Grammatical Approaches to Design

by

S. Mullins, J. R. Rinderie

EDRC 24-27-90

Grammatical Approaches to Design

Scott Mullins
Research Assistant
Mechanical Engineering Department
Purdue University
West Lafayette, IN. 47907

James R. Rinderle
Associate Professor
Mechanical Engineering Department
Carnegie Mellon University
Pittsburgh, PA. 15213

Abstract

A grammar is a definition of a language written in a transformational form. To the extent that design requirements and designed artifacts can be represented by some language, and to the extent that design is a transformation from function to form, grammars may facilitate the development of theories and methods for design. Furthermore, the computational complexity of various grammatical formalisms may provide a foundation upon which to base complexity measures in design. We discuss grammatical formalisms and give examples of how grammars might facilitate design automation.

The syntax and lexicon of a formal language are analogous to the configuration and the components in an engineering design. Similarly, the complexity of alternative grammatical formalisms is related to the complexity of design, and to the specific representations chosen. Attribute grammars are convenient for managing relationships among engineering parameters, even when configurations are not known a priori. In this way the grammatical formalisms provide a bridge between conventional rigid parameterizations and ad hoc design representations.

1. Introduction

The degree of automation in mechanical design is modest in comparison with that in other domains, such as integrated circuit design. One difficulty in automating mechanical designs is the highly integrated, tightly coupled nature of mechanical devices which precludes a direct application of various *decompose and transform* methodologies [Rinderle 87] [Finger 89]. It is also the case that relationships between three dimensional geometry and configurational alternatives preclude *morphological* methods often inherent in many expert system applications. Alternative representations for the function and form of mechanical devices and alternative methodologies for transforming function to form are needed to advance the state of the art of mechanical design education and to provide a foundation upon which to base designer assistance systems. One approach to alternative representations is based on formal languages [Sony 80] [Fitzhorn 86].

The *transformational* nature of grammars is critical from a design perspective since design can be viewed as a transformation of functional requirements to a physical device [Mostow 85] [Rinderle 87]. Furthermore, grammars can be thought of as formal computational procedures for generating and computing the elements of a language, perhaps a language of mechanical design. I this light, grammars are relevant to engineering design.

The formalism of grammars has been the subject of a great deal of research lately in the field of design theory and design automation. There are several reasons why the idea of language formalisms for mechanical design seem attractive.

- Formal methods facilitate the characterization of the mechanical design problem more precisely.
- The absence of a fixed structure in mechanical design decision making, e.g. hierarchical decomposition, makes formal but less rigid methods attractive.
- Recent work in the representation of geometry as strings or graphs [Fitzhorn 86] makes methods which operate on strings or graphs useful.
- A grammatical approach to design makes it possible to leverage related work in computer science.

We proceed with a review of grammatical formalisms in the context of design.

2. An Example of a Design Grammar

A grammar is a structured way of describing the relationships between the entities of a language, whether that language be English prose, a computer programming language, a shape language or perhaps a language of mechanical function and form. A grammar defines the syntactic structure or rules of the language and hence provides the framework upon which to interpret the language. English grammar, for example, specifies a structure among sentences, noun phrases, modifiers, adjectives, adverbs, and nouns that make it possible to unambiguously interpret phrases such as:

I hit only him.
Only I hit him.

A grammar itself specifies only the syntax of a language. The syntactic correctness of an element of the language is necessary but not sufficient to insure that the element is a meaningful message. The interpretation of the language completes the "string to thing" transformation. Interpretation or the *semantics* of the language can be defined as anything pertaining to the meaning. The semantics of a language are based upon both the syntactical information gained from the grammatical analysis and from the lexical information inherent in the individual components of the language. For natural languages the lexical information is contained in the dictionary definition of the words. For mechanical designs the lexical information can be thought of as the behaviors and features associated with individual components of the design.

We present a simple design grammar to illustrate some key points. The grammar generates two dimensional, truss-like bridge structures which are constructed from triangular substructures and spans. Adjacent triangular structures are understood to be connected at their apexes as well as at the base. Triangular structures connect to the

mid-point of an adjacent span. The grammar is designed to prohibit adjacent spans since such a structure would not be stable. The bridges can be of any length. An example of an acceptable structure is shown in Figure 1.

Since any bridge structure can be described as a sequence of triangular structures and spans **we may** describe **any** specific structural configuration as **an** arbitrary length **sequence of the characters t and 5. Since these and only these symbols will be** present in **a** complete description **of a** bridge **structure, we say that these are the *terminal*** *symbols* of the bridge language.

The grammar will generate sequences of t's and s's which are valid bridge structures. If we begin with a generic bridge, indicated as B, we may write transformation rules which elaborate the generic bridge into one of possibly many allowable basic configurations comprised of substructures. Each substructure is subsequently elaborated into other substructures and ultimately into *terminal* components.

A bridge, B, for example, consists of a rigid substructure, R, which in turn can either end in a span or not end in a span, R^ES* ^DES respectively. A rigid section whicft docs not end in a span must consist of a triangular structure alone or some rigid structure followed by a triangular structure. Similarly, a rigid structure which does end in a span must consist of either a single span or some rigid structure ending in **a** span, however, the rigid structure preceding the span must itself not end in **a** span so as to guarantee stability between adjacent sections. These compositional rules or *productions* in the bridge grammar may be shown as:

```
B       -*     R
R       —>     RNES ' RDES
RNES    ->     T       I RT
RDES    "*     S       I RNESS
S       ->     S
T       ->     t
```

The I indicates that the symbol on the left hand side of the rule can be transformed into either of the strings on the right hand side. The symbols, B, R, R^ES'^{R}DES' T' ^'^{donoc} appear in the final language but are useful in constructing the final language. These symbols are referred to as the *non-terminal symbols* of the language. The symbol B is a special symbol called the *start symbol* from which the entire bridge language can be generated. The interpretation of the individual symbols, apart from their relationship to the other members of the string, is the *lexical* information of the language.

The semantics of the language define the interpretation of the string as a whole. The implicit triangle-triangle and triangle-span connections, for example make it possible to understand how a string of t's and s^{9}s can be thought of as a bridge. Semantics specified in this way are called the *static semantics* because the interpretation is the same no matter what the overall form of the string. The bridge corresponding to the string, **t** s **t t** t_f is shown in Figure 1 along with a tree that shows how the string is derived using the grammar. The start symbol, B, representing a generic "bridge,^{11} is at the top of the tree and the "leaves" of the tree are the symbols in the string.

We have just seen a simple example of how a design grammar can be used to generate a design alternative. If instead we dunk of the grammar operating in reverse, we will be able to determine whether a particular design alternative, in this case represented by a string of *Vs* and s's, could have been generated by the grammar and therefore determine whether or not a design alternative is valid. In this case we are using the grammar to *parse* an element of the language. The bridge grammar productions were chosen to preclude any bridge configuration with two adjacent spans. An "invalid" bridge represented, for example, by the string, t s s 11, can not be parsed to recover the start symbol and therefore, the string is not part of the bridge language, and is therefore, not a valid bridge configuration.

Parsing not only provides a test of validity but also results in the identification of a

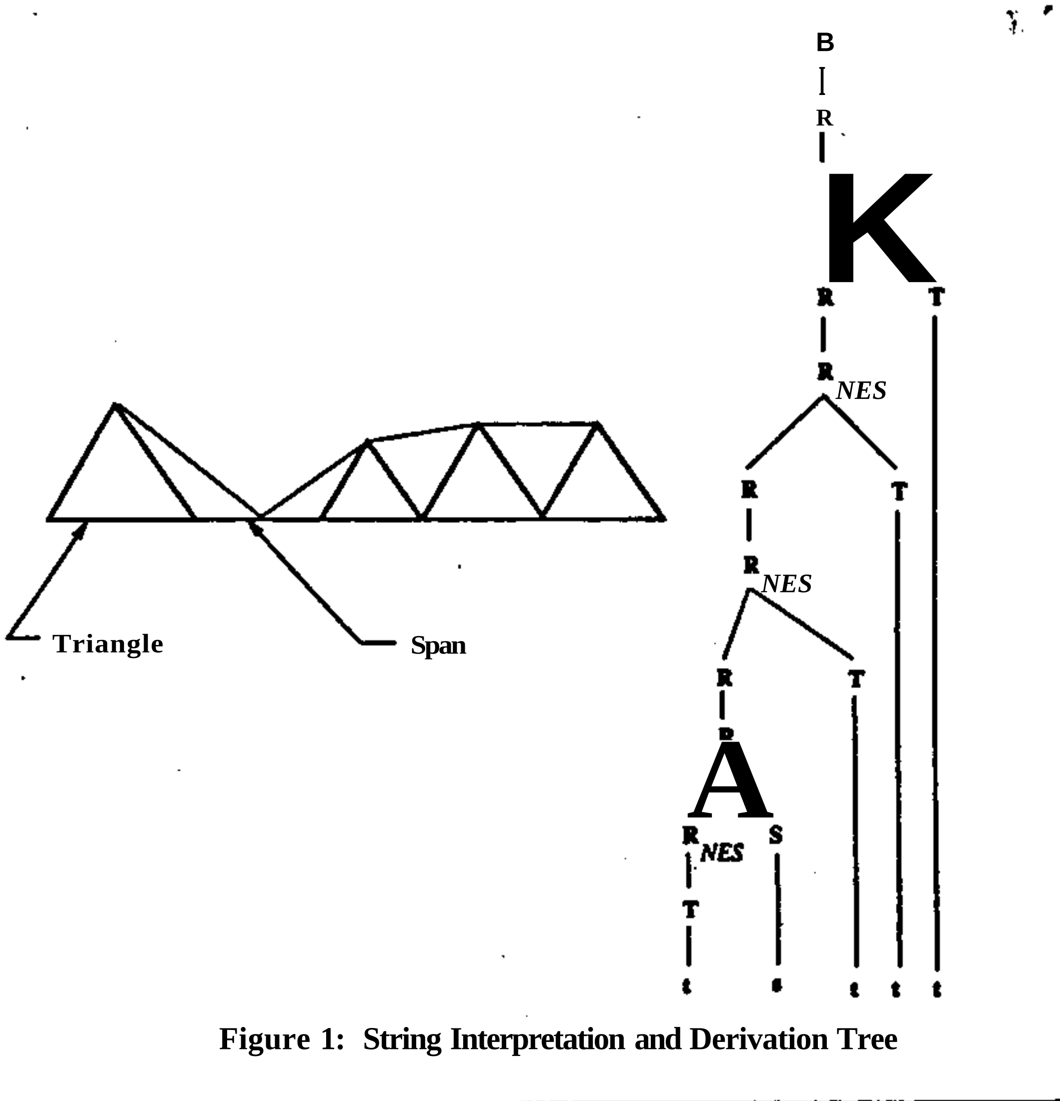


Figure 1: String Interpretation and Derivation Tree

sequence of productions which produces the string from the start symbol. The productions themselves and the meaningful non-terminal symbols which they produce are themselves useful in interpreting and operating on the language.

Although grammars may be used to parse a string as is common in the the interpretation of the English language or in the translation of computer languages, they may also be used to generate elements in the language. In the context of engineering design, generation corresponds to the identification of design alternatives [Stiny 80]. Controlling a grammar so as to focus the generation of alternatives is a key issue.

3. Theoretical Properties of Grammars

Grammars are both quite general and powerful. They allow for a very compact representation of a possibly infinite set of strings. Furthermore, the transformational paradigm inherent in grammars allows for a wider variability than is possible with other types of procedural formalisms. The generality, however, has a direct impact on the difficulty in parsing and in determining the characteristic of a language. The properties of various grammatical formalisms were first explored by Chomsky [Chomsky 66] and their computational properties were derived through their relationship to Turing machines [Moll 88] [Revesz 83]. We discuss various of the grammatical formalisms and their characteristics as a foundation upon which to discuss the complexities of mechanical design formalisms.

3.1. String Grammars

The bridge grammar presented in the previous section is an example of a string grammar in that it prescribes transformations on, and therefore, relations among, symbols in a string. String grammars are used to specify the syntax of natural languages and of most programming languages. In general the formalisms of other types of grammars, such as graph grammars, shape grammars, and structure grammars, are extensions to the string grammar formalisms.

String grammars are defined in terms of two sets of characters, a set of productions which transform strings of characters into alternative strings and a special character called the start symbol. Formally a string grammar consists of four elements, also called a 4-tuple:

$G = \{N,T,P,S\}$

where N is a set of non-terminal symbols
T is a set of terminal symbols with $N \cap T = \emptyset$
P is a set of productions
S is a special symbol called the start symbol.

The representational power of a string grammar depends strongly on the form of the allowable productions in the grammar. The form of the productions also have a significant impact on the computational complexity of characterizing a language or parsing an element of that language. In the following sections we will look at various grammatical formalisms and discuss these characteristics.

3.2. Context-Free Grammars

In a context free grammar the left hand side of all productions must consist of a single non-terminal symbol, i.e., they take the form:

$$v \rightarrow s_1 s_2 \ldots s_n \text{ with } v \in N, s_k \in N \cup T$$

Each production causes a single non-terminal symbol v to be replaced by a string of terminal and non-terminal symbols. In this case the firing of a production depends on a single symbol, i.e. the *context* of the non-terminal symbol is not relevant The bridge design grammar in the previous section is a context-free grammar. A derivation or parse for a context-free grammar is always representable by a tree such as shown in Figure 1 because a single symbol is always replaced by one or more symbols. In this sense it is a hierarchical derivation which continues until the string contains only terminal symbols.

The formal definition for the bridge grammar presented in the last section is:

$N = \{R, R_{NES}, R_{DES}, S, T\}$
$T = \{s, t\}$
$5 = \{B)$

and the productions for the grammar are the transformation rules as defined before. The bridge grammar is infinite in that there is no limit to the possible length of the strings that can be generated.

It is important to note that while the bridge *grammar* is context-free the bridge design problem itself is not The placement of the spans and trusses do in fact depend on the type of bridge element located next to it This is due to the constraint that spans in the bridge may not be adjacent This "physical" context-sensitivity is taken care of by a careful choice of the non-terminal symbols and productions. As will be demonstrated later, alternative grammatical formulation may incur severe computational penalties.

Two grammars Gj and G_2 are said to be equal iff they can generate the same language, *L(G{)* = $L(G_2)$. In general, the problem of determining whether or not two grammars generate the same language is undecidable.[1] The problem of determining whether an arbitrary string is a member of the language of a context free grammar is decidable. Moreover, the time complexity for the parse is at worst $O(n^3)$, where n is the length of the string of terminal symbols being parsed.

Several other theoretical characteristics of context-free grammars are worth noting. A grammar is said to be ambiguous if there are multiple parse trees for the same string in a language. The question of whether or not a grammar is ambiguous is undecidable. It is also undecidable, in the general case, as to whether or not the language of one context-free grammar properly subsumes the language of another grammar, or L(Gj) c $L(G_2)$.

[1]A property is said to be *decidable* if an algorithm can be written that will compute that property in i finite amount of time for any member of that class of problems. A property can be undecidable for the general case and be decidable for specific problems in that class.

Despite this, given a finite language L_x and a context-free grammar *G* it is dccidable whether or not Lj *Q L^G)* and I, *nL^G)* = 0. It is also decidable if the language of a grammar, *Up) is infinite. The procedure is to find a non-terminal symbol A** such that **there is a derivation:**

A -*...•-> *XAY*

This **type of derivation denotes a recursive grammar.**

The proofs for **these** properties **can be found** in **[Revesz** 83]. **In general the** proofs follow from the relationship between grammars and Turing machines.

The preceding theoretical results along with **the** ones presented later in the paper are important in understanding the implementational difficulties associated with any particular model of the design process. The type of grammar needed to generate or parse a particular language is not always obvious, as will become more apparent in the next section. A strictly hierarchical evolution of a design can be easily captured in a context-free grammar. Since context-free grammars have the most advantageous computational properties of all the grammars this is an attractive possibility, it is, however, unrealistic to expect that all design problems, particularly mechanical design problems, can be simplified to this extent Nevertheless, a careful formulation of the grammar used to model the design process will mediate against the computational complexity of the problem.

3.3. Context-Sensitive Grammars

The definition for context-sensitive grammars takes the same form as for the context-free grammars. That is, a CSG also consists of a four-tuple;

G* = *{N9T9P9S)

where the symbols have the same meaning as for the CFG.

Context-sensitive grammars differ from context-free grammars in the restrictions placed on the form of the productions. Whereas the productions in context-free grammars have a single non-terminal symbol on the left hand side, context-sensitive grammars can have strings of terminals and non-terminals that contain at least one non-terminal symbol. In a CSG, the productions replace one of the non-terminals in the left hand side with a string of terminal and non-terminal symbols.

The basic form of the productions for a CSG is [Revesz 83]:

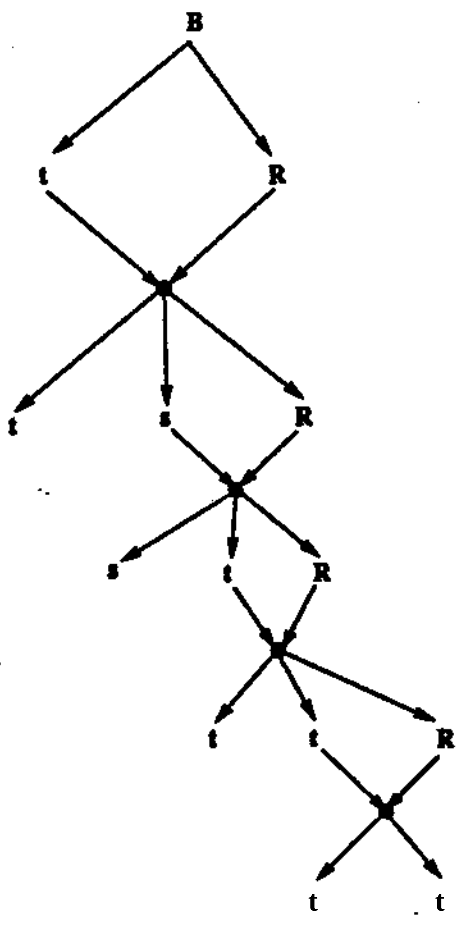
$^{s}l^{v}$ $^{S}2$ "* ^{5}1^2

where *v e N* and $S_{i\%}$ S_2* and *P* arc strings in *(N* u T), and *P * X*

Context-sensitive grammars provide **a** convenient bridge between string grammars and graph grammars. Even though the final output of a context-sensitive grammar is a string of non-terminals, the derivation for it is most easily represented by a graph rather than a tree. The graph representation for a context-sensitive grammar derivation is shown in

Figure 2. This grammar is a context-sensitive alternative to the context-free bridge grammar presented in the previous section. The derivation graph shown produces the same string, and therefore the same bridge as the first bridge grammar example.

A careful examination of the productions in Figure 2 will also reveal that there is more than one derivation graph and therefore more than one sequence of productions that will produce the string shown. This is an example of grammatical ambiguity. If the sequence of productions is critical for the interpretation of the string, for instance if the production sequence for the bridge grammar were a guide to the fabrication of the bridge, then this ambiguity becomes a significant feature of the representation. If there is more than one way to put together a bridge, this may in fact provide a more accurate model of the domain. It is, however, important that this issue be recognized and understood.



Productions:

B —*• sR I tR
SR —*> stR I st
tR —^ tsR I ttR | ts | tt I tRt

Non-terminals: R

Terminals: t,s

String - tsttt

Figure 2: Derivation Graph for a Context-Sensitive Grammar

It may seem at first glance that the context-sensitive bridge grammar is simpler than the context-free grammar. While the CFG has more productions and intermediate symbols than the CSG, the CFG has certain computational advantages over the CSG. This is an important point in any planned implementation of a design grammar. Context-sensitive grammars can explicitly incorporate aspects of inter-component coupling, at the expense of computational complexity, however, this is not the only way that this coupling can be handled. A cleverly chosen grammar can ease these difficulties.

It is provable that context-sensitive grammars are inherently more powerful than context-free grammars, meaning that there are languages that can be generated by CSG's that can not be generated by CFG's. However, it is impossible to determine whether or not an arbitrary language can or can not be generated by a context-free grammar.

Context sensitive grammars are more complex than context free grammars. Any question that is undecidable for context-free grammars is also undecidable for context-sensitive grammars. Other interesting properties of context-sensitive grammars are that it is undecidable as to whether or not an arbitrary CSG is infinite and whether or not an arbitrary CSG generates an empty language (i.e. $L(G) = \emptyset$). A positive property of context-sensitive grammars is that the membership problem is decidable. In other words, for any arbitrary string of terminal symbols P finding out if $P \in L(G)$ is possible, but not in polynomial time.

3.4. The Chomsky Hierarchy

Context-free grammars and context-sensitive grammars fit into a broader family of grammar types called the *Chomsky hierarchy*. This hierarchy orders the grammars according to their generative power and according to their computational complexity. It should not be surprising that there is a direct relationship between these two properties in that as the generative power increases the difficulty associated with computing it also becomes greater. The hierarchy is based upon restrictions applied to the form of the production rules used in defining the grammar.

The grammars in the Chomsky hierarchy are said to be Type i if they conform to the following restrictions:

- i=0: Phrase Structure Grammars.
  No restrictions to the production rules. Type 0 grammars have the greatest generative power. For an arbitrary Type 0 grammar determining whether or not a string w is derivable from a string y is undecidable. This is the membership problem for arbitrary grammars. Graph grammars generally fall into this category.
- i=1: Context Sensitive Grammars.
  Every production rule has the form

$$Q_1AQ_2 \rightarrow Q_1PQ_2$$

where A € *N* and the strings *$Q_{l9}P,Q_2$* are in (W u T)*[2], and *P * X* (the null string). The only exception to this last restriction is that the start symbol may be in the production *S* -> X but then S may not appear in the right hand side of any production. Type 1 grammars are also loiown as *length-increasing* grammars.

- /=2: Context Free Grammars.
  The production rules are restricted to the form:

  A -» FwhcreA € Wand/Msastringin $(N \cup T)^*$

- i=3: Left Recursive Grammars.
  The production rules are restricted to the form:

  A- RB or A -> R* where *A_9B e N* and the string/? is in *T**.

  These are also known as the *regular* or *finite-state* grammars.
  LR grammars are unambiguous.

A language is said to be Type i if it can be generated by a Type i grammar. The'following assertions about the hierarchy of the languages can be made [Revesz 83].

meaning that every Type 3 language is also Type 2_f and so forth.

The importance of this classification scheme is that it provides a simple test for any grammatical model used in a design system. By comparing the productions in the design grammar to the restrictions in the Chomsky hierarchy it is possible to quickly determine some important computational properties of the design system.

It is of interest in our discussion to note that Type 0 grammars are Turing equivalent. In other words they display the same computational complexity and power as do Turing machines. This fact forms the basis for many of the assertions regarding the computational properties associated with the grammatical formalisms discussed in this paper. Additionally it has been postulated that design is inherently Turing equivalent [Fitzhorn 88] [Harrison 74]. As such it is interesting to note the number of undecidable properties associated with Turing machines. These difficulties certainly mirror those of design in that it is often very difficult to determine whether or not a particular functional specification can be met A detailed discussion of Turing machines is presented in [Lewis 81].

[2]The symbol * indicates the set of all possible combinations of the symbols contained in the set. For this case it indicates that *P*, *$Q_{x\%}$* and *Q_2* can be any combination of terminal and non-terminal symbols, including combinations that contain repeated symbols. The inclusion of strings that have repeated members makes this different than the power set

3-5. Graph Grammars

To apply the string grammar formalism to design requires that the design be reprcsentable as some string of symbols. Mechanical designs however, consist of many different components configured in space and are more naturally represented as graphs. Although any graph can be represented as a string, doing so often obscures useful topological information, e.g. physical adjacency, which is explicit in a graph structure. It is useful, therefore, to consider an extension of the string grammar formalisms to graphs.

The concepts underlying graph grammars are fundamentally the same as those for string grammars. Differences arise only in specifying unambiguously how each of the productions is to operate. Specifically questions of graph matching, sub-graph removal and the *embedding* or attachment of some new graph to the parent graph.

A graph consists of a set of vertices connected together by a set of arcs. Formally we say that a graph *g* is defined as *g = (V,B)* where *V* is the set of vertices in the graph and *B* the set of arcs. In this context vertices are also referred to as nodes or knots and the arcs are also called edges. There can be labels for the nodes and/or the arcs taken from the node label alphabet, Z_n, and the arc label alphabet, Z_{fl}, respectively. The arcs can b$ directed, in which case the graph is a *di-graph,* or they can be undirected. A self-loop in a graph is an arc that starts and ends at the same node.

There are several operations that apply to graphs that are important in the definition of graph grammars. We say that a graph *g = (V,B)* is *isomorphic* (equal) to graph *h* s (V*,fl*) iff there is a function *Hf* such that *F is bijective from Vto V* and from BtoB*. By bijective we mean that there is both a one-to-one and an *onto* correspondence between the two sets. That is, for every node in *g* there is *one* equivalent node in *h* and vice versa and for every arc between two nodes in *g* there is *one* equivalent arc in *h* and vice versa . The definition of isomorphism can include or exclude the use of node and arc labels.

A graph *g* is said to be included in graph G if there is a *subgraph* of G, A, such that *g* is isomorphic to *h.* In any graph G there could be several subgraphs that are isomorphic. Finding a particular subgraph in a larger graph is called graph matching and is np-completc[3] for the general case.

The graph difference operation, given that a graph *g* is included in a graph G, is defined to be:

$$G - g = (V^{**}, B^{**})$$

where $G = (V,B)$

$$V^{\cdot} = V - V^{\cdot}$$

and *B*** is the set of all the arcs in *B* for which neither of their nodes has been deleted. In

[3]Np-compIete means that (he problem is <u>not</u> solvable in polynomial time. NP stands Tor *non-deterministic polynomial.*

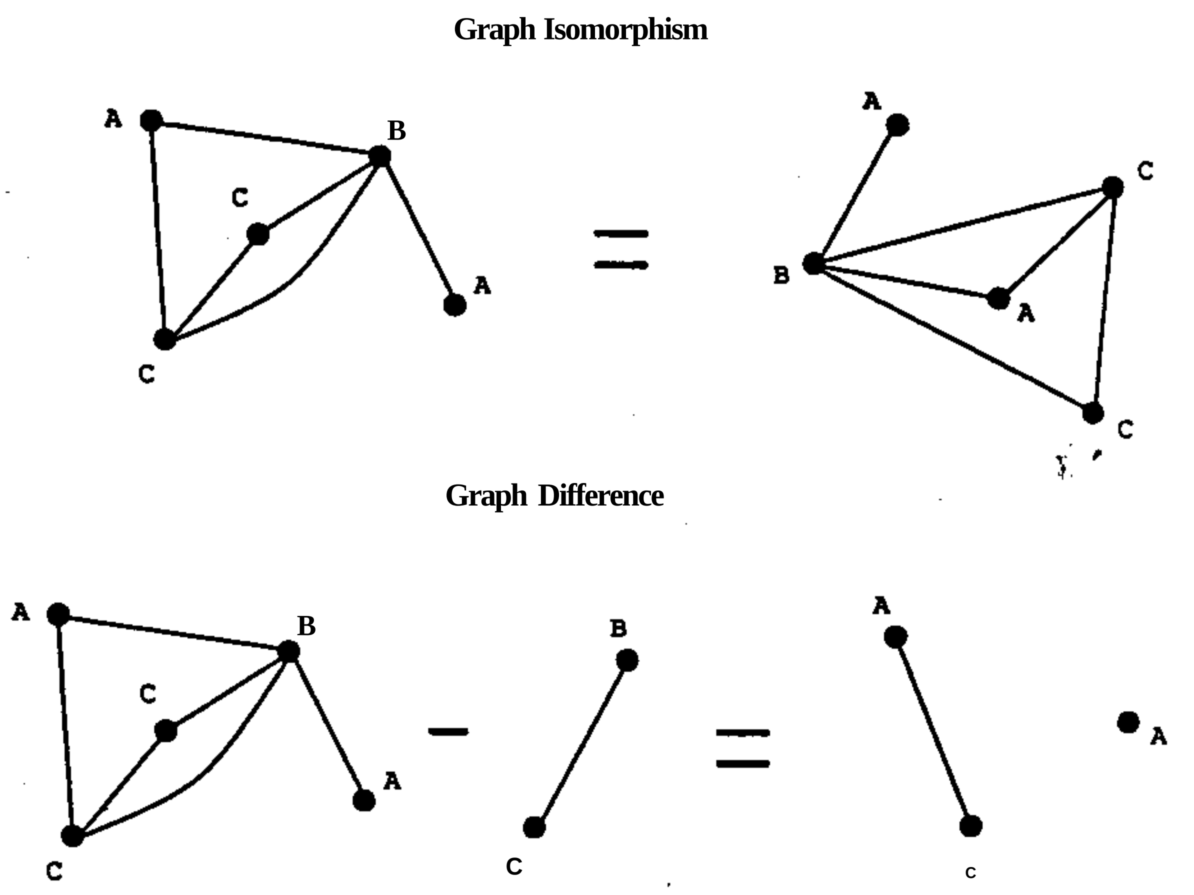


Figure 3: Graph Isomorphism and Graph Difference

other words the difference operation is performed on the graph *G* by removing the nodes and arcs of *g* from it including any dangling arcs left by the removal of the nodes. Figure 3 shows these operations as performed on an example graph. It should be obvious that there are two subgraphs in the graph of Figure 3 on which the graph difference operation could be performed. Either of these subgraphs could be removed by the graph difference operation. In the graph difference and inclusion operations defined above, the graph *G* is called the *parent* graph and graph *g* is called the *target* graph. The set of nodes directly connected to the target graph by an arc is called the *neighborhood* of the target graph.

A graph grammar is defined as a four-tuple in much the same way as were string grammars. [Ehrig 78] [Nagl 78] [Ehrig 86] [Nagl 86].

$G = (Z_n, Z_a, P, S)$

where £„ is the set of all node labels.

Z_a is the set of all arc labels.

P is the set of productions of the grammar.
S is the start graph.

The productions of a graph grammar *G* are defined by the triple (g_t, g_d, E) where g_t is the target graph of the production, g_d is the *daughter graph*, and *E* is the *embedding function* of the production. The basic approach to graph rewriting using grammars defined in this way is called the LEARRE method. LEARRE is an acronym that stands for the sequence of operations performed in applying a production: Locate an isomorph of the target graph g_t in the parent graph, establish the Embedding Area in the parent graph, Remove g_t, Replace g_t with the daughter graph g_d, and Embed it to the parent graph. The embedding rule *E* for a production determines how the daughter graph will be attached to the parent graph. Figure 4 shows the process of applying a production to a graph.[4] For the production shown in Figure 4 the embedding rules are:

1. Connect node(1_c) to *neigh*(1_a)
2. Connect node(1_d) to *neigh*(1_b)

Where *neigh* indicates the neighborhood of the node label in parenthesis. The subscripts of the node labels in the production shown in Figure 4 are used only to clarify the explanation of the embedding rules. The subscripts do not play a role in the matching, graph difference, or embedding operations. Figure 4 is a good example of why graph grammars are of interest. Graphs have the ability to explicitly represent the connectivity between elements of the language. The graphs in Figure 4 could be represented by strings but much would be lost in terms of our ability to extract information from the representation. Representational convenience and computational complexity are both critical to any graph grammar implementation.[5] This motivates the study of well defined classes of graph grammars for which certain properties are known.

There are many specialized variants of the graph grammar formalism, including the edge-label controlled grammars and node-label controlled (NLC) grammars [Rozenberg 86] [Janssens 82]. In these formalisms the productions are restricted to replacing a single arc or a single node, respectively.

4. Attribute Grammars

Grammatical formalisms are a convenient means to generate or to verify the validity of configurations of mechanical components. When we think of a description of a design, however, we think not only of a configuration but also of a specification of parameters or attributes of the components which comprise that configuration. The formalism of

[4]The graph production shown in Figure 4 is a *behavior preserving* transformation as applied to bond graphs. Bond graphs are used to model the dynamics of physical systems. Since bond graphs are graphs they are convenient in the formalism of graph grammars. A more detailed discussion of grammars for bond graphs is presented in [Finger 89].

[5]Habel [Habel 86] notes that "It is easy to introduce a complicated mechanism of graph rewriting so that the general framework becomes nearly infeasible, the theoretical achievements are rare and poor, and the best you can learn is the undecidability of everything interesting."

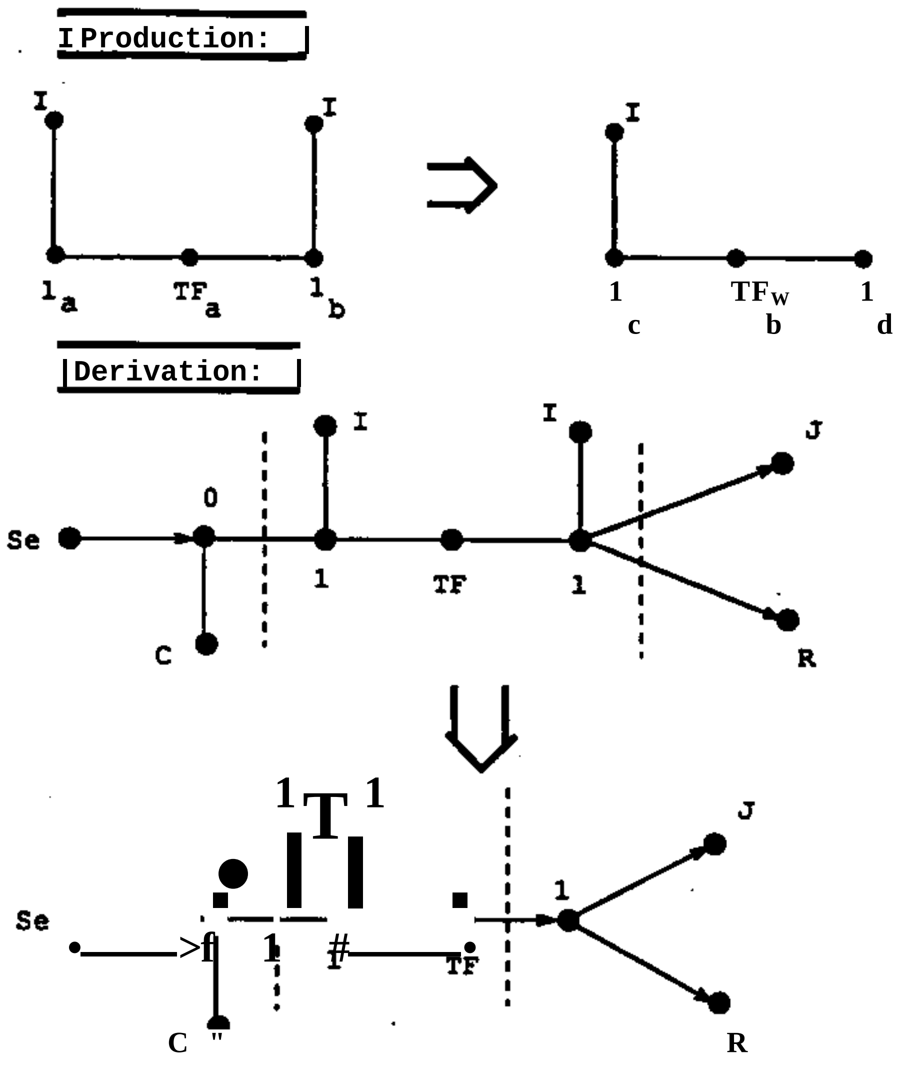


Figure 4: Application of a graph production

attribute grammars is convenient for including, propagating, and computing engineering attributes of components and configurations.

In an attribute grammar, we associate with every symbol in the grammar a set of attributes and ultimately values for those attributes. The attribute-value pairs for any symbol can be related to the attribute-value pairs for other symbols. The rules, or *attribution relationships,* among the attributes are not directly associated with any string or graph fragments produced by the grammar but rather they are associated with the individual productions that generate these language fragments. As a result, relationships among attributes of symbols need not be static or invariant but rather may depend on the specific productions employed to produce the language fragment

Formally an attribute grammar is a grammar where every symbol, *X* € *(NKJT)* has associated with it some finite set of attributes denoted *AttrfX)* which are referred to individually as*XM* where *a e AttrQc)*. We also associate with each production,/? € P, a set of attribution rules which specify relationships among attributes. These attribution rules may, in the most general case, assert an arbitrary relationship among any or all of the attributes of any or all of the symbols used in the production. For a more complete description see [Knuth 68] and [Deransart 88].

Consider briefly how an attribute grammar could be useful in the context of engineering design. A boom grammar is a simplified version of the bridge grammar described previously: It generates only sequences of connected triangular structures. We associate with each triangular structure and with collections of triangular structures, attributes corresponding to forces, sizes and weights. The modified grammar, including the definition of attributes and attribution relations is shown in Table 1.

The attributes in this grammar correspond to the dimensions and forces in an end loaded boom structure, as shown in Figure 5. The attribution relations among ihe' boom dimensions and forces are very simple and fall into a special case of attributional relationship referred to as *inherited.* In this case the attributes associated with the symbol on the right hand side of any production depend only on the attributes of the symbols of the left hand side of the production. As a result, when the grammar is used to generate designs (from the generic boom start symbol, B_9 and attributes corresponding to the desired length, height and end load) it is possible to compute substructure dimensions and forces in parallel with the grammatical evolution of the boom configuration without iterating.

If alternatively the attributes of symbols on the left hand side of all productions depend only on the attributes of the symbols on the right hand side, we would say that the attributes are *synthesized.* The *Weight* attribute in the boom grammar is a synthesized attribute. Grammars which are wholly inherited or wholly synthesized are extreme cases which correspond respectively to grammars in which all attributes can be computed during generation without backtracking or computed during parsing without backtracking. In many cases neither of these extremes apply, however, it is still often possible to compute the values for all attributes without iteration. In these cases the dependency among attributes is not cyclic and the attribute grammar is said to *be well defined.* It is obvious that the boom grammar is well defined because all dimensions and forces can be computed as we "proceed" down the derivation tree shown in Figure 5 and that all weights can later be computed as we "climb" the tree.

In cases where a circular dependency among attributes does exist, simultaneous consideration of attributional relationships will generally be required. Although simultaneous solution of this sort might not be difficult (depending on the complexity of the attributional relations) it will preclude an evaluation of all attribute values in parallel with a parse or generation of a design alternative.

Grammar Definition

$G = \{(B), (t), P, B\})$

Grammar Symbols **and** Attributes

B:	Boom Section		
	Attributes		
		J	**length**
		.h	**height**
		.al	**attachment length**
		J	**end load**
		.W	**weight**
t:	Triangle Sectioni **(and lefit attachment)**		
	Attributes		
		.1	**length**
		.h	**height**
		.al	**attachment length**
		$.F_a$	**force in attachment links**
		$.F_b$	**force in base strut**
		$.F_u$	**force in** angled (upright) members
		.W	**weight**

Grammar Pro Auctions and Attribution Relations

B	-+	t	tl	=	B.1
			th	m	**B.h**
			th	=	B.h
			t.al	=	**B.al +1.1/2**
			^{L}Fa	=	**B.F*B.1/B.h**
			$t.F_b$	=	**B.F*B.l/(B.h*2)**
			$t.F_u$	=	$B.F*\sqrt{1+(B.l/(2*B.h))^2}$
			tW	=	$f(t.l, th, tal, t.F_a, t.F_b, t.F_u)$
B	->	tB	tl	=	mintC! * B.h, B.I)
			th	=	B.h
			tal	=	**B.al + Ll/2**
			$t.F_a$	=	**B.F*B.1/B.h**
			$t.F_b$	=	**B.F*(B.l-tl/2)/th**
			$t.F_u$	=	$B.F\sqrt{1+(tl/(2*th))^2}$
			tW	=	$f(tl, th, tal, t.F_a, t.F_b, tFy)$
			$B_r l$	=	**B.1 - tl**
			$B_r h$	=	B.h
			$B_r al$	=	**tl/2**
			$B_r F$	=	**B.F**
			B.W	=	$B_r W + tW$

Table 1: Attribute Grammar Table

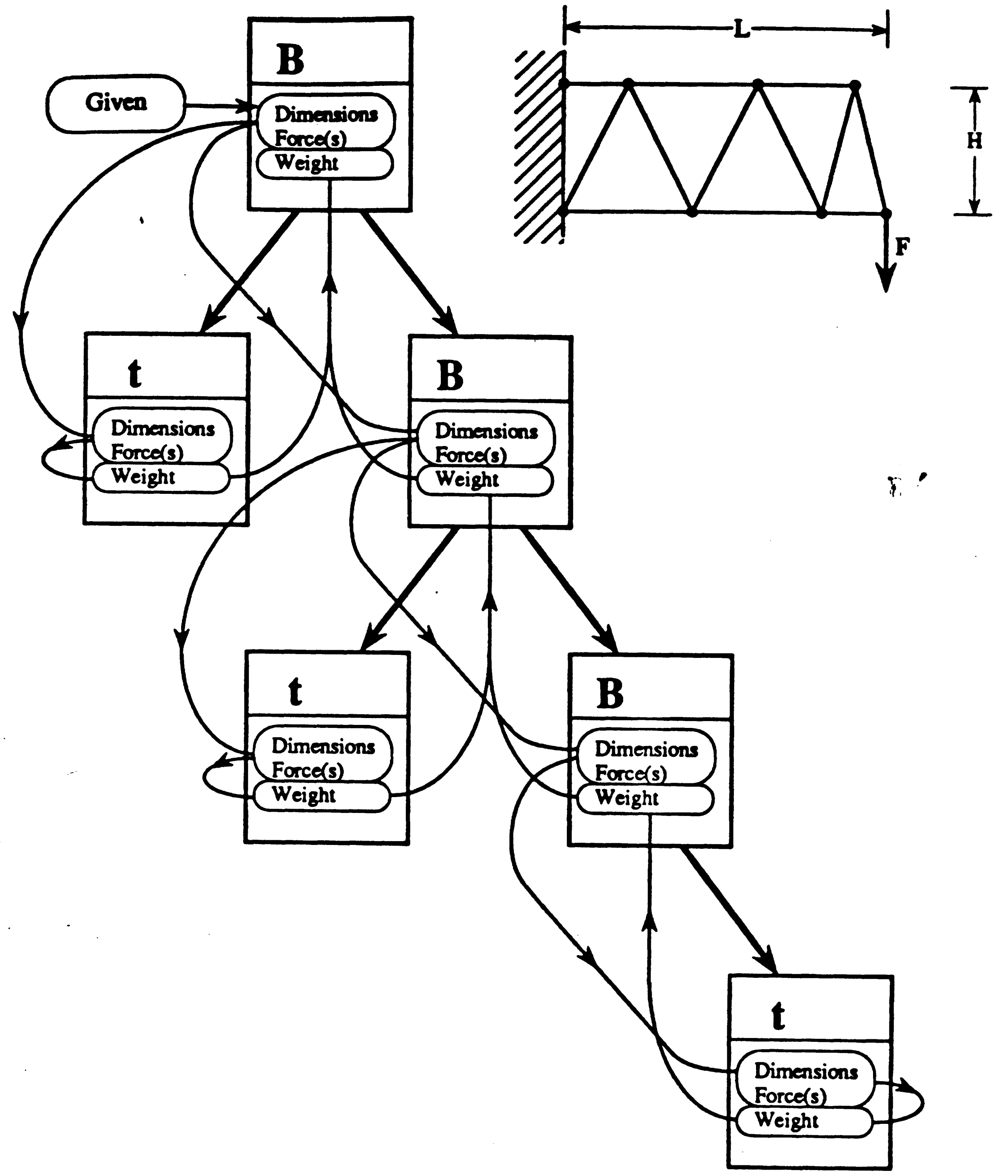


Figure 5: Attribute Grammar

Attribute grammars are as powerful a method of semantic analysis as any that can be imagined in that the value of one attribute can depend, at least indirectly, on die value of any other attribute anywhere in the tree [Knuth 68]. The formal restriction of attributes to the specification of purely semantic, rather than lexical, properties is unimportant The attribute values associated with a symbol can specify information that is purely local **to that symbol. An example of this can be seen in Figure 6 where attributes** associated with **the I elements correspond to the inertia of the mechanical components** they each **represent**

The attribute grammar formalism applies generally to graph grammars. The extension of attribute grammars to graphs has been formalized in the Node Label Control grammars, but the extension to more general graph grammars is possible [Kaplan 87]. We demonstrate their utility with an example based on the bond graph.

Figure 6 shows the production again. For this example the graph is taken to represent a gear pair used in a mechanical transmission: The transformer (TF) indicates that the speed is reduced and I elements represent the inertia of the gears. The symbols in the production have the following attributes associated with them: V

TF.r	reduction ratio
Li	mass moment of inertia

The attribution rule for the production is:

$$\boldsymbol{I_3.i} \quad \boldsymbol{= I_1.i + I_2.i/(TF_1.r)^{1}}$$

$$TF_2.r \quad = TF_xjr$$

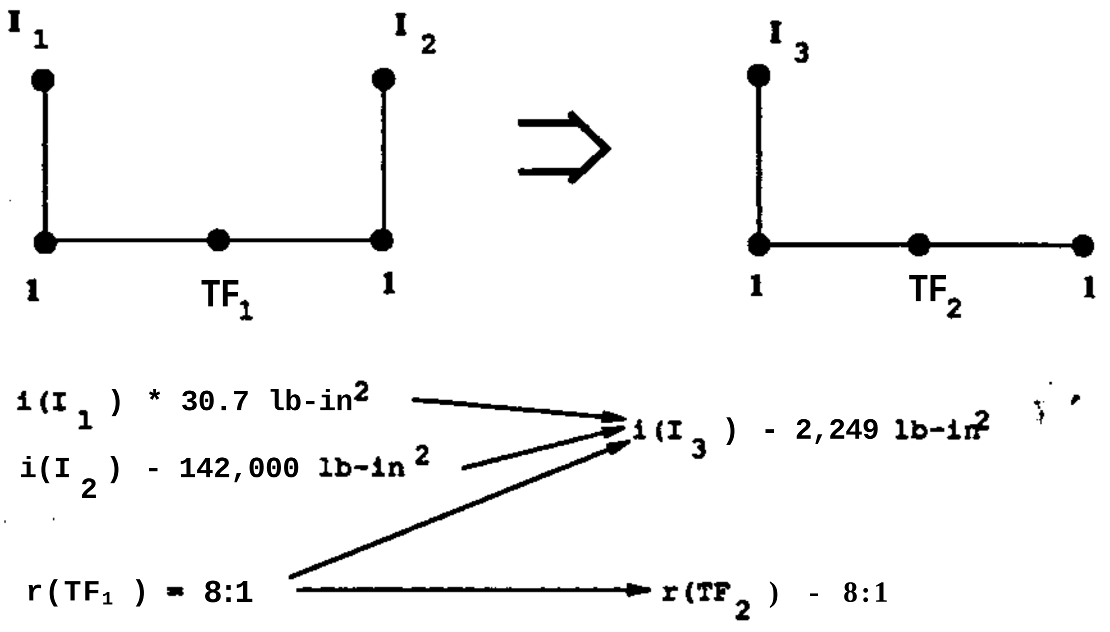
The result of applying the production is shown in Figure 6.

The production in Figure 6 exemplifies some of the parallels mentioned earlier. Mechanical components have properties, such as inertia that effect the behavior of the design as a whole. How the component is connected to the rest of the design determines what this effect will be. For the bond graph example, the component properties are attributes (lexical information) and the configuration is established by the syntax of the graph grammar. The attribution rules (the semantics) of the language provide a model of the overall behavior.

In the context of engineering design it is quite natural to think of these attributes as parameters of components, however, the attributes themselves need not have physical significance. In many cases attributes are used to maintain some internal state of a generation or parse. In this case the attributes may be used as a mechanism to interpret the significance of various strings in a language. In this case attributes may be used to enhance the semantics of a language.

5. Semantics and Grammars

The semantics of a language are derived from the syntactical relationships between the elements of the language (as defined by the grammar) and from the *lexical* information associated with each element in the language. A lexicon in natural language is a dictionary that defines the individual meanings of words. For the bond graph language



I_1
I_2
I_3
1
TF_1
1
1
TF_2
1
$i(I_1)$ * 30.7 lb-in^2
$i(I_2)$ - 142,000 lb-in^2
$i(I_3)$ - 2,249 lb-in^2
$r(TF_1)$ = 8:1
$r(TF_2)$ - 8:1

Figure 6: Attributed Graph Grammar Example Production

mentioned earlier the node TF symbolizes a transformation of velocity and force associated with a gear pair for example. The lexical information of the symbol is then combined with the way it is connected to the rest of the graph to determine what it does in the graph. For our purposes certain parallels can be drawn between the way meaning is derived from a language and the way behavior is exhibited by mechanical designs. The lexical information corresponds to the behaviors and properties of the individual components that make up the design. The syntax of a mechanical design domain describes the allowable configurations of components. The semantics depend on both types of information and results in a behavior of the design. This, however, is only one paradigm for the use of grammars in design. Another, which will be used for the example in the final section of the paper, is to treat the desired behavior of the artifact as the string to be parsed. The result of the parse corresponds to a physical description of an artifact that exhibits that behavior. In this way the functional specification is effectively transformed into a description of a physical artifact The syntactic and lexical parallels are the same but the meaning of semantics for this case is somewhat less clear. Suffice it to say that the semantics still describe the behavior of the artifact but the semantics are defined almost completely in terms of the syntax and lexicon.

There are two types of semantic information, static semantics and dynamic semantics. The static semantics of a language are defined such that there is always one fixed way of interpreting a given clement or group of elements of the language. The semantics for the bridge grammar presented earlier provides a good example of static semantics. The interpretation of the *Vs* and the s's does not depend on the form of the parse tree or the order of the productions. A triangle is always connected by its apex to the adjacent elements and the span is always connected from its midpoint The dynamic semantics of the language, however, depend on the productions which were used to produce the string or graph. Attribute grammars are useful for interpreting the dynamic semantics of a language. The attributes of the symbols in the language are defined to capture the semantic state. *Semantic rules,* which are directly analogous to attribution relations, are associated with each of the productions in the grammar. The semantic rules prescribe how semantic variables are related and therefore can be used to capture the evolving meaning of a language element

The attribute grammar approach attaches meaning to the productions in the grammar. Alternatively we may find a *procedural semantics* in which the semantics arc determined by functions or procedures attached to each *symbol* in the language.

I '

6. Example: A Grammatical Approach to Logic Circuit Design

Logic circuit design displays many of the same problems associated with mechanical design. While it is certainly possible to design and implement a circuit in which each function of the circuit is performed by an individual component or group of components, it is desirable to integrate as many of the functions as possible. Many of the difficulties associated with geometry are absent from logic circuit design, thus providing us with a more convenient domain in which to test some of our ideas. Furthermore, logic circuit design is well understood, providing a bench mark by which to measure the success of alternative approaches to design.

We wish to design a combinatorial logic circuit to be implemented with inverters and two input AND, and OR gates. The specification for the logic circuit is given as a truth table in Figure 7 where all combinations of inputs not explicitly represented in the table must produce a zero or false output The method we demonstrate uses two different grammars. The first is a string grammar which is used to parse a string representation of the truth table. The parse provides not only a test of validity (i.e., is the function attainable) but also identifies the productions in the grammar necessary to produce that string. The parse tree is a graph which represents a functionally correct logic circuit and may, depending on the string grammar, represent a canonical form of the desired functionality. Although functionally correct, this tree is an ineffective implementation approach. We therefore, use the tree as the starting symbol in a graph grammar which is used to generate functionally equivalent but more economical implementations of that function.

The string to be parsed, taken from the table shown in Figure 7, is represented as:

0110x0100x1110

The x's signify that a separate line in the truth table has been started. For this grammar

AT = {AND, OR, N)

r={o,i,xj

A	B	C	D	Output
0	1	0	0	1
0	1	1	0	1
1	1	1	0	1

Figure 7: Truth Table for the Example

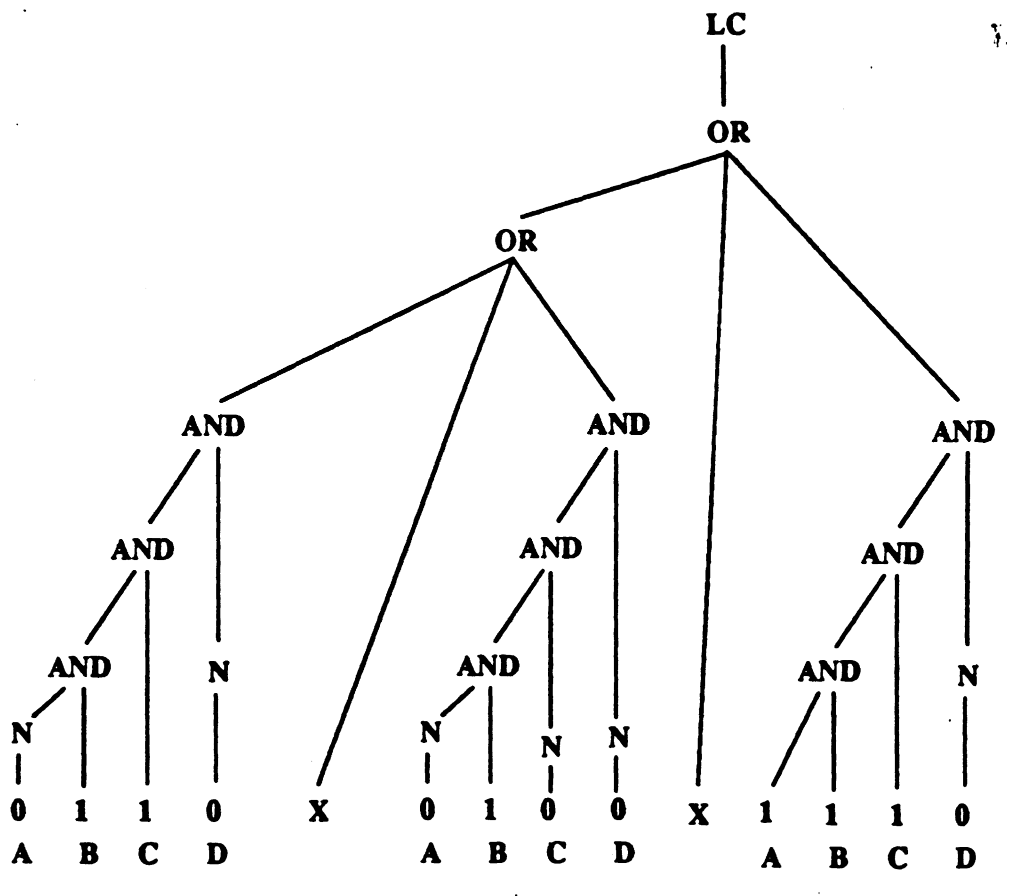


Figure 8: Parse Tree for the Example

S = {LC}

AND stands for an **AND** gate, OR stands for an OR gate, and N stands for a NOT gate. For a more detailed explanation of logic circuits and their design see [Hill 81]. The productions for the grammar are:

1:	LC	->	OR
2:	OR	->	OR,x,ANDIAND,x,AND
3:	AND	-*	AND.N I AND, 111,AND I **N,N** I N_fl 11 ,N 11,1
4:	N	->	0

The language of the grammar is any string of Ts, O's, and x^{f}s. Any size truth table can be accommodated by the grammar.[6]

As the parse proceeds the input label (e.g. A, B, etc.) associated with each terminal symbol is maintained as an attribute for use in the subsequent design minimization grammar. The parse tree for the truth table in Figure 7 is shown in Figure 8. The parse tree corresponds to a logic circuit that has the required functionality however, that circuit is not minimal. y '

The implementation minimization grammar is then used to take advantage of the function sharing possible in these circuits. The parse tree generated by the string grammar is then prepared to start the implementation minimization grammar by deleting, nodes labeled *x* (and the dangling arcs) and by replacing the 1's and O's with the input variable they represent. The input variable names are shown underneath the terminal symbols in Figure 8. The grammar is very similar to a graph grammar in the way in which graphs are rewritten, however, it is not formally a graph grammar because each production is actually an infinite family of productions specified using place holders (e.g. Q).

The productions for the graph grammar arc based on the method of mintemis [Hill 81] and are shown in Figure 9. The symbols Q, R, and S are used as place holders in the productions for sub-graphs. These symbols indicate that those portions of the graph must match with any other sub-graph in the production that carries the same label. The embedding rules are simple tree-replacement rules as in NLC grammars. The productions used in this grammar are all function preserving because replacing the target graph with the daughter graph doesn't change the input/output relationship for the logic circuit

The derivation for the graph grammar is shown in Figures 10 through 12. Figure 10 shows the intermediate graph resulting from firing Production 2. Figure 11 shows the graph after firing Productions 2,7, and 2 in that sequence. The final graph, shown in Figure 12 is obtained by firing Productions 5,4,2,7,2,1,4 and 5. There are then no more matches for the productions and so the generation of alternatives is complete and the grammar terminates. This graph represents the minimal logic circuit for the truth table

***For simplicity of presentation, the grammar does not parse pathologically small truth tables nor will it parse strings that begin or end with an x or that do not contain any x's.**

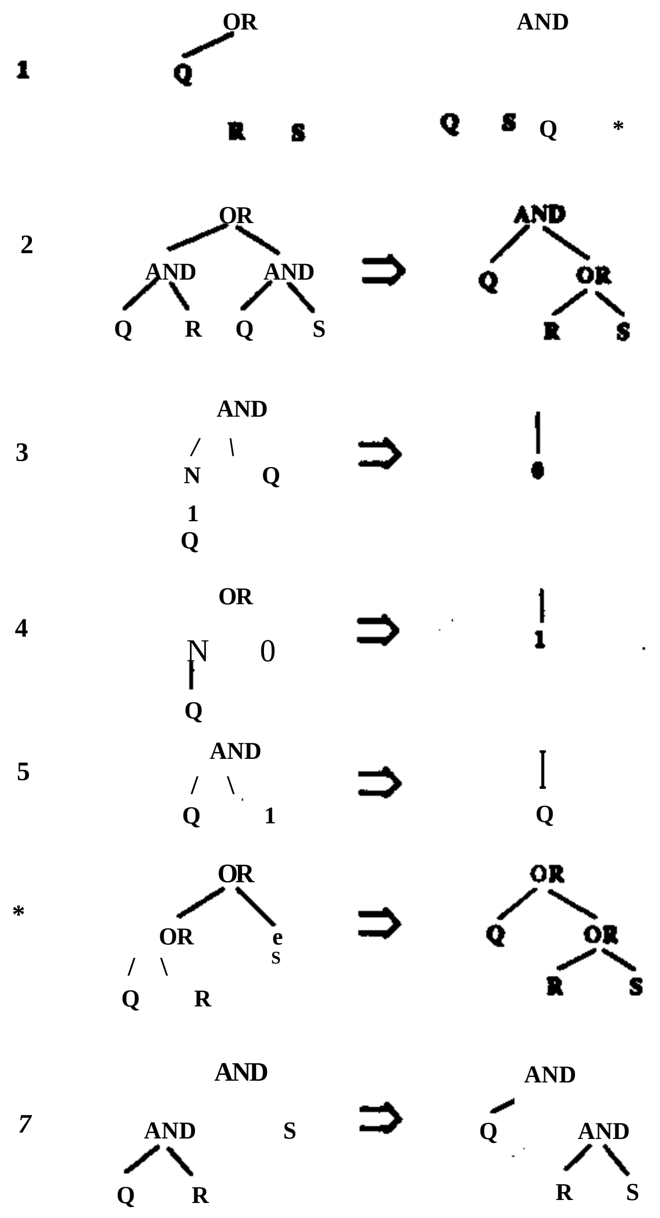


Figure 9: Graph Productions for the Example

given although the result is not unique.

This is a simple example of the transformational approach to design. The string representing the desired logic circuit behavior is transformed, through the use of the two grammars, into a physical description of a circuit with the required function. This grammatical approach results in a physically realizable, minimal AND-OR-NOT, solution to the (somewhat) practical problem of combinatorial logic synthesis. This is in keeping with the goal of designing systems that take foil advantage of the functionality and interactions of its components.

This grammar' however, has two significant drawbacks. The first is that there is no control structure to the firing of the graph production rules. This means that the grammar will take a long time to generate solutions to even simple design problems. This drawback is common to almost all graph grammars because of the difficulty of the embedded graph matching problem. The second difficulty with the grammar is that it is limited to using AND, OR, and NOT gates. The grammar can be augmented to include other types of gates but this will increase the combinatorial difficulties associated with the grammar. This is another example of the power/generality trade-off that is inherent in most formalisms: The domain of the grammar can be expanded but only at the expense of the ease of computation.

7. Discussion and Conclusions

The paradigm of design as a transformational process is a powerful one. Grammars are useful in so far as they provide a framework for this transformational process, "however, there remain many critical issues in applying grammatical formalisms to design. Some of these are:

- Representation of function and behavior of mechanical designs.
- Representation of the form and structure of mechanical devices.
- Relationships between function and form.
- The generation of *good* designs.

Most common representations of behavior and geometry of mechanical devices are based either on a rigid parameterization or are completely ad hoc. Other representations have more recently been suggested with varying levels of utility and generality [Beitz 84] [Lai 87] [Fenves 87]. Representations based on formal grammars are attractive because it is possible to represent both a wide range of configurational alternatives as well as the traditional engineering parameters. Because the grammatical approaches are applicable in general to string and graph representations, broader representations of behavior [Rinderlc 87] and geometry [Pinilla 89] [Fitzhorn 86] are anticipated. Grammars will provide a formalism with which to perform operations and transformations on these representations.

Structuring relationships between behavior and geometry and the focused generation of good design alternatives are perhaps more difficult problems. Although grammars by their nature are suited to the generation of alternatives, there are only rare instances of their use in design [Fleming 87]. While it is easy to generate syntactically correct structures, it is difficult to generate useful or optimal structures. Attribute grammars are useful in that they may be used to manipulate engineering equations, however, issues of semantics and ^{H}style^{H} must be addressed. Style in the english language refers to the formation of sentences and paragraphs that are not only syntactically and semanticaily

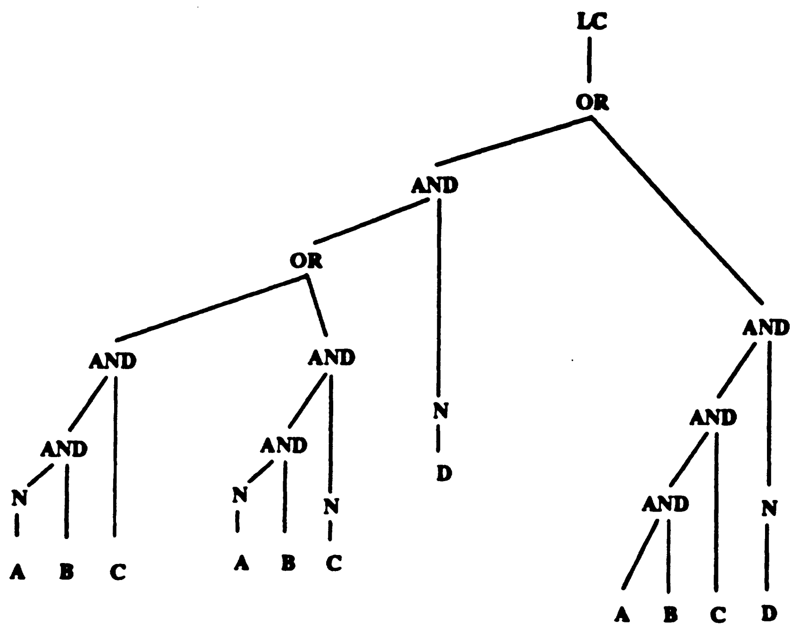


Figure 10: An Intermediate Graph in the Parse

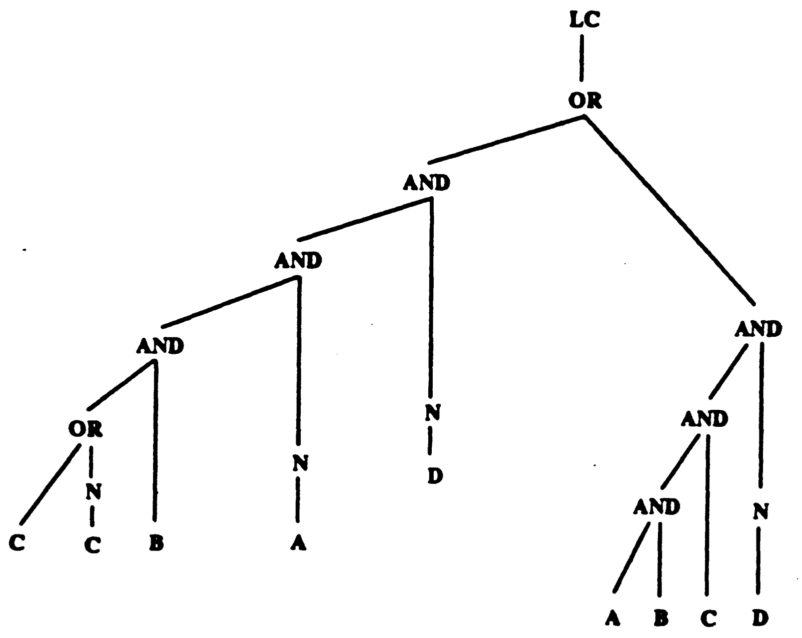


Figure 11: Subsequent Intermediate Graph in the Parse

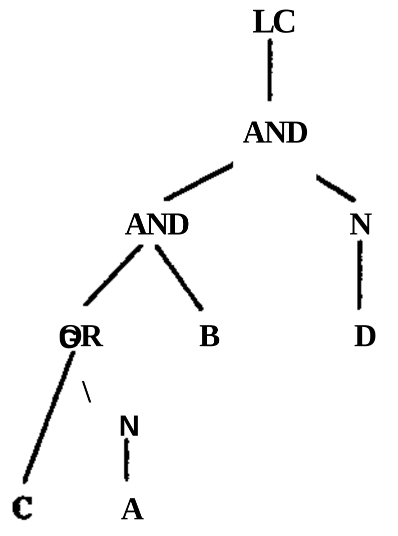


Figure 12: Final Graph

correct (i.e. they "sound" all right and convey the idea that was intended) but also are clear and concise. In mechanical engineering a good "style" often consists of components that display a great deal of functional integration. Generation focusing principles such as this must be integrated into the grammar early in the design process. To facilitate this integration, grammatical control strategies can be employed. The use of programmed grammars is one one approach to controlling grammatical generation [Bunke 79].

A programmed grammar and certainly a designer must reason about relationships between behavior and geometry at a level more abstract than detailed geometry or function. Grammars support reasoning with abstractions since the transformational paradigm inherent in grammars makes it natural to transform and elaborate abstract entities providing a hierarchy of detail and refinement It is peiiiaps this characteristic of grammars which will yield a near term benefit by bringing together detailed geometric and behavioral representations with the abstract reasoning exhibited by practicing designers.

The relationships between computational complexity of grammars and design might be discouraging, particularly given the frequency with which properties of languages turn out to be undecidablc. It seems natural, however, that a model of the design process would have such properties. Design by its very nature is not deterministic and is not strictly analytical. If design itself is undecidable, or at best np-complete, why wouldn't the general model for it be also? Although designers address this issue daily, their design problem solving strategies must be identified and made explicit so as to simplify grammatical approaches to design and to facilitate the development of practical design automation strategics.

Acknowledgments

The authors are pleased to acknowledge the support of the Design Theory and Methodology Program of the National Science Foundation (NSF Grants DMC-84-51619 and DMC-88-14760) and the Engineering Design Research Center at Carnegie Mellon University (NSF Grant CDR-85-22616).

References

[Beitz84]
P. Beitz A G. Paul, *Engineering Design,* Springer- Verlag. 1984.

[Bunke79]
H. Bunke, "Progrtnuncd Graph Grammars,'9 in *Graph Grammars and Their Application to Computer Science,* Springer-Verlag, 1979, Lecture Note Series in Computer Science

[Chomsky 66]
N. Chomsky, *Syntactic Structures,* Mouton & Co. 1966.

[Deransart 88]
P. Deransart et al. *Attribute Grammars,* Springer-Verlag, 1988.

[Ehrig78]
H. Ehrig, Introduction to the Algebraic Theory of Graph Grammars,M in *Graph Grammars and Their Application to Computer Science and Biology,* Springer-Verlag, 1978, Lecture Note Series in Computer Science

[Ehrig 86]
H. Ehrig, Tutorial Introduction to the Algebraic Approach of Graph Grammars," in *Graph Grammars and Their Application to Computer Science,* Springer-Verlag, 1986, Lecture Note Series in Computer Science

[Fenves 87]
S. Fenves & N. Baker, "Spatial and Functional Representation Language for Structural Design," in *Expert Systems in Computer-Aided Design,* Elsevier Science, 1987.

[Finger 89]
S. Finger & J. Rinderle, "A Transformational Approach to Mechanical Design Using a Bond Graph Grammar," *Proceedings, Design Theory and Methodology Conference,* ASME, Montreal, September 1989.

[Fitzhorn 86]
P. Fitzhorn, "A Linguistic Formalism for Engineering Sob'd Modeling," in *Graph Grammars and Their Application to Computer Science,* Springer-Verlag, 1986, Lecture Note Series in Computer Science

[Fitzhorn 88]
P. Fitzhorn, "A Computational Theory of Design," *Design Computing,* January 1988.

[Fleming 87]
U. Fleming, "More than the Sum of Parts: the Grammar of Queen Anne Houses," in *Environment and Planning B: Planning and Design,,* 1987, vol. 14

[Habel86]
A. Habel & H. Kreowski, "May We Introduce to You: Hyperedge Replacement," in *Graph Grammars and Their Application to Computer Science,* Springer-Verlag, 1986, Lecture Note Series in Computer Science

[Harrison 74]
M. Harrison, "Some Linguistic Issues in Design," in *Basic Questions of Design Theory,* North-Holland Publishing. 1974.

[Hill 81]
F. Hill SL G. Peterson, *Introduction to Switching Theory and Logical Design,* Wiley & Sons, 1981.

[Janssens 82]
D. Janssens k G. Rozenberg, "Graph Grammars With Node-Label Controlled Rewriting and Embedding," in *Graph Grammars and Their Application to Computer Science,* Springer-Verlag, 1982, Lecture Note Series in Computer Science

[Kaplan 87]
S. Kaplan, "Incrmental Attribute Evaluation on Node-Label Controlled Graphs," Technical report R-87-1309, University of Illinois at Urbana-Champaign, May 1987.

[Knuth68]
D. Knuth, "Semantics of Context-Free Languages," in *Mathematical Systems Theory,* Springer-Verlag. 1968.

[Lai 87]
K. Lai A W. Wilson, MFDL • A Language for Function and Rationalization in Mechanical Design," in *Computers in Engineering,* ASME, 1987.

[Lewis 81]
R. Lewis A C. Papadimitriou, *Elements ofthe Theory ofComputation,* Prentice-Hall 1981.

[MoU 88]
R. Moll, M. Arbib, A Kfoury, *An Introduction to*

Formal Language Theory, Springer-Veriag, 1988.

[Moitow851
Mottow, *U* Towaid Better Models Of The Design Process," *The AI Magazine,* Spring 1985.

[Nagl78J
M. NagL "A Tutorial and Bibliographic Surrey On Graph Grammars/* in *Graph Grammars and Their Application to Computer Science and Biology,* Springer-Verlag. 1978, Lecture Note Series in Computer Science

[Nagl86]
M. NagL "Set Theoretic Approaches to Graph Grammars," in *Graph Grammars and Their Application to Computer Science,* Springer-Verlag, 1986, Lecture Note Series in Computer Science

[Pinilla89]
J. Pinilla, S. Finger, & F. Prinz, "Shape Feature and Recognition Using an Augmented Topology Graph Grammar," *Proceedings of the 1989 NSF Engineering Design Research Confer*[illegible].AmhemMA^June 1989.

[Revesz83]
G. Revesz, *Introduction to Formal Language Theory,* Mcgraw-HilL 1983.

[Rinderle87]
Rinderk, *J.* R^ "Funcdoo and Form Relttionshipi: A Basis for Preliminary Design," *Proceedings from the NSF Workshop on the Design Process.* Waldron, M. B., ed, Ohio State University, Oakland, CA, Febniary 8-10 1987, pp. 295-312.

[Rozenberg86]
G. Rozenberg, "An Introduction to die NLC Way of Rewriting Graphs," in *Graph Grammars and Their Application to Computer Science,* Springer-Verlag, 1986, Lecture Note Series in Computer Science

[Stiny 80]
Stiiiy, G., "Introduction to Shape and Shape Grammars,* *Environment and Planning B,* Vol. 7. July 1980, pp. 343-351.