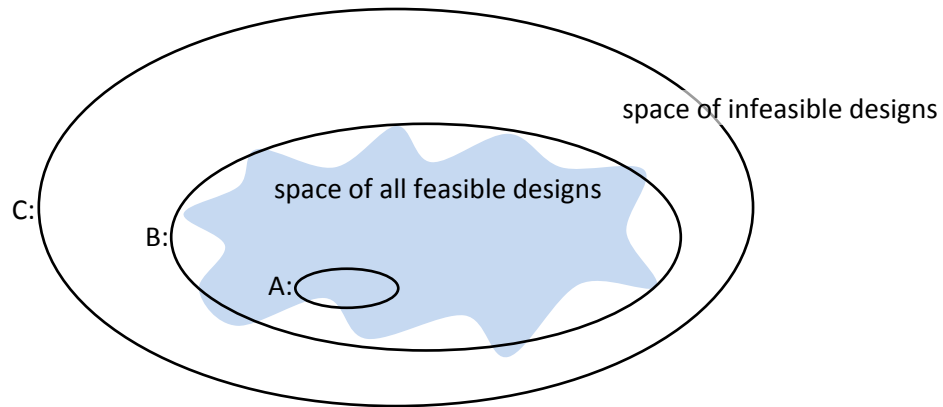
Figure 19: How can the validity of a grammar rule set be determined? In the Venn Diagram, we compare three grammar rule sets that define the solutions captured in A, B, and C.

validity would need to prove that a set of grammar rules (like those shown in Section 6), or the formalism for recognizing and applying the grammar rules (as shown in Sections 3, 4, and 5) properly captures all possible valid designs. Imagine the different grammar rule sets or formalisms that define three different regions, A, B, and C in Figure 19. Clearly the grammar defining Region B is better than A and C. But in cases where B is not available, C is preferred even though a search method would be required to distinguish between feasible and infeasible solutions. It is even possible that in the absence of B and C, A would be accepted even though it only defines a portion of the design space. In the case of the five rule neural network grammar rule set, it is unclear what portion of feed-forward neural networks can be created. The authors of that work feel confident that all feasible feed-forward neural networks can be created, but a proof for this has yet to be devised.

The validity of this formal approach for recognizing and applying grammar rules (as shown in Sections 3, 4, and 5) could be validated as complete if it were capable of generating all possible graphs. More specifically, the formalism should be capable of transforming any valid graph into any other graph. This has been accomplished on multiple occasions, but the results are uninteresting. There are an infinite set of possible graphs, and when labels are considered the space is even larger. Given that any number of grammar rules can be created to make the arbitrary transformation, we find the exercise arduous and the results unsatisfying.

Perhaps a "first principles" approach would be more successful than an empirical one, but one has yet to be devised. Simply, grammar rules can be created to add or remove arcs, nodes and/or labels. On a controlled and limited domain, it can be shown that a single rule (shown in Figure 20a) can create all possible graphs between a seed of n unconnected nodes and the complete graph. The rule in Figure 20a adds an arc between any two unconnected nodes (the **induced** Boolean discussed in Section 3.1 qualifies that nodes 1 and 2 are not connected for the rule to be valid). For a seed of six nodes (as shown in Figure 20b), the complete graph can be reached by consecutively invoking the rule exactly fifteen times. After this point, the rule is no longer valid and no more transformations can be performed. In Figure 20c, the search tree is illustrated. The first level has fifteen possible states, but the branching factor reduces all the way to one at the second-to-last level. Within this tree, there are nearly 73 million unique graphs – all of which are reachable from the seed through application of the single grammar rule.
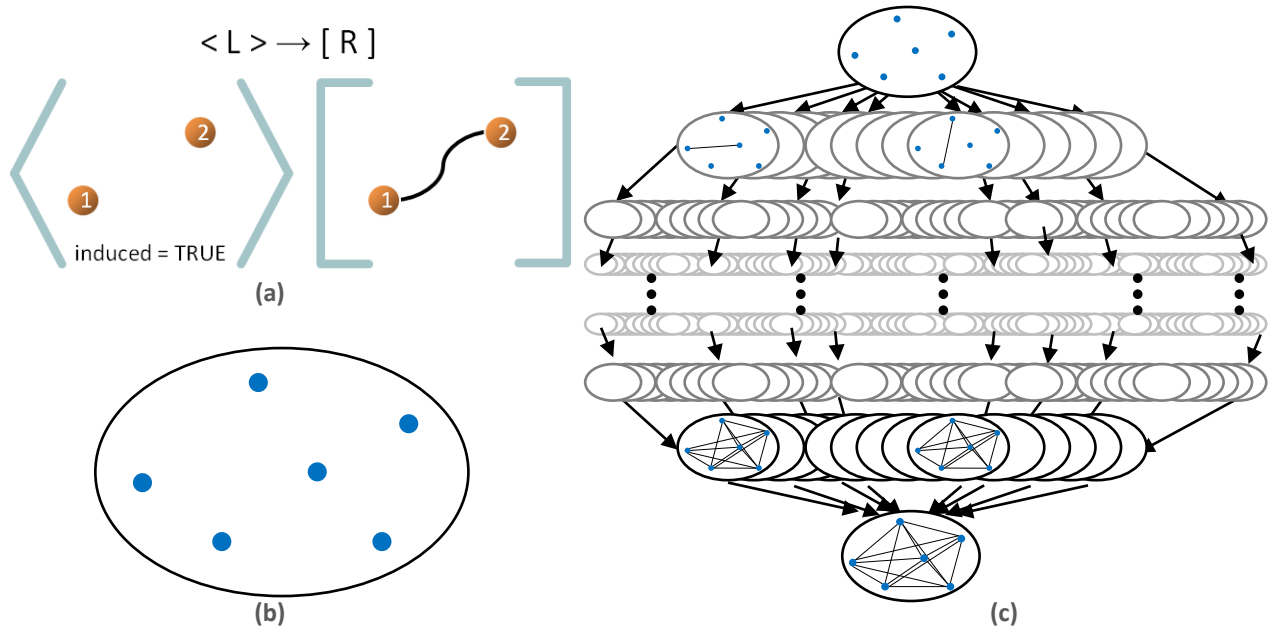
Figure 20: a) a single grammar rule that adds an arc between two unconnected nodes. b) For a seed consisting of n unconnected nodes, c) the search tree terminates with the complete graph after n(n-1)/2 levels.

The validity of the formalism discussed here or the validity of a particular grammar rule set seems to be, at best, application specific. Given the large search spaces that will likely result, researchers in design may be happy to produce designs which appear optimal for a set of needs. In a way, the pursuit of a valid approach or sufficient representation is akin to nonlinear optimization where algorithms are developed and compared empirically. Essentially, the proof or guarantee that a global optimal solution is found is impossible without visiting each solution in the infinite set.

As a result, this method selectively combines published theoretical computer science accomplishments [6, 7, 22 and G. Rozenberg, 24, 25, 27, 28, 40-42] to create a practical foundation for open-ended design problems. The graph grammar concept is beneficial in that it empowers researchers to create algorithms capable of creating and comparing solutions that range in complexity as opposed to traditional representation approaches where an array of parameters restricts the variability of the solution space. In the future, comparison to traditional representations should show that a grammatical approach can create a richer, more varied set of solutions.

8 Conclusions

This paper has presented the development of a graph grammar framework for generative systems. The research brings together advances from theoretical computer science [42], engineering design theory [38], artificial intelligence [10], and shape grammar research [11]. The holistic view of graph transformations offered in this paper allows researchers to develop and test rules easily. Unlike many artificial intelligence methods and the representations offered therein to capture a particular problem domain, the graph grammar approach is particularly useful for design or synthesis. This is

because design produces unknown entities of arbitrary complexity. The most famous automated synthesis approach is likely the XCON system [43] developed to create custom computer systems for Digital Equipment Corporation's VAX systems. This system is in fact rule-based where hundreds of production rules are used to build feasible computer systems. There is no coincidence that the approach endorsed in this paper and the approaches used in past generative systems are rule-based. As has been noted [39], rule-based systems naturally work in a forward-chaining manner building on previous states. Backward chaining is essentially impossible in design since nothing is known of the final artifact in advance.

The methodology presented in this paper has been implemented in a software tool known as GraphSynth [31] which represents the first publicly available graph grammar system created specifically for generative systems (shown in Figures 2e,13, 14, 15, 16, 17, and 18). While there have been numerous computational implementations of specific graph transformation systems, this approach captures a wide variety of transformations (see Figure 10) so that the rules, and the graphs can be stored as static XML files. The procedures for graph recognition and application follow the methods discussed in Sections 3 and 5 which combine the state of the art of graph grammar systems into an efficient implementation. In the future, the implementation may be extended to hypergraphs [44] in which the arcs may connect to more than two nodes or extended to hierarchical graphs where nodes may contain an entire graph within themselves. For many application domains, the computational structures defined in GraphSynth for nodes and arcs may be insufficient in terms of the data and methods stored therein. One can however define subclasses of the node and arc of GraphSynth to represent additional details. This has in fact been the approach adopted in the examples shown in Section 6.

In Figure 4, it is shown how the recognition, choose, and apply functions can be successfully invoked to create a search tree of candidate solutions. This begs the question about how the tree should be searched to identify the best candidate. In the example in Section 6, simple tree traversal methods are used (breadth first and best first), but for larger design spaces, it is likely that specific search methods will need to be developed. Of course, for a given domain the construction of a meaningful heuristic and transition cost function has also an obvious effect on the choice of search as well as the size of the search tree.

Furthermore, values of non-discrete parameters within a graph would seem to mandate a need for real number linear or non-linear programming methods. Optimization has been used throughout the design of complex systems (one could argue that it was developed to do just that). In optimization, a space of valid solutions is searched to find the candidate with the best performance or objective function. Optimization functions on the basis that the design problem is representable by a fixed set of numbers (either integer or real). This provides a clear motivation for the creation of new methods that can search a more open-ended space of candidates in which those candidates are not bound by traditional optimization but rather are organized by the transitions presented in a set of rules that define the space of solutions. Current research [45] is being undertaken to establish these new search methods that borrow from both optimization and tree-searching methods.

Acknowledgements

References

[1]   J. Pfaltz and A. Rosenfeld, "Web Grammars," in *Proc. 1st Int. Joint Conf. on Artificial Intelligence*, Washington, D.C., May 1969, 1969, pp. 609-619.

[2]   H. Ehrig, *Fundamentals of algebraic graph transformation*, 1st ed. New York: Springer, 2006.

[3]   J. Seemann and J. W. von Gudenberg, "Extension of UML sequence diagrams for real-time systems," *Unified Modeling Language,* vol. 1618, pp. 240-252, 1999.

[4]   K. Holscher, P. Ziemann, and M. Gogolla, "On translating UML models into graph transformation systems," *Journal of Visual Languages and Computing,* vol. 17, pp. 78-105, Feb 2006.

[5]   E. K. Antonsson and J. Cagan, *Formal Engineering Design Synthesis*: Cambridge University Press, 2001.

[6]   M. Lowe, "Algebraic Approach to Single-Pushout Graph Transformation," *Theoretical Computer Science,* vol. 109, pp. 181-224, Mar 1993.

[7]   R. Heckel, J. M. Kuster, and G. Taentzer, "Confluence of typed attributed graph transformation systems," *Graph Transformations, Proceedings,* vol. 2505, pp. 161-176, 2002.

[8]   J. Schaefer and S. Rudolph, "Satellite design by design grammars," *Aerospace Science and Technology,* vol. 9, pp. 81-91, 2005.

[9]   H. Göttler, *Graphgrammatiken in der Softwaretechnik: Theorie und Anwendungen (Informatik-Fachberichte)*: Springer, 1988.

[10]   L. Bolc and J. Cytowski, *Search Methods for Artificial Intelligence*: Academic Pr, 1992.

[11]   G. Stiny, "Introduction to Shape and Shape Grammars," *Environment and Planning B-Planning & Design,* vol. 7, pp. 343-351, 1980.

[12]   G. Cagdas, "A shape grammar: The language of traditional Turkish houses," *Environment and Planning B-Planning & Design,* vol. 23, pp. 443-464, Jul 1996.

[13]   G. Stiny, "Ice-Ray - Note on Generation of Chinese Lattice Designs," *Environment and Planning B-Planning & Design,* vol. 4, pp. 89-98, 1977.

[14]   M. Agarwal and J. Cagan, "A blend of different tastes: the language of coffeemakers," *Environment and Planning B-Planning & Design,* vol. 25, pp. 205-226, Mar 1998.

[15]   R. Krishnamurti and C. Giraud, "Towards a Shape Editor - the Implementation of a Shape Generation System," *Environment and Planning B-Planning & Design,* vol. 13, pp. 391-404, 1986.

[16]   S. C. Chase, "A model for user interaction in grammar-based design systems," *Automation in Construction,* vol. 11, pp. 161-172, 2002.

[17]   T. W. Knight, "The Generation of Hepplewhite-Style Chair-Back Designs," *Environment and Planning B-Planning & Design,* vol. 7, pp. 227-238, 1980.

[18]   H. Koning and J. Eizenberg, "The Language of the Prairie - Wright,Frank,Lloyd Prairie Houses," *Environment and Planning B-Planning & Design,* vol. 8, pp. 295-323, 1981.

[19]   N. Hartsfield and G. Ringel, *Pearls in Graph Theory: A Comprehensive Introduction*. library: Academic Pr, 1990.

[20]   R. Diestel, *Graph Theory (Graduate Texts in Mathematics, 173)*. library: Springer, 1997.

[21]   S. Skiena, "Graph Isomorphism," in *Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica* Reading, MA: Addison-Wesley, 1990, pp. 181-187.

[22]   H. Ehrig, G. Engels, H. J. Kreowski, and G. Rozenberg, *Handbook of Graph Grammars and Computing by Graph Transformation: Applications, Languages and Tools*. library: World Scientific Publishing Company, 1999.

[23]   J. Arora, *Introduction to Optimum Design, Second Edition*: Academic Press, 2004.

[24]   H. Ehrig, H. J. Kreowski, A. Maggioloschettini, B. K. Rosen, and J. Winkowski, "Transformations of Structures - an Algebraic Approach," *Mathematical Systems Theory,* vol. 14, pp. 305-334, 1981.

[25]   A. Glikson and J. A. Makowsky, "NCE graph grammars and clique-width," in *Graph-Theoretic Concepts in Computer Science*. vol. 2880, 2003, pp. 237-248.

[26]   M. Nagl, "Bibliography on Graph Rewriting-Systems (Graph-Grammars)," *Lecture Notes in Computer Science,* vol. 153, pp. 415-448, 1983.

[27]   D. Janssens and G. Rozenberg, "Graph-Grammars with Neighborhood-Controlled Embedding," *Theoretical Computer Science,* vol. 21, pp. 55-74, 1982.

[28]   P. M. Vandenbroek, "Algebraic Graph Rewriting Using a Single Pushout," *Lecture Notes in Computer Science,* vol. 493, pp. 90-102, 1991.

[29]   A. Soman, S. Padhye, and M. I. Campbell, "Toward an automated approach to the design of sheet metal components," *AI EDAM,* vol. 17, pp. 187-204, 2004.

[30]   P. Sridharan and M. I. Campbell, "A study on the grammatical construction of function structures," *AI EDAM,* vol. 19, pp. 139-160, 2005.

[31]   M. I. Campbell, "The Official GraphSynth Site," University of Texas at Austin, 2006.

[32]   B. Müller, J. Reinhardt, and M. T. Strickland, *Neural networks : an introduction*, 2nd updated and corr. ed. New York: Springer-Verlag, 1995.

[33]   J. C. F. Pujol and R. Poli, "Evolving the topology and the weights of neural networks using a dual representation," *Applied Intelligence,* vol. 8, pp. 73-84, Jan 1998.

[34]   V. Maniezzo, "Genetic Evolution of the Topology and Weight Distribution of Neural Networks," *Ieee Transactions on Neural Networks,* vol. 5, pp. 39-53, Jan 1994.

[35]   C. Vempati and M. I. Campbell, "An Automated Approach to Generate Neural Network Topologies," in *Artificial Neural Networks in Engineering*, St. Louis, MO, 2007.

[36]   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. 19, 2008.

[37]   T. Kurtoglu and M. I. Campbell, "A Graph Grammar Based Framework for Automated Concept Generation," in *International Design Conference, Design'06*, Dubrovnik, Croatia, 2006.

[38]   G. Pahl and W. Beitz, *Engineering Design: A Systematic Approach*: Springer Verlag, 1996.

[39]   P. H. Winston, *Artificial Intelligence Edition (Addison-Wesley series in computer science)*. Reading, Mass.: Addison-wesley, 1984.

[40]   H. Ehrig, *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. library: Springer, 2006.

[41]   M. Nagl, "Formal Languages of Labeled Graphs," *Computing,* vol. 16, pp. 113-137, 1976.

[42]   P. Padawitz, "Graph-Grammars and Operational Semantics," *Theoretical Computer Science,* vol. 19, pp. 117-141, 1982.

[43]   J. Mcdermott, "R1 (Xcon) at Age 12 - Lessons from an Elementary-School Achiever," *Artificial Intelligence,* vol. 59, pp. 241-247, Feb 1993.

[44]   A. Habel, *Hyperedge replacement : grammars and languages*. Berlin ; New York: Springer-Verlag, 1992.

[45]   J. Patel and M. I. Campbell, "An Approach to Automate and Optimize Concept Generation of Sheet Metal Parts by Topological and Parametric Decoupling," *Journal of Mechanical Design,* vol. in review, 2009.