Grammars for Engineering Design

Thesis by
Andrew B. Wells

In Partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy

California Institute of Technology
Pasadena, California

1994

(Defended September 27, 1993)

Acknowledgments

My wife, Loey Werking Wells, has been the major source of joy in my life ever since we became reacquainted with each other at about the same time as I started seriously working on this dissertation. Her presence has given me the strength to press on with my work and her encouragement has helped me to complete this project in a relatively timely manner. While I was a graduate student, we saw each other for the first time in six years, fell in love, moved in together, became engaged, got married, and moved from Pasadena. She has opened me up to new experiences and stimulated my thinking immeasurably. I know that I am the luckiest person in the world to have her love and support.

I also wish to thank my parents, Michael and Phyllis Wells, for all their praise and for encouraging me to try new and different things. They have put up with many surprises from me without criticizing and have been the best parents possible.

At Caltech, my advisor, Dr. Erik Antonsson, has provided invaluable advice, encouragement, and financial support. Although my research was sidetracked many times, he put up with my diversions and gently pushed me back on course. During my time at Caltech, he was available for consultation whenever I needed him.

The joints and links used to construct the robot arms of Chapter 6 were originally proposed by I-Ming Chen in [16]. He also wrote the original *Mathematica* code used to draw arms formed using these modules. I am deeply indebted to him for discussing his work with me and for allowing me to use, modify, and expand upon his code.

The various members of the Society of Professional Students, SOPS, deserve special recognition. Having these other graduate students around me, sharing in our common experiences, giving each other advice, and getting to know one another kept me from going completely crazy over the past four years. Thanks to Andrew, Kevin, Ted, Wen-Jean, Al, Jim, Will, Dan, Michael, Bob, Howie, Brett, Beth, I-Ming, and all the others!

Grammars for Engineering Design

by

Andrew B. Wells

In Partial Fulfillment of the
Requirements for the Degree of
Doctor of Philosophy

Abstract

Just as grammars for natural languages use rules to form grammatical sentences from a dictionary of words, grammars for engineering design use rules to make structures from a dictionary of shapes, properties, labels, and other elements. Engineering grammars may be used to help the designer generate and evaluate ideas and concepts during the conceptual phase of the design process.

A formal definition of a grammar is given and some properties of grammars are discussed. Natural language grammars, shape grammars, and engineering grammars are defined. Some group-theoretic properties of shapes and operations are derived. It is shown that some sets of shapes form Boolean algebras under the standard regularized set operations. Polygonal tracings, which are extensions of the outlines of two-dimensional polygons, form a ring under the shape union and convolution (or generalized Minkowski sum) operations. A subset of polygonal tracings which includes all convex tracings, along with the convolution and shape scaling operations, form a vector space over the real numbers. The implications for grammar rules which use these types of shapes and operations are discussed.

Grammars and expert systems are compared and contrasted. While the formalisms have similar definitions, some explicit differences exist. Furthermore, when the customary uses of the two systems are compared, large differences are evident. It is concluded that grammars are more well-suited to generating many alternative designs and searching large, unexplored design spaces, while expert systems function best in well-known domains when only one design is required.

The formation and modification of grammatical rules is discussed, focusing on the relationships between form and function in design. Several strategies which could be used for the search for optimal designs in a grammar's language are considered. The significance of transformations used to apply rules is discussed.

An extended example of grammars used to generate configurations of modular reconfigurable robot arms is presented. The grammars generate all non-isomorphic assembly configurations, while simultaneously calculating kinematic properties of the arms. Several methods of quickly searching for arms to satisfy various requirements are discussed.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Conceptual Design

In the early stages of engineering design, a designer first identifies a set of functional requirements and performance parameters which designs must satisfy to be acceptable. Then he or she enters the conceptual design phase, in which a large number of conceivable designs and solution methods are considered and quickly (and often heuristically) evaluated. From this phase emerges a small collection of design alternatives, which the designer can go on to develop and evaluate more thoroughly.

Many methods are used to generate concepts. A group of designers might brainstorm. Existing designs might be analyzed and modified or naturally-occurring phenomena might be copied. An expert might be consulted. Possible methods of achieving a partially functioning design might be combined. Expert systems might be used to generate a design automatically. In practice, all these methods have biases toward particular types of designs [95]. Only some methods can generate a large number of alternatives and few allow easy evaluation of the possibilities that are generated.

1.2 Form, Function, and the Need for New Systems

Every engineering design's physical structure and functionality are interrelated. In the conceptual design phase, most existing formal systems do not do a good job of representing form and function. If a design's form changes, then its functionality will probably also change. However, the two properties are not directly linked: a design's functionality might

be affected differently if a structural change were applied in different places, or it might not be affected at all. There are few systems which can generate a broad range of alternative designs, evaluate them, and represent the form/function relationships in them. Grammars form one class of systems that can perform all these functions and thus help to fill the need for formalisms for conceptual design.

1.3 Grammars

Just as grammars for natural languages use rules to form grammatical sentences from a dictionary of words, grammars for engineering design use rules to make structures from a dictionary of shapes, properties, labels, and other elements. Engineering grammars may be used to help the designer generate and evaluate ideas and concepts during the conceptual phase of the design process.

As will be discussed in Chapter 2, grammars use rules to arrange, combine, or modify symbols in specific ways. For example, a natural language grammar uses grammatic rules to combine words in order to form sentences. Similarly, a shape grammar uses rules to arrange and modify shapes in order to produce new shapes and forms. A simple example of a shape grammar is shown in Figure 1-1. In this shape grammar, pentagons may be added to a shape at any location denoted by a small circular marker. There is also a rule to erase the markers, leaving only lines. The evolution of a shape as the rules of the grammar are applied is pictured in Figure 1-2. Note that the resultant shape is just one of many shapes which could be derived using the grammar's rules. For engineering use, grammars can be defined on the domains of shapes, engineering properties, labels, colors, or almost any other domain or combination of domains. Several uses of grammars in engineering will be described later in this dissertation.

1.4 Overview of the Dissertation

In the chapters to come, it will be shown that grammars are useful tools for engineering design. In Chapter 2, a formal definition of a grammar is given and some properties of grammars are discussed. Natural language grammars, shape grammars, and engineering grammars are defined.

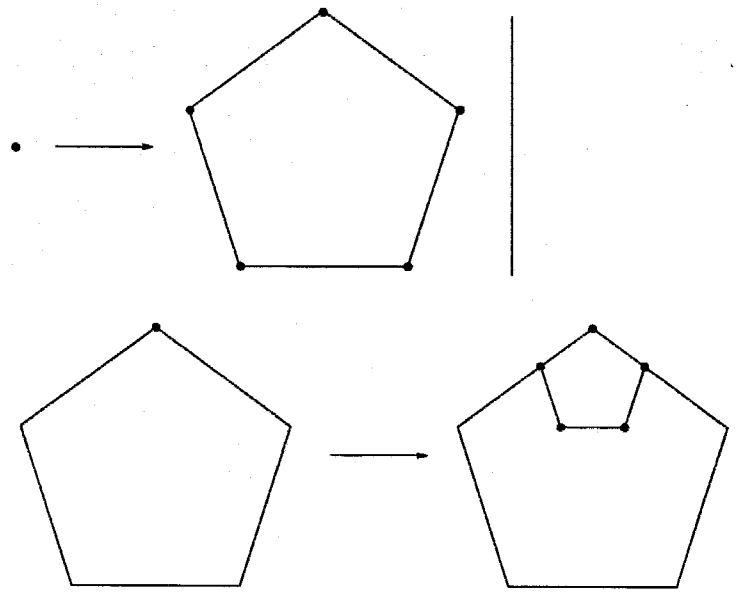


Figure 1-1: Rules for shape grammar using pentagons.

In Chapter 3, some group-theoretic properties of shapes and operations are derived. It is shown that some sets of shapes form Boolean algebras under the standard regularized set operations. Polygonal tracings, which are extensions of the outlines of two-dimensional polygons, form a ring under the shape union and convolution (or generalized Minkowski sum) operations. A subset of polygonal tracings which includes all convex polygonal tracings, along with the convolution and shape scaling operations, form a vector space over the real numbers. The implications for grammar rules which use these types of shapes and operations are discussed.

Grammars and expert systems are compared and contrasted in Chapter 4. While both formalisms share the same basic definition, some explicit differences exist. Furthermore, when the customary uses of the two systems are compared, large differences are evident. It is concluded that grammars are more well-suited to generating many alternative designs and searching large unexplored design spaces, while expert systems function best in well-known domains when only one design is required.

The formation and modification of grammatical rules is discussed in Chapter 5, concentrating on the relationships between form and function in design. Several strategies which could be used for the grammar-directed search for an optimal design are considered. The

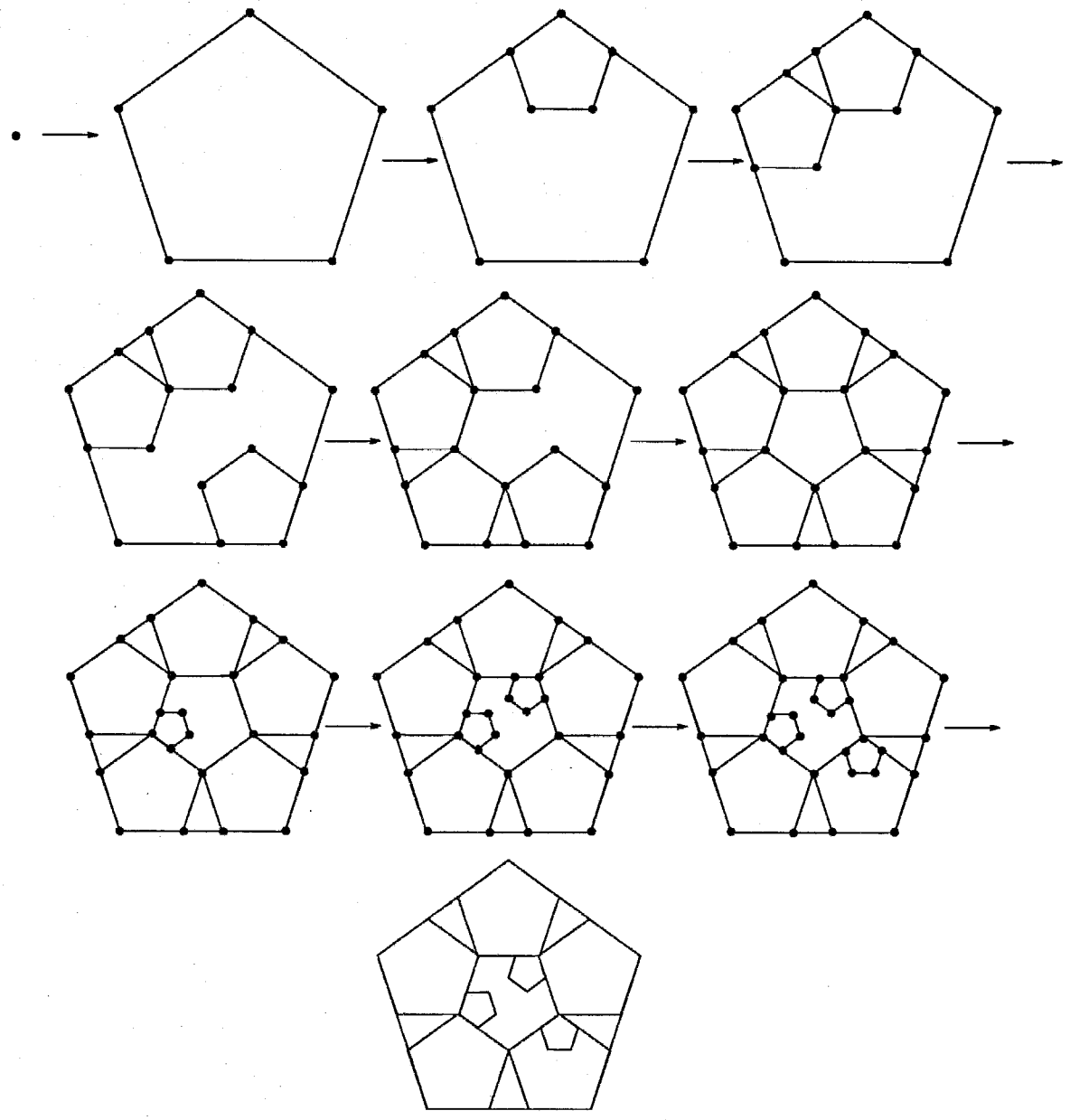


Figure 1-2: The evolution of a shape as rules are applied to it.

significance of transformations used to apply rules is discussed.

An extended example of grammars used to generate configurations of modular reconfigurable robot arms is presented in Chapter 6. The grammars generate all non-isomorphic assembly configurations, while simultaneously calculating kinematic properties of the arms. Several methods of quickly searching for arms to satisfy various requirements are discussed.

In Chapter 7, related work in several fields is discussed. Chapter 8 presents some concluding remarks, while many of the terms used in the dissertation are defined in Appendix A. Examples and definitions of the group-theoretic terminology needed to understand Chapter 3 are presented, along with definitions of some mapping properties and language-theoretic structures.

1.5 Contributions Made in the Dissertation

Several significant contributions to the field of design theory will be discussed in this dissertation. The group-theoretic properties of certain classes of shapes and shape operations have important implications for the use of those shapes and operations. While some of the more basic properties have been proven, it has never been shown that some shapes are members of rings or vector spaces.

Grammars and expert systems have many similarities and some very important differences. Until now, these two formalisms have never been compared or differentiated explicitly. Some new conclusions are also made about the suitability of each formalism for solving particular design problems.

Also contained in this dissertation is one of the most complex examples yet of the use of grammars in the mechanical engineering domain. The modular robot arm grammars of Chapter 6 are some of the first to operate on both the geometric and the kinematic properties of a system.

Finally, the appropriate uses of grammars are discussed in several contexts. While grammars have been used by others to address various engineering and architectural problems, little attention has been paid to the ways in which grammars are used to search design spaces or to the proper choice of rules for grammars. In this dissertation, those and other issues are considered in greater depth than they have been before.

Chapter 2

Introduction to Grammars

2.1 Grammars

In an informal sense, a grammar is a collection of rules which are used to govern the combination of words into sentences. However, grammars are not restricted solely to the domains of words and spoken and written languages. They may be defined in almost any domain, as will be demonstrated below. The formal definition of a grammar is introduced first. Then properties which can be deduced from the structure of grammar rules are discussed. Next, building on those definitions, grammars for natural languages, shapes, and engineering applications are introduced. Finally, several types of grammars which may be used to solve engineering design problems are discussed.

2.2 Formal Properties of Grammars

The formal definition of a grammar applies in all domains, be they textual, mathematical, visual, or any other area. The type of production rules determines the expressibility of a grammar and the properties which can be determined for it.

2.2.1 Definitions

In leading up to the formal definition of a grammar, some terminology must be defined [84].

Definition 2.1 (Alphabet) An *alphabet* X is a nonempty finite set of symbols.

The symbols which form an alphabet may be from any domain. They may be letters, numbers, words, shapes, variable names, colors, labels, or any other representation that one wishes to choose. In the development that follows, let X^* denote the set of all finite length strings formed by members of X. This set includes the *empty string*, typically denoted ϵ. The set of all finite, nonempty strings formed by members of X, denoted X^+, is defined to be $X^* - \epsilon$. Note that in this dissertation, the words "string," "sentence," and "design" will be used interchangeably when a string of symbols is being used to represent any type of engineering or visual design. Similarly, the words "symbol," "component," and "shape" will be used to denote members of the alphabet being used.

Definition 2.2 (Grammar) A *phrase structure grammar* (or grammar, for short) G is a quadruple (T, N, S, P), where T and N are finite alphabets, and

1. T is the *terminal alphabet* for the grammar;
2. N is the *nonterminal alphabet* for the grammar, and $T \cap N = \emptyset$;
3. S is the *start symbol* for the grammar, and $S \in (T \cup N)^*$;
4. P is the set of *productions*, or rules, for the grammar. P is a set of pairs (y, z), usually written $y \rightarrow z$, where y is a string in $(T \cup N)^*$ containing at least one nonterminal symbol and z is any string in $(T \cup N)^*$.

The first element of a production is called its *left-hand side* and the second element is called the production's *right-hand side*. If there are two or more rules which share the same left-hand side, they may be represented by a single rule with multiple elements on its right-hand side. The single rule's left-hand side is the same as that of the original rules, while its right-hand side consists of all the right-hand sides of the original rules, separated by vertical lines. For example, if a grammar has productions $y \rightarrow v$ and $y \rightarrow w$, then the two productions can be represented by the rule $y \rightarrow v|w$. Some examples of grammars in several domains will be presented in Sections 2.3 through 2.5.

In some informally defined grammars, terminal and nonterminal symbols might not be explicitly distinguished. In these grammars, all symbols presented are taken to be members of the nonterminal alphabet. To make these grammars comply with the formal definition given above, every symbol in the grammar must include an implicit, invisible,

marker. Furthermore, the rule set must include implicit "finishing up" rules which erase the nonterminal symbols and replace them with terminal symbols which look the same but lack the invisible markers. These implicit rules are generally applied at the end of a derivation. In this type of informally defined grammar, strings formed by applying production rules can be finished and output at any time simply by using the implicit production rules to erase the invisible markers.

Sometimes, a grammar will be a quintuple with fifth element, F, a set of allowable transformations, as discussed by Carlson in [12]. If, in the course of using the grammar, there is a string $x_1x_2\ldots x_i\ldots x_m$ (where every $x_j \in (T \cup N)^*$) and there is a transformation $f \in F$ such that $f(x_i) = y$, where y is the left-hand side of a production (y, z), then using the production, the string may be rewritten as $x_1x_2\ldots f^{-1}(z)\ldots x_m$, where f^{-1} is the inverse of the transformation f. If a set of allowable transformations is not specified, then the user of the grammar may decide which transformations may be desirable according to the context of the application, as Stiny discusses in [113].

Definition 2.3 (Derive) Let G be a grammar and let $y, z \in (T \cup N)^*$. Then y *directly derives* z (or z is directly derived from y), written $y \Rightarrow z$, if z can be obtained from y by replacing an occurrence in y of the left-hand side of some production by its right-hand side. Furthermore, y *derives* z (or z is derivable from y), written $y \overset{*}{\Rightarrow} z$, if $y = z$ or if there is some sequence of strings $w_1, w_2, \ldots, w_n$, with $w_1 = y$ and $w_n = z$ such that for all $i \in \{1, 2, \ldots, n-1\}$, w_i directly derives w_{i+1}.

Definition 2.4 (Language) A *language* generated by grammar G, denoted $L(G)$, is the set of terminal strings which is derivable from the start symbol, S, of the grammar: $L(G) = \{z | z \in T^*, S \overset{*}{\Rightarrow} z\}$.

The richness of a language in a particular domain is often determined by the complexity of the grammar's productions, as described in the next section.

2.2.2 The Chomsky Hierarchy

The form of a grammar's production rules determines the complexity of its language. Four basic types of grammars exist, determined by the forms of their most complex production rules. The simplest type of grammar is the *regular grammar*, which has rules of the form

$n \rightarrow mt$ or $n \rightarrow t$, where $n, m \in N$ and $t \in (T \cup \epsilon)$. Note that each of n, m, and t is a single member of the grammar's alphabet, so a string generated by a regular grammar can never have any more nonterminal symbols than the starting symbol does. The production rules given here, with right-hand sides of rules always having nonterminals to the left of terminals, define a *left-linear grammar*; a right-linear grammar has rules in which nonterminals always appear to the right of terminals in the right-hand sides of all the productions. Both left-linear and right-linear grammars are regular grammars.

Just up the complexity scale from the linear grammars are the *context-free grammars*. These grammars have productions of the form $n \rightarrow v$, where $n \in N$ and $v \in (N \cup T)^*$. These grammars are called context-free because a single nonterminal appears on the left-hand side of each rule: no attention is paid to the context of the symbol.

Next come the *context-sensitive grammars*. Production rules in this type of grammar are of the form $unv \rightarrow uwv$, where $n \in N$, $u, v \in (N \cup T)^*$, and $w \in (N \cup T)^+$. Note that w may not be the null element ϵ. The rules check for nonterminals appearing in some context of finite length and replace those nonterminals with strings of nonzero length.

Finally, the most complex grammar is called a *general grammar*, also known as an unrestricted or phrase structure grammar. The form of rules in a general grammar is unrestricted. While this type of grammar gives the greatest freedom in defining transformations, it will be shown below that many properties of general grammars are unknown.

Grammars are also known by a type number. Regular grammars are of type 3, context-free grammars are type 2, context-sensitive are type 1, and general grammars are type 0. It can be shown that each type of grammar (where the grammar has no rule whose right-hand side is the empty string) is properly contained in the grammar of lower type number. For example, production rules for a regular grammar also satisfy all restrictions on the type of rules allowed in context-free grammars (and, for that matter, in context-sensitive and general grammars as well). If the set of languages generated by type i grammars is denoted $\mathcal{L}_i$, then:

$$\mathcal{L}_3 \subsetneq \mathcal{L}_2 \subsetneq \mathcal{L}_1 \subsetneq \mathcal{L}_0.$$

This is called the *Chomsky hierarchy*, named for linguist Noam Chomsky, who was a pioneer in the field of formal language theory [18]. Note that the above inclusion is predicated upon the assumption that the regular and context-free grammars under consideration have no

rules in which the right-hand side is the empty string ϵ. Languages generated by grammars of type 2 or 3 which *do* have right-hand sides of rules which contain the empty string are only properly contained in the set of languages generated by general grammars.

More complex grammars need not have more complex languages. Instead, the more complex a grammar, the sharper the available delineation between sentences which are part of the grammar's language and those which are not. For example, it is trivial to define a linear grammar which generates all possible finite strings of terminals. Let n be the single member of the set of nonterminals and let the starting symbol $S = n$. Then only the rules $n \rightarrow nt$ and $n \rightarrow \epsilon$, where $t \in T$, need be defined in order to be able to generate all finite strings of terminals. A more complex grammar could have the ability to "draw a fine line" between strings, generating a number of complex strings, but also refraining from generating other complex strings which are unwanted, as discussed by Grune and Jacobs [47]. Nonetheless, absent any other measure of the complexity of a language, the level of its grammar on the Chomsky hierarchy could be a useful measure.

2.2.3 Decidability and Other Properties

One area in which the complexity of a grammar does make a difference is in determining the properties of its language. For example, a user may want to find out if a particular string can be generated by a grammar. Alternatively, he or she may want to know if the language defined by a grammar is properly contained within another language generated by a different grammar of the same type.

In the context to be used here, a problem is *decidable* if there is a general algorithm which always produces an answer to the problem. Given languages of the various types, conclusions can be made about the decidability of various problems based on the level of a language in the Chomsky hierarchy, as discussed by Hopcroft and Ullman [53]. First, consider the question: "Is a string y contained in language L?" For languages generated by regular, context-free, or context-sensitive grammars, this problem is decidable. However, the problem is undecidable for languages generated by general grammars. It is even more difficult to construct an algorithm to determine if the language generated by any grammar is empty. Such algorithms may be constructed only for context-free and regular grammars; the problem is undecidable for the other two types.

Some problems are decidable only for regular grammars. For example, a general algorithm can determine whether the languages generated by two grammars are equivalent, if one is contained within the other, or if their intersection is non-null only if the grammars are both regular. Note that this conclusion does not imply that the equivalence of the languages of two grammars of type less than 3 can never be determined. Rather, it says that there is no single algorithm which may be applied in every situation which will always be able to produce an answer. Regular languages are also the only type for which it can be determined by a general algorithm whether the language contains all possible (finite and infinite) strings formed from members of an alphabet (and therefore whether the strings generated by the grammar will always be of finite length).

Closure properties can also be posited for the various classes of languages. Consider the set of strings defined by the intersection of two languages of the same type. It may be shown that the intersection of two context-free languages is not necessarily a context-free language. However, the regular, context-sensitive, and general classes of language are all closed under intersection. As for the complement operation, only languages generated by regular grammars have complements which can be generated by other grammars of the same type. Finally, all four types of grammar are closed under substitutions and homomorphisms. A *substitution* f is a mapping of an alphabet onto subsets of another alphabet, so for any x in alphabet A, the substitution $f(x)$ replaces x with some y in B^*, the set of combinations of symbols in alphabet B. A *homomorphism* h is a substitution in which each substituted $h(x)$ contains a single symbol from alphabet B [53]. Thus if symbols are substituted in a grammar of type i, the resulting grammar is still of type i.

2.3 Natural Language Grammars

A grammar for natural language is probably the most familiar. Linguists and computer scientists have also been exploring the formal aspects of natural language grammars for many years, their goal being to create parsers for spoken or written language. Natural language grammars exist for all written or spoken languages. In general, their nonterminal symbols are concepts, such as "sentence," "subject," "verb phrase," "preposition," "past tense suffix," and "adjective." The terminal symbols are actual words. The starting symbol is the nonterminal "sentence" and the set of rules is, for almost all languages, very large. A few examples of rules from the English language are:

$$
\begin{aligned}
\textit{sentence} &\rightarrow \textit{subject predicate} \\
\textit{subject} &\rightarrow \textit{adjective noun} \\
\textit{predicate} &\rightarrow \textit{verb direct\_object} \\
\textit{noun} &\rightarrow \text{cat} \mid \text{house} \mid \text{tree} \mid \cdots \\
&\vdots
\end{aligned}
$$

Many attempts have been made to create a grammar which will give all, and only all, sentences in the English language. Similar attempts, led by Chomsky, have been made to create parsers, or automatons which will accept written or spoken input sentences and return the meanings of the sentences [17]. Some ambitious researchers, including R. C. Berwick, are trying to design a parser which can understand many different natural languages. They hypothesize that all natural languages have the same underlying structure. In grammatical terms, they believe that all the rules which combine nonterminal concepts are identical in every language. Only the rules which substitute in instances of actual words would need to be modified in order to parse different languages [5, 6]. The idea that related but radically different strings may be formed by carefully changing a relatively small number of rules is explored further in Sections 2.4.2 and 5.2.3.

2.4 Shape Grammars

A *shape grammar* is a grammar in which the basic elements are shapes. As will be discussed in Section 3.2, shapes can be any combination of points, line segments, planar sections, or solids. Shape grammars were first proposed by Stiny and Gips [45, 112, 128], and have subsequently been studied and extended by them and many other authors. See Section 7.2 for a discussion of the use of shape grammars to date.

2.4.1 Definitions

Shape grammars satisfy every part of the definition of a grammar given in Section 2.2.1. The alphabets for shape grammars are composed exclusively of shapes. These shapes can be of zero, one, two, or three dimensions, as discussed by Stiny [127].

While shape grammars do not typically have a set of allowable transformations defined, several types of transformation are useful for generating non-obvious or interesting shapes [128]. For example, allowing scaling enables a single rule to apply to shapes of different sizes. Rotation and reflection transformations allow even more shapes to match the left-hand sides of production rules. Additionally, translation allows rules to apply at any position in a shape and is usually necessary in order to generate sizable numbers of shapes. A shape grammar which transforms and arranges symbols from an alphabet has been called a *structure grammar* by Carlson and others [12, 13].

2.4.2 Properties

With a small number of shape rules and the availability of the transformations discussed above, a shape grammar can generate a large number of different shapes. One especially interesting property of shape grammars is that rules can apply to *emergent* shapes, or shapes which are not placed by a single rule but are formed from elements of other shapes. For example, in Figure 2-1, even though only small hexagons are used in the shape production rules, the grammar can produce, among other shapes, diamonds, six-pointed stars, and larger hexagons. This example also uses translation, rotation, and scaling transformations when applying its rules.

Using certain operations in a shape grammar guarantees some closure properties. As

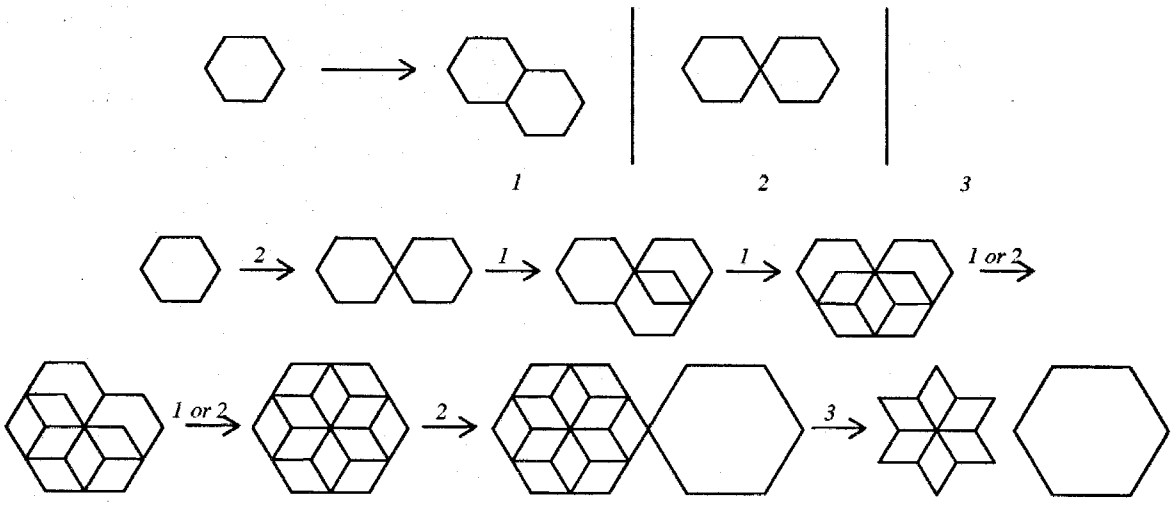


Figure 2-1: A shape grammar using hexagons and some emergent shapes.

will be discussed in Chapter 3, various sets of shapes are closed under several basic shape operations. Therefore, if a shape grammar uses only shapes from a certain set and combines them (via its rules) in certain ways, then all shapes in the grammar's language will be guaranteed to be members of the set.

Changing a single rule or a small number of rules in a shape grammar (or any other type of grammar) can produce very interesting results. For example, by changing the basic shape in the rules of the grammar in Figure 2-1 from a hexagon to a square, radically different shapes may be obtained. Using proper analysis, shape rules may be modified in informative ways. For example, in [69] Koning and Eizenberg present a grammar which generates house plans in Frank Lloyd Wright's Prairie style. In [66], Knight modifies some of the shape relations in the rules from the first paper. The resulting grammar generates plans in Wright's later Usonian style. Thus, by changing rules, some insight as to the nature of a specific design or problem may be gained.

2.5 Extensions to Shape Grammars

Shape grammars may be extended so that their alphabets and rules contain both shapes and other symbols. If a grammar is extended to work with engineering properties, it is known as an *engineering grammar*. Problems in engineering design typically have more

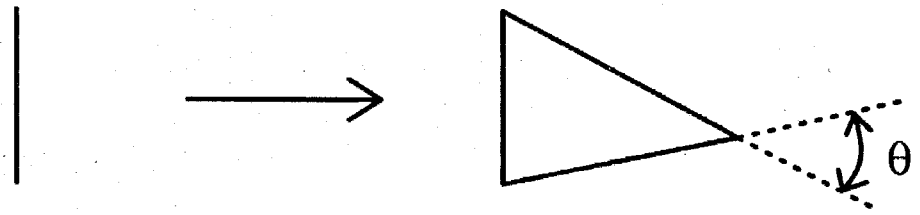


Figure 2-2: Rule for a parametric grammar for tiling a plane with triangles.

than one aspect. For example, when designing a structure the designer must specify its geometry, but must also find its strength, its stiffness, its cost, and other factors which affect how the structure will perform while and after it is manufactured. Furthermore, all the aspects of a design are related in relatively complex ways. Engineering grammars are multidimensional analogs of symbolic grammars. Their alphabets and production rules are defined in more than one domain. These domains could include shapes, words, labels, parameters, graphs, kinematic properties, material properties, colors, or any other factor of interest in design. Examples of grammars in many of these domains will be discussed in Chapter 7. An engineering grammar is a type of *parallel grammar*, a grammar which uses alphabets and rules in more than one domain. The distinguishing property of parallel grammars is that a production in one domain may necessitate a change in another domain. Changes may be made in several domains by, for example, using a production rule defined on multiple domains, or by using a rule in a single domain and placing a symbol in another domain which enables a rule in the second domain to be used.

The set of extended grammars contains many other types of grammars. *Parametric grammars* have rules in which shapes or other properties may be changed by varying one or more parameters. For example, consider the rule for the shape grammar used to tile a plane with triangles shown in Figure 2-2. The starting symbol for this grammar is a single triangle and the only rule is one which adds two lines, with an intersection angle denoted by parameter θ, to an existing line to form another triangle. Implicit in this rule is the requirement that the new lines which are added may not cross any other line (though they may be coincident with other lines) and may not lie within or contain another triangle. It is also required that the existing line either be maximal (*i.e.*, not be contained within another, longer, line) or have endpoints which are intersections with other lines. The parametrized

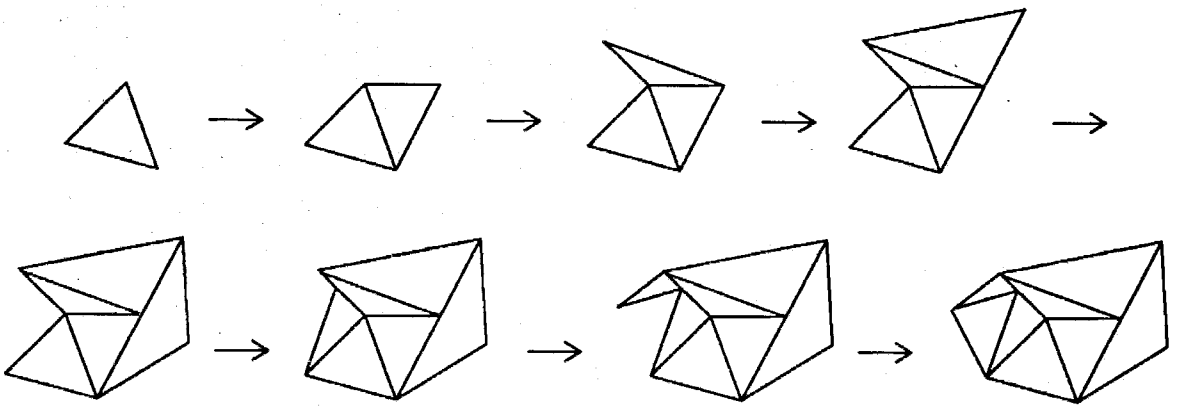


Figure 2-3: The production of a tiling using the parametric grammar.

angle allows the designer to tile the plane with many different sizes of triangle, as shown in Figure 2-3.

Graph grammars are another extension of shape grammars. As discussed by Ehrig and Nagl in [29, 91], graph grammars are used to construct labeled networks of nodes and edges. These grammars have been used to model finite-state automata and to represent the topology of shapes by Fitzhorn and Longenecker [33, 35, 74].

Fitzhorn has claimed [34] that the design process is equivalent to the operation of a Turing machine whose strings describe artifacts of design. However, it can be shown [53] that the set of strings generated by any Turing machine is a recursively enumerable set, and thus may also be generated by a general grammar. Therefore, if the conjecture is true, then grammars can be used to solve any design problem. It may be argued that the design process has either more or less complexity than a general grammar. For example, there are design problems, such as the choice of a ball bearing for a particular application, which might be modeled by a context-sensitive grammar. The starting symbols could be a number of variables standing for the load and speed to which the bearing is to be subjected, the required life, the reliability required of the bearing, and other factors. Context-free parametric rules could first be used to substitute numbers into these variables (with the designer substituting in the appropriate numbers needed to solve a particular problem). Next, context-sensitive rules could be used to generate, for example, the L_{10} life required of the bearing, or its reliability factor. Finally, other context-sensitive rules could be used

to find appropriate bearings in a catalog. The whole process of choosing a bearing for a particular application, described by Shigley and Mischke in [109], could thus be modeled with a context-sensitive grammar.

Conversely, it may be argued that some design tasks are too difficult to be modeled by a general grammar. For example, consider the problem: "Generate all possible types of mechanism which will produce a certain motion." Not only is the number of solutions to this problem infinite, but the number of methodologies which could be used to solve the problem is also infinite, in general. This type of problem is unsolvable even by humans (whose reasoning abilities can exceed those of a Turing machine), so it is inadvisable, and probably impossible, to solve problems of this sort using general grammars. Problems which require some sort of nonlinear reasoning in order to generate a solution are also not well-suited for solution by general grammars.

2.6 Conclusions

Grammars can be defined in many domains or combinations of domains. Engineering grammars in particular are able to manipulate shapes, properties, parameters, and labels. This ability can give the grammars great expressiveness in their domains. In particular, grammars can be used to explore large design spaces using relatively few rules. In the coming chapters, some properties of shape grammars and engineering grammars will be explored. It will be shown that grammars can be powerful design tools when used properly.

Chapter 3

Group-Theoretic Properties of Shape Operations

3.1 Introduction

In the following, several group-theoretic properties of operations on shapes are introduced and proved. For a more complete introduction to group theory, including the definitions of groups, rings, vector spaces, and the like, see Appendix A.

3.2 Shapes and Boolean Algebras

Following the logic of Stone [133], a Boolean ring can be constructed from shapes and the operations of symmetric difference, $\diamond$, and intersection, $\cap$. This ring can then be extended to define a Boolean algebra. This process was originally applied to shapes by Stiny [126].

In the following, shapes are considered as sets of points, line segments, plane segments, and solids. Then the operations on shapes can be considered to be equivalent to operations on sets, so that the operations have an intuitive meaning. A similar course was followed by Requicha and Tilove [99], who showed that regular sets (sets which are equal to the closure of their interior) form a Boolean algebra under regularized union, intersection, complement and difference operators.

Definition 3.1 (Set of Shapes) A *set of shapes*, U_{ij}, $i \leq j$, is the set of all elements of dimension i lying in $\mathbb{R}^j$.

For example, U_{12} is the set of all lines and curves in the plane, while U_{33} is the set of all solids in three-dimensional space. See Figure 3-1 for some examples. Operations may only

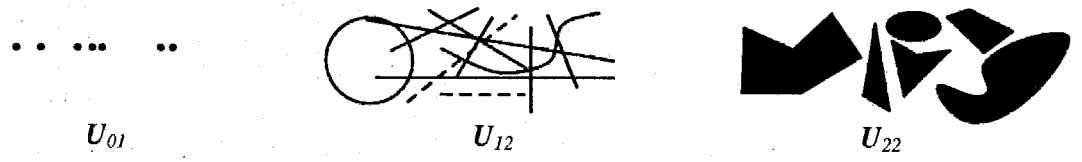


Figure 3-1: Some members of different sets of shapes.

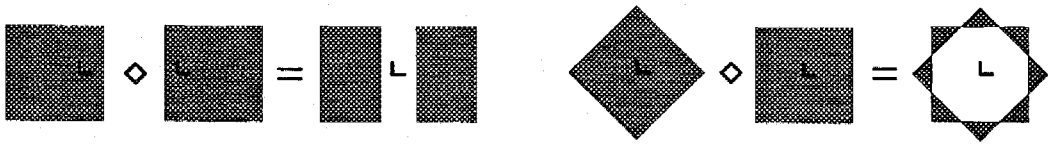


Figure 3-2: The symmetric difference operation. The L-shaped symbol denotes the origin of the coordinate system.

be defined on shapes which are members of the same set of shapes. Stiny presents a more complete discussion of these sets in [125].

Definition 3.2 (Symmetric Difference Operation) The *symmetric difference operation*, $\diamond$, is a binary operation on sets of shapes,

$$\begin{aligned} \diamond : U_{ij} \times U_{ij} &\rightarrow U_{ij} \\ (A, B) &\mapsto A \diamond B = \{X \in U_{ij} \mid X \subseteq A \vee X \subseteq B, X \not\subseteq A \cap B\}, \end{aligned}$$

where $\vee$ denotes the logical "or" of two statements.

The symmetric difference of two shapes is equivalent to performing an "exclusive or" on the shapes or taking the union modulo 2. The symmetric difference of A and B is the set of shapes which is contained in either A or B, but not both. See Figure 3-2 for some examples from U_{22}.

Lemma 3.1 *Sets of shapes, U_{ij}, form an abelian group under the symmetric difference operation, with identity equal to the empty shape, $\emptyset$.*

Proof. The symmetric difference operation is commutative since for all $A, B \in U_{ij}$,

$$\begin{aligned} A \diamond B &= \{X \in U_{ij} \mid X \subseteq A \vee X \subseteq B,\ X \not\subseteq A \cap B\} \\ &= \{Y \in U_{ij} \mid Y \subseteq B \vee Y \subseteq A,\ Y \not\subseteq B \cap A\} \\ &= B \diamond A. \end{aligned}$$

The symmetric difference operation is associative since for all $A, B, C \in U_{ij}$,

$$\begin{aligned} A \diamond (B \diamond C) &= \{X \mid X \subseteq A \vee X \subseteq B \diamond C,\ X \not\subseteq A \cap (B \diamond C)\} \\ &= \{X \mid X \subseteq A \vee X \subseteq \{Y \mid Y \subseteq B \vee Y \subseteq C,\ Y \not\subseteq B \cap C\}, \\ &\qquad X \not\subseteq (A \cap \{Y \mid Y \subseteq B \vee Y \subseteq C,\ Y \not\subseteq B \cap C\})\} \\ &= \{X \mid X \subseteq A \vee X \subseteq B \vee X \subseteq C,\ X \not\subseteq B \cap C,\ X \not\subseteq A \cap B,\ X \not\subseteq A \cap C, \\ &\qquad X \not\subseteq A \cap B \cap C\} \\ &= \{X \mid X \subseteq A \vee X \subseteq B \vee X \subseteq C,\ X \not\subseteq B \cap C,\ X \not\subseteq A \cap B,\ X \not\subseteq A \cap C\} \\ &= \{X \mid X \subseteq C \vee X \subseteq A \diamond B,\ x \not\subseteq C \cap (A \diamond B)\} \\ &= (A \diamond B) \diamond C. \end{aligned}$$

The sets contain additive identity $\emptyset$ since for $A \in U_{ij}$,

$$\begin{aligned} A \diamond \emptyset = \emptyset \diamond A &= \{X \in U_{ij} \mid X \subseteq A \vee X \subseteq \emptyset,\ X \not\subseteq A \cap \emptyset\} \\ &= \{X \in U_{ij} \mid X \subseteq A,\ X \not\subseteq \emptyset\} \\ &= \{X \in U_{ij} \mid X \subseteq A\} \\ &= A. \end{aligned}$$

The two-sided inverse of an element is just the element itself, *i.e.*, $A^{-1} = -A = A$ for all $A \in U_{ij}$, since

$$\begin{aligned} A \diamond A &= \{X \in U_{ij} \mid X \subseteq A \vee X \subseteq A, X \not\subseteq A \cap A\} \\ &= \{X \in U_{ij} \mid X \subseteq A, X \not\subseteq A\} \\ &= \emptyset. \end{aligned}$$

■

Lemma 3.2 *The intersection operation is distributive over the symmetric difference operation.*

Proof. Left-distribution:

$$\begin{aligned} A \cap (B \diamond C) &= A \cap \{X \in U_{ij} \mid X \subseteq B \vee X \subseteq C, X \not\subseteq B \cap C\} \\ &= \{X \in U_{ij} \mid X \subseteq A \cap B \vee X \subseteq A \cap C, X \not\subseteq B \cap C\} \\ &= \{X \in U_{ij} \mid X \subseteq A \cap B \vee X \subseteq A \cap C, X \not\subseteq A \cap B \cap C\} \\ &= \{X \in U_{ij} \mid X \subseteq A \cap B \vee X \subseteq A \cap C, X \not\subseteq (A \cap B) \cap (A \cap C)\} \\ &= (A \cap B) \diamond (A \cap C). \end{aligned}$$

Right-distribution:

$$\begin{aligned} (A \diamond B) \cap C &= C \cap (A \diamond B) \\ &= (C \cap A) \diamond (C \cap B) \\ &= (A \cap C) \diamond (B \cap C). \end{aligned}$$

■

Note that in the use of both the symmetric difference operation and the intersection operation, the results of the operation are confined to lie within the set under consideration. For example, consider two lines that cross in the plane in the set of shapes U_{12}. Their intersection is said to be the empty shape because the actual intersection is a point, which is not contained in U_{12}. This confinement of operations to the set of shapes under consideration makes the operations behave like the regularized operations, discussed by Requicha in [98],

which are required for constructive solid geometry modeling systems.

Theorem 3.3 *Sets of shapes, U_{ij}, along with the operations of intersection and symmetric difference, form a Boolean ring with intersection identity equal to the set of shapes.*

Proof. Lemma 3.1 has proven that the set U_{ij}, together with the symmetric difference operation, is an abelian group with identity element $\emptyset$. The intersection operation on shapes, $\cap$, is associative because for all $A, B, C \in U_{ij}$, $(A \cap B) \cap C = A \cap (B \cap C)$ by definition. Lemma 3.2 has proven that shape intersection is both left- and right-distributive over symmetric difference.

For all $A \in U_{ij}$, $A \cap A = A$.

For all $A \in U_{ij}$, $A \cap U_{ij} = U_{ij} \cap A = A$, so U_{ij} is the identity under intersection.

Therefore, sets of shapes, U_{ij}, along with the operations of intersection and symmetric difference, form a Boolean ring with intersection identity equal to the set of shapes. ■

Note that no elements in the Boolean ring of shapes are invertible except the universal set U_{ij}. Note also that all elements of the Boolean ring of shapes satisfy the Boolean ring property that $A + A = \emptyset$ for all $A \in U_{ij}$ (where "+" is the ring's "addition" operation, which is the symmetric difference in this ring) because every element of U_{ij} is its own inverse, as shown above.

Now, thanks to a theorem by Stone [133], the Boolean ring of shapes can be converted into a Boolean algebra. From the intersection and symmetric difference operations, two additional operations, the binary union operation, $\cup$, and the unary complement operation, $'$, can be derived.

Theorem 3.4 (Stone) *Given a Boolean ring of shapes with identity under intersection, U_{ij}, the introduction of the binary operation $\cup$ and the unary operation $'$ through the equations*

$$A \cup B = A \diamond B \diamond (A \cap B)$$

$$A' = A \diamond U_{ij}$$

converts the Boolean ring into a Boolean algebra in which

$$A \cup B = B \cup A$$

$$A \cup (B \cup C) = (A \cup B) \cup C$$

$$(A' \cup B')' \cup (A' \cup B)' = A.$$

The old operations may be expressed in terms of the new ones through the equations

$$A \diamond B = (A \cap B') \cup (A' \cap B) = (A' \cup B'')' \cup (A'' \cup B')'$$

$$A \cap B = (A' \cup B')'.$$

See [133] for a proof of this theorem and further discussion of Boolean algebras. In particular, that article introduces the ordering relation $<$, and consequently shape inclusion. Then, it goes on to deal with atoms, or members of the ring which are included in all other members of the ring. In the ring of shapes U_{ij}, there are no atoms (no matter what value the indices i and j have). If there were atoms, a basis for the ring could be defined.

Subrings of the Boolean rings of shapes U_{ij} can also be found. For instance, as Stone points out, for any $A \in U_{ij}$, the subclass $s(A)$, composed of all elements $B \in U_{ij}$ such that $A \cap B = B$, is a subring of U_{ij}. For example, in U_{33}, if A is a cube, then $s(A)$ is the set of all solids which fit entirely inside the cube. A subring may also be formed of shapes having similar characteristics. For example, the set of all sets of horizontal and vertical lines in the plane forms a subring of U_{12}.

Homomorphisms between Boolean rings of shapes and other rings can be defined. As Stone proves, if an algebraic system R is homomorphic to a Boolean ring of shapes U_{ij} with respect to any of the pairs of operations $\diamond$ and $\cap$, $\cup$ and $'$, or $\cup$ and $\cap$, then R is homomorphic

to U_{ij} with respect to all four operations $\diamond$, $\cap$, $\cup$, and $'$. Furthermore, if the above is true, then R is a Boolean ring with identity under $\cap$ and the homomorphism $U_{ij} \rightarrow R$ carries the additive and multiplicative identity elements of U_{ij} into the corresponding additive and multiplicative identity elements of R.

3.3 Shapes and Rings

This section discusses the properties of shapes under the operations of union and convolution. Following the lead of Guibas, Ramshaw, and Stolfi [48], shapes are taken to be polygonal tracings. Next, operations on the tracings are defined so that the union operation corresponds directly with the set union of shapes while the convolution operation is a generalized form of the Minkowski sum and difference operations described in [43, 49].

The following is partially a restatement of formalisms presented by Guibas *et al.* in [48]. All the italicized definitions, both numbered and unnumbered, and nomenclature introduced in this section are adapted with little or no change from that paper. Sections 3.3.1 and 3.3.2 mostly present work done by Guibas *et al.* Most of the original content of those two sections is the work related to the mapping **T**, defined below. Additionally, all numbered definitions in the rest of this chapter present terms previously defined by Guibas *et al.* Furthermore, the results in Theorem 3.8 and Lemmas 3.9, 3.11, and 3.12 were presented, but not proved, in that paper. Some unnecessary material from the original article has been omitted while enough of the notation and definitions have been retained to show that shapes, thought of as polygonal tracings (along with the operations of union and convolution), form a ring.

3.3.1 Basic Definitions

Definition 3.3 (Oriented Line) An *oriented line* o is a straight line in $\mathbb{R}^2$ along with one of the two unit vectors parallel to it, called the line's *orientation*.

The orientation of line o is denoted by $\hat{o}$. The *opposite* of a line o, denoted o°, is the oriented line which is coincident with o but has opposite orientation.

Definition 3.4 (State) A *state* s is the combination of a point in $\mathbb{R}^2$, denoted $\dot{s}$, and a tangent, denoted $\bar{s}$, which is an oriented line containing $\dot{s}$.

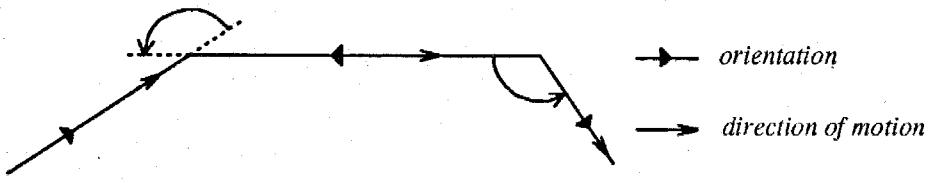


Figure 3-3: A trip and the symbols used to depict it.

The orientation of a state s, denoted $\hat{s}$, is just the orientation of its tangent.

Definition 3.5 (Trip) A *trip* is a time-ordered sequence of states. The sequence must be smooth: position and orientation must be continuous functions of time. Whenever the vector derivative of position with respect to time is nonzero, it must be collinear with the orientation.

Think of a two-wheeled cart being pushed across a field. At any fixed time, the cart has a position and orientation. A trip is a record of where the cart travels. The position and orientation must be continuous functions of time so that the cart's velocity is always finite. The cart's velocity must be collinear with its orientation in order to avoid skidding sideways. However, the velocity and orientation may be parallel but opposite; in that case, the cart is moving backward. See Figure 3-3 for the depiction of a trip.

Definition 3.6 (Tour) A *tour* is a trip in which the first and last states coincide in position and orientation.

Definition 3.7 (Move) A *move* is a trip in which orientation is constant while position changes, tracing a straight line segment.

A forward move occurs when motion is in the direction of the orientation, while a backward move occurs when the directions of motion and orientation are opposite.

Definition 3.8 (Turn) A *turn* is a trip in which position is constant while orientation changes.

A turn may be either to the left (counterclockwise) or to the right (clockwise).

The forward move or left turn which begins with state x and ends with state y is denoted by $\langle xy \rangle$. A backward move or right turn from state y to state x is denoted $-\langle xy \rangle$. Of course,

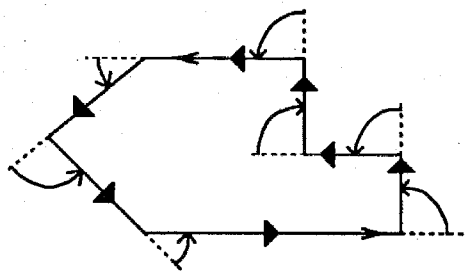


Figure 3-4: A simple polygonal tour.

in any case, either $\dot{x} = \dot{y}$ (turn) or $\hat{x} = \hat{y}$ (move). The states x and y are called the *extremal states* of the move or turn.

The concatenation of a finite number of moves and turns is called a *polygonal trip*. A polygonal trip which starts and ends at the same state is called a *polygonal tour*. The states traveled through on a polygonal tour define a polygon, with edges corresponding to moves and vertices corresponding to turns. A polygonal tour may be constructed from any polygon by concatenating forward moves along the polygon's edges with left or right turns of less than 180° between the edges at each vertex. This type of polygonal tour, where all motion is forward and all turns are less than 180°, is called a *simple polygonal tour*. See Figure 3-4 for an example. A mapping between polygons and simple polygonal tours will be defined below. Note that any simply-connected closed two-dimensional contour can be approximated to an arbitrary degree by a polygon (and hence a simple polygonal tour) if the lengths of the polygon's edges are small enough. Thus, all of the following results which are proved for polygons are also valid for simply-connected closed contours.

Instead of considering states as ordered in time (though the restrictions given in the definition of a trip still apply), think of a trip as simply a signed multiset of the states traversed. A *multiset* is a set in which elements may occur with any multiplicity (whereas in a set, elements may occur only once). The number of times a given state s occurs in a trip is just equal to the number of forward moves and left turns passing through s minus the number of backward moves and right turns passing through s. An extremal state of a move or turn only contributes $\pm 1/2$ to the total number of times the state appears. For example, in the forward move or right turn $\langle xy \rangle$, states x and y occur with multiplicity $1/2$, while all the states in between occur with multiplicity one. In the backward move from y

to x, x and y have multiplicity $-1/2$, while the other states have multiplicity -1. These facts provide the rationale for denoting the moves $\langle xy \rangle$ and $-\langle xy \rangle$ respectively. The sum, or concatenation, of two moves or two turns can be found:

$$\langle xy \rangle + \langle yz \rangle = \langle xz \rangle.$$

Thus, in an intuitive manner, the following can be written:

$$\langle xy \rangle + -\langle xy \rangle = \emptyset,$$

where $\emptyset$ denotes the empty multiset, also called the *null tracing*.

Definition 3.9 (Partial Polygonal Tracing) A *partial polygonal tracing* is a signed multiset of states that can be expressed as the sum of finitely many moves and turns, each taken with arbitrary multiplicity.

The set of all possible partial polygonal tracings in the two-dimensional plane will be denoted T_p^2. If each move and turn has integer multiplicity, the partial tracing is said to be *integral*. If each state is entered just as often as it is left, then the partial tracing is said to be *balanced*. A balanced tracing corresponds to a tour. An integral and balanced partial polygonal tracing is called simply a *tracing*. A *null tracing* is one in which each state occurs with zero multiplicity. The set of all possible tracings in the two-dimensional plane will be denoted T^2.

Now a mapping between polygons and tracings will be defined. Polygons are considered to be simply-connected, compact, closed sets of points in $\mathbb{R}^2$ with boundaries made up of finite numbers of straight line segments. The notation $\mathcal{C}$ will be used to refer to the set of all possible polygons (the letter stands for *compact, connected* and *closed*). The boundary of polygon $\mathcal{P}$ is denoted by $\partial\mathcal{P}$ and the corners of the polygon are denoted by $\partial\partial\mathcal{P}$. For any point $x \in \partial\mathcal{P}$, let the vector $V(x)$ denote the unit vector parallel to the boundary at x such that in the neighborhood of x, $\mathcal{P}$ lies to the left of $V(x)$. If $x \in \partial\partial\mathcal{P}$, then $V(x)$ will have two values, one for each edge of $\mathcal{P}$ which contains x. Otherwise, $V(x)$ will be single-valued. For any point $y \in \partial\partial\mathcal{P}$, let $\alpha(y)$ and $\beta(y)$ be the two distinct unit vectors obtained from $V(y)$. Denote by $(\alpha(y), \beta(y))$ the set of all unit vectors lying between $\alpha(y)$ and $\beta(y)$ which

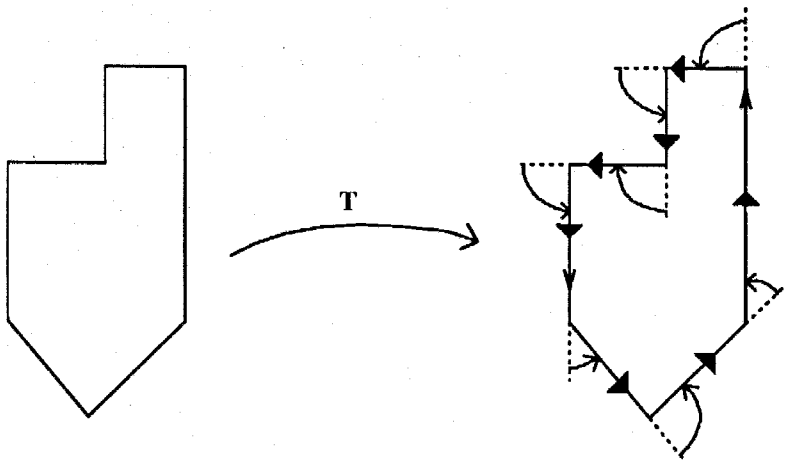


Figure 3-5: The mapping **T**.

span an angle of less than 180 degrees. This set does not contain $\alpha(y)$ or $\beta(y)$, only the vectors in between. The boundary of a polygon will be mapped into a counterclockwise, forward tracing with the operator **T**:

$$\begin{aligned} \mathbf{T} : \mathcal{C} &\rightarrow T^2 \\ \mathcal{P} &\mapsto \mathbf{T}(\mathcal{P}) = \{x, y \mid \dot{x} \in \partial\mathcal{P},\ \hat{x} = V(\dot{x}),\ \dot{y} \in \partial\partial\mathcal{P},\ \hat{y} \in (\alpha(\dot{y}), \beta(\dot{y}))\ \}. \end{aligned}$$

An example of this mapping is shown in Figure 3-5.

Claim 3.5 *The mapping* **T** *described above is injective.*

Proof. Two things must be proven. First, it must be shown that **T** actually produces tracings when applied to polygons. Then it must be shown that if $\mathbf{T}(\mathcal{A})$ and $\mathbf{T}(\mathcal{B})$ are two counterclockwise, forward tracings and $\mathbf{T}(\mathcal{A}) = \mathbf{T}(\mathcal{B})$, then $\mathcal{A} = \mathcal{B}$.

Consider polygon $\mathcal{P} \in \mathcal{C}$. The set of states obtained from the mapping $\mathbf{T}(\mathcal{P})$ consists of two subsets. The first subset is the set of all states contained in moves, each with multiplicity one (so it contains the moves plus an additional multiplicity of 1/2 for each of their extremal states). The second subset is the set of all states contained in turns, minus the extremal states, each with multiplicity one. By subtracting a multiplicity of 1/2 from the extremal states of each move in the first set and adding those states with multiplicity of 1/2 to the turns, a collection of moves and turns is obtained which satisfy the definition of a

partial polygonal tracing. From the definition of $\mathbf{T}$, it is obvious that the states completely traverse the boundary of P in a counterclockwise direction, so the partial polygonal tracing is balanced and the moves are all forward. Since it was shown above that all states have multiplicity one, the partial polygonal tracing is integral. Therefore, $\mathbf{T}(\mathcal{P})$ is a tracing.

Now consider counterclockwise forward tracings $\mathbf{T}(\mathcal{A})$ and $\mathbf{T}(\mathcal{B})$, with $\mathbf{T}(\mathcal{A}) = \mathbf{T}(\mathcal{B})$ and $\mathcal{A}, \mathcal{B} \in \mathcal{C}$. Since the tracings are equal, $\{\dot{x} \mid x \in \mathbf{T}(\mathcal{A})\} = \{\dot{y} \mid y \in \mathbf{T}(\mathcal{B})\}$. Therefore, they define the same boundary. Since polygons $\mathcal{A}$ and $\mathcal{B}$ are closed, compact and simply-connected, they each must have one and only one boundary. Since the boundaries are the same, the polygons must be the same. Therefore, the mapping from polygons to tracings is one-to-one, or injective. ■

Since the mapping $\mathbf{T}$ is injective, another mapping, $\mathbf{T}^{-1}$, may be defined from T^2 to $\mathcal{C}$. This mapping maps counterclockwise, forward, non-self-crossing tracings in which each state has multiplicity ± 1 and each turn is through an angle less than 180 degrees to closed, compact, simply-connected polygons. This inverse mapping is done by following a procedure similar to that used in the final part of the above proof. The inverse mapping can be extended to tracings which cross themselves and/or have states with multiplicity of magnitude greater than one. The extension will be described below, after the concept of a winding number is introduced.

The mapping $\mathbf{T}$ can also be extended to generate tracings in which all moves are backward and traversal is clockwise. To do so, the notion of a *negative polygon* shall be introduced. Negative polygons will be used later on in this section when discussing the Minkowski difference of shapes and (with a different definition) in Section 3.4 when discussing the scaling of shapes by real numbers.

3.3.2 Functions of States

Now define four *state counting functions* of a partial tracing A at state s:

$$\begin{aligned}
\mu^+(s) &= \lim_{m \to s^+} \{(\text{forward moves out of } m) - (\text{backward moves into } m)\} \\
\mu^-(s) &= \lim_{m \to s^-} \{(\text{forward moves into } m) - (\text{backward moves out of } m)\} \\
\tau^+(s) &= \lim_{t \to s^+} \{(\text{left turns out of } t) - (\text{right turns into } t)\}
\end{aligned}$$

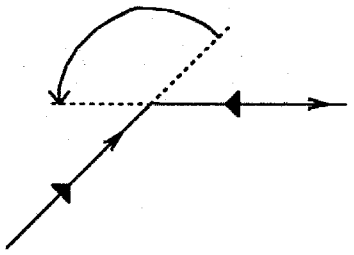


Figure 3-6: A move-turn-move sequence.

$$\tau^-(s) = \lim_{t \to s^-} \{(\text{left turns into } t) - (\text{right turns out of } t)\}.$$

In the above definitions, $\lim_{m \to s^+}$ means the limit as state m approaches state s from ahead of s. Similarly, $\lim_{m \to s^-}$ means the limit as state m approaches state s from behind of s, $\lim_{t \to s^+}$ means the limit as state t approaches state s from the left of s, and $\lim_{t \to s^-}$ means the limit as state t approaches state s from the right of s. As an example of how these state counting functions are defined, consider the move-turn-move sequence pictured in Figure 3-6. The partial tracing is composed of a forward move, a left turn, and a backward move. All the states f in the forward move except the one at its end (and at the start of the turn) have $\mu^+(f) = \mu^-(f) = 1$ and $\tau^+(f) = \tau^-(f) = 0$. The state at the end of the forward move and start of the turn, e, has $\mu^+(e) = 0$, $\mu^-(e) = 1$, $\tau^+(e) = 1$, and $\tau^-(e) = 0$. All the states t in the turn except for the extremal states have $\mu^+(t) = \mu^-(t) = 0$ and $\tau^+(t) = \tau^-(t) = 1$. The state at the end of the turn and the start of the backward move, s, has $\mu^+(s) = -1$, $\mu^-(s) = \tau^+(s) = 0$, and $\tau^-(s) = 1$. Finally, all the states b in the backward move, except for the one at its start, have $\mu^+(b) = \mu^-(b) = -1$ and $\tau^+(b) = \tau^-(b) = 0$.

A move or turn passing through s is considered to be both entering and exiting the state. If the partial tracing referred to by a state counting function is not obvious, it is included as a subscript, as in $\mu_A^+(s)$. A tracing A is completely determined (or described) by the four functions $\mu_A^\pm(s)$ and $\tau_a^\pm(s)$. The functions $\mu(s)$ and $\tau(s)$ can also be defined as

$$\mu(s) = \frac{\mu^+(s) + \mu^-(s)}{2}$$
$$\tau(s) = \frac{\tau^+(s) + \tau^-(s)}{2}.$$

These quantities can be interpreted as the amounts of moving and turning, respectively, that take place at state s. Note that the multiplicity of state s in a partial tracing is equal to $\mu(s) + \tau(s)$.

A partial tracing is integral if and only if $\mu^+(s)$, $\mu^-(s)$, $\tau^+(s)$ and $\tau^-(s)$ are all integers for every state s in the partial tracing. A partial tracing is balanced if and only if

$$\mu^+(s) - \mu^-(s) + \tau^+(s) - \tau^-(s) = 0$$

for all states s in the partial tracing.

In the following, summation over a multiset A will be defined as

$$\sum_{s \in A} f(s) = \sum (\mu(s) + \tau(s)) \cdot f(s),$$

where $f(s)$ is any function of a state. The notation $[xy]$ will be used to denote the same multiset as $\langle xy \rangle$, but where x and y each have multiplicity one instead of $1/2$. Similarly, let $[xy) = [xy] - \{y\}$, $(xy] = [xy] - \{x\}$, and $(xy) = [xy] - \{x\} - \{y\}$. Note that $[xx] = \{x\}$, $[xx) = (xx] = \emptyset$, and $(xx) = -\{x\}$.

Now define the *ray out of* s as the set of all states with tangent $\bar{s}$ which are located ahead of state s. For partial tracing A and state s, the *winding number of* s *with respect to* A, denoted $\omega_A(s)$, is defined to be the total number of moves in A which cross the ray out of s from right to left minus the number of moves which cross it from left to right. If A is a tracing, then the winding number of a state s is the number of times a cart goes around s (in a counterclockwise direction) while traversing A. Formally,

$$\omega_A(s) = \sum_{r \in \langle s\infty \rangle} \sum_{x \in \langle\langle rr^\circ \rangle\rangle} \mu_A(x) + \frac{\tau_A(s) + \tau_A(s^\circ)}{2},$$

where $\langle\langle xy \rangle\rangle$ denotes the multiset $\langle xy \rangle + \ -\langle yx \rangle = (xy) + \ -(yx)$ and states in $\langle\langle ss^\circ \rangle\rangle$ have multiplicity $1/2$ because s only has multiplicity $1/2$ in $\langle s\infty \rangle$. For example, in a counterclockwise-direction non-self-intersecting simple polygonal tracing P, the winding number of state s is zero if $\dot{s}$ is not located inside of P, one if $\dot{s}$ is inside P, and $1/2$ if $\dot{s}$ is on the tour itself. See Figure 3-7 for an example.

Using the winding number, the mapping $\mathbf{T}^{-1}$ can be extended to apply to tracings which

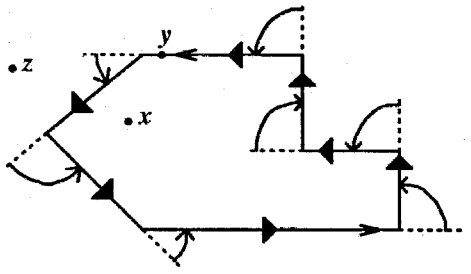


Figure 3-7: The winding numbers of states x, y, and z are 1, 1/2 and 0, respectively.

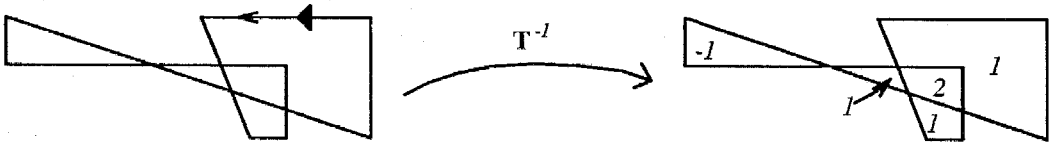


Figure 3-8: A self-crossing tracing composed of turns and forward moves, mapped through $\mathbf{T}^{-1}$ to stacks of polygons. The number of polygons on each stack (equal to the winding number in the tracing) is indicated.

cross themselves and/or have states with multiplicity of magnitude greater than one. The tracings still must have turns through angles of less than 180 degrees. A tracing will, in general, define several polygonal regions in the plane, separated by moves and turns. Inside each region, all states s will have the same integer winding number $\omega(s)$. The regions can be thought of as "stacks" of polygons, the number of polygons equal to the winding number. The polygons may be negative, as discussed below. See Figure 3-8 for an example.

3.3.3 Operations on Tracings

Now that some terminology has been introduced, several operations on tracings can be defined. It will be demonstrated that two of these operations satisfy the properties of operations on a field of tracings. This result implies that some of the more intuitive aspects of operations on shapes can be formalized. In addition, as with the Boolean ring of shapes, homomorphisms between the set of tracings (or shapes) and other fields in which shapes are described may be discovered which allow designers to work with other aspects of the shapes beside their forms.

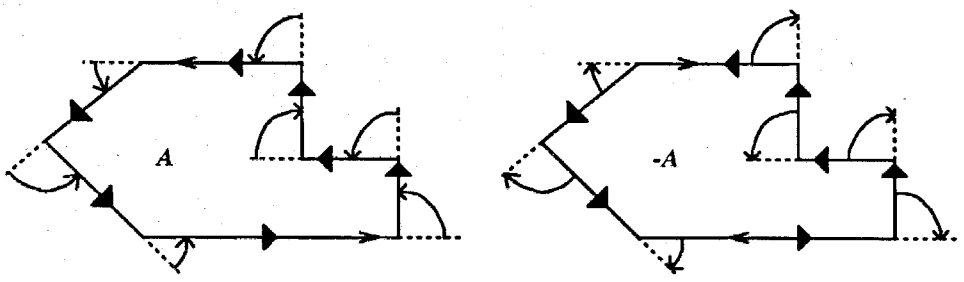


Figure 3-9: A tracing and its inverse.

Definition 3.10 (Tracing Addition Operation) The *tracing addition operation*, also called addition or sum and denoted $+$, is a binary operation on partial polygonal tracings

$$\begin{aligned} + : T_P^2 \times T_P^2 &\rightarrow T_P^2 \\ (A, B) &\mapsto A + B = \{x \mid x \in A \vee x \in B\} \end{aligned}$$

where each state x has multiplicity equal to the sum of its multiplicities in A and B.

The sum of n copies of tracing A is denoted nA. Similarly, the inverse of a tracing A, which is just its time reversal, is denoted $-A$. The inverse of a tracing corresponds to a path traced in reverse order and is simply an extension of the definition of a backward move (or right turn). If the tracing A is non-self-intersecting, is a simple polygonal tour and is counterclockwise (so that all states inside the tracing have winding number equal to 1), then the tracing $-A$ is clockwise (so all states inside the negative tracing have winding number equal to -1). See Figure 3-9 for an example. It should be apparent that $A + (-A) = \emptyset$ for any tracing A. One can use $A - B$ to stand for $A + (-B)$. Similarly, for each tracing P which is generated by a polygon $\mathcal{P}$ (*i.e.*, $P = \mathbf{T}(\mathcal{P})$), the negative tracing $-P$ is generated by the negative polygon $-\mathcal{P}$. A negative polygon can thus be thought of as the "additive inverse" of a polygon as it was originally defined.

The tracing addition operation is closely linked to the shape union operation discussed in Section 3.2. The shape union of two positive non-self-intersecting polygons $\mathcal{P}$ and $\mathcal{Q}$, or $\mathcal{P} \cup \mathcal{Q}$, can be found by constructing the counterclockwise simple polygonal tours of the

polygons, $\mathbf{T}(\mathcal{P})$ and $\mathbf{T}(\mathcal{Q})$, and calculating their sum. Then the definition can be written:

$$\mathcal{P} \cup \mathcal{Q} = \{y \mid y = \dot{x},\ x \in T^2,\ \omega_{\mathbf{T}(\mathcal{P})+\mathbf{T}(\mathcal{Q})}(x) > 0\}.$$

Since $\mathcal{P}$ and $\mathcal{Q}$ are contained in $\mathcal{C}$, they are closed, compact and simply-connected. Therefore, so are the sets of all the states with winding number greater than zero (located on and inside their respective tracings). The tracing addition operation just combines the two sets of states together, so the resulting set of states is closed and compact, but may either be simply-connected (if the original two polygons intersect) or have two unconnected areas (if they are disjoint). Taking the positions of all those states, either one polygon or the original two polygons is obtained. Thus, the set of points defined by the operation is really a polygon.

In a similar manner, the complement of non-self-intersecting polygon $\mathcal{P}$ can be constructed by finding all the points not contained in the corresponding counterclockwise polygonal tracing $\mathbf{T}(\mathcal{P})$:

$$\mathcal{P}' = \{y \mid y = \dot{x},\ x \in T^2,\ \omega_{\mathbf{T}(\mathcal{P})}(x) = 0\}.$$

Note, however, that the set of points $\mathcal{P}'$ is neither simply-connected nor bounded. Therefore, the mapping $\mathbf{T}(\mathcal{P}')$ is not defined. While this loss somewhat reduces the utility of tracings for representing shape operations, it by no means implies that tracings are useless. The boundedness issue could be resolved by restricting the shapes considered to a fixed subset of $\mathbb{R}^2$, since every real-world application of shape operations has limits on the amount of space available. Furthermore, once shapes are restricted to a fixed area of the real plane, any holes or inclusions in shapes can be removed by connecting them to the boundary of the space with infinitesimally close pairs of (not necessarily straight) lines. This resulting polygon can then be converted to a tracing. The lines connecting the holes with the boundaries will have two moves through them, one in each direction. For an example, see Figure 3-10. The points in the plane with winding number greater than zero are then exactly those points contained in $\mathcal{P}'$.

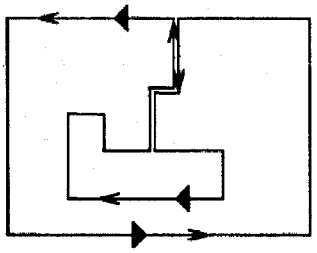


Figure 3-10: A fixed subset of $\mathbb{R}^2$ with a hole represented by a gap.

The intersection of non-self-intersecting polygons $\mathcal{P}$ and $\mathcal{Q}$ is just the area that is inside both of the corresponding counterclockwise polygonal tracings $\mathbf{T}(\mathcal{P})$ and $\mathbf{T}(\mathcal{Q})$:

$$\begin{aligned}\mathcal{P} \cap \mathcal{Q} \;=\; & \{y \mid y = \dot{x},\, x \in T^2,\, (\omega_{\mathbf{T}(\mathcal{P})+\mathbf{T}(\mathcal{Q})}(x) > 1) \vee \\ & (\omega_{\mathbf{T}(\mathcal{P})+\mathbf{T}(\mathcal{Q})}(x) = 1,\, \omega_{\mathbf{T}(\mathcal{P})}(x) = \omega_{\mathbf{T}(\mathcal{Q})}(x) = 1/2)\}.\end{aligned}$$

The first clause of this definition gathers all points which are inside one polygon and either inside or on the boundary of the other. The second clause gathers the points which are on the boundaries of both polygons.

Finally, since the symmetric difference of two shapes is just the set of points contained in one of the shapes but not both:

$$\begin{aligned}\mathcal{P} \diamond \mathcal{Q} \;=\; & \{y \mid y = \dot{x},\, x \in T^2,\, (\omega_{\mathbf{T}(\mathcal{P})+\mathbf{T}(\mathcal{Q})}(x) = 1/2) \vee \\ & (\omega_{\mathbf{T}(\mathcal{P})+\mathbf{T}(\mathcal{Q})}(x) = 1, \omega_{\mathbf{T}(\mathcal{P})}(x) \neq \omega_{\mathbf{T}(\mathcal{Q})}(x))\}.\end{aligned}$$

In this definition, the first clause gathers all points on the boundary of only one polygon. The second clause gathers all points which are inside one polygon but not the other.

Now that the connections between tracing addition and the standard shape operations have been demonstrated, a group-theoretic property which is also true for polygons (if tracing addition is replaced by shape union) is presented.

Lemma 3.6 *Tracings, along with the tracing addition operation $+$, form an abelian group.*

Proof. For any tracings $A, B, C \in T^2$, the following statements stem directly from the definition of the tracing addition operation:

$$A + (B + C) = (A + B) + C = A + B + C;$$

$$A + \emptyset = \emptyset + A = A;$$

$$A + (-A) = -A + A = \emptyset;$$

$$A + B = B + A.$$

Therefore, $(T^2, +)$ forms an abelian group with identity equal to the null tracing and inverse of any tracing equal to its reversal. ∎

Now consider another operation on tracings, one which displays behavior reminiscent of multiplication: the tracing convolution operation.

Definition 3.11 (Convolution Operation) The *convolution operation*, or convolution, $*$, is a binary operation on partial polygonal tracings

$$\begin{aligned} * : T_P^2 \times T_P^2 &\rightarrow T_P^2 \\ (A, B) &\mapsto A * B = \{c \mid \hat{c} = \hat{a} = \hat{b},\ \dot{c} = \dot{a} + \dot{b},\ a \in A,\ b \in B, \\ &\qquad a \text{ or } b \text{ is part of a turn}\}. \end{aligned}$$

If a has multiplicity m and b has multiplicity n, then $c = a * b$ will have multiplicity mn.

At least one of the states must be part of a turn because if A and B have parallel moves, then there would be infinitely many pairs of states in those moves whose positions would sum to the same state in the convolution. See Figure 3-11 for an example of the convolution of two tracings. It is obvious from the above definition that the convolution operation is commutative: $A * B = B * A$ for all $A, B \in T_P^2$.

The state counting functions for a convolution can be found using the following formulas:

$$\mu^{\pm}_{A*B}(s) = \sum_{\substack{\dot{x} + \dot{y} = \dot{s} \\ \hat{x} = \hat{y} = \hat{s}}} \mu^{\pm}_A(x)\,\tau_B(y) + \mu^{\pm}_B(x)\,\tau_A(y)$$

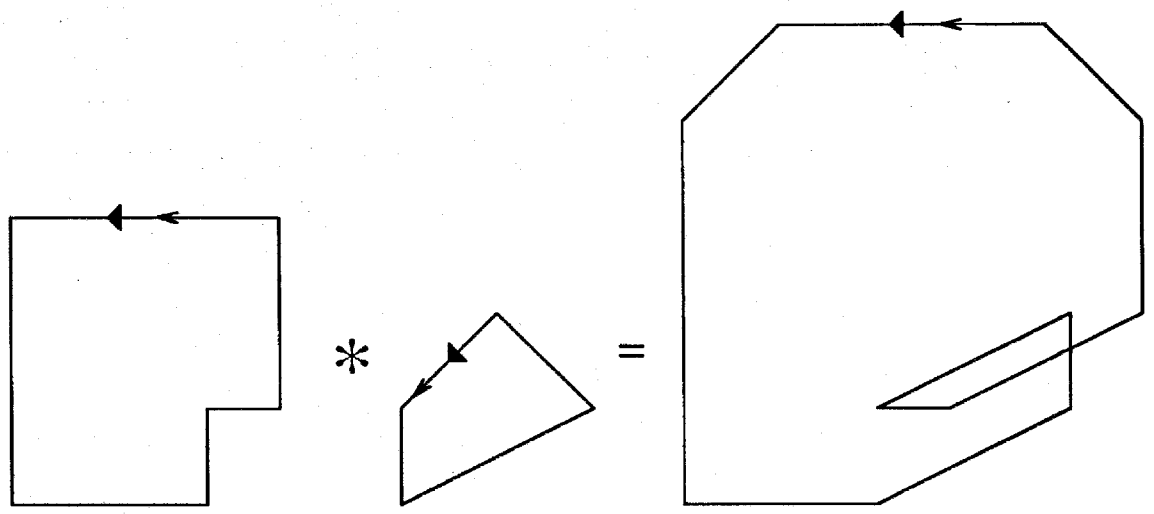


Figure 3-11: The convolution of two tracings. The origin of the coordinate system is at the lower left corner of all three tracings.

$$\tau^{\pm}_{A*B}(s) \;=\; \sum_{\substack{\dot{x}+\dot{y}=\dot{s} \\ \hat{x}=\hat{y}=\hat{s}}} \tau^{\pm}_{A}(x)\,\tau^{\pm}_{B}(y).$$

Lemma 3.7 *The convolution operation distributes over the addition operation. For any* $A, B, C \in T^2_P$,

$$A*(B+C) = (A*B)+(A*C) \text{ and } (B+C)*A = (B*A)+(C*A).$$

Proof.

$$\begin{aligned} A*(B+C) &= \{d \mid \hat{d}=\hat{a}=\hat{b},\ \dot{d}=\dot{a}+\dot{b},\ a\in A,\ b\in B \vee b\in C,\ a \text{ or } b \text{ is part of a turn}\} \\ &= \{d \mid (\hat{d}=\hat{a}=\hat{b},\ \dot{d}=\dot{a}+\dot{b},\ a \text{ or } b \text{ is part of a turn}) \\ &\qquad \vee(\hat{d}=\hat{a}=\hat{c},\ \dot{d}=\dot{a}+\dot{c},\ a \text{ or } c \text{ is part of a turn}), \\ &\qquad a\in A,\ b\in B,\ c\in C\} \\ &= (A*B)+(A*C). \end{aligned}$$

Since convolution is commutative, the second claim of the lemma is a direct result of the first. ■

It is fairly obvious from the above that if $A = \sum_i A_i$ and $B = \sum_j B_j$, where A_i and B_j are moves or turns and $\sum$ denotes the tracing addition of all the trips, then $A * B = \sum_{i,j} A_i * B_j$, *i.e.*, the convolution of two sums is equal to the sums of the convolutions. Therefore, the convolution of two tracings can be considered as the sum of all the combinations of two trips, one from each tracing, where one or both of the trips is a turn. The convolution operation was defined for both of these cases. The convolution of a left turn t_1 at point p_1 sweeping arc a_1 with another left turn t_2 at point p_2 sweeping arc a_2 is the left turn at point $p_1 + p_2$ sweeping arc $a_1 \cap a_2$. The convolution of a forward move $m = \langle xy \rangle$ with a left turn t at point p sweeping an arc which includes the orientation of m is a forward move from $x + p$ to $y + p$. If the arc of t does not contain the orientation of m, then the convolution is the null tracing.

Since the convolution of two states with multiplicities m and n respectively is a state with multiplicity mn, convolutions involving backward moves and right turns are also easily described. The convolution of two right turns is a left turn (because, since each state in a right turn has multiplicity -1, their convolution has multiplicity $-1 \cdot -1 = +1$). Similarly, the convolution of a right turn and a backward move is a forward move, the convolution of a right turn and a forward move is a backward move, and the convolution of a right turn and a left turn is a right turn.

In the sequel, convolutions of two tracings (which are, by definition, integral and balanced) will mainly be considered. Therefore, the following theorem is valuable.

Theorem 3.8 *The sum or convolution of two tracings is a tracing.*

Proof. It must be shown for any two tracings A and B that $A + B$ and $A * B$ are both integral and balanced. Since both A and B are integral and balanced, every state in A and B occurs with integer multiplicity and $\mu^+(s) - \mu^-(s) + \tau^+(s) - \tau^-(s) = 0$ for all s in A or B.

Consider $A + B$ first. The multiplicity of any state in $A + B$ is equal to the sum of that state's multiplicities in A and B. Since the multiplicities of any state in A and B must be integers, the multiplicity of the state in $A + B$ must be an integer. Therefore, $A + B$ is integral. Since $\mu^{\pm}(s)$ and $\tau^{\pm}(s)$ are just sums of states near s, it may be concluded that

$\mu^{\pm}_{A+B}(s) = \mu^{\pm}_A(s) + \mu^{\pm}_B(s)$ and $\tau^{\pm}_{A+B}(s) = \tau^{\pm}_A(s) + \tau^{\pm}_B(s)$. Therefore,

$$\begin{aligned}
&\mu^+_{A+B}(s) - \mu^-_{A+B}(s) + \tau^+_{A+B}(s) - \tau^-_{A+B}(s) \\
&\quad = \mu^+_A(s) - \mu^-_A(s) + \tau^+_A(s) - \tau^-_A(s) + \mu^+_B(s) - \mu^-_B(s) + \tau^+_B(s) - \tau^-_B(s) \\
&\quad = 0 + 0 \\
&\quad = 0.
\end{aligned}$$

Thus the sum of two tracings is a tracing.

Now consider the convolution of two tracings. Since the multiplicity of any state $s \in A*B$ is equal to the product of multiplicities of states in A and B, s must have integer multiplicity, so $A * B$ is integral.

Next check that $A * B$ is balanced. For any state $s \in A * B$,

$$\begin{aligned}
&\mu^+_{A*B}(s) - \mu^-_{A*B}(s) + \tau^+_{A*B}(s) - \tau^-_{A*B}(s) \\
&= \sum_{\substack{\dot{x}+\dot{y}=\dot{s} \\ \hat{x}=\hat{y}=\hat{s}}} \mu^+_A(x)\,\tau_B(y) + \mu^+_B(x)\,\tau_A(y) \;-\; \sum_{\substack{\dot{x}+\dot{y}=\dot{s} \\ \hat{x}=\hat{y}=\hat{s}}} \mu^-_A(x)\,\tau_B(y) + \mu^-_B(x)\,\tau_A(y) \\
&\quad + \sum_{\substack{\dot{x}+\dot{y}=\dot{s} \\ \hat{x}=\hat{y}=\hat{s}}} \tau^+_A(x)\,\tau^+_B(y) \;-\; \sum_{\substack{\dot{x}+\dot{y}=\dot{s} \\ \hat{x}=\hat{y}=\hat{s}}} \tau^-_A(x)\,\tau^-_B(y) \\
&= \sum_{\substack{\dot{x}+\dot{y}=\dot{s} \\ \hat{x}=\hat{y}=\hat{s}}} \mu^+_A(x)\,\tau_B(y) + \mu^+_B(y)\,\tau_A(x) \;-\; \sum_{\substack{\dot{x}+\dot{y}=\dot{s} \\ \hat{x}=\hat{y}=\hat{s}}} \mu^-_A(x)\,\tau_B(y) + \mu^-_B(y)\,\tau_A(x) \\
&\quad + \sum_{\substack{\dot{x}+\dot{y}=\dot{s} \\ \hat{x}=\hat{y}=\hat{s}}} \tau^+_A(x)\,\tau^+_B(y) \;-\; \sum_{\substack{\dot{x}+\dot{y}=\dot{s} \\ \hat{x}=\hat{y}=\hat{s}}} \tau^-_A(x)\,\tau^-_B(y) \\
&= \sum_{\substack{\dot{x}+\dot{y}=\dot{s} \\ \hat{x}=\hat{y}=\hat{s}}} \mu^+_A(x)\,\tau_B(y) + \mu^+_B(y)\,\tau_A(x) - \mu^-_A(x)\,\tau_B(y) \\
&\qquad + \mu^-_B(y)\,\tau_A(x) + \tau^+_A(x)\,\tau^+_B(y) - \tau^-_A(x)\,\tau^-_B(y).
\end{aligned}$$

At each x and y such that $\dot{x} + \dot{y} = \dot{s}$ and $\hat{x} = \hat{y} = \hat{x}$,

$$\mu^+_A(x)\,\tau_B(y) + \mu^+_B(y)\,\tau_A(x) - \mu^-_A(x)\,\tau_B(y) + \mu^-_B(y)\,\tau_A(x) + \tau^+_A(x)\,\tau^+_B(y) - \tau^-_A(x)\,\tau^-_B(y)$$

$$
\begin{aligned}
&= [\mu_A^+(x) - \mu_A^-(x) + \tau_A^+(x) - \tau_A^-(x)]\,\tau_B(y) \\
&\qquad + [\mu_B^+(y) - \mu_B^-(y)]\,\tau_A(x) + [\tau_B^+(y) - \tau_B(y)]\,\tau_A^+(x) + [\tau_B(y) - \tau_B^-(y)]\,\tau_A^-(x) \\
&= 0 \cdot \tau_B(y) + [\mu_B^+(y) - \mu_B^-(y)]\,\tau_A(x) + \frac{\tau_B^+(y) - \tau_B^-(y)}{2}\,\tau_A^+(x) + \frac{\tau_B^+(y) - \tau_B^-(y)}{2}\,\tau_A^-(x) \\
&= [\mu_B^+(y) - \mu_B^-(y)]\,\tau_A(x) + [\tau_B^+(y) - \tau_B^-(y)]\,\frac{\tau_A^+(x) + \tau_A^-(x)}{2} \\
&= [\mu_B^+(y) - \mu_B^-(y)]\,\tau_A(x) + [\tau_B^+(y) - \tau_B^-(y)]\,\tau_A(x) \\
&= [\mu_B^+(y) - \mu_B^-(y) + \tau_B^+(y) - \tau_B^-(y)]\,\tau_A(x) \\
&= 0 \cdot \tau_A(x) \\
&= 0.
\end{aligned}
$$

So

$$
\sum_{\substack{\dot{x}+\dot{y}=\dot{s} \\ \hat{x}=\hat{y}=\hat{s}}} \mu_A^+(x)\tau_B(y) + \mu_B^+(y)\tau_A(x) - \mu_A^-(x)\tau_B(y) + \mu_B^-(y)\tau_A(x) + \tau_A^+(x)\tau_B^+(y) - \tau_A^-(x)\tau_B^-(y)
$$

is equal to zero because each of the terms of the sum is equal to zero. Therefore,

$$
\mu_{A*B}^+(s) - \mu_{A*B}^-(s) + \tau_{A*B}^+(s) - \tau_{A*B}^-(s) = 0
$$

for all $s \in A * B$. So the convolution of two tracings is balanced. Thus the convolution of two tracings is a tracing. ■

Note that tracing A can be translated by vector x by taking the convolution $A * X$, where X is a tracing at point $\dot{x}$ which is a 360° left turn. This translation of A by x is denoted by $(A)_x$. It may be shown that for any two vectors x and y and any two tracings A and B, $(A)_x * (B)_y = (A)_y * (B)_x = (A * B)_{x+y}$.

Any tracing B may be considered as the sum of its moves and turns ($B = \sum_j B_j$, where B_j is a move or a turn). A move or turn B_j in tracing B can be considered as the sum of moves or turns between neighboring states in B_j: $B_j = \sum_k b_k$, where b_k is a move or turn between two adjoining states. The incremental moves are equal to the set $\{c \mid c = \lim_{\dot{x}-\dot{y}\to 0}\langle xy\rangle,\ x, y \in B_j\}$. The incremental turns are equal to the set $\{d \mid d = \lim_{\hat{x}-\hat{y}\to 0}\langle xy\rangle,\ x, y \in B_j\}$. But the limit of the trip in this definition is just a sin-

gle state. Therefore, a tracing B may be considered as the sum of its states. Thus the convolution of two tracings A and B can be written as

$$A * B = A * \sum_{s \in B} s = \sum_{s \in A} s * B.$$

That is, the convolution of two tracings is equal to the sum of the convolutions of one tracing with every state of the other. Furthermore,

$$A * B = \sum_{r \in A,\, s \in B} r * s.$$

So the convolution of two tracings is just the sum of the convolutions of the states of the tracings.

The convolution operation has a close link to the Minkowski sum operation described, among other places, in [49]. The Minkowski sum of two polygons is simply the set of points which is obtained by the vector addition of each point on one of the polygons with each point in the other. Note the similarity of this definition to the equation immediately above. As pointed out by Ghosh in [43], the Minkowski sum of two non-self-intersecting polygons $\mathcal{P}$ and $\mathcal{Q}$, written $\mathcal{P} \oplus \mathcal{Q}$, can be found by constructing the counterclockwise simple polygonal tracings of the polygons and finding their convolution. In the above notation:

$$\mathcal{P} \oplus \mathcal{Q} = \{y \mid y = \dot{x},\, x \in T^2,\, \omega_{\mathbf{T}(\mathcal{P}) * \mathbf{T}(\mathcal{Q})}(x) > 0\}.$$

See Figure 3-12 for a Minkowski sum example.

Similarly, the Minkowski difference of polygons $\mathcal{P}$ and $\mathcal{Q}$, written $\mathcal{P} \ominus \mathcal{Q}$, is defined to be the intersection of the translations of $\mathcal{P}$ by each point in $\mathcal{Q}$ and can be found by using the convolution operation:

$$\mathcal{P} \ominus \mathcal{Q} = \{y \mid y = \dot{x},\, x \in T^2,\, \omega_{\mathbf{T}(\mathcal{P}) * \mathbf{T}'(\mathcal{Q})}(x) \geq 1\},$$

where $\mathbf{T}'(\mathcal{Q})$ is the counterclockwise tracing of $\mathcal{Q}$ with all backward moves. See Figure 3-13 for a Minkowski difference example. Note that this conforms with the definition of Minkowski subtraction used by Serra in [108], and not that used by Ghosh, Haralick, *et al.* [43, 49]

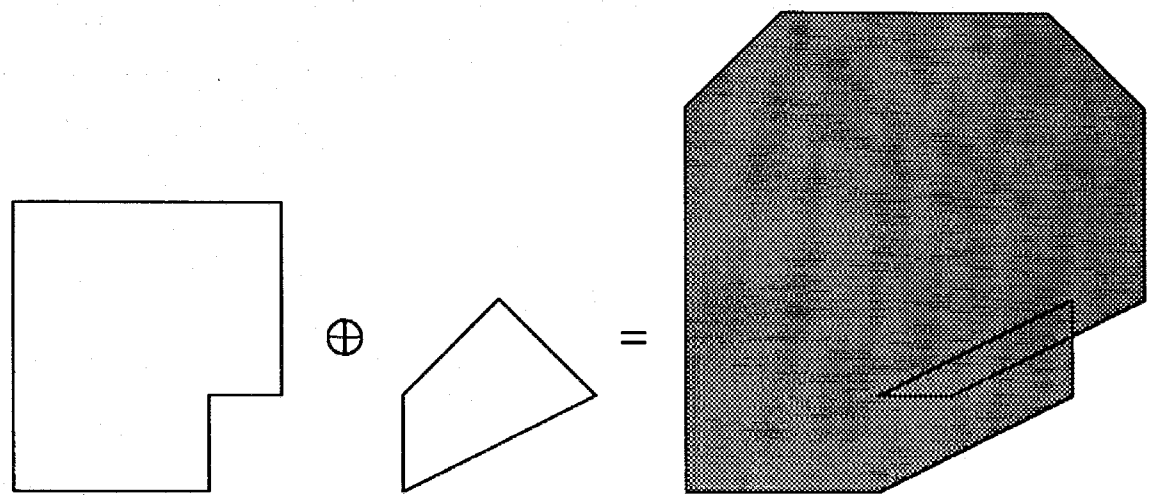


Figure 3-12: The Minkowski sum of two polygons, distinguished by the shaded area, is the region of the convolution of the polygons' tracings which has winding number greater than zero. The moves of the convolution are shown in black. The origin of the coordinate system is at the lower left corner of all three polygons.

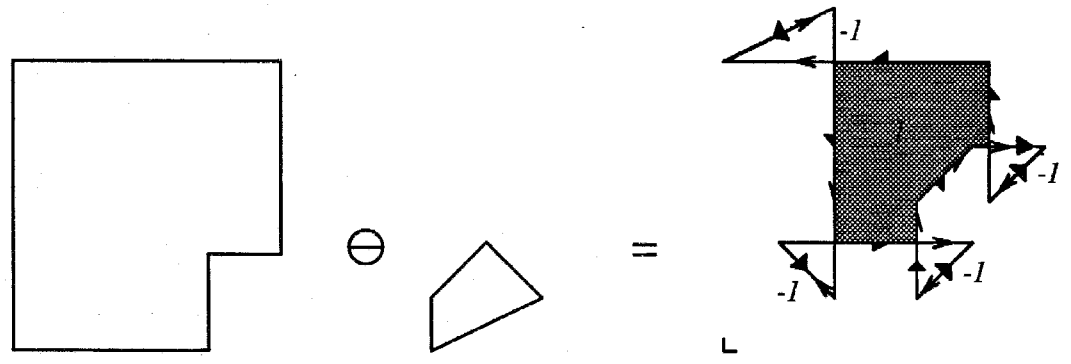


Figure 3-13: The Minkowski difference of two polygons, distinguished by the shaded area, is the region of the convolution of the tracing of the first polygon with the inverse of the tracing of the second which has winding number greater than or equal to one. The winding number of each region is shown. The origin of the coordinate system is at the lower left corner of all three polygons.

(although the definition could be easily modified to conform with that in the latter articles).

Lemma 3.9 *The convolution of tracings is associative,* i.e., *for any tracings* $A, B, C \in T^2$,

$$A * (B * C) = (A * B) * C.$$

Proof.

$$\begin{aligned}
A * (B * C) &= A * \{d \mid \hat{d} = \hat{b} = \hat{c},\ \dot{d} = \dot{b} + \dot{c},\ b \in B,\ c \in C,\ b \text{ or } c \text{ is part of a turn}\} \\
&= \{d \mid \hat{d} = \hat{a} = \hat{b} = \hat{c},\ \dot{d} = \dot{a} + (\dot{b} + \dot{c}),\ a \in A,\ b \in B,\ c \in C, \\
&\qquad a \text{ or } b \text{ or } c \text{ is part of a turn}\} \\
&= \{d \mid \hat{d} = \hat{a} = \hat{b},\ \dot{d} = \dot{a} + \dot{b},\ a \in A,\ b \in B,\ a \text{ or } b \text{ is part of a turn}\} * C \\
&= (A * B) * C.
\end{aligned}$$

■

As discussed above, a tracing convoluted with a tracing at a point which sweeps out a 360° arc just serves to translate the first tracing by the position of the point. Therefore, the identity under convolution is the tracing at the origin which traces out a 360° arc, which will be denoted $\langle\langle 0 \rangle\rangle$. Given all the above information, the conclusion in the following theorem may be reached.

Theorem 3.10 *Tracings,* T^2, *along with the convolution and addition operations, form a commutative ring with identity.*

Proof. Lemma 3.6 proved that tracings form an abelian group under the addition operation with identity equal to the empty shape. Lemma 3.9 proved that the convolution operation is associative. Lemma 3.7 proved that the convolution operation distributes over the shape addition operation. The convolution operation has been demonstrated to be commutative and to possess an identity, $\langle\langle 0 \rangle\rangle$. Therefore, tracings, along with the convolution and addition operations, form a commutative ring with identity. ■

3.3.4 Discussion

The ring structure of tracings under addition and convolution has some important consequences. Most obviously, any number of tracings combined using the two operations is also a tracing. For example, consider a shape grammar which treats shapes as polygonal tracings. If the rules of the grammar use any combination of tracing addition and convolution (which, as can be seen from the operations derived from these two, allows many commonly-used shape operations and some very expressive but not-so-commonly-used ones), then all shapes generated by the grammar will also be tracings.

Furthermore, some elementary algebra can be done on the ring of shapes under tracing addition and convolution. Consider a situation in which one wishes to sum or convolute two shapes to obtain a third. If the desired shape and one of the two precursor shapes are known (where both of the known shapes are arbitrary summations and/or convolutions of other shapes), then the other precursor shape can be uniquely determined. This ability to "work backward," or to subtract or divide shapes, is only possible where the shapes have the structure of a ring.

The discussion in the previous section of the relations between operations on tracings and associated operations on two-dimensional polygons raises the question of whether the group-theoretic properties derived are valid for shapes in dimensions higher than two. Consider the basic operations on shapes: union, intersection, complement, symmetric difference and Minkowski sum and difference. As discussed in Section 3.2, shapes form a Boolean ring under the first four operations cited. This property holds true no matter what the dimension of the shapes. The use of Minkowski sum and difference operations in spaces of arbitrary dimension is discussed extensively by Matheron and Serra in [76, 108]. Both operations produce valid shapes, though the operations might produce shapes which are not closed and compact. The Minkowski sum and difference of arbitrary shapes with a generalized sphere are discussed by Rossignac and Requicha in [106]. The authors present algorithms which calculate a superset of the boundary of the Minkowski sum or difference of a shape and a generalized sphere, but they do not concern themselves with finding the exact boundary because they are concerned with the graphical representation of shapes rather than the shapes' properties. The elements of the superset which are not elements of the actual boundary of the shape are all inside the boundary, so the superset may be displayed on a

computer using z-buffering and only the actual boundary of the shape will be displayed.

Given the above work, it may be concluded only that it is possible to use the various shape operations in three-dimensional spaces. The ring structure, however, is not present in any of the above formalisms because arbitrary subtraction and division types of operations are not supported. Nonetheless, both tracings and the convolution operation might be extended to three dimensions by extending the mapping **T** and its inverse. This extension needs further investigation.

3.4 Shapes and Vector Spaces

In this section, operations on the set of monostrofic tracings (tracings, defined formally below, which are a generalization of the set of convex tracings) are discussed. These tracings, along with the convolution operation introduced in the previous section, will be shown to form a vector space over the real numbers, where multiplication of a tracing by a real number can be thought of as scaling.

3.4.1 From Multiplication of Tracings to Addition of Monostrofic Tracings

In Section 3.3, the union and convolution of tracings were used as "addition" and "multiplication" operators, respectively, to form a ring of shapes. In this section, the convolution operation will be used differently. Rather than using it as a multiplication of shapes, it will now be considered an additive operation. The domain under consideration will also be restricted to monostrofic tracings, defined below. Furthermore, different definitions for the negative of a tracing and the multiplication of a tracing by a scalar will be given.

Definition 3.12 (Monostrofic Tracing) A tracing is *monostrofic* if it takes on every orientation exactly once.

The set of monostrofic tracings will be denoted by M. Note that the moves in a monostrofic tracing are ordered by their orientations and that every turn is in the same direction. Also, every polygonal tracing which is non-self-intersecting and convex and in which all states have multiplicity one (*e.g.*, the tracings which are maps of convex polygons) are monostrofic. In

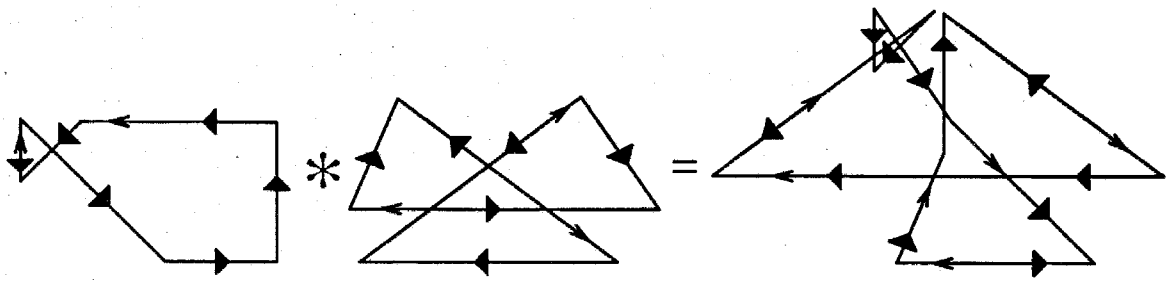


Figure 3-14: The convolution of two left-turning monostrofic tracings.

what follows, only the set of left-turning monostrofic tracings will be considered. The same results could be obtained for the complementary set of right-turning monostrofic tracings.

Lemma 3.11 *The convolution of two monostrofic left-turning tracings is a monostrofic left-turning tracing.*

Proof. Since all turns in the two original tracings are to the left, all turns in their convolution are also to the left (since the convolution of two turns is a turn and the convolution of a move and a turn is a move). Furthermore, since each of the two monostrofic tracings has turns that span a single left turn of 360 degrees and the turns in a convolution are just intersections of the arcs of turns in the two tracings, the turns in the convolution span a single left turn of 360 degrees. Therefore, states in the convolution may only have the same orientation if they are on the same move. These facts imply that the convolution is monostrofic. ■

The convolution of two left-turning monostrofic tracings is shown in Figure 3-14. If $\mathcal{P}$ and $\mathcal{Q}$ are convex polygons, then the corresponding counterclockwise tracings $\mathbf{T}(\mathcal{P})$ and $\mathbf{T}(\mathcal{Q})$ have only left turns and forward moves, so they are monostrofic. Their convolution is also monostrofic, and since the convolution of a left turn with a forward move is a forward move, all moves in the convolution are also forward. Since the moves in the convolution are ordered by their orientations, the convolution must also be convex (because if it was self-intersecting, then its turns would have to span two complete counterclockwise rotations instead of one; and if it had a concave corner it would have to have a right turn). The convolution can be mapped back into a convex polygon, this one equal to the Minkowski sum of the original two polygons. See Figure 3-15 for an example.

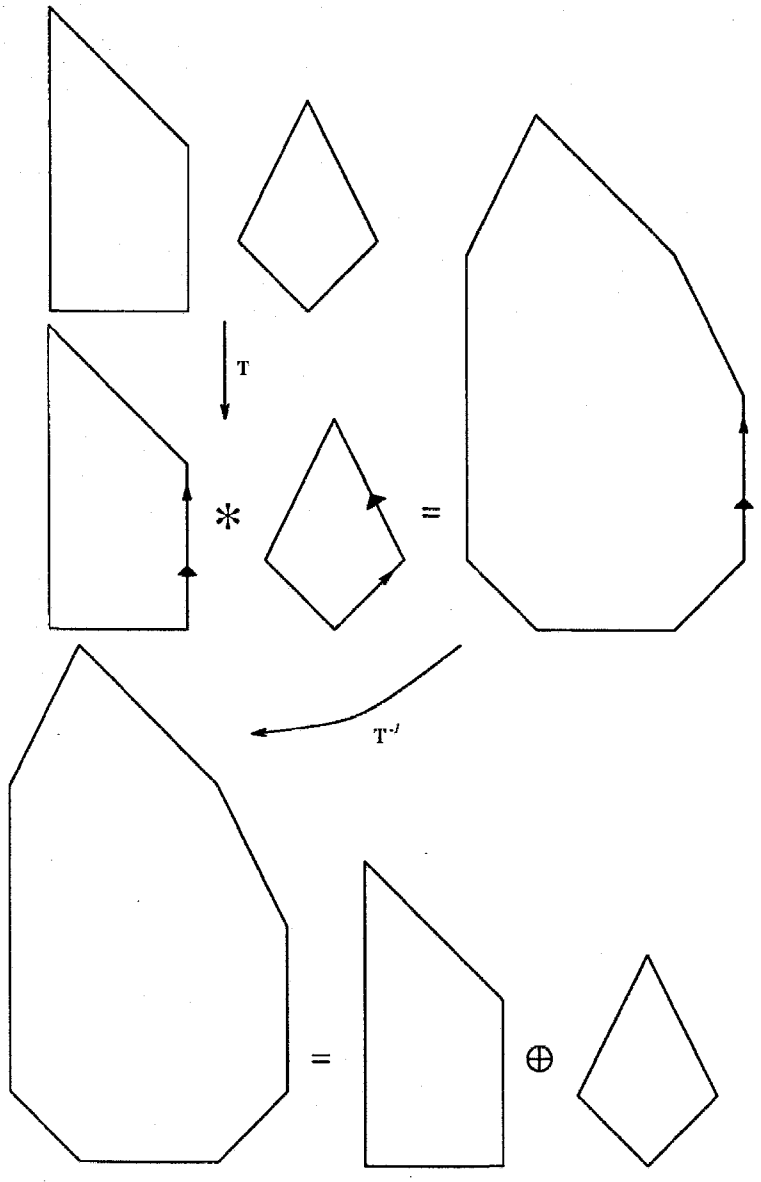


Figure 3-15: Two convex polygons, mapped into convex monostrofic tracings, convoluted, then mapped back to a convex polygon. The resulting polygon is equal to the Minkowski sum of the original two polygons.

An even more powerful property of monostrofic tracings which relates to convex regions of the tracings was stated by Guibas *et al.* in [48] and is proved below.

Lemma 3.12 *Monostrofic left-turning tracings have no region of winding number greater than one. Furthermore, they have at most one region with winding number equal to one, and that region is convex, bounded, and either to the left (in the orientation sense) of all sides enclosing it or to the right of all the sides.*

Proof. If a monostrofic tracing had a region with winding number greater than one, then it would encircle a region two or more times in a counterclockwise direction. But since the sum of all the turns in a monostrofic tracing is only a 360 degree left turn, the greatest amount of clockwise turning possible is a single rotation. Therefore, it is impossible for a region of a monostrofic tracing to have winding number greater than one.

Similarly, if there were two or more regions with winding number equal to one, each would have to be encircled in a counterclockwise direction by a partial tracing containing only left turns. Again, in order to completely encircle a region the tracing must go through a complete 360 degree left turn. However, the tracing can only go through a single turn, since it has exactly 360 degrees of left turns. Therefore, it is impossible for there to be more than one region with winding number equal to one.

If a region of winding number equal to one exists in a monostrofic tracing, then it must be bounded because there must be moves which enclose it, since the definition of the winding number of a state is the number of counterclockwise turns the tracing makes about the state. All sides of the region are moves, and their directions must be such that they form a counterclockwise encirclement of the region. Each corner of the region is either a turn or an intersection of two moves.

In order to show that a region of winding number one is convex and either to the left or right of all sides bounding it, the combinations of edges and corners bounding the region must be considered. If the region has a concave corner, then it corresponds to a turn because every corner of the region corresponding to the intersection of two moves (not a turn) must measure less than 180 degrees (see Figure 3-16 for an illustration). One of the edges which is an extremal state of the turn at a concave corner must be a forward move and one must be a backward move (because if both were in the same direction, the turn would have to be to the right). At a convex corner of the boundary defined by a turn, the two moves which

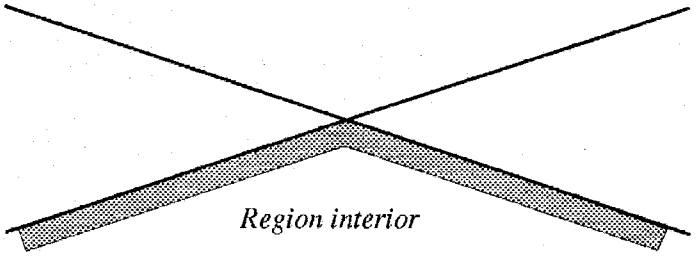


Figure 3-16: If a corner of a region is defined by the intersection of two moves (without a turn between them), then that corner must be convex.

are extremal states of the turn must be in the same direction, since the turn between them must be less than 180 degrees.

Now consider a corner of the boundary defined by the intersection of two moves (not by a turn). As discussed above, this corner is convex. Consider the move toward this corner and its relation to the move after the corner. It is known that the direction of the move after the corner is away from the corner, but its orientation must be determined in order to know if the orientation may change from forward to backward (or vice versa) at this sort of corner. The move after the corner comes from the right of the move before the corner (with respect to direction, not necessarily orientation). Therefore, there must be at least one move of orientation opposite the first move between the two moves, since the only way to move to the right with only left turns is to switch orientation. However, if the move after the corner is oriented opposite to the move before the corner, then there must be more than 180 degrees of left turning between them because a left turn of less than 180 degrees is needed just to orient the move after the corner in the same direction as the move before. Therefore, the two moves must be in the same direction, either forward or backward.

It has been shown that the orientation of the tracings around a region of winding number one must change at a corner if and only if the region has a concave turn at that corner. Therefore, there must be an even number of concave turns, so that the orientations of the tracings around the region match up. However, if there is one or more pair of concave turns in the boundary of the counterclockwise-traversed region, the same orientation must be taken more than once because any pair of turns through concave corners leaves the orientation further to the left while making no headway in the trip around the interior

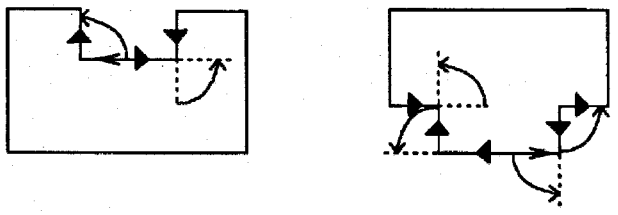


Figure 3-17: Two examples of turns through concave corners. In the first example, the concave turns are consecutive, while they are separated in the second example.

of the region. See Figure 3-17 for an illustration. Since monostrofic tracings are being considered, two states on separate moves must not have the same orientation. Therefore, the region of winding number one can not have any concave corners. Furthermore, the orientations of the moves around the region must be all forward or all backward, since orientation could only change at a concave corner. Thus the region is either to the left (if the moves are forward) or to the right (if the moves are backward) of all the sides. ■

Now a new definition for the negative of a tracing will be given. Recall that in the previous section the negative was defined to be the tracing with the same orientations but traversed in the opposite direction. The negative will now be given a different meaning.

Definition 3.13 (Negative of a Monostrofic Tracing) The *negative* of monostrofic tracing A, denoted $-A$, is the monostrofic tracing in which all points in A are reflected through the origin. In other words,

$$-A = \{b \mid \hat{b} = \hat{a},\ \dot{b} = -\dot{a},\ a \in A\}.$$

See Figure 3-18 for an example. It can be shown that the negative of a monostrofic tracing is the tracing's inverse under convolution.

Lemma 3.13 *For all left-turning monostrofic tracings A,*

$$A * (-A) = -A * A = \langle\langle 0 \rangle\rangle.$$

Proof. There is one-to-one correspondence between the turns of A and $-A$; they each are left turns through the same angle and located equidistant in opposite directions from the

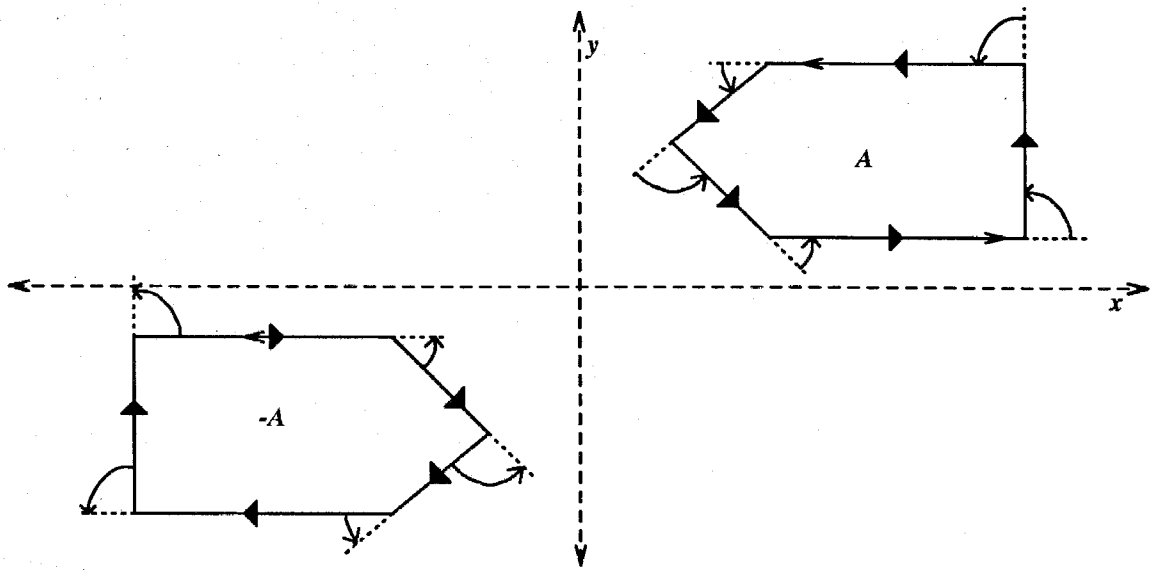


Figure 3-18: A monostrofic tracing and its negative.

origin. Therefore, the convolution of the turns is the collection of all the turns at the origin, or $\langle\langle 0 \rangle\rangle$.

Each move in A convolutes with the two turns located at the end points of its image in $-A$. Note that this convolution produces two moves of multiplicity $\pm 1/2$ with the same orientation, collinear and touchinging at the origin. Similarly, convoluting moves in $-A$ with turns in A produces two moves of multiplicity $\mp 1/2$ with the same orientation, collinear and touching at the origin. The moves all cancel out since their multiplicities are negatives of each other. Thus all that remains is the identity tracing $\langle\langle 0 \rangle\rangle$. ■

Given the results proved above, it is a simple step to deduce that these tracings have a group-theoretic structure.

Lemma 3.14 *Monostrofic, left-turning tracings, along with the convolution operation, form an abelian group.*

Proof. In Lemma 3.11, it was shown that left-turning monostrofic tracings are closed under convolution. The convolution of tracings was proven to be associative for all tracings (not just monostrofic ones) in Lemma 3.9. It is obvious that the tracing $\langle\langle 0 \rangle\rangle$ is still the identity, and it was shown in Lemma 3.13 that each left-turning, monostrofic tracing has a two-sided inverse. Finally, the definition of the convolution operation implies that it is commutative.

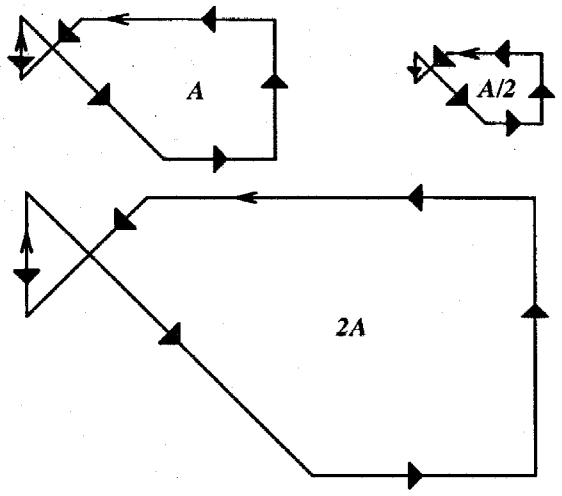


Figure 3-19: A monostrofic tracing scaled by factors of 1/2 and 2.

From all these facts, it can be concluded that the left-turning, monostrofic tracings form an abelian group. ■

3.4.2 Scaling of Convex Shapes

Now a scaling operation on monostrofic tracings may be defined.

Definition 3.14 (Scaling Operation) The *scaling operation*, or scaling, denoted $\cdot$, is an operation on real numbers and left-turning monostrofic tracings

$$\begin{aligned} \cdot : \mathbb{R} \times M &\rightarrow M \\ (r, A) &\mapsto r \cdot A = \{b \mid \hat{b} = \hat{a},\ \dot{b} = r \cdot \dot{a},\ a \in A\}, \end{aligned}$$

where $r \cdot \dot{a}$ denotes the multiplication of both components of $\dot{a} \in \mathbb{R}^2$ by the real number r.

Where there is no ambiguity, the scaling of tracing A by real number r will be denoted by rA for simplicity of notation. Recall from Definition 3.13 that the negative of a monostrofic tracing is obtained by reflecting each state in the tracing through the origin. The scaling operation defined here simply extends the definition to scaling factors other than -1. See Figure 3-19 for some examples.

It turns out that with the convolution and scaling operations, monostrofic tracings have a very interesting property.

Theorem 3.15 *The abelian group of monostrofic, left-turning tracings under the convolution operation, along with the scaling operation, forms a vector space over the ring of real numbers.*

Proof. Consider left-turning, monostrofic tracings A and B and real numbers r and s. First, it must be shown that scaling distributes over convolution:

$$\begin{aligned}
r(A * B) &= \{d \mid \hat{d} = \hat{e},\ \dot{d} = r \cdot \dot{e},\ e \in A * B\} \\
&= \{d \mid \hat{d} = \hat{e},\ \dot{d} = r \cdot \dot{e},\ e \in \{c \mid \hat{c} = \hat{a} = \hat{b},\ \dot{c} = \dot{a} + \dot{b},\ a \in A,\ b \in B, \\
&\qquad a \text{ or } b \text{ is part of a turn}\}\} \\
&= \{c \mid \hat{c} = \hat{a} = \hat{b},\ \dot{c} = r \cdot (\dot{a} + \dot{b}),\ a \in A,\ b \in B,\ a \text{ or } b \text{ is part of a turn}\} \\
&= \{c \mid \hat{c} = \hat{a} = \hat{b},\ \dot{c} = r \cdot \dot{a} + r \cdot \dot{b},\ a \in A,\ b \in B,\ a \text{ or } b \text{ is part of a turn}\} \\
&= \{c \mid \hat{c} = \hat{y} = \hat{z},\ \dot{c} = \dot{y} + \dot{z},\ y \in rA,\ z \in rB,\ y \text{ or } z \text{ is part of a turn}\} \\
&= rA * rB.
\end{aligned}$$

Next, the convolution of two scalings of a tracing must be shown to be equal to the tracing scaled by the sum of the two scaling factors:

$$\begin{aligned}
rA * sA &= \{c \mid \hat{c} = \hat{y} = \hat{z},\ \dot{c} = \dot{y} + \dot{z},\ y \in rA,\ z \in sA,\ y \text{ or } z \text{ is part of a turn}\} \\
&= \{c \mid \hat{c} = \hat{a} = \hat{b},\ \dot{c} = r \cdot \dot{a} + s \cdot \dot{b},\ a \in A,\ b \in A,\ a \text{ or } b \text{ is part of a turn}\}.
\end{aligned}$$

If states a and b are both parts of turns and $\hat{a} = \hat{b}$, then they must occupy the same position $\dot{a}$ (because a monostrofic tracing takes on each orientation exactly once). Therefore, the convolution of the two states has position $r \cdot \dot{a} + s \cdot \dot{a} = (r + s) \cdot \dot{a}$. If only one of the states is part of a turn, then the other state is part of a move into or out of the corresponding turn in its tracing. Consider a move $\langle xy \rangle$ and its possible convolutions with the turn ending at state x and the turn beginning at y. Note that each convolution will produce a move with

multiplicity 1/2 (if turns are to the left and the move is forward) because the states at the ends of the turns only have multiplicity 1/2. There are four possible convolutions:

1. $r \cdot x * s\langle xy \rangle$,

2. $r \cdot y * s\langle xy \rangle$,

3. $s \cdot x * r\langle xy \rangle$,

4. $s \cdot y * r\langle xy \rangle$.

The first convolution produces a move (of multiplicity 1/2) from $(r+s)\cdot x$ to $r\cdot x + s\cdot y$. The fourth convolution produces a move from $r\cdot x + s\cdot y$ to $(r+s)\cdot y$, so their sum is a single move of multiplicity 1/2 from $(r+s)\cdot x$ to $(r+s)\cdot y$. The second and third convolutions produce similar moves, so the resulting sum of all four convolutions (and the only move with this orientation) is a move of multiplicity one from $(r+s)\cdot x$ to $(r+s)\cdot y$. If this process is completed for all states, the set of states left is $\{d \mid \hat{d} = \hat{a},\ \dot{d} = (r+s)\cdot \dot{a},\ a \in A\}$, *i.e.*, $(r+s)A$. Therefore, $rA * sA = (r+s)A$.

It is simple to show that scaling factors multiply:

$$\begin{aligned} r(sA) &= \{b \mid \hat{b} = \hat{a},\ \dot{b} = r \cdot \dot{a},\ a \in sA\} \\ &= \{b \mid \hat{b} = \hat{a},\ \dot{b} = (rs) \cdot \dot{a},\ a \in A\} \\ &= (rs)A. \end{aligned}$$

With the above facts proved, it may be concluded that left-turning, monostrofic tracings under convolution form a module over the real numbers. But the real numbers have multiplicative identity 1 and form a division ring (because every number except zero has a multiplicative inverse). Therefore, left-turning, monostrofic tracings under convolution form a vector space. ■

The above theorem is an important result because, as discussed below, it enables a designer to work with monostrofic tracings in a way very similar to the way he or she works with vectors in Euclidean space. This ability will offer new freedom in working with shape and form in a parametric domain.

3.4.3 Consequences of the Vector Space Structure

Since monostrofic tracings form a vector space under convolution and scaling, they have an underlying structure which can be exploited. For example, consider the convolution of two monostrofic tracings. If the result is convoluted with one of the original tracings, the resulting tracing will be equal to the convolution of that original tracing, scaled by a factor of two, with the other tracing. This example demonstrates the vector-like qualities of these tracings: two of them can be scaled and convoluted to produce a whole range of tracings. All the resulting tracings have moves with orientations the same as in the original two tracings; the only difference is in the relative lengths of the moves from the respective original tracings. Furthermore, the original two tracings form a basis for a subspace which contains all of the resulting tracings (since all resulting tracings, and the convolution of any of these tracings, can be obtained from the convolution of the original two tracings, scaled). Most of these properties were touched on by Ghosh in [43], but the formal structure which enables their use was unexplored in that paper. See Figure 3-20 for an example.

These properties are exploited for computer graphics use by Kaul and Rossignac in [59]. There, the authors linearly interpolate between two convex two-dimensional polygons $\mathcal{P}$ and $\mathcal{Q}$ by taking the scaled Minkowski sum $t \cdot \mathcal{P} \oplus (1-t) \cdot \mathcal{Q}$. In Section 3.3.3, it was shown that the Minkowski sum of two polygons is the set of points in the convolution of their respective tracings with winding number greater than zero. It was also proved in Section 3.4.1 that the convolution of two tracings corresponding to convex polygons is a convex tracing. Thus the Minkowski sum of the two convex polygons corresponds to the points contained within the convolution of their corresponding tracings. The interpolating tracing is simply $t \cdot \mathbf{T}(\mathcal{P}) * (1-t) \cdot \mathbf{T}(\mathcal{Q})$. This family corresponds to the dotted line between A and B in the example in Figure 3-20. As discussed above, this family of tracings has moves with orientations corresponding to those in the original two tracings. They are also members of the subspace spanned by the two tracings $\mathcal{P}$ and $\mathcal{Q}$. Since they are members of the subspace, they vary continuously with t, so the interpolation is indeed a smooth one. While the authors of the original article relied on examples to demonstrate that the interpolation is smooth, by using the vector space property of monostrofic tracings, it can be proven to be smooth.

Kaul and Rossignac move on to discuss interpolations between three-dimensional poly-

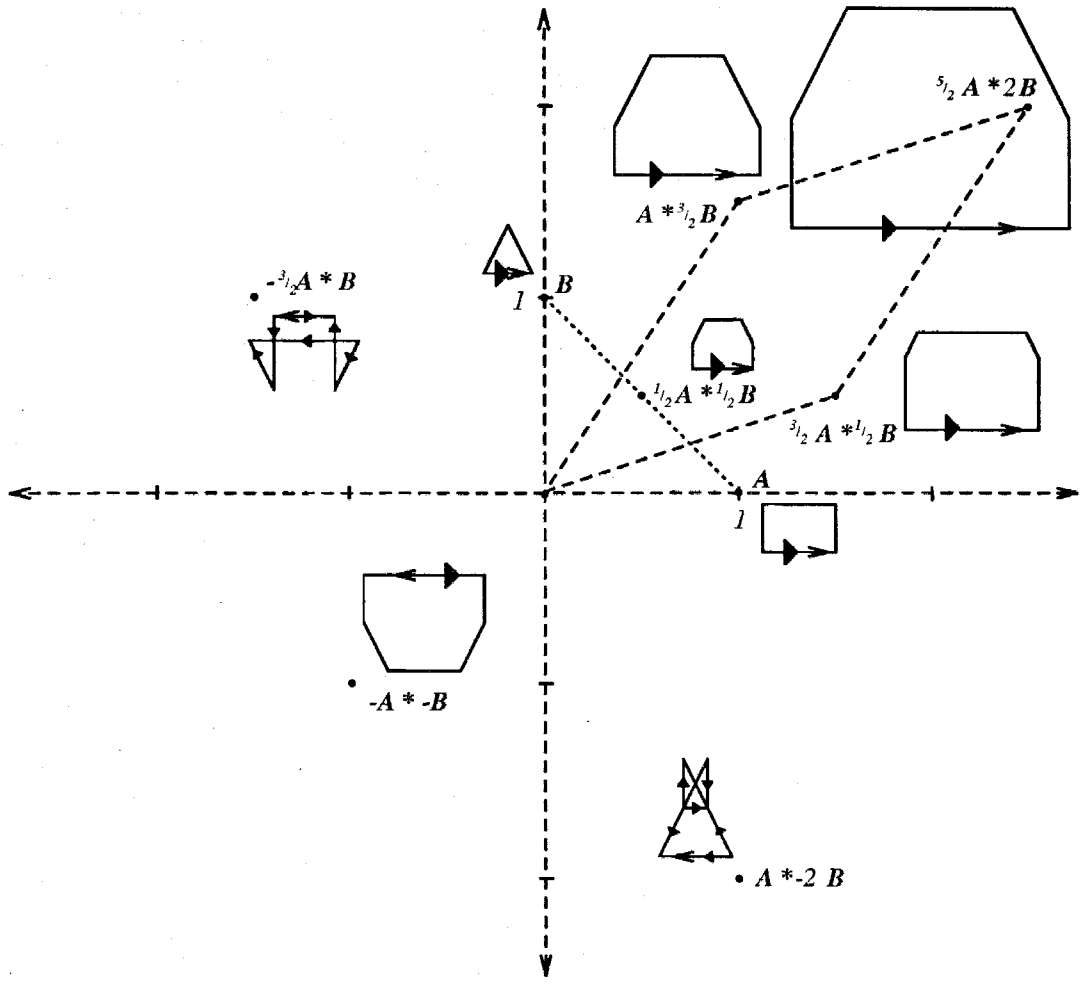


Figure 3-20: Scalings and convolutions of two convex tracings.

topes and between polytopes which have concave corners. Two-dimensional polygons with concave corners can not be represented by monostrofic tracings, as discussed in the proof to Lemma 3.12. Therefore, they lack a vector space structure, though they can be represented by tracings and so have the structure of a commutative ring, as discussed in Section 3.3. Consequently, the ability to scale by any real number is lost because for tracing A which has a concave corner, $A * (-A) \neq \langle\langle 0 \rangle\rangle$. However, positive scalings produce no difficulties, so two tracings can still be interpolated, even if one or both is concave. The interpolated tracing may be self-intersecting, have regions of winding number greater than one, and be expressible as a polygon through a mapping akin to $\mathbf{T}^{-1}$ only in the crudest way: by taking all points contained inside or on the tracing (a method which is equivalent to taking the Minkowski sum of the polygons). Nonetheless, this crude mapping does allow interpolation between polygons which may be concave.

Interpolating between three-dimensional solids using the convolution operation would require extending the formalism of tracings to include planar areas. Instead, consider the Minkowski sum of two polyhedra. As for concave two-dimensional polygons, the Minkowski sum does not admit much group-theoretic structure. Nonetheless, it should be obvious by extension of the two-dimensional results that the Minkowski sum of two convex solids is itself a convex solid. Two solids can even be interpolated using the Minkowski sum operation, though once again the interpolation may, for some or all values of the interpolating variable, be self-intersecting. The authors of [59] get around this problem by dropping the interior edges through z-buffering.

Most monostrofic tracings with four or more moves can be decomposed into the convolution of two monostrofic tracings. This decomposition is not unique, and each separate decomposition (in which the two tracings are not scalings of each other or of a pair of other tracings which convolute to the original tracing) forms another basis for a subspace of shapes, of which the original shape is a member. Homomorphisms can be defined between such subspaces (or the full space of monostrofic tracings) and other vector spaces. It can be proven that some other sets and their operations form vector spaces by finding bijections (mappings which are one-to-one and onto) between the sets and a vector space. For example, consider the set of positive and negative convex polygons along with the scaling and Minkowski sum operations. It has already been shown that the operation $\mathbf{T}$ maps such

polygons into convex tracings. Furthermore, for any convex tracing, there is a unique convex polygon which can be obtained by the mapping $\mathbf{T}^{-1}$. Thus there is a bijection between the two sets and so convex polygons form a vector space. It can be shown that for any two such polygons $\mathcal{P}$ and $\mathcal{Q}$, $\mathbf{T}(\mathcal{P}) * \mathbf{T}(\mathcal{Q}) = \mathbf{T}(\mathcal{P} \oplus \mathcal{Q})$, so in this case the Minkowski sum and convolution operations correspond, as was shown in Figure 3-15. It can be concluded that the operations used on convex polygons by Kaul and Rossignac are allowable because they form a vector space themselves (and thus no translation into the realm of tracings is actually necessary).

The vector space properties of monostrofic tracings are also useful in making rule sets for grammars. For example, assume a designer wants a shape grammar whose language contains only convex shapes. He or she can choose the nonterminal elements of the grammar to be monostrofic, left-turning tracings. The rules for the grammar can contain any combination of scaling and convolution of the tracings. The final rule applied in any derivation using the grammar extracts the convex polygon using the mapping $\mathbf{T}^{-1}$. A grammar like this is assured of obtaining only convex shapes in all its derivations.

Furthermore, if a designer wishes to derive a specific polygon or type of polygon, he or she can attempt to work backward, determining what rules and transformations must be applied to the starting tracing to produce the final shape. Although he or she is not guaranteed to find a derivation for the desired shape, he or she can apply the "inverse" of any rule to a tracing in order to find out what it might have looked like at an earlier stage of its derivation (if it is indeed derivable by the grammar). This reversability property of the rules is only available when dealing with grammar elements which have the structure of a group, ring, or vector space (*e.g.*, grammars with elements discussed in this and the preceding two sections). If a class of shapes can not be proven to have at least the structure of a group (especially that every member of the class has an inverse), then the inverse of all the rules may not be defined. The group-theoretic properties discussed above thus allow the designer much more freedom in working with shape grammars: he or she is able to "work from both ends" toward a goal, applying the rules of the grammar to the starting shape and applying the inverses of the rules to a desired shape.

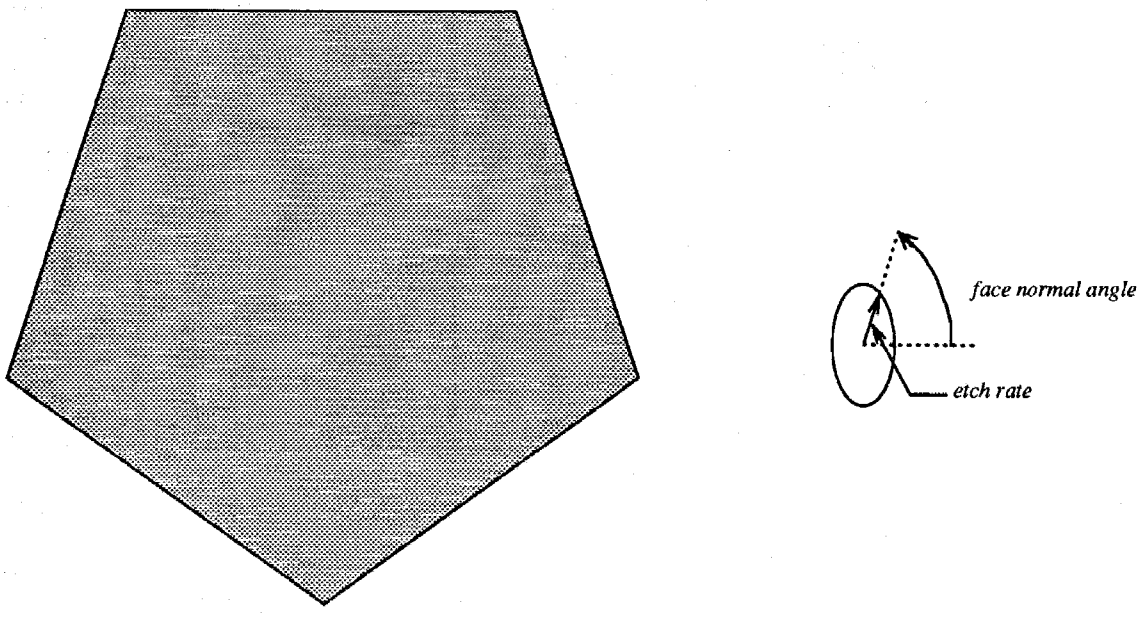


Figure 3-21: A feature to be etched and an etch rate diagram.

3.5 Consequences of the Algebraic Structures of Shape Operations

As an illustration of the utility of the group-theoretic properties of shapes and shape operations, consider the chemical etching of features on semiconductor chips. Features are typically cut on semiconductor chips using photolithography. The chips are then exposed to certain acids, which change the features by etching them. Most acids etch semiconductor material at different speeds in different directions, eating away material more quickly along some crystal axes than others. It is possible to make etch rate diagrams, graphic depictions of the rate of material dissolution at each possible face normal angle, for any combination of semiconductor material and acid. Using these diagrams, the shape of a feature which has been etched for a specified time may be derived [54]. One way to derive the resultant shape is to use the formalism of the vector space of monostrofic tracings discussed in Section 3.4.

Consider a convex feature $\mathcal{F}$ which is to be etched away by an acid with convex etch rate diagram $\mathcal{R}$, shown in Figure 3-21. The shape of the feature after being etched for unit time is the area with winding number equal to one in the convolution $\mathbf{T}(\mathcal{F}) * -[\mathbf{T}(\mathcal{R})]$. If the feature is etched for an additional time unit, its shape will be the area of winding

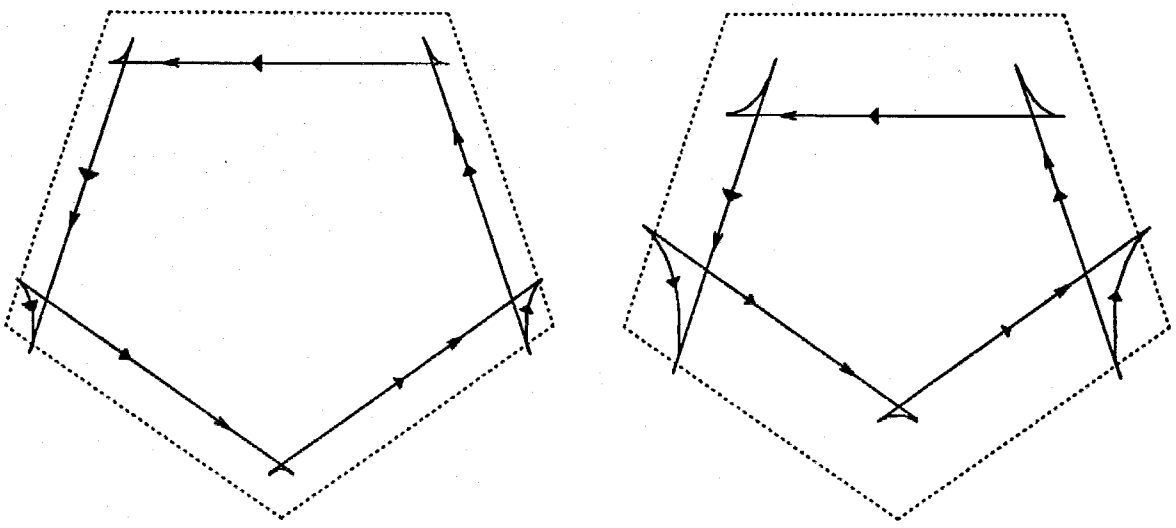


Figure 3-22: Convolutions used to determine the shape of the etched feature. The dotted line is the original feature. The convolution on the left has been etched for one time unit. The convolution on the right has been etched for two time units.

number equal to one in the convolution

$$\{\mathbf{T}(\mathcal{F}) * -[\mathbf{T}(\mathcal{R})]\} * -[\mathbf{T}(\mathcal{R})] = \mathbf{T}(\mathcal{F}) * -2[\mathbf{T}(\mathcal{R})],$$

as shown in Figure 3-22. Furthermore, if a specific final feature $\mathcal{G}$ is desired, to be obtained after etching time t, then the smallest initial feature which must be cut by photolithography is $\mathbf{T}^{-1}[\mathbf{T}(\mathcal{G}) * t\mathbf{T}(\mathcal{R})]$, as shown in Figure 3-23. Thus it may be seen that the group-theoretic properties of shapes and shape operations can be quite helpful in solving real-world problems.

Grammars that employ operations used in defining an algebraic structure on shapes or tracings are guaranteed to produce only shapes or tracings which are also members of the respective structures. Thus it can be proven that only realizable shapes will be generated when using the rules. As discussed above, homomorphisms between the sets of shapes and other descriptions of the shapes can also be defined. Operating in the homomorphic world might allow the designer to describe and represent the shapes in a more simple way than their physical representation in $\mathbb{R}^n$. For example, a designer could work with tracings, but determine the properties of the shapes corresponding to the tracings (*e.g.*, the area or

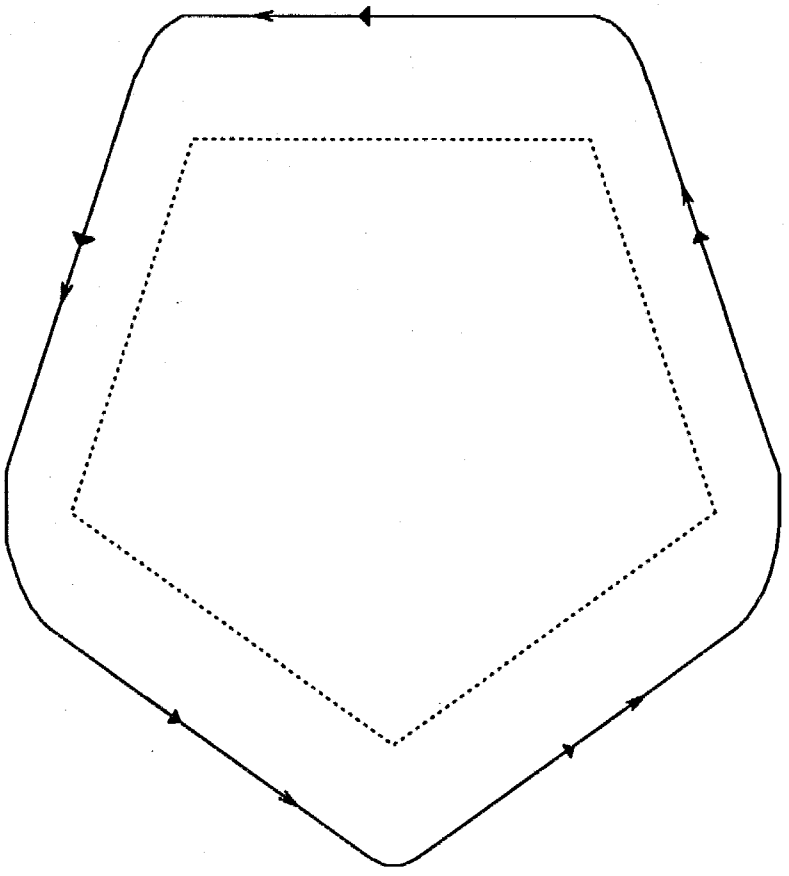


Figure 3-23: Convolution used to determine a required initial feature shape. The dotted line represents the shape of the desired feature. The convolution has the shape of the initial feature required to produce the desired feature after two time units of etching.

moment of inertia) and use these properties to determine if a partially-derived shape will satisfy the function and performance requirements for the design.

The operations and properties discussed above serve to add flexibility to the designer's use of shape in his or her designs. By guaranteeing that shapes which have had rules applied to them are still members of a particular class, the above properties enable the designer to experiment with combinations of shapes (using the operations discussed) without concern that the final shape may not be realizable or may not fall in a particular class. Grammars which exploit these group-theoretic properties can absolutely ensure that all shapes in their languages satisfy certain basic requirements of form, which reduces the size of the design space which the designer must search.

Furthermore, the guarantee that all shapes in a language are members of a certain set enables some generally undecidable properties of a language to be easily decided. For example, as discussed in Section 2.2.3, the problem of determining if a given language is contained within another is undecidable by any general algorithm. However, in some cases it may be easy to check if a language which is restricted to a certain set of shapes is contained in another language. The properties of the grammar might be used to get around some of the decision properties which would otherwise be required.

Chapter 4

Grammars and Expert Systems: Similarities and Differences

4.1 Introduction

Many formalisms for aiding the engineering designer have been proposed. A number of design tools being developed are based on expert systems [3, 22, 27, 37, 77, 82, 90, 136]. In this chapter, grammars are compared with expert systems in order to evaluate the suitability of each formalism for use as an engineering design tool. This will necessarily be a high-level comparison because, as will be discussed below, expert systems are only defined informally. Furthermore, both grammars and expert systems vary so widely that no canonical prototype of either system exists. Therefore, this presentation will be relatively informal and consist mostly of opinion, though the two systems will be treated as even-handedly as possible. Expert systems are defined first, and then the many similarities between expert systems and grammars are discussed. Then the formalisms are distinguished by laying out their differences, both in formal definition and in customary use.

4.2 Formal and Informal Definitions

Expert systems have been designed to solve many different classes of problems. In general, an expert system is a computer program which uses rules and facts to solve problems which could otherwise only be solved by human experts, as discussed by Ullman and Dietterich in [136]. Unfortunately, as pointed out by Humpert, expert system and artificial intelligence

researchers can not agree on the specific, formal characteristics of an expert system [55]. Instead, they "know one when they see one": each individual determines for him- or herself if a program satisfies his or her own specific criteria which set an expert system apart from other computer programs.

Nonetheless, a general, informal set of criteria was put forward by a group of expert system designers in [7]. In this introductory article on expert systems, the authors presented some characteristics which almost all expert systems share.

- An expert system has expertise in its field. It can solve problems quickly using an efficient methodology.
- Expert systems manipulate symbols as well as the numerical variables with which computers typically work.
- Expert systems can solve a range of problems in their domains of expertise. They robustly handle problems by using general-purpose reasoning when faced with patterns which fail to satisfy more specialized and powerful rules.
- An expert system solves problems which might be too difficult for a non-expert human to solve. The solution method could be difficult to understand, or the problems might be very complex.
- Expert systems, when faced with a poorly-formulated problem, are able to transform the given knowledge into a representation in which it is easier to apply the rules and facts available.
- An expert system can describe to a user why it uses a certain rule at a given time. It is able to explain the reasoning behind its conclusions by enumerating the sequence of rules applied in order to reach the conclusions.
- An expert system is suited to a particular task. Different expert systems use different methodologies to represent and manipulate their knowledge. Therefore, two expert systems which use different methodologies may have strikingly different performance when presented with the same problem.

Existing expert systems satisfy each of these criteria to different extents. For example, an expert system might only be able to solve very general problems in its domain or it might not be able to recognize data presented to it in a certain way. Nonetheless, all expert systems seem to share these characteristics.

There are three main classes of expert systems, differentiated by the way that knowledge is represented: rule-based, frame-based, and logic-based [57]. The knowledge of rule-based expert systems is contained in the production rules used to manipulate data. In these systems, a controller checks if the data match the left-hand side of a rule. If they do, then the rule is activated and it transforms the database. In contrast, frame-based expert systems concentrate on forming relationships between different data elements. Any data object might be a member of several intersecting classes, any of which might be matched with the left-hand side of a rule. Like rule-based systems, logic-based expert systems also concentrate their knowledge in rules. However, logic-based system rules are based on predicate logic, whereas rule-based systems rules need not be. Logic-based systems are typically used for deductions: given a data set, a logic-based system will deduce other facts, based wholly on the knowledge contained in the database. On the other hand, rule-based systems are usually used to transform the data; knowledge that directs the transformation may be implicit in the rule. A particular expert system is usually a mixture of all three of these classes, but typically one knowledge representation feature dominates.

Recalling the definitions in Chapter 2, observe that the definition of a grammar is much more formal and explicit than that of an expert system. As will be seen in the following sections, some grammars can satisfy the criteria to be expert systems and some expert systems can, in turn, satisfy the definition of a grammar. However, in customary use, the explicit definition of a grammar gives it some powers which an expert system can not match.

4.3 Similarities Between Grammars and Expert Systems

Grammars share several properties with expert systems because they are both production systems, which use rules to modify strings of symbols. These symbols could be words, shapes, mathematical symbols, or any other set of elements. Rule-based expert systems, with their domain knowledge encoded in their rules, attempt to satisfy goals by sequentially

applying the rules to a starting set of data. Similarly, grammars apply rules to a starting symbol. Both systems can manipulate symbols and numerical values and can function with a range of inputs (as long as the inputs are of the type specified for the particular system). They are able to apply general rules when there is no match to any of the more specialized rules in the set of transformations. They can both keep track of the rules applied and back up a derivation or conclusion by listing the rules and the context in which they were applied.

Grammars share an additional property with frame-based expert systems (a property which strictly rule-based expert systems lack). Both frame-based expert systems and grammars are able to distinguish relations between different elements of the transformed input and they may classify elements in several different ways. Grammar rules can apply to any substring of the string of symbols being manipulated. They can also apply in the different domains being used, if the grammar uses more than one domain or if there are parallel grammars operating in tandem. Expert systems and grammars can both distinvguish elements differently in different contexts.

4.4 Explicit Differences Between Grammars and Expert Systems

While there are some striking similarities between grammars and expert systems, as detailed above, there are also differences between the two production systems. In this section, the differences which necessarily stem from the definitions of the two are discussed. In the following section, differences which are results of how the two systems are customarily used will be examined.

4.4.1 Emergent Properties

Although both grammars and frame-based expert systems can classify elements in several different ways, the expert systems are more limited in their scope of classification than the grammars are. As discussed in Section 2.4.2 and by Stiny [126], grammars can exploit the emergent properties of the strings which they produce. Specifically, rules may be applied to a string in a way which is not obvious from the outset. Expert systems, on the other hand, have their data structures rigidly defined from the beginning (this definition is necessitated

by the computational implementation of an expert system, even a frame-based one). Thus only a finite number of patterns need to be checked against the left-hand sides of an expert system's rules.

Note that this difference has both a positive and a negative aspect for the use of grammars. On the plus side, grammars can recognize and modify patterns which expert systems may not notice. On the minus side, grammars may be faced with situations in which the left-hand sides of their rules match a huge number of patterns in the string, causing the size of the design space to be searched with the grammar to grow very (and perhaps unmanageably) large or dense.

4.4.2 Formal Structure

Expert systems lack the formal mathematical structure which grammars possess. The formal properties of various types of grammars have been studied extensively by many researchers [47, 84, 85]. Most importantly, the rules of grammars are formed so that every string, sentence, or design produced by the grammar is "grammatical." In other words, every sequence of terminals that is contained in the language defined by a grammar will "work" in the context of the problem. For example, a grammar for the Spanish language will produce only well-formed Spanish sentences (though it may not output *all* such sentences). Similarly, a shape grammar which operates on lines in a plane will produce only combinations of lines, with no points or plane segments. Grammar rules are formulated so that a sentence of the grammar satisfies the constraints of its domain. Expert system rules are initially formulated only to emulate the process a human follows when trying to solve a problem in the domain. After these original expert system rules are tested, they are modified or augmented so that a satisfactory result is produced for every set of test data. However, satisfactory results are *not* guaranteed for all possible sets of data, only the ones which have been tested and found to work.

There are other grammatical properties which expert systems lack. For example, it can be determined whether a particular string, sentence, or design can be derived by a grammar if the grammar is low enough on the Chomsky hierarchy, as discussed in Section 2.2.2. Based on the complexity of the language and following the lead of Mullins and Rinderle, an attempt can also be made to estimate the richness of the design space that will be explored when

constructing sentences with the grammar [86]. Furthermore, the type of Turing machine which would be required to write all the expressions of the grammar can be determined, as demonstrated by Fitzhorn [34].

The properties defined by the grammar type are not the only advantage of the grammatical formalism. In Chapter 3, it was shown that certain families of shapes have algebraic structures. If the rules for a grammar operating on shapes are defined completely in terms of the elementary operations which determine the algebraic structure, then it can safely be asserted that the shapes generated will be members of the desired set. Also, if a bijection can be defined between the shapes and another domain (especially a domain in which there is a measure of how well a given design satisfies specified performance criteria), then operations can be done in that other domain while doing parallel, legal, operations in the shape domain. This ability to work in more than one domain allows the designer to be more function-oriented in his or her application of the rules and ought to result in faster generation of designs. Even if only a homomorphism from shape to function can be found, the location in function-space could allow a designer to evaluate designs which are in a language more easily than if he or she had to evaluate them based on shape information alone.

4.5 Differences in the Customary Uses of Grammars and Expert Systems

While grammars and expert systems necessarily have the similarities and differences discussed above, other differences arise in their customary use. Three differences will be discussed in this section. First, expert systems typically derive their rules from human expert behavior, while grammars have rules rooted more solidly in fundamental principles. This difference makes the two formalisms appropriate for different types of problem solutions. Second, while data structures in frame-based expert systems can be flexible, grammars usually prove to have even greater capabilities. Third, grammars are customarily used to solve problems in different domains from expert systems.

4.5.1 Rule Sets

Expert systems and grammars usually take different approaches to solving a problem. The application of grammatical rules is, in general, undirected. Since grammars rely on well-constructed sets of rules in order to keep the size of the language small, there is typically little control over the sequence in which rules are applied. A supervisory grammar or human director, using one of the techniques outlined in Section 5.3.2, may be needed in order to direct a grammar toward a given goal. Alternatively, the rule set for a grammar could be elegantly defined in order to encode a search strategy directly into the rules.

On the other hand, expert systems are generally very goal-directed. Their goal is usually to find one answer to a problem, not many possible answers. For this reason, expert systems tend to use meta-rules, rules which govern the use of the expert rules, to direct their searches. The meta-rules can be used to govern the use of rules in the knowledge base, so expert systems are able to have a larger number of complex rules for handling special cases in their rule sets. Thus, expert systems are usually better suited to solving problems for which the domain is complex but has been explored and solution methods for problems are fairly well-defined. Conversely, grammars function best when they are used to explore large areas of a relatively unknown design space.

An important difference between grammars and expert systems may be seen in the way that rules are typically created for the two formalisms. Rule sets in expert systems are usually gleaned from the observation of a number of humans with expertise in the field under consideration. Alternatively, a neural network rule base might be inductively "learned" from examples in a domain by adapting neuron weights. These rules are all patterned after the response of expert designers to specific design problems, so they might not be able to deal with unforeseen inputs in a reasonable manner. Thus the rules of expert systems often are heuristics or "rules of thumb." While these rules could be valid, they may not be provably so, as pointed out by Adeli [2]. Grammars, on the other hand, usually have rules sets which are based on physical principles or underlying design philosophies. The rules must all contribute to the end purpose of the grammar: to produce a set of sentences, strings, or designs which are all grammatical. Grammar rules must be product-oriented in order to produce valid results. Expert system rules tend to be more process-oriented, derived by looking over the shoulders of designers and mimicking their behavior. In [28],

Dym and Levitt present a typology of engineering knowledge. Domain knowledge runs the spectrum from fundamental knowledge, through phenomenological knowledge, analytical models, numerical models, component descriptions, to experiential knowledge. The first few types of knowledge have closer ties to the root causes of real-world phenomena, while the last few types deal with how humans interpret the effects of those phenomena. Grammar rules tend to be based on the former knowledge types, expert system rules on the latter.

4.5.2 Data Structures

As mentioned above, expert system data structures tend to be relatively rigid, while a string being modified by a grammar can be considered in almost any manner. This flexibility can lead to the discovery of emergent structures in designs being generated by grammars, as discussed in Section 2.4.2. However, in order to recognize these emergent structures, the representation of the design being generated by a grammar must be sufficiently rich. Thus richness in manipulation of data can usually be gained only by sacrificing ease of its representation.

The left-hand side of a rule must usually match the data being manipulated so that the rule can apply in an expert system. At most, expert systems might allow symbols in rules to match a number of possible data values. Grammars allow transformation of data before applicable rules are chosen, as discussed at length by Carlson [12], so grammatical rules can apply in a larger number of cases than expert system rules. Grammars also usually allow parameters, symbols, and the like to be used more readily than expert systems do. Expert systems currently handle uncertainty by using formalisms such as fuzzy or probabilistic variables, as described by Wood, Antonsson, and Beck [138], or by using constraint propagation, as described by Ward, Lozano-Pérez, and Seering [137], all of which may be used by grammars as well.

4.5.3 Problem Types Addressed

As discussed by Ullman and Dietterich in [136], expert systems are primarily used for problems which have a limited range of solutions, which can be broken down into subproblems, which have constraints independent of the solutions, and which have reliable data for a knowledge base. Thus they operate best in the detail part of the design process. Grammars

may be used in all stages of problem solving, but they are used to their best advantage in conceptual design, where a design philosophy has been settled upon but an embodiment has had little consideration.

Furthermore, grammars and expert systems are employed in situations in which different results are desired. If only one satisfactory solution to a design problem is desired, then a designer is likely to use an expert system if one is available. However, if the designer wants to explore a range of designs, evaluate a number of designs against each other, or search for an optimal design, then he or she would be better off using a grammar to generate a range of alternatives.

4.6 Conclusions

The two formalisms discussed here seem very similar on their surface. However, once their differences are explored and their customary uses are taken into account, expert systems and grammars stand in very different lights. For tasks in the early stages of the design process, grammars tend to be superior thanks to their richness of expression. Later in the design process, when a design concept has been decided upon but an embodiment needs to be constructed, expert systems can often give results faster than grammars, though a good grammar may produce a richer set of solutions with better performance. Both formalisms are, and will continue to be, useful tools for the designer.

Chapter 5

Using Grammars Effectively

5.1 Introduction

The following is a general overview of some techniques which can be used when designing and using grammars. Most of this chapter is of necessity an informal description of ways to use grammars, since each design problem has a different goal and therefore needs different tools. Some of those tools will be discussed below. First, methods of properly choosing appropriate rules for a grammar are discussed. Next, strategies for searching large design spaces are explored. Finally, the role of transformations in grammatical derivations is discussed.

5.2 Choosing Rules for Grammars

As was discussed in Section 4.5.1, the most important step in the definition of a grammar is the proper generation of its rules. Rules must always be chosen with the final design in mind because all designs in a grammar's language *must* satisfy some basic set of functional requirements, as pointed out below. The design task thus changes from one of creating a single design which satisfies specified criteria to one of creating rules which can be used to generate a large number of satisfactory designs. While the latter task may be more difficult than the former, it also enables the designer to make a more thorough search of the design space. Several types of rules are available for the designer's use, and are discussed below. Once a grammar is defined, its rules may be modified (provided they remain well-formed) in order to refocus the search of the design space.

5.2.1 Proper Formation of Rules

When defining a grammar, the important step comes not when deciding which symbols to use in the alphabet, but when defining the set of rules. The proper use of a grammatical formalism requires some thought about how the rules will function. If a production system is to be a grammar, all of the strings which the system produces must be "grammatical." In other words, the members of a grammar's language must all satisfy some set of requirements or be alike in some specific way. For this reason, a designer should know what properties he or she wants grammar-generated strings to share before he or she defines all of the grammar's rules.

For example, consider a grammar which generates three-dimensional shapes. If the shapes generated by the grammar are to be used as real-world constructions, they must satisfy some criteria. At a minimum, the shapes must be regular: every point of a shape's boundary must be adjacent to the shape's interior. In other words, there may be no "dangling" faces or edges on a shape (see Requicha and Tilove's work in [98, 99] for a discussion of operations which can be used in rules and whose use ensures the production of only regular shapes). If the shapes are regular, then they may be physically constructed. If the constructions based on the shapes from the grammar must satisfy some other criteria (for example, they must have only 90 degree intersections between their edges, or they must be able to be produced from a single block of material using only milling-type removal operations), then the rules must be more restrictive and must be formulated more carefully in order to satisfy all the constraints imposed.

5.2.2 Types of Rules

Rules may have several functions in the sentence derivation process. In general, the application of a rule will have implications in many domains. For example, an engineering grammar rule which attaches a girder to a partially-formed truss also changes the weight and stiffness of the truss and could also change its style. Conversely, a stylistic rule which governs the points at which girders may be attached will also affect the load-bearing capacity of the truss. All rules are, to some extent, both functional and stylistic. The interaction of geometry and functionality is a characteristic of all engineering design problems.

Grammars are well-suited to the exploration of design spaces because grammars are

able to represent and work with form/function interactions so well. Parallel grammars can function in several domains at once, with a rule in one domain triggering rules in other domains. A designer using parallel grammars can select rules in different domains in order to satisfy a number of functional requirements. For example, consider a grammar for constructing gear trains. This grammar must have geometric rules for determining the relative placement and the shapes of gears. It must also have kinematic rules which govern the relative angular velocities of the gears. It could also have engineering rules which calculate the pitch and face width required of gears in order to transmit a desired amount of power. A designer could use these parallel grammars to create a gear train which satisfies several requirements. The designer could first apply a kinematic rule in order to obtain a desired speed reduction in the first stage of the gear train. Parallel geometric and engineering rules would then be applied in order to determine gear sizes and shapes. Next, the designer could use a geometric rule to specify the size or location of another gear. This rule would enable kinematic and engineering rules to be applied in order to determine the gear ratio and tooth sizes. Finally, the designer could use an engineering rule to determine the pitch of the last gear in the train. Then kinematic and geometric rules could be used to define the gear ratio of the entire train and the sizes and positions of all the gears. The designer can work in the domain of his or her choice; parallel rules in several domains enable stylistic and functional aspects of the design to be related and manipulated in parallel.

5.2.3 Modification of Rules and Alphabets

Once rules have been chosen and some candidate designs have been generated by a grammar, a designer might feel a need to modify some of the rules. There are many reasons for such an urge. The designs being generated might not share the characteristics desired. The designs may lie in a part of the design space which is far removed from the area in which the designer wants his or her designs to lie. The designer may decide that the characteristics common to the designs in the language are not precisely the ones needed in order to satisfy some functional requirements, but that some related characteristics might be. Interesting results can be obtained by modifying only a small subset of a grammar's rules. For example, consider a grammar for steel trusses which uses pieces of steel plate in its alphabet. The grammar could have a rule for welding the plates at right angles along their edges, producing

channels and beams with L-shaped cross sections. That rule could be modified to allow the edge of one plate to be welded perpendicular to another plate along the second plate's center line, producing beams with T- and I-shaped cross sections. The simple modification of one rule could seriously affect the optimum geometry of a truss for a particular application.

Alternatively, and perhaps for some of the same reasons, a designer might want to modify the grammar's alphabet (and consequently the form, but not the substance, of its rules). For example, he or she might want to substitute blocks with rounded edges and corners for ones with sharp angles, or triangles for squares, or one color for another. For each modification of a grammar's alphabet, the rules must also be modified to handle the new symbols in the alphabet in a way similar to the way in which the old symbols were manipulated. Consider, for example, a gear train grammar in which spur gears are replaced by bevel gears. Rules would have to be modified in order to make successive shafts at right angles. However, the basic representations of the gear trains in other domains (gear ratios, pitches, diameters, etc.) remain unchanged in this new grammar. The new grammar has operations and symbols which are analogs of those in the original grammar, as discussed by Knight in [63].

Rule modification has produced some interesting results. For example, Koning and Eizenberg's grammar used to generate architectural plans in the style of Frank Lloyd Wright's Prairie houses [69] was modified by Knight by changing the shape relations in some of its rules [66]. The resulting grammar generated houses in Wright's later Usonian style. Here, rule modification serves as a tool for style analysis, highlighting the small number of stylistic changes necessary to generate very different designs.

5.3 Searching the Design Space With Grammars

Throughout this dissertation, the design process has been characterized as a search of a design space. This space is very large, having dimension equal to the number of independent design decisions which may be made; the space could have an infinite number of dimensions. The concept of the design space and the designer's search through it has become the prominent paradigm in the realm of design theory, as discussed by Woodbury *et al.* in [140].

Since the size of a search space in a design problem can be very large (and perhaps

infinite), efficient methods for reducing the size of the space to be searched are needed. The choice of a satisfactory design through the exhaustive enumeration and evaluation of all possible designs (called the generate-and-test method of design) is only successful in very small design spaces or in the fortuitous circumstance in which a good design happens to be evaluated early in the process. Grammars, too, can generate a large number of design possibilities. However, the choice of grammatical rules in itself serves to reduce the searched area of the design space to a set of designs which all satisfy some basic criteria, as discussed in Section 5.2.1. Tradeoffs must be made when choosing the portion of a design space to search. However, a designer may achieve good results if he or she uses a grammar in a logical manner.

5.3.1 Tradeoffs in Grammar-Directed Searches

Compromises must be made between the expressiveness of a grammar and the speed of the search of the design space which its use enables. Some grammars can be used to generate enormous numbers of designs; in many languages there are an infinite number of grammatical strings. If a relatively thorough search is to be made in a reasonable amount of time, some limitations must be imposed on the size of the design space to be considered, the size of the language, or the number of designs evaluated.

The creator of a grammar must walk a fine line. On one hand, the grammar should be able to generate designs in every area of the design space. But on the other hand, the generated designs must be able to be evaluated relatively quickly. One resolution to this dilemma is to direct the grammar's rule application in order to quickly find a family of near-optimal designs, as discussed in the next section. Another resolution is to create more restrictive rules for the grammar. Since the designs generated by the grammar will be evaluated using some measure, it would be to the designer's advantage to generate designs which tend to maximize that measure. For example, a grammar which generates beam cross sections and then evaluates them based on bending stiffness could have its rules modified so that stiffer beams are generated. Since stiffness is directly related to the moments of inertia of a beam's cross section, the moments can be increased by ensuring that material is generally added to the beams far from the beams' centroids.

An alternative to restriction of the rules is the restriction of the design space. If the

possibilities available to the designer are reduced, then all searches of the smaller design space are faster. Furthermore, a grammar used to search this new design space could have fewer rules because there are fewer alternative design variable values that must be evaluated. However, a designer must not be too quick to reduce the size of the design space. Grammars are meant to stimulate the designer's imagination and creativity by showing many alternative designs so that the designer does not focus on one design too early in the design process. A single design selected early in the conceptual design phase might perform worse than one chosen after a broader search. Therefore, the designer must not restrict the design space too much, because he or she could be sacrificing good designs for faster results [1, 95].

5.3.2 Strategies for Applying Rules

As mentioned in the previous section, the number of designs to be evaluated might have to be limited if both the design space and the language of the grammar being used for design are large. Each design in the design space has a unique measure of "optimality," or satisfaction of all functional requirements and performance parameters specified by a customer. However, the measure of optimality varies in a highly irregular way throughout the design space. Thus, typically no standard methods of multivariable optimization can be easily used to find the best design in a design space. Instead, some method must be used to direct the application of a grammar's rules in order to sample a range of designs or to generate a number of designs in one of the more optimal areas of the design space.

The order and locations of rule applications may be controlled in several ways. An explicit, predetermined scheme may be devised which calls rules in a prespecified manner. At the opposite end of the spectrum, a human (preferably one with some knowledge about the design space being searched) could choose which rules to apply at certain times. Somewhere in between these two methods is the use of a nondeterministic system, like a meta-grammar or an expert system, to control rule applications. Each of these controllers could follow some of the search strategies outlined below.

Occasionally, a design space will have a great deal of underlying structure. In these rare cases, it might be possible to use some straightforward search strategies. For example, if each rule application corresponds to the setting of one design variable, then multivariable

optimization might be used to help find the best designs in the design space. If the design variables are independent, then a designer might be able to choose which one of a number of applicable rules to apply at a given time simply by finding the one that would produce the most optimal design after the application of that single rule. However, these sorts of design spaces do not appear very often in real design problems.

Another simple way to direct a search uses the tree-like structure of evolving grammar-generated designs. Consider a representation of a grammar as a graph-theoretic tree. The root of the tree is the starting symbol and the leaves are the members of the language generated by the grammar. The root has one or more branches corresponding to the application of one of the grammar's rules to the starting symbol. Each of these branches, in turn, has more branches, all the way out to the leaves. If there is a way to evaluate partially-formed designs accurately, the designer could use the information gained from the evaluation to "prune" the branches which produce designs that are sub-optimal. Note that this strategy depends on the assumption that all the leaves beyond one node will meet the design criteria better than all the leaves beyond another node if the first node satisfies the criteria better than the second. In design problems, this is usually an incorrect assumption, but this method of directing a search may nonetheless be of use in some cases where the assumption is incorrect or known to be false. For example, if all of the designs in a language will satisfy the criteria demanded of them and the designer wants to create a "good-looking" design, he or she could prune branches from the grammar's tree based on subjective aesthetic rules.

The number of incomplete designs to which no more rules can be applied—branches with no leaves—might be decreased by properly choosing the order in which rules are applied. If two rules are to be applied to an evolving design, but incomplete designs might be produced after the application of one of them, then when generating the grammar's tree it makes sense to use the rule which produces incompletable designs first. Then the other rule can be applied, producing a smaller number of incomplete branches. For example, if a rule may be used in m ways, of which n resulting designs will be incompletable, and a second, independent, rule may be used in p different ways, then applying the first rule followed by the second produces $(m-n)p+n$ designs, of which n are incompletable. By contrast, if the order of rule application is reversed, mp deigns are generated, of which np are incompletable. Note that these results will be valid only if the use of one rule

does not affect the incompletability of any designs produced by the other. Nonetheless, an analysis of the effects of different rules on an evolving design can help to reduced the number of candidate designs generated. This scheme is sometimes used in rule-based parsers for natural language, as discussed by Fong in [38].

If a designer wants to sample a range of designs from the grammar's language, he or she can use a Monte Carlo-type method of design generation. The designer chooses the number of designs that he or she wants to view. Beginning with the starting symbol, for each string of nonterminals, all possible productions are determined and one is applied at random. The process is repeated the desired number of times. This procedure produces random paths from the root of the grammar's tree out to the desired number of leaves. If certain rules are known to produce better designs than others, the procedure could be augmented. Instead of giving each possible production an equal chance of being applied at each step, each production (or each rule) can be given a weighting factor by which a random number is multiplied. The production with the highest product is then applied.

In a design space in which optimal designs tend to be clustered, it can be profitable to use a simulated annealing technique to find good designs. Simulated annealing is an optimization analog to the annealing of metals or ceramics. In the annealing of metals, alloys with coarse grains are held at elevated, but subliquid, temperatures for extended periods. The high temperatures allow the material to soften and the grains to shrink in size because smaller grains have lower total potential energy. The metal is maintained at a relatively high temperature at the start of the process in order to stimulate the flow of grain boundaries. As time goes on, the temperature is reduced slowly so that the smaller grains can enter a relatively stable equilibrium state. The resulting material is less brittle than the original mixture and has greater toughness. If the temperature is reduced very slowly, then the material will have infinitesimal grain sizes and a global minimum potential energy. If the temperature is dropped more quickly, potential energy could end up only at a local minimum.

Simulated annealing is an analogous process in which optimality is maximized rather than minimizing potential energy. Just as the molecules of a metallic compound vibrate more at higher temperatures, the chosen path through a design tree changes more at higher simulated annealing temperatures. In the early steps of a grammatical derivation, the

temperature is high and many different paths are randomly tested. Paths with higher optimality are preferred, so the path is switched to a better one when it is found. As the temperature is lowered, the random search for more optimal paths is narrowed to those paths close to the most optimal one found so far. As the temperature is lowered to zero, the path stops changing, and the designer is left with a design which is at a local optimum in the design space. If the temperature is lowered slowly enough, the path describes the globally optimum design. For a more detailed description of the simulated annealing process and an example of a grammar used to densely pack shapes into a predefined area, see the work of Cagan *et al.* [9, 10, 11].

A close analysis of a grammar's rules might also provide some direction to the design derivation process. If several different rules could apply to each partially-derived design, then the designer could use Taguchi's methods [8, 134] to determine which rules most robustly generate designs which satisfy the design requirements. In this situation, the sets of rules are treated as parameters which may be varied. Since each design problem solved using a particular grammar might have different requirements, the measure of optimality is treated as a random variable. By choosing appropriate combinations of rules and analyzing the resulting optimality of the designs (under the different measures of optimality), the designer can discover which rules most robustly produce optimal designs. These findings could then be used to direct future design derivations or to define weighting factors for rules in a weighted random search scheme.

5.4 Transformations

There are usually many opportunities to apply rules to a partially-derived string. Most of these opportunities are made possible by the ability to transform the partially-derived string in order to make a portion of it match the left-hand side of a rule. As discussed in Sections 2.2.1 and 2.4.1, even if no transformations are formally defined in a grammar it might be necessary, at the discretion of the designer, to use some transformations. In shape and engineering grammars, the ability to translate a shape in order to align it with a shape in a rule is almost always necessary. Rotation, reflection, and scaling are three other valuable and much-used transformations for grammars involving shapes. The use of these

transformations helps broaden the range of situations in which a rule is applicable, which in turn increases the size of the design space searched by a grammar.

Transformations can also help maintain, in a compact format, information on how a design is generated. In order to reconstruct a design generation process, a designer needs to know the order in which rules were applied to the starting symbol and the transformation which was necessary to apply each rule. Thus, for the transformations mentioned above, only nine pieces of data can describe each step in the derivation process (because scaling can be described by one piece of data, a screw—represented by a vector of length six—and its magnitude can describe the rotation and translation, and one other piece of data can point to the rule used).

5.5 Conclusions

In order to use grammars effectively, some attention must be paid to the selection of rules, the search strategies employed, and the transformations used. By selecting rules properly, unique and interesting designs may be generated by a grammar. If the design space is searched in a well-chosen manner, good designs may be recognized relatively quickly. Using various transformations allows the designer a more compact representation of both the rules of a grammar and the designs generated by it. These conclusions may only scratch the surface; there may be additional properties of grammars which can be discovered during a more thorough investigation into these areas.

Chapter 6

A Grammar for Reconfigurable Modular Robot Arms

6.1 Introduction

In this chapter, a grammar for the configuration of reconfigurable modular robot arms is introduced and discussed. These robot arms are made up of a small set of link and joint components and are meant to perform a large number of different tasks by being disassembled after one task and reassembled in a configuration more well suited to carrying out the next task. Although several different prototype module sets have been constructed, researchers have presented few algorithms for the optimal configuration of an arm for a given task. While no such algorithm is presented in this chapter, a grammatic formalism for enumerating the non-isomorphic configurations of an arm with a given number of links and joints is presented. By finding only non-isomorphic arms, the search space for the optimization of arms to specific tasks is reduced by orders of magnitude. The grammatic formalism is compared to an existing formalism based on the symmetry groups of matrices and is found to perform comparably and produce not only arm configurations (like the existing formalism) but also forward kinematics, which the existing formalism does not automatically produce.

The grammars in this chapter were inspired by the work of I-Ming Chen [15, 16], who first proposed the set of links and joints used here. He also wrote the original *Mathematica* code used to draw the arms formed using these links and joints. His method of generating non-isomorphic arms is discussed in depth in the next section.

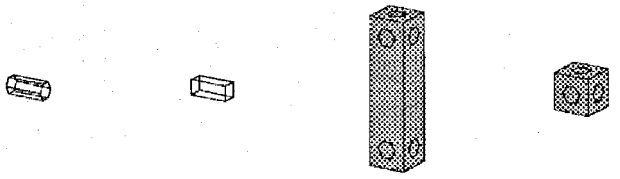


Figure 6-1: Joints and links used in this chapter for constructing robot arms.

6.2 Background

While robots are presently used to perform a wide variety of tasks, a single robot is, in general, suited to only a small number of tasks. Some robots are capable of manipulating large objects but can not position the objects with high accuracy. Other robots can accurately pick and place objects at high speed but have very limited workspaces. Ideally, one would like to have a robot which could perform many different tasks. However, robot geometry, and consequent dynamics, preclude today's typical arms from operating in all but the most restricted workspaces. For this reason, several teams have designed reconfigurable modular robot arm systems in recent years [20, 40, 107, 141]. These systems share several characteristics. They all consist of rigid links and of joints with at least one degree of freedom which connect the links. The links and joints can be separated and reconnected in several different ways to produce arms with very different geometries and kinematic and dynamic behaviors. Some scheme is used to decide on a configuration for an arm. When an arm is configured, its forward kinematics (the transformation between joint variables and end-effector position) are obtained and the inverse kinematics (the transformation, not necessarily unique, between end-effector position and joint variables) are derived. The generation of an arm's forward kinematics is relatively straightforward, but all proposed methods of finding inverse kinematics rely on numerical simulation so that the methods can apply to all possible configurations of an arm [60, 107].

Given a set of joints and links, it is possible to construct a very large set of arms. Take, for example, the set of joints and links shown in Figure 6-1. There are two types of joints, one which rotates about its axis, called an R-joint or rotary joint and represented by a transparent hexagonal prism, and one which translates along its axis, called a P-joint or prismatic joint and represented by a transparent square prism. There are also two types of

links, a long rectangular block, called an L-link, and a cube, called a C-link. Joints may be attached to the C-link at the center of any of its six faces. Joints may be attached to an L-link at either square end and near either end of all four rectangular faces. The joint attachment points are denoted by circles on the blocks in the figure.

A designer might want to use three L-links and two R-joints to make a serially connected robot arm. He or she has ten choices for the location of an R-joint on the first link, ten choices for the location of the other end of the R-joint on the second link, five choices for the location of the second R-joint at the other end of the second link, and ten choices for the other end of the second R-joint on the third and final link. Multiplying, it is clear that there are a total of five thousand possible configurations for the arm! Of course, many of these configurations are just rotations of one another; they look the same and have the same kinematics. Two arms which can be made to look identical by rotating or translating their bases and/or moving some or all of their joints are called *isomorphic* assembly configurations. As will be shown in Section 6.4, there are only 21 non-isomorphic assembly configurations of two links and one joint. This example demonstrates why a brute-force method of enumerating all possible configurations is a problematic way to start deciding how an arm should be configured: only a fraction of all configurations are non-isomorphic.

Unfortunately, most of the reconfigurable robot prototypes discussed above use such a brute-force, generate-and-test method in the initial stages of determining the best configuration for a task. The general strategy employed is to generate all the possible configurations of the arm, find their forward kinematics, then determine the optimal arm for the task. The measure of optimality may be determined differently for different tasks. An arm may be evaluated not only on its cost and size, but also on the size of its workspace, its ability to manipulate objects at a given point in the workspace, or its use of only modules which are available. Furthermore, though many arms may satisfy all performance requirements, they can not, in general, be obtained directly from the requirements; they must be generated and tested against the requirements. There is a pressing need for a method of generating candidate configurations which pares down the number of generated arms to a minimum and simultaneously aides optimization.

One such method, the one which first motivated the writing of the grammar that follows

and which is discussed by Chen and Burdick in [15, 16], starts by labeling each possible connection point on a link with a number, from one to the number of connection points. Next, a graph representation of each arm is formed, with links corresponding to vertices of the graph and joints corresponding to edges. This graph is then transformed into a matrix, which the authors call the "assembly incidence matrix" (AIM), with each column corresponding to an edge and each row corresponding to a vertex. If an edge touches a vertex in the graph, the corresponding entry in the matrix takes on the connection point number of the link which is represented by the vertex. If an edge does not touch a vertex, the corresponding entry in the matrix is set to zero.

With this transformation from arms to graphs to matrices, the authors have prepared their tools for generating candidate assembly configurations. Next, they begin generating candidate AIMs. For a given number (and possibly given types) of links and joints, they generate all possible graph representations of arms. Then for each possible graph, they generate all allowable AIMs. In doing so, they make use of the Pólya Counting Theorem, which allows them to determine the number of non-isomorphic placements of a given set of joints on a link. For each link, they generate a candidate joint placement and compare it to any placements already generated for that link. If the candidate or any one of its transformations by members of the rotational symmetry group for the given link matches the placements already generated, or if the number of placements already generated is equal to the total number possible (given by the Pólya Counting Theorem), then the candidate is rejected; otherwise, it is added to the list of placements. With the resulting list of non-isomorphic joint placements for each link, all the allowable AIMs can be constructed by choosing one placement for each link in an arm.

Finally, all of the AIMs for each graph are winnowed down. Each candidate AIM, along with all the permutations of its rows and columns and its transformations and permutations by members of the rotational symmetry group for the arm, is compared to all the AIMs already generated for the same graph. If none of the transformations match, then the candidate is added to the list of AIMs. Once the complete set of AIMs for all non-isomorphic arms has been generated, each AIM can be translated back into the realm of arms and its forward kinematics can be generated.

There remain several problems with the method of generating arms described above.

First, it retains much of the brute-force enumeration technique that is supposed to be escaped. Many candidate joint placements must be generated and tested against a library of already-generated placements. Then many candidate AIMs must go through the same process. The number of candidates, and hence the computational intensity, can be undesirably high. A second problem with the above method is that, while all possible non-isomorphic arms are generated, they may not be evaluated until the end of the process. Without an idea of what an arm looks like and what its kinematics are, it is impossible to evaluate its suitability for a task. Even if the first AIM generated corresponds to an arm which could satisfy all functional requirements and performance parameters specified for it, all the other AIMs will still be generated, even though they will be unneeded. Along with this lack of ability to evaluate the arms until the end of the process comes the inability to reduce the number of AIMs generated while the process is running: even if some of the AIMs generated by a graph would be obviously unable to perform a task, they are still generated. Finally, the above process requires the translation from arm to graph to AIM, then back through graph to the set of completed arms. It would be more instructive if actual arm configurations were proposed without the translation through two additional domains.

The problems discussed above can be avoided while generating non-isomorphic arms if two grammars are used instead of the above system or brute-force generation. In the sections that follow, a grammar which generates link-joint-link combinations will first be presented. Then a grammar which uses those combinations to build non-isomorphic arms of any required length which satisfy a variety of user-specified properties is introduced. The grammatical method of generating arms will be shown to avoid many of the problems of the above method through a parallel structure which produces the kinematics and the configurations of all non-isomorphic arms simultaneously.

6.3 Grammar to Generate Dyads

The first grammar for modular robots will be used to generate dyads, or link-joint-link combinations. The second grammar, discussed in the next section, will use these dyads to build up arms. Given a library of links and joints, the task of this first grammar is to generate all possible dyads, along with several mathematical descriptions of their relative

orientation and motion. Some of these dyads will be isomorphic (the set contains all dyads which are unique up to a transformation of the first link's coordinate frame), but in the second grammar the dyads will be used in a method such that all the generated arms will be non-isomorphic.

The mathematical descriptions which will be generated with this grammar include the Plücker coordinate of the joint's axis of motion (directed toward the second link) with respect to the first link's coordinate frame, P, the homogeneous transformation between the first and the second link, $\mathbf{R}_2^1$, and the twist between the links, ξ.

The *Plücker coordinate* of a line is a 6-vector (a vector of length six) whose first three elements are the vector orientation of the line (typically, this orientation 3-vector is normalized to unit length). The last three elements are the vector which is given by the cross product of the orientation of the line with a vector from the origin to the line. For example, consider the line which is parallel to the vector $v = (a, b, c)^T$ and which goes through the point p. If the cross product $v \times p = (x, y, z)^T$, then the Plücker coordinate of the line is $P = (a, b, c, x, y, z)^T$. The i^{th} component of Plücker coordinate P will be referred to as P_i, so $P = (P_1, P_2, P_3, P_4, P_5, P_6)$.

The *homogeneous transformation* between frames i and j is represented by a 4×4 matrix $\mathbf{R}_j^i$ (which is a member of the special Euclidean group, $SE(3)$), with form:

$$\mathbf{R}_j^i = \begin{pmatrix} & R_j^i & & d \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

where R_j^i is the 3×3 rotation matrix between the i^{th} and the j^{th} frame (and is a member of the special orthogonal group, $SO(3)$) and d is the 3-vector from the origin of the i^{th} frame to the origin of the j^{th}. A homogeneous coordinate of point c in the j^{th} frame, expressed as a 4-vector in which the first three elements are equal to the coordinates of the point in the j^{th} frame and the last element is unity, may be expressed in the i^{th} frame by the homogeneous vector $\mathbf{R}_j^i c$.

The *twist* between two links is a 6-vector ξ of the form

$$\xi = \begin{pmatrix} v \\ \omega \end{pmatrix},$$

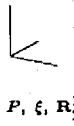


Figure 6-2: The starting symbols of the dyad grammar.

(where v and ω are both 3-vectors) which can be manipulated to form a homogeneous transformation between points in the frame of the first link which move with the second link as the joint variable changes and the corresponding points in the reference configuration, when the joint variable is equal to zero. If the homogeneous representation of a point which moves with the second link in the frame of the first link is denoted by $p(\theta)$, where θ is the value of the joint variable and $\theta = 0$ in the reference configuration, then:

$$p(\theta) = e^{\hat{\xi}\theta} p(0),$$

where

$$\hat{\xi} = \begin{pmatrix} & \hat{\omega} & & v \\ 0 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & -\omega_3 & \omega_2 & v_1 \\ \omega_3 & 0 & -\omega_1 & v_2 \\ -\omega_2 & \omega_1 & 0 & v_3 \\ 0 & 0 & 0 & 0 \end{pmatrix},$$

and where for 3-vector v, v_1, v_2, and v_3 denote the first, second, and third components of the vector, respectively. Note that this defines not only $\hat{\xi}$, but also $\hat{\omega}$. The matrix $\hat{\omega}$ is a member of $so(3)$, the group of 3×3 skew-symmetric matrices.

The grammar's starting symbols, shown in Figure 6-2, are the coordinate frame which will be used for the first link and markers for a Plücker coordinate, the twist, and the homogeneous rotation matrix of the dyad. The first rule places either an L-link or a C-link on that frame, as shown in Figure 6-3. The first link, or base link, of an arm will be depicted as a transparent box in order to differentiate it from the other links in the arm and to allow the base coordinate frame to be seen. All other links will be depicted as shaded boxes. The base frame is a left-handed coordinate frame and its z-axis will always be shown drawn toward the top of the page. Only some of the attachment points on these two links are

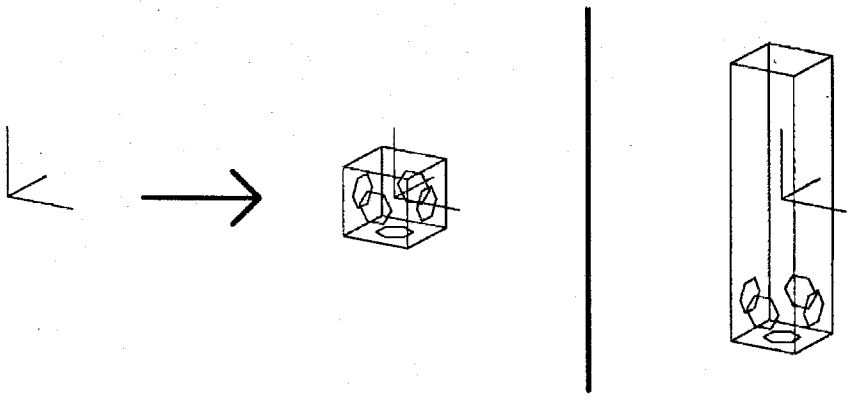


Figure 6-3: Rule to place either a C-link or an L-link on the starting symbol.

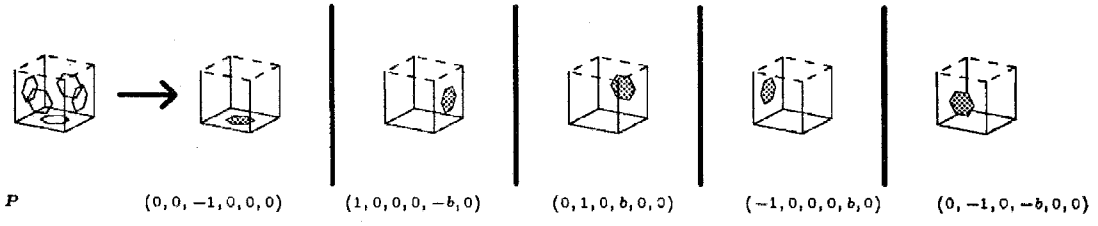


Figure 6-4: Rule to choose the joint attachment point on the base link.

shown. Joints may be put on the link only at those points in order to keep from generating pairs of dyads in which one is the other's reflection through the x-y plane. Therefore, the attachment points are used to mark the locations at which joints may be placed using the next rule.

The second rule chooses an attachment point for the joint and adds the Plücker coordinate of the joint with respect to the base coordinate frame to the body of knowledge about the dyad. As shown in Figure 6-4, the rule erases all the attachment points but one from the link. It labels the remaining attachment point with the Plücker coordinate, P, of the line through the attachment point which is perpendicular to the link's surface on which the

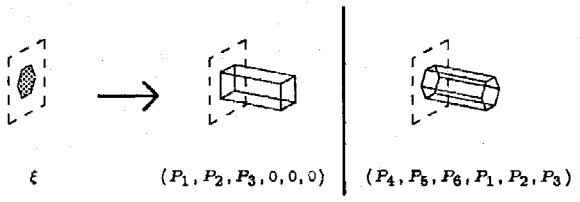


Figure 6-5: Rule to place a joint on the available attachment point.

point lies. In the figure, this labeled attachment point is represented by a filled circle on the link. The Plücker coordinates are given below each link on the right-hand side of the rule. The parameter b in the Plücker coordinates is equal to the shortest distance from the plane through the centers of the four holes on the side of the first link to the origin of the base coordinate frame. Note that the tops of the links in the figure are represented by dashed lines. This line style denotes that there may or may not be a link edge where the lines are. Thus, the links in the rule stand for both C-links and L-links. Two additional parameters will be used in this grammar. The shortest distance from the plane through the centers of the four holes on the side of the second link to the origin of the second link's coordinate frame will be denoted c. The length of a joint (when at zero extension, if a prismatic joint) will be denoted a. If the first [resp. second] link of the dyad is a C-link, then b [resp. c] is equal to zero since the coordinate frame lies in the plane of the attachment points along the sides.

The next rule simply places a joint on an available attachment point, as shown in Figure 6-5. This rule also gives a value to the twist of the link, ξ, since it is now known where a point attached to the movable end of the joint will move when the joint variable becomes nonzero. If a new type of one degree of freedom joint was to be added to the arm, only this rule (to add the joint and its twist) and the previous rule (to add the Plücker coordinate of the axis of the joint's motion) would have to be modified. Since both joint types under consideration move along axes which coincide with the major axes of the joints, the Plücker coordinate of the axis of the joint's motion may be added in the previous rule instead of this one.

In the final three rules, the second link is attached to the joint and the rotation matrix of the dyad is given a value. The first of these rules attaches either a C-link or an L-link

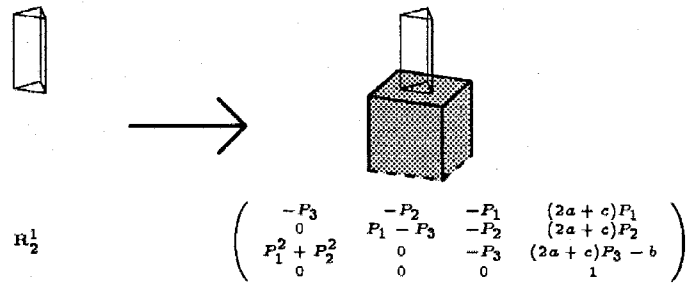
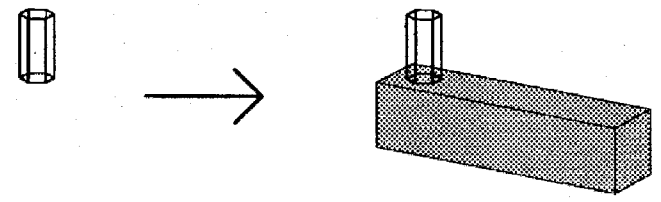


Figure 6-6: First rule to attach a link to a joint.

with its long axis parallel to the joint, as shown in Figure 6-6. In the figure, the triangular prism represents any joint and the cube with the four dashed edges once again represents the possibility (but not necessity) of having actual edges there. Thus the rule states that either a C-link or the end of an an L-link may be attached to the free end of any joint. The parameters P_i in the homogeneous rotation matrix refer to components of the Plücker coordinate P of the joint. The parameters a, b, and c were introduced in the discussion of Figure 6-4. The coordinate frame for the second link has its origin at the center of the link and its z-axis pointing toward the joint. If the joint is located on the bottom of the first link (*i.e.*, if it is on the face in the $-z$ direction in the coordinate frame of the first link), then the x- and y-axes of the coordinate frame of the second link are parallel to and in the same direction as the corresponding axes in the first link's coordinate frame when the dyad is in its reference configuration (*i.e.*, when the joint variable is equal to zero). If the joint is not located on the bottom of the first link, then the x-axis of the second link's coordinate frame points in the same direction as the z-axis in the first link's coordinate frame.

The second rule to attach the second link to the joint connects an R-joint to the side of an L-link, as shown in Figure 6-7. If the joint is attached to the bottom of the first link, then in the reference configuration the z-axis of the second link's coordinate frame is parallel to and oriented in the direction opposite the first link's x-axis, while the second link's x-axis is parallel to and is oriented in the same direction as the first link's z-axis. Otherwise, the z-axes of the first and second link coordinate frames are parallel and share the same orientation in the reference configuration, while the x-axis of the second link is



$$R_2^1 \quad \begin{pmatrix} -P_1 & P_2 & P_3 & 2aP_1 - cP_3 \\ -P_2 & -P_1 - P_3 & 0 & 2aP_2 \\ -P_3 & 0 & P_1^2 + P_2^2 & (2a+b)P_3 - (b+c)(P_1^2 + P_2^2) \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Figure 6-7: Second rule to attach a link to a joint.

parallel to and in the direction of the joint.

The final rule for attaching the second link to the joint, and the final rule of the dyad grammar, is shown in Figure 6-8. It connects a P-joint to the side of an L-link, which is at one of four positions. The parameter p in the rotation matrices is equal to $P_1^2 + P_2^2$. The z-axis of the second link's coordinate frame is located along the long axis of the link, directed toward the joint. The x-axis is parallel to the joint, directed toward the first link.

If each separate case on the right-hand side of a rule is counted as a separate rule, fifteen different rules have been presented. These rules allow the formation of every possible dyad. Furthermore, no two dyads generated by different rules are identical or can be made to look the same solely by moving their joints. Only by rotating an entire dyad can two be made to look identical. The language of this grammar consists of ninety dyads, shown in Figures 6-9 through 6-11. The homogeneous rotation matrices, twists, and Plücker coordinates of the dyads are not shown, but are nonetheless an integral part of the dyads. This set of ninety dyads will be used as "building blocks" to construct arms in the next grammar.

6.4 Grammar to Generate Arms

Now the tools are available to generate all possible non-isomorphic robot arms. Dyads and their rotations and twists have already been generated. All which must be done in this grammar is to string together dyads and their associated numerical representations.

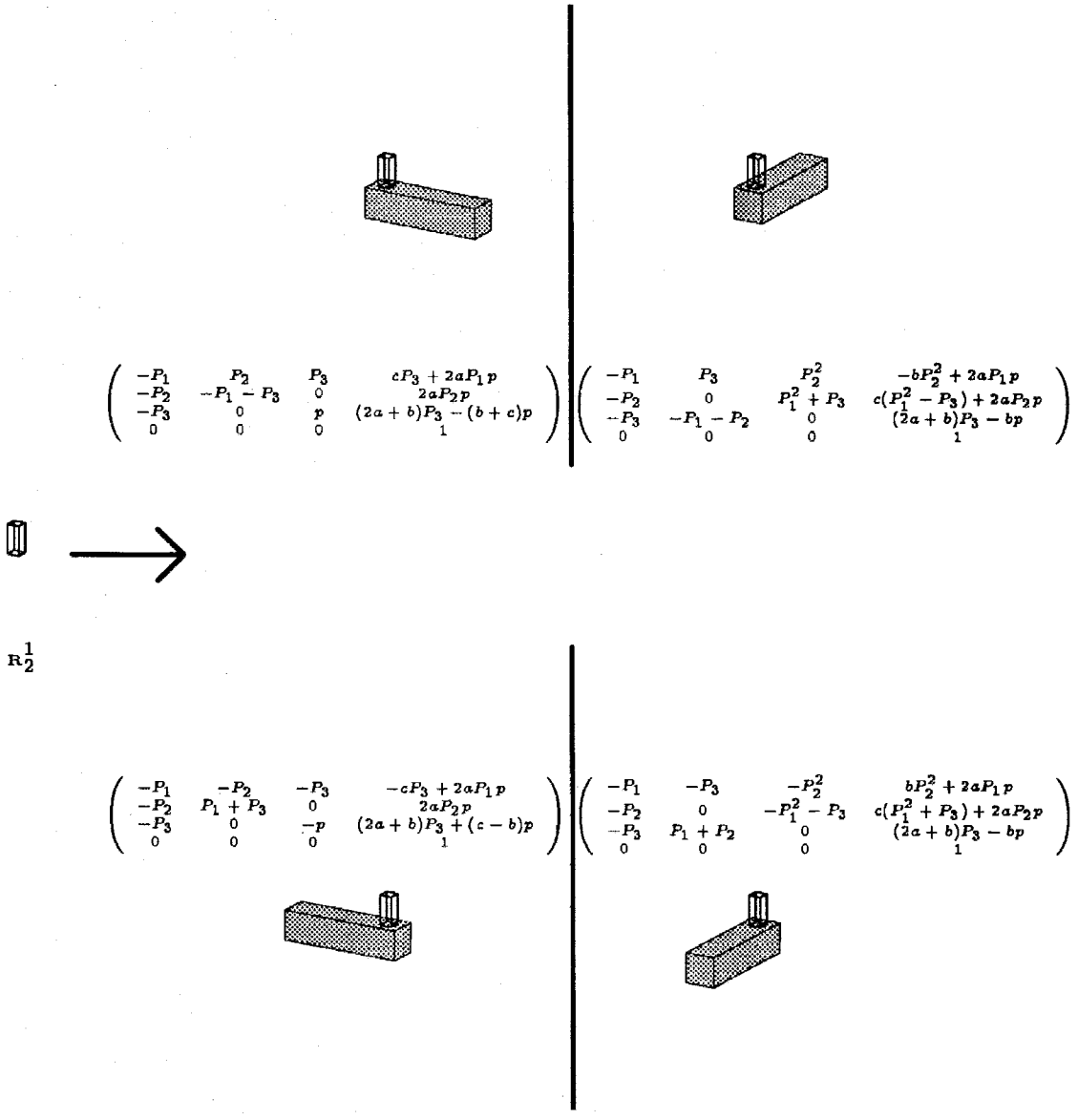


Figure 6-8: Third rule to attach a link to a joint.

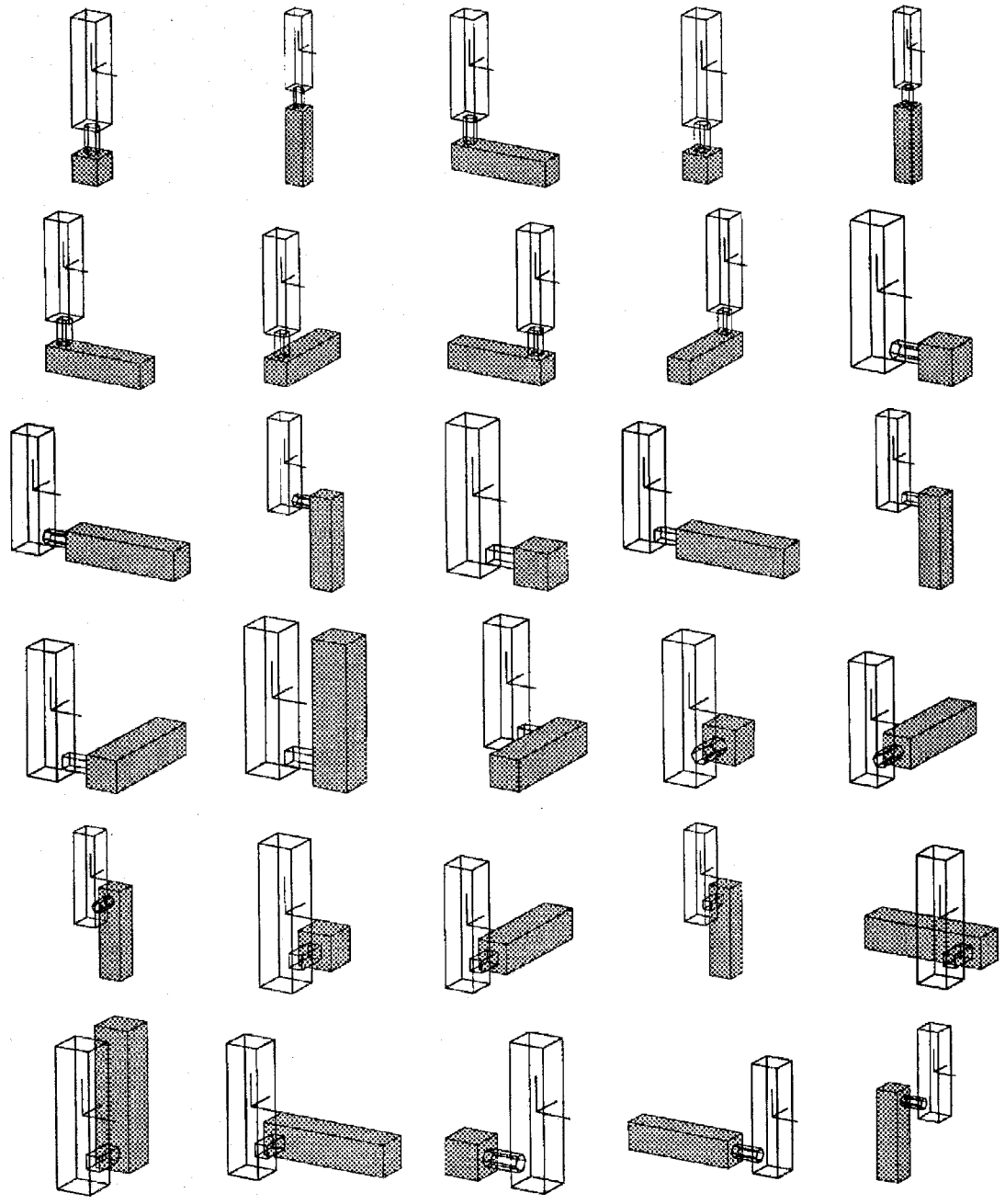


Figure 6-9: Dyads generated by the grammar.

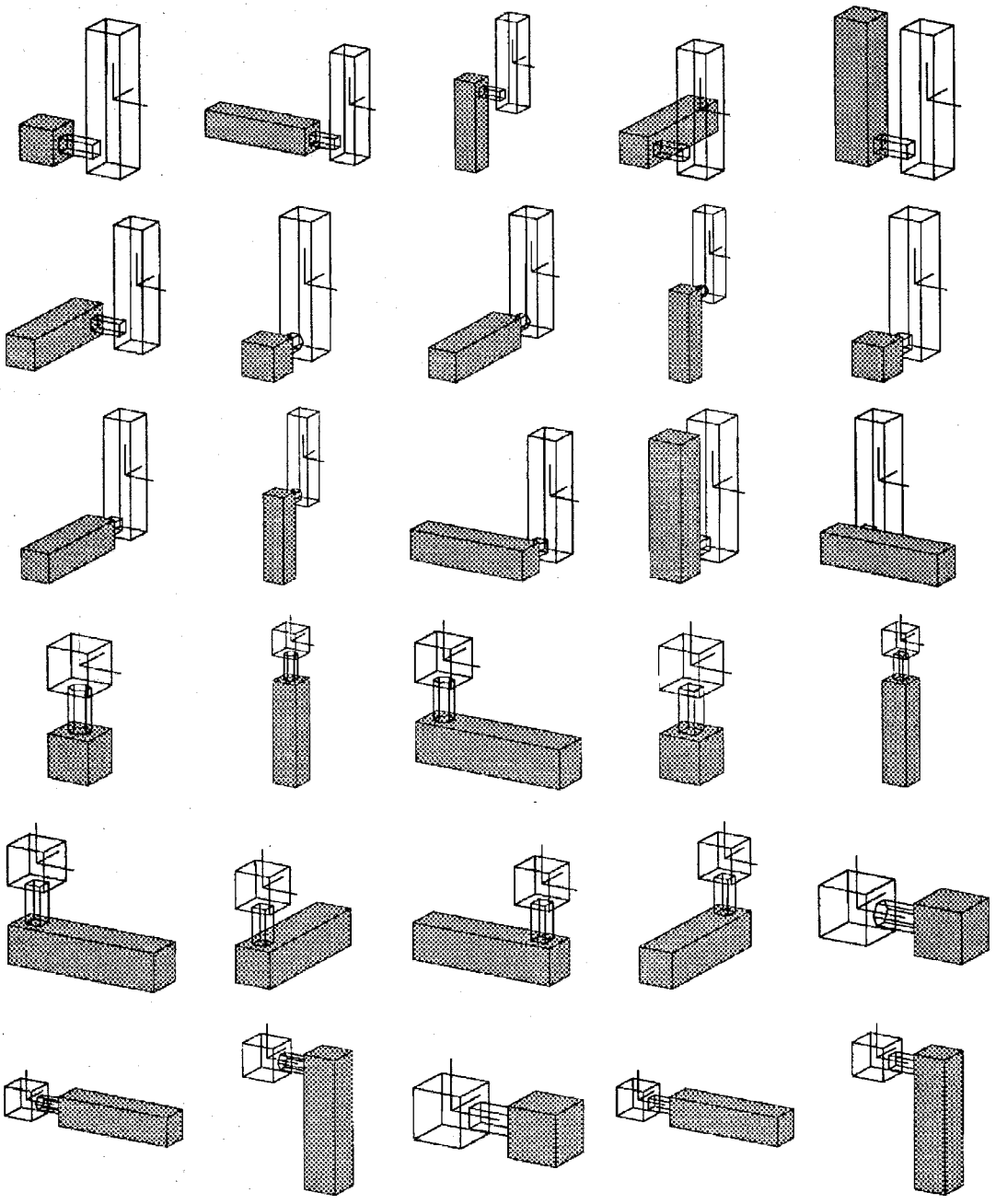


Figure 6-10: Dyads generated by the grammar, continued.

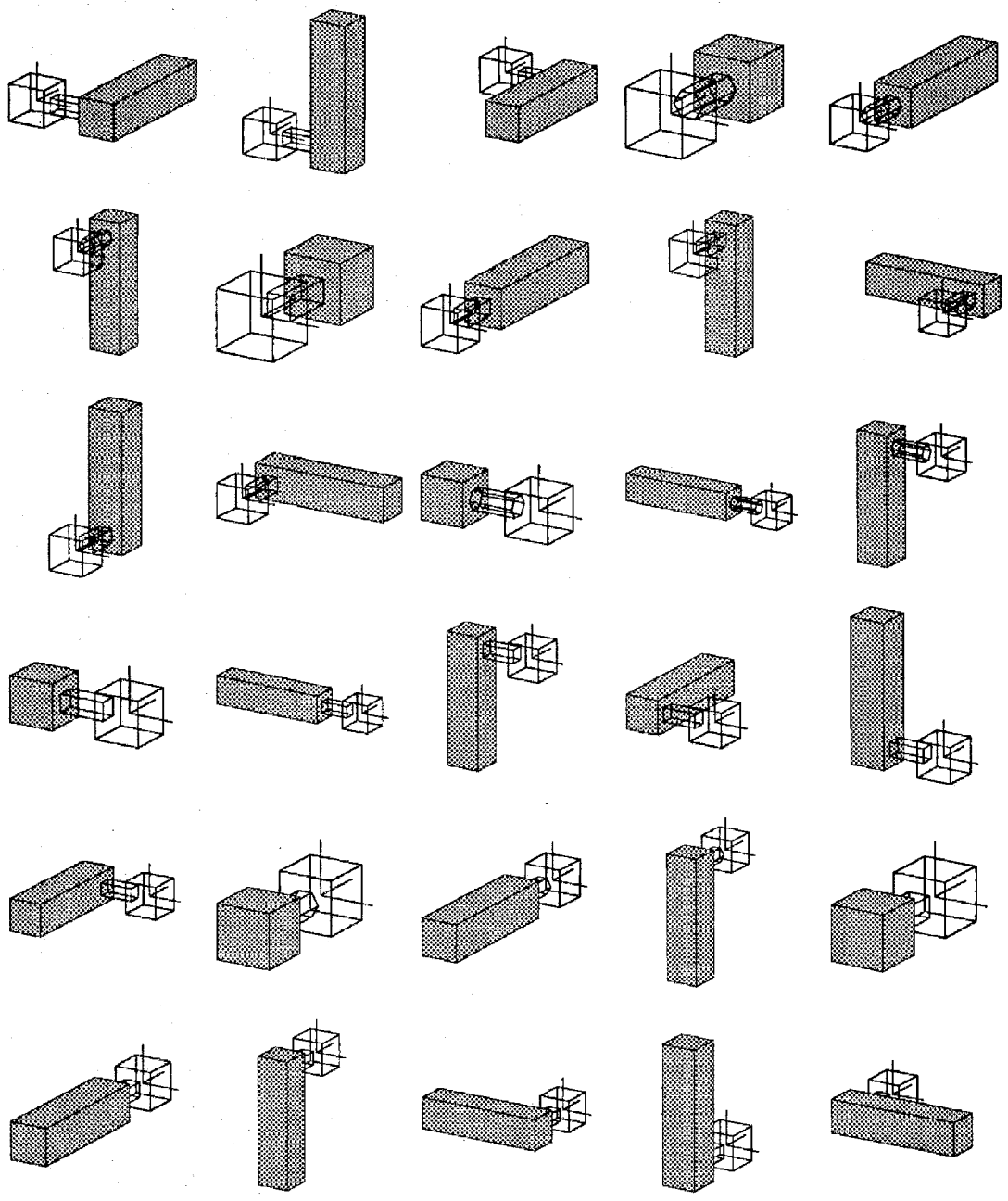


Figure 6-11: Dyads generated by the grammar, continued.

$\mathbf{R}^1,\ \xi,\ \mathbf{E}^1(\theta),\ \mathbf{T}^1(\theta),\ \mathbf{J}(\theta)$

Figure 6-12: The starting symbols of the arm grammar.

The starting symbols for this grammar once again include a coordinate frame. Also included are markers for several lists of data about the arm, as shown in Figure 6-12. A new element will be added to the end of each list whenever a link is added to the end of the arm. This data will be used to ensure that only non-isomorphic arms are generated, to evaluate the arm as it is generated, and to find the motion of the complete arm once it is finished. The lists contain the following data:

$\mathbf{R}_i^1$ The homogeneous transformation matrix between the base coordinate frame and the coordinate frame of the i^{th} link when all joints are in their reference configurations.

ξ_i The twist vector for the i^{th} joint when all other joints are in their reference configurations, expressed in the base frame.

$\mathbf{E}_i^1(\theta)$ The homogeneous transformation matrix between points in the base coordinate frame and points in the i^{th} coordinate frame, expressed in terms of the joint variables, in the frame of the i^{th} link.

$\mathbf{T}_i^1(\theta)$ The homogeneous transformation matrix between the base coordinate frame and the i^{th} coordinate frame, expressed in terms of the joint variables, in the base frame.

$\mathbf{J}_i(\theta)$ The $6 \times i$ Jacobian matrix for the arm up to the i^{th} joint, expressed in terms of the joint variables. This matrix is used to relate the velocities of the first i joints to the velocity of the $(i+1)^{st}$ coordinate frame.

The first rule places a dyad so that the dyad's base coordinate frame is coincident with the starting symbol, as shown in Figure 6-13. Not all dyads are allowed to be placed this way: this rule only places two classes of dyads: those whose joint axes are in the $-z$ direction in the base frame and whose second link is either a C-link or an L-link which has its long

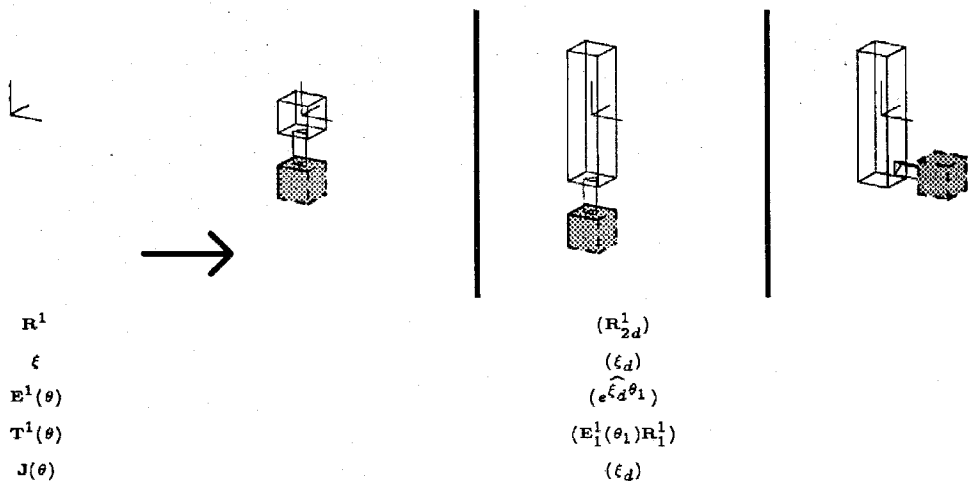


Figure 6-13: Rule to place a dyad on the starting symbol.

axis in the $-z$ or $+x$ direction in the base frame, and those dyads which have an L-link as a base link and whose joint axes are in the $+x$ direction in the base frame. This distinction is made in order to avoid generating arms which are isomorphic but rotated about an axis of the base frame. Note in the figure that the shaded links of the first two alternatives on the right-hand side of the rule have five solid edges and five dashed ones each (the hidden vertical edge in both alternatives is also solid). Since the solid edges in the figure must correspond to the actual edges of the second link, the only links which may be used are L-links which extend in the $-z$ or $+x$ directions in the base frame or C-links. As the first dyad is placed, the rule also starts putting data into each of the lists. The parameters ξ_d and $\mathbf{R}^1_{2d}$ are used to denote, respectively, the twist vector and the homogeneous rotation matrix belonging to the dyad being added in a rule. The variable θ_1 is the value of the joint variable of the first joint in the arm (which is the joint on this dyad). All of the one-joint arms in the language of this grammar are generated by applying this rule alone. These arms, shown in Figure 6-14 comprise the entire set of non-isomorphic configurations of two links and one joint.

Now three rules which can be used to lengthen the arm are introduced. Essentially, each rule aligns the coordinate frames of the first link of one of the dyads among the ninety in the library and the last link of the arm being generated. The first of these rules, shown in

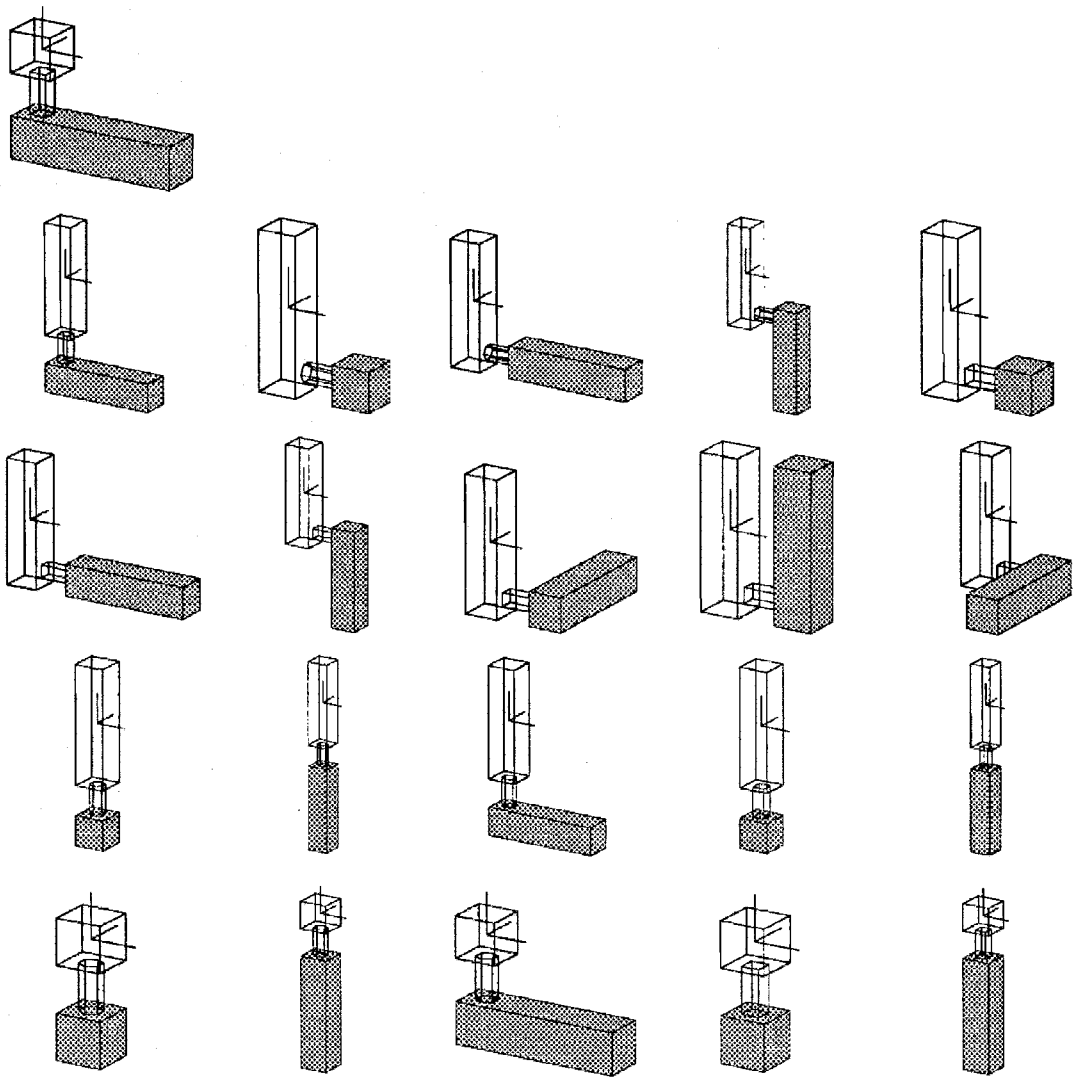


Figure 6-14: The 21 non-isomorphic dyads generated by the grammar.

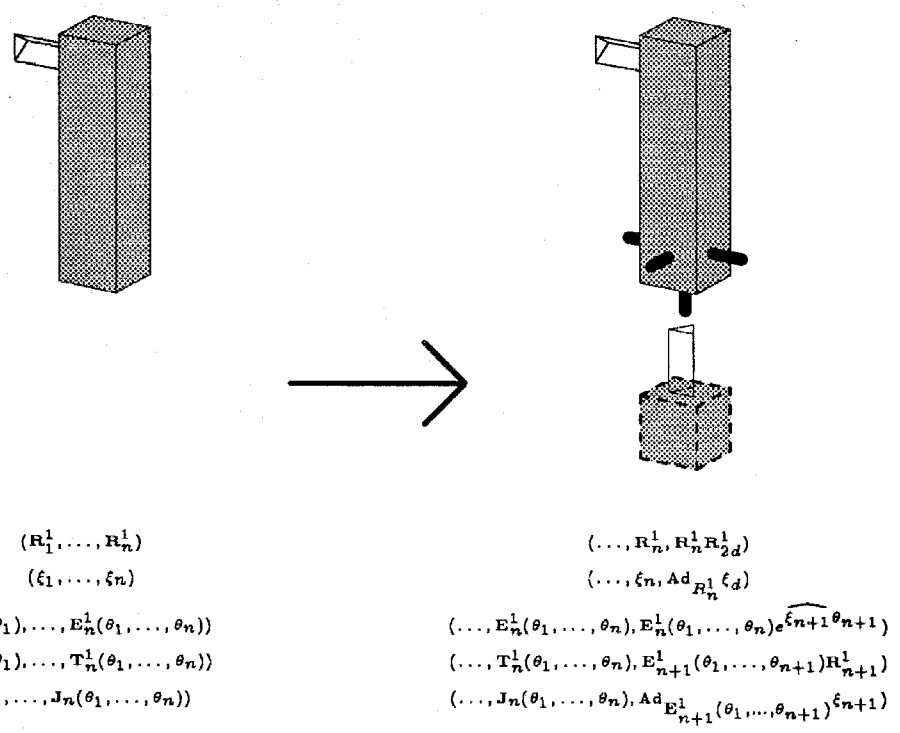


Figure 6-15: First rule to place a dyad on the end of an arm.

Figure 6-15, adds a new joint and link to the end of an arm which is terminated by an L-link with a joint on one of its long sides. The right-hand side of the rule in the figure represents a superposition of an L-link which is the first link of a dyad onto the L-link which is the last link of an arm. The thick lines at the end of the link denote the possible locations of the dyad's joint and the triangular joint connected to the cube with dashed edges denotes any joint attached in any way to another link. Thus the rule replaces the end of the arm with any of the dyads which have an L-link as their first link. As the arm is lengthened, the lists of data about the arm must also be lengthened, as shown in the figure. The variable n is the number of joints in the arm before the rule is applied; after the rule's application, the total number of joints in the arm is $n + 1$. One new piece of notation is introduced here: the *adjoint transformation* of a motion $\mathbf{R}$, denoted $\mathrm{Ad}_{\mathbf{R}}$. The adjoint is a 6×6 matrix which transforms twists from a base coordinate frame to a moving coordinate frame. If homogeneous transformation matrix $\mathbf{R}$ is formed from 3×3 rotation matrix R and 3-vector

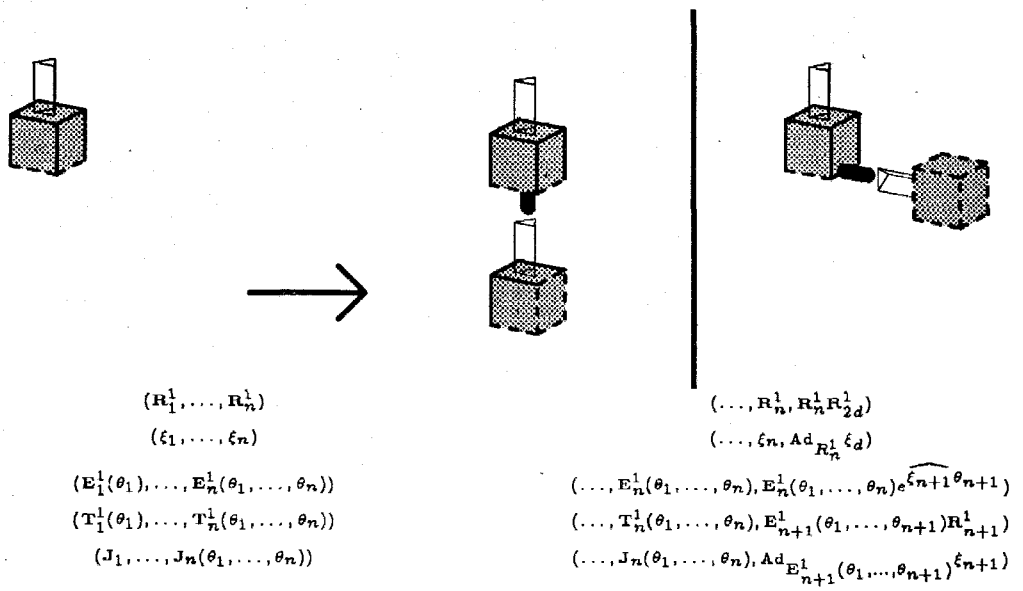


Figure 6-16: Second rule to place a dyad on the end of an arm.

p, then

$$\mathrm{Ad}_{\mathbf{R}} = \begin{pmatrix} R & \hat{p}R \\ 0 & R \end{pmatrix},$$

where, as defined above, $\hat{p}$ is a skew-symmetric 3×3 matrix whose elements are contained in vector p.

The second rule used to lengthen the arms, shown in Figure 6-16, adds a new joint and link to the end of an arm whose last joint is attached to the last link on one of the link's square sides. In this figure, the shaded cube with the dashed lower edge and the joint attached to the top represents either a C-link or an L-link connected to the previous link by a joint on a square face. On the right-hand side of the rule, the lines again represent the directions in which the joint of the dyad to be added may face. Again, the cube with five solid sides and five dashed sides in the first alternative on the right-hand side represents a C-link or an L-link whose long axis is in the vertical direction or one and only one of the horizontal directions, while the triangular joint attached to the dashed cube (in the second alternative on the right-hand side of the rule) again represents any permissible combination of joint and link. Implicit in the first alternative on the right-hand side of the rule is the

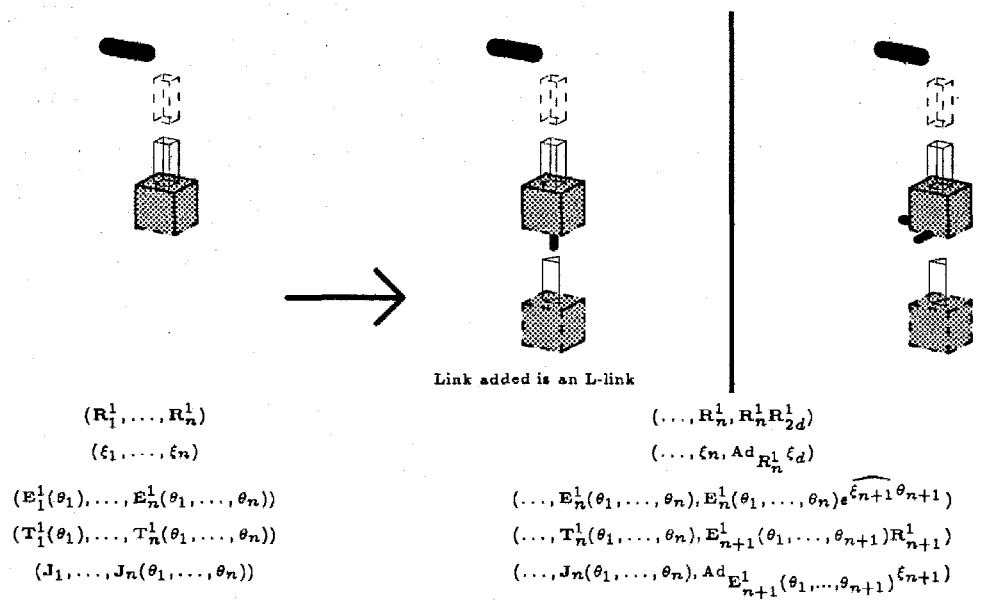


Figure 6-17: Third rule to place a dyad on the end of an arm.

fact that each link has its own coordinate frame. Therefore, the only horizontal direction in which an added L-link's long axis may lie is the +x direction, in the frame of the previous link.

An L-link's long axis may lie in one of the other three horizontal directions if and only if the arm satisfies an additional condition, as shown in the left-hand side of the third and final rule used to lengthen the arm, shown in Figure 6-17. The condition is that the final link in the arm shall not be able to rotate about its axis that is aligned with the axis of motion of the arm's final joint unless such rotations of 90, 180, or 270 degrees would produce different-looking arms. Furthermore, the final link must be either a C-link or an L-link which has the final joint attached to one of its square faces. If these conditions are satisfied, then an L-link whose long axis lies in the +y, −x, or −y direction (in the final link's coordinate frame) may be added, along with a joint aligned with the prismatic joint. In the first alternative on the right-hand side of the figure, those links are symbolized by a cube with two solid edges and the remaining edges dotted. Note that while a C-link satisfies the graphic criteria of the first alternative (since it has edges where the two solid lines are and fills the volume of the shaded cube), it is nonetheless not to be used in the context of

this addition: only L-links may be added. Additional joints, whose axes lie in the +y, −x, or −y directions, along with any new final link which matches one of the patterns in the dyad language, may also be added. These additions are shown in the second right-hand alternative.

The requirement that the final link not be able to rotate about its own axis that is aligned with the axis of motion of the arm's final joint unless such rotations of 90, 180, or 270 degrees would produce different-looking arms is represented graphically in Figure 6-17. The thick line at the top of the left-hand side of the rule represents a joint or link perpendicular to the final joint. The dashed P-joint represents the possibility of having one or more P-joints in the arm, aligned with the final joint, between the perpendicular link or joint and the final joint. If the final joint were an R-joint, then the final link would be able to rotate about its axis that is aligned with the joint's axis; therefore, the final joint must be a P-joint, as shown. Similarly, if there were an R-joint earlier in the arm which was aligned with all the joints following it, then the final link would be able to rotate about its axis (which would be aligned with the R-joint's axis and all the joint axes in between). Thus there can be no R-joints whose axes are aligned with the final joint's axis without there being some link or joint which is not aligned with those axes in between. Since the base of the arm could also be rotated, there must be at least one joint or link which is not aligned with the final joint in order to keep the final link and joint from rotating about their common axis as the arm's base is rotated. Combining these conditions produces the graphic criteria of the figure.

Now that all of the grammar's rules have been presented, it may be shown that its language contains all non-isomorphic robot arms.

Proposition 6.1 *Any two robot arms from the language of the grammar described above, derived using different sequences of rule application and/or different choices of dyads to add using the rules, are non-isomorphic* (i.e., *they can not be made to look the same by rotating or translating their bases and/or moving their joints). Furthermore, every non-branching robot arm constructed from the joints and links used in the grammar (and in which no L-link has two joints attached at one end) is isomorphic to an arm in the language of the grammar.*

Proof. It will first be proven by contradiction that no two arms in the grammar's language are isomorphic. Then all possible joint and link combinations are considered and shown to be able to be constructed using the rules of the grammar.

Assume that two arms in the grammar's language are isomorphic and are generated by different sequences of rule applications and/or different choices of dyads added by the rules. Since the two arms are isomorphic, their sequences of link and joint types must be the same, they must have identical workspaces, and one must be able to be moved to look like the other no matter what the other's configuration. The two arms' lengths must be the same and the sequences of rules applied and dyads chosen must be identical until some specific rule application or choice of dyad. The two partially-derived arms must be able to be made to look the same solely by changing the joint variables on one of the arms. Either the last joint of the partially-derived arm must be moved, an earlier joint must be moved, or multiple joints must be moved. If only the last joint must be moved to make an arm look like the other, then there must be a dyad added by the final rule which can be transformed into a different dyad solely by changing its joint variable. If an earlier joint must be moved to make an arm look like the other, then there must have been at least one dyad added by a previous rule which looks the same when its joint variable is changed. Furthermore, the final dyad added must be able to be transformed into another dyad (the dyad at the end of the other arm) solely by rotating about and translating along the axes of any earlier joints whose joint variables can be changed.

As noted in the previous section, no dyad in the language of dyads can be made to look like another simply by changing its joint variable. Instead, an entire dyad must be rotated about the z-axis of its base frame in order to make it look like another. This property can be ascertained by inspection of the dyads in the language, shown in Figures 6-9 through 6-11, or by consideration of the dyad grammar's rules. Therefore, the different dyads at the ends of two partially-derived isomorphic arms must be able to be rotated as units about the z-axes of the next-to-last links and be made to match up.

If the next-to-last links were L-links attached to the previous links by joints on their long faces, then rotating about the z-axis of one of the next-to-last links (which is the link's long axis) would move that joint on its rectangular face, so the previous links in the arm would not match up with those in the other arm unless the *previous* dyad (the one containing that joint) could be rotated about the same axis and match up with itself or some other dyad. But since the final dyads in the partially-derived arms are the first which are different, that previous dyad must also match up with itself when it is rotated. Since that previous dyad

does not lie along a single axis, it can not be rotated about that axis and be made to look the same. Therefore, the final dyads in the two partially-derived arms can not be added using the first rule to lengthen arms, pictured in Figure 6-15.

If the next-to-last links were attached to the previous links by joints on their square faces, then they would have to be added in the second or third rules for lengthening arms and, in order to match up, the must be able to rotate about the axes of those joints on the square faces. The dyads added in the third rule may not rotate about that axis because of the requirements of the left-hand side of the rule. Therefore, neither of the final dyads in the two partially-derived arms can be added using the third rule. Furthermore, rotating the dyads added using the second rule would only produce the same dyads or the dyads which are added in the third rule. Thus the second rule also can not be used to add the final dyads. Finally, no dyad added in the starting rule of the grammar can be rotated to match another starting dyad. There are no other opportunities to add dyads, so therefore it must be concluded that two arms which are formed using different sequences of rule applications and/or different choices of dyads to add using the rules are non-isomorphic.

To show that every non-branching robot arm constructed from the grammar's set of joints and links is isomorphic to an arm in the grammar's language, consider the set of all possible links with one or two joints attached. Arms could be defined solely by listing the order of their links, where and what types of joints are attached to each link, and the relative orientations of successive links. In fact, this method of defining arms is used by Chen [16]. Thus if the grammar can generate every member of the set of links with joints attached, then it can generate every possible arm.

First consider the set of all possible base links and first joints. An arm may start or end with either a C-link, an L-link with a joint attached on a square face, or an L-link with a joint attached on one end of a rectangular face. These are the only three unique configurations which a base link may have: any base link is isomorphic to one of the configurations. The first rule of the grammar adds links and joints in exactly these three configurations. Furthermore, the rule adds both R-joints and P-joints, so all possible base links and first joints can be generated by this grammar. Additionally, the rule adds second links in those three configurations as well. If the joint is attached to a square face on the first link then only one of the four possible orientations of an L-link with the joint attached

to one of its rectangular faces may be used for the second link. The other three orientations may be obtained by rotating the base link about the axis of the joint. On the other hand, all possible orientations are allowed if the joint is attached to a rectangular face on the base link because the base link can not be rotated without looking different.

The same reasoning lies behind the other rules of the grammar. If the arm or a joint can be rotated and still look the same as it originally was, then only certain orientations of joints perpendicular to the last joint or of L-links attached on their rectangular faces may be added. Conversely, an arm which can not be moved and made to look the same may have all possible combinations of link and joint added to it. Those combinations, enumerated by the dyad grammar, include four different orientations of an L-link whose long axis is perpendicular to a P-joint, and one orientation each of an L-link with long axis perpendicular to an R-joint, an L-link with long axis aligned with an R- or P-joint, and a C-link attached to an R- or P-joint. The other three orientations of the latter combinations need not be used because they look exactly the same or, in the case of the L-link perpendicular to the R-joint, can be transformed into the other configurations solely by rotating the joint which is added. Therefore, links are added by the rules at every possible relative orientation to the final link of the partially-completed arms. Furthermore, the final link in a finished arm can also be added at any possible orientation to the next-to-last link; and the final joint can be attached to a C-link, the square face of an L-link, or the rectangular face of an L-link.

Now all that remains is to show that the grammar can generate all possible links which have two joints attached to them. The first rule used to lengthen an arm applies to all L-links connected to the previous link by a joint on a rectangular face. The rule allows the attachment of joints at any position on the other end of the link, so the rule generates every L-link with the first joint on a rectangular side. The second rule used to lengthen arms applies to all links connected to the previous link by a joint on a square face. The third and final rule applies to the subset of those links that are on an arm which can not be moved and still look the same. If the link is a member of that subset, then joints may be attached at any position on the other end of the link. If the links is not a member of the subset, then it can be moved so that it looks like the joint connected to the next link was added in one of the other three configurations. Thus these rules generate every link with the first joint on a square side. Therefore all possible links with two joints connected to them can be

Number of joints	Exhaustively enumerated arms	Grammar-generated unique arms	Grammar-generated L-link, R-joint arms
1	100	21	4
2	10,000	753	28
3	1,000,000	~28,000	196
4	100,000,000	~1,000,000	1372
5	10,000,000,000	~40,000,000	9604
6	1,000,000,000,000	~1,500,000,000	67,228

Table 6-1: The total number of possible arms, the number of unique arms, and the number of unique arms constructed from only L-links and R-joints which have a given number of joints.

generated by the grammar. It can thus be concluded that every non-branching robot arm constructed from the joints and links used in the grammar (and in which no L-link has two joints attached at one end) is isomorphic to an arm in the language of the grammar. ■

The above rules form the core of the arm generation grammar. Additional, constraining, rules could be added to the rule set in order to generate arms of a particular style or for a particular purpose. Constraining rules are discussed in [12]. While the grammar defined by the above rules is a context-sensitive one, constraining rules add even more context which the left-hand sides of rules must satisfy in order to be applicable. For example, the above rules could be constrained by changing them so that wherever a generic joint appears in the rules (represented above by triangular prisms) it is replaced by an R-joint. Then all of the generated arms would have only R-joints and no P-joints. Similarly, the rules could be modified so that the only links are L-links. Then all the links in the generated arms would be L-links. Some arms of various lengths constructed using only R-joints and L-links are shown in Figures 6-18 and 6-19. These types of rules could be valuable if there were only a small number of a certain type of module available. They also allow the number of arms generated to be cut drastically, as discussed below, while still providing a range of kinematic behaviors.

Using only the eight original rules given above (where separate cases on the right-hand side of a rule count as separate rules), a huge number of unique, non-isomorphic arms can be produced. Table 6-1 gives the number of unique arms (all of which are generated by this grammar) which have a given number of joints. It also shows the number of unique arms

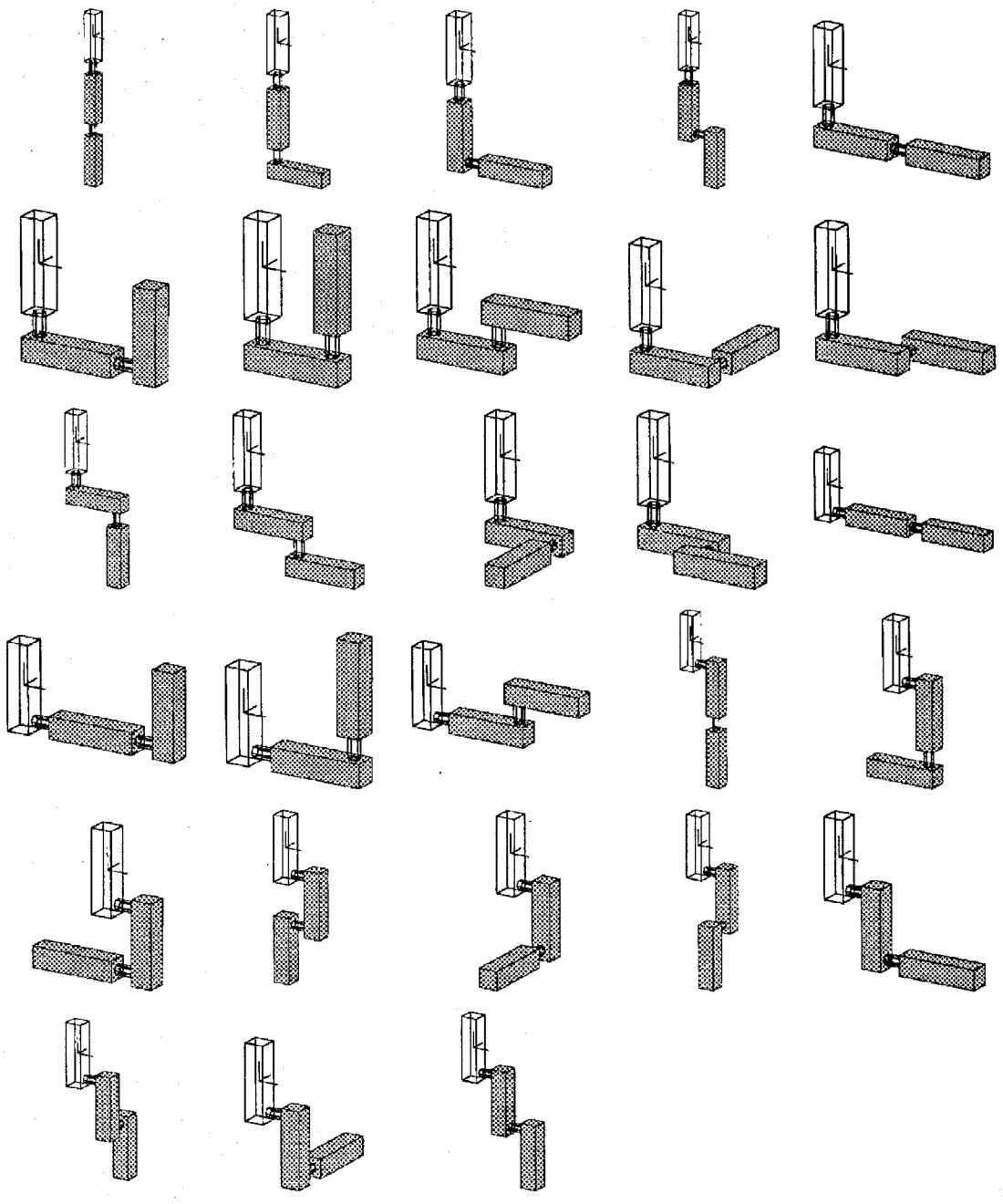


Figure 6-18: All of the arms generated by the grammar when using only three L-links and two R-joints.

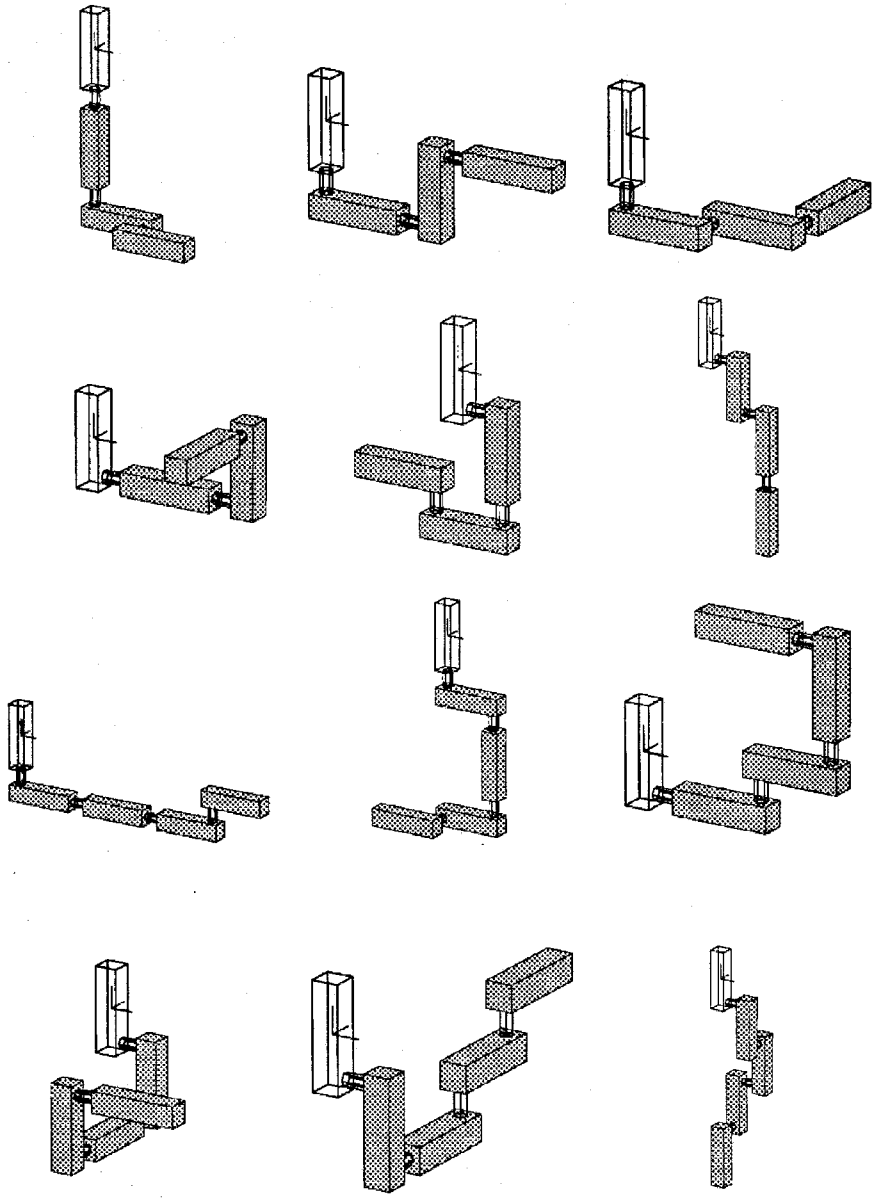


Figure 6-19: Some arms generated by the grammar using only four of five L-links and three or four R-joints.

which may be formed solely of L-links and R-joints and the number of arms which would be generated by a brute-force enumeration technique. The latter technique would choose one of five attachment points on the first link, choose one of two joints, choose one of the two styles of links for use as the second link in the arm, choose one of one of five attachment points on the second link, choose one of the five attachment points on the other end of the second link (or simply one of the other five points if the second link is a C-link), choose between the two joints, etc. Note that this brute-force enumeration technique is "smart" in that it only looks for joint attachment points on the opposite ends of L-links and only initially checks five of the six attachment points on the C-link (although the grammatical method only checks one of them). Less intelligently thought out enumerations could generate many more candidate arms. Even with this brute-force technique, a huge number of arms would have to be evaluated (and remembered) in order to find only the non-isomorphic ones. The numbers in the last column of the chart agree with those found using Chen and Burdick's method [16]. Chen and Burdick do not give the numbers of unique arms using both types of links and joints, enumerated here in the third column of the table.

Note that all L-links in a generated arm which are attached to two joints have the joints at opposite ends of the link (the same is true for an arm generated by exhaustive enumeration as described above). Rules could be written which would allow the attachment of two joints to connection points on the same end of an L-link, but there is no compelling reason to do so. All of the arm's kinematic properties would be duplicated by an arm in which the L-link under consideration was replaced by a C-link. Allowing two joints on the same end of an L-link would constrain the motion of the actual arm since the unused end of the L-link would interfere with other links.

Constraining rules, such as the one restricting the alphabet to L-links and R-joints, need not be confined to the geometric domain. A designer could, for example, decide that he or she wants only arms which are non-redundant, *i.e.*, for which the number of degrees of freedom of the final link is equal to the number of joints. The amount of redundancy of an arm (which is equal to the number of joints minus the number of degrees of freedom of the final link) is equal to the dimension of the null space of the Jacobian matrix **J**. The dimension of the null space is the number of linearly independent 6-vectors whose product, when multiplied by the Jacobian, is the zero vector. Before adding each dyad, the null

space of the Jacobian can be checked to see if it has dimension greater than zero, and if it does then it can be concluded that the dyad does not match the left-hand side of the non-redundant arm rule.

Constraining rules on other aspects of the arm, such as its cost or its size, could also be included. An attempt can also be made to impose constraints in order to make arms which are able to complete some task (since performing tasks is, after all, the purpose of a robot arm). An arm's workspace is the set of all points which can be reached by the end of the arm. Since the transformations $\mathrm{T}_n^1(\theta_1, \ldots, \theta_n)$ between the base frame and the frame of each link (*i.e.*, the forward kinematics) are defined, the position of the end of the arm (or of a hand attached to it) can easily be found. Unfortunately, there is no way, in general, to automatically find the inverse kinematics of the arm (the joint variables necessary to reach a specific point). Nonetheless, the grammar's rules could be constrained to generate only arms in certain styles which have proven to provide large workspaces, such as elbow manipulators, which were proven by Paden and Sastry to have large, well-connected workspaces [94].

The grammar could also be used for experimenting with arm setups. Many arms of different styles could be generated (for instance, by choosing randomly from among the rules available) and estimates of their workspaces could be made using their forward kinematics. Then those workspaces could be evaluated for size, connectedness, ability of the arm to manipulate an object near a certain point, or other properties in order to determine which rules produce the most optimal arms (for example, the Taguchi method could be used to test which rules most robustly produce arms with good workspaces, as discussed in Section 5.3.2). Then the grammar could be modified with constraining rules which restrict it to the generation of a family of arms which are in the same style as the optimal arms (and therefore might be the most optimal for the task).

6.5 Comparison With Existing Methods

As can be seen from the comparison in Table 6-1, the grammatical method of generating modular robot arm configurations enjoys a distinct advantage over any brute-force, exhaustive enumeration technique. Furthermore, this method compares favorably with the technique discussed above which uses symmetry groups of assembly incidence matrices. By

using a grammar to generate arms, it is guaranteed that no two arms which are isomorphic will be generated. Therefore, it is unnecessary to do any checking of the completed arms, which affords this methodology a computational advantage. By generating arm kinematics along with the configuration, candidate arms can be evaluated as they are generated, even if the arms are not yet of the required length, rather than having to wait until all of the full-length arms are generated before getting into the kinematics. Thus the grammar allows increased flexibility with less computation than existing methods.

This grammar also demonstrates some of the general advantages of using grammars to solve engineering problems. The parallel rules of many grammars can help to speed the generation and evaluation of design alternatives. Emergent properties, like the shapes of the arms or forward kinematic descriptions of the ends of the arms, might not be foreseen by the designer, but become evident when grammars are used. The early stages of the design process can be structured by grammatical formalisms in such a way as to significantly ease the designer's workload. By giving some thought to the rules used to manipulate shapes and properties, generation and evaluation of designs can be done relatively easily.

Chapter 7

Related Work

7.1 Introduction

Much other work has been done in many of the fields discussed in this dissertation. In the sections below, several applications of shape and engineering grammars will be discussed. Additionally, references to other discussions of pertinent issues from the preceding chapters will be presented.

7.2 Shape Grammars

The original descriptions of shape grammars, some of their properties, and the measurement of the aesthetics of the shapes produced by them were published by Gips and Stiny in [45, 112]. A short introduction by those authors to shape grammars and parametric shape grammars may be found in [117]. Mitchell gives a very thorough and interesting introduction to the use of formal grammars in architectural design, along with many examples [79]. Another overview of the many possible applications of shape grammars is presented by Stiny in [122].

7.2.1 Applications of Shape Grammars

Early shape grammars were mostly conceived to generate art [129], so there was some study into the aesthetics of grammar-generated shapes [115]. However, the scope of shape grammars soon increased. Grammars have been devised for Chinese lattice design [114], Hepplewhite-style chair backs [61], Indian formal gardens [132], Japanese tea rooms [62],

and bungalow-style houses [24]. Artificial landscapes can be drawn on computers using grammatical rules [39] and houses in Frank Lloyd Wright's Prairie style can be designed [69]. Wooden block puzzles can even be designed and solved [4].

Shape grammars can also be used for analysis of existing designs. For example, in [36] Flemming analyzes a building with a seemingly unstructured design. Flemming created a grammar with a relatively small number of simple rules that would generate a family of building designs, one of which was that of the building under consideration. When considered in light of the grammar's rules, the deep structure of the building's design becomes obvious. Thus, creating a grammar can help a designer understand a particular problem more completely.

In [79, 116, 129, 131], Stiny, Mitchell, and Gips present a grammar that generates plans for villas in the style of the Roman architect Palladio. Detailed plans can be generated and all allowable plans with a certain floor area can be enumerated. Furthermore, the plans may be evaluated based on some aesthetic and functional criteria presented in Palladio's writings on architecture. After all the plans are ranked, they are compared to the plans of villas which were actually laid out by Palladio. Not only do all of the villas designed by Palladio belong to the language, they tend to be the ones which are ranked toward the top of the list. This example emphasizes the fact that grammars are very useful for enumerating and ranking solutions to design problems.

7.2.2 Modifications of Shape Grammars

Both rules and alphabets of shape grammars can be modified in order to create new grammars. In [63], Knight presents a method of modifying rules by substituting one shape for another. The resulting new rules are functional analogs of the original ones, but use the new shapes instead. This modification method was used by Knight in [67], where a grammar which made patterns like those on a particular style of Greek pottery had been defined. New shapes were substituted into the rules, which then generated patterns like those on pottery from a later period. The temporal procession of designs can often be mimicked by substituting modified shapes in some design rules.

Another way of modeling the change in a design is by changing the shape relations in a grammar's rules. As discussed in Section 5.2.3, by modifying the spatial relations of some

shapes in the grammar which generates house layouts in the style of Frank Lloyd Wright, Knight's new grammar can generate layouts in one of his later styles [64, 65, 66].

7.2.3 Rules and Representations in Shape Grammars

Great attention must be paid to shape operations if they are to have desirable properties like those in Chapter 3. Most importantly, shapes should be maximal sets of points, lines, or planes which have infinite numbers of subshapes, as discussed extensively by Stiny [120, 121, 123, 125]. Algorithms have been devised by Krishnamurti to heed this advice when shapes are being operated on by a computer [70]. If shape operations are defined appropriately, then sets of shapes can form Boolean algebras, as asserted by Stiny in [126] and discussed in detail in Section 3.2.

Some attention has been paid to the transformations which must be used to make the left-hand side of a rule match up with a shape. Spatial relations in grammars are discussed by Stiny in [113, 118]. Krishnamurti gives an algorithm for determining all possible instances in which a rule can be applied to a shape when a given set of transformations is allowed [71]. However, the consequences of allowing only a limited set of transformations have been explored more deeply with regard to engineering grammars by Carlson, Woodbury, *et al.* [12, 13, 14, 140].

7.3 Engineering Grammars

Engineering grammars generally use alphabets from multiple domains and have rules which operate in parallel. The use of grammars with parallel rules has been discussed by Stiny in [119, 124, 126]. There have been some interesting applications of engineering grammars and they are well-suited for searching design spaces. Much work has gone into defining representations for physical objects and properties in engineering grammars, but the interrelation of form and function in design remains an overriding concern of developers of grammars.

7.3.1 Applications of Grammars in Engineering

Many unique engineering grammars have been devised. Members of one family of grammars can be used to generate topological models of three-dimensional objects, as discussed by Fitzhorn and Longenecker. Engineering grammars can represent the faces, edges, and corners of any solid which can be deformed into a sphere [33]. A later grammar can represent the topology of any realizable shape [72, 74]. Other grammars in the same family use parametric graphs to generate all possible constructive solid geometry and boundary-representation solid models [35].

Other engineering grammars, devised by Finger and Rinderle, are able to construct bond graphs which are used for modeling the behavior of electromechanical systems [31, 32]. Electronic circuits or computer logic gates can also be constructed with grammars [85, 86, 100]. A grammar can also be used to represent and combine features on an object to be injection molded. Then another grammar can be used to parse that representation and design a mold for the part [104].

7.3.2 Structures and Representations

Many different structures have been proposed to try to solve or mitigate some of the problems, discussed by Fenves and Baker in [30], which arise when using parallel grammars in multiple domains. For example, structure grammars, defined by Carlson, Woodbury, *et al.* are shape grammars in which emergent shapes can be recognized and a predefined set of transformations can be used [12, 13, 14, 140]. Augmented topology graph grammars, defined by Pinilla, Finger, and Prinz, use graphs whose edges are labeled with relationships between the nodes [96], while Rinderle's attribute grammars couple properties to objects [101]. Chuang and Henderson's grammar for representing a solid's topology even makes a double translation, from boundary representation of the solid into a graph, and thence into a web [19].

Engineering grammars should be able to represent both the structure of a design and how it functions. Since these two properties are interrelated, care must be taken when devising rules for engineering grammars so that proper form/function relationships are maintained. Several researchers have explored these problems [102], while Hoover and Rinderle also discuss the sharing of functions between different parts of an object [52]. The structure

of an object is like the syntax of a sentence: if well-defined, it can be understood. The function of an object is more akin to the semantics, or meaning, of a sentence; it may not be immediately obvious. It is partially for this reason that extracting functions from a design is like parsing a sentence, as discussed by Murakami and Nakajima [87, 88, 89].

7.3.3 Searching the Design Space

Several researchers have described features required of computer tools for design [136, 139]. Others have discussed the uses of grammars in the design process [130] and possible ways of presenting the designer with the results of a grammar-generated search of the design space [68]. Creativity and the types of production rules which mimic it are discussed in [21]. Since the products of design are claimed to be equivalent to the outputs of a Turing machine, and since a Turing machine's behavior may be duplicated by a general grammar, some conclusions about the complexity of a design problem might be drawn from the complexity of the grammar used to solve it, as discussed by Fitzhorn and Longenecker [34, 73].

Only a few researchers have described ways to search design spaces using rule-based systems or grammars. In [9, 10, 11], Cagan *et al.* describe the shape annealing process, which checks multiple paths through the design tree in order to find more optimal ones. Another way of directing a search through a design tree, described by Navichandra and Marks, is to evaluate all the alternative branches at every node and to follow the one which produces the most optimal partially-completed solution to the design problem [93], though this strategy may not work in all cases. While the generate-and-test search method is always available [37], it has been concluded by Fong, in the natural language parsing field, that a much better way of finding optima in a short time is to, if at all possible, apply rules using a strategy which aims to reduce the number of applicable productions in the following step [38]. If, after applying rule a, there are only two or three ways to apply rule b, then that is better than applying rule b in a dozen different ways, then applying rule a to each of the resulting strings.

7.4 Other Grammars

Picture languages, which are shape production systems used for shape representation and recognition, are similar to shape grammars and are described by Narasimhan in [92], but have fallen into disuse as the shape recognition field is taken over by neural network analyzers. In another domain, grammars which generate or parse music have been created by Roads [103]. Grammars have also been used extensively as the basis for graph rewriting systems, as discussed by Ehrig and Nagl [29, 91], because graphs are useful representations of finite automatons, which are related to Turing machines in that they are another representation of an ideal computer.

7.5 Expert Systems

A basic introduction to expert systems may be found in [7, 57]. The major concepts and assumptions underlying expert systems are discussed in [50]. See Taylor [135] for an interesting introduction to expert systems and the broader field of artificial intelligence.

7.5.1 Desirable Features of Expert Systems

Different expert systems excel at different tasks. Adeli discusses the types of tasks well-suited to solution by an expert system in [2]. Existing expert system environments are compared in [3] and recommendations are made regarding the best utilities for solving particular types of problems. Methods of solving particular problems, and features required in expert systems which are used to solve the problems, are discussed in [77]. Other strategies and tools for solving particular types of problems, for searching a design space, and for reasoning using expert system knowledge, are compared in [110, 111].

7.5.2 Engineering Applications

There have been many applications of expert systems to engineering design. For an introduction to expert systems from an engineering perspective, see Dym [27] or Taylor [135]. Different forms of design knowledge representation are discussed by Dym and Levitt in [28] and a frame-based expert system for checking fire code compliance in architectural plans is introduced. A general expert system which builds truthful statements for use in design by

drawing conclusions from a knowledge base is presented by Dietterich and Ullman in [22]. Some examples of expert systems in various fields may be found in [135].

The structure and function of objects are considered and related in several expert systems. In [25], Doyle presents a system which takes the temporal behavior of an electromechanical system as an input and outputs theories about how the system works and the components which could be used to construct it. Mechanical devices and their functions are represented in a knowledge base in [26]. These devices can be modified by the expert system to make them function in a desired manner. A rule-based system presented by Hirschtick [51] can extract design features and make suggestions to the designer of ways to improve the design. Designs can also be modified by an expert system of Murthy and Addanki [90] that reasons from physical principles to make its conclusions. Finally, a well-developed expert system for laying out paper paths through photocopiers has been described by Mittal, Dym, Morjaria, and Araya in [80, 81, 82, 83].

7.6 Shape Operations

The claim that regular sets under regularized operations form a Boolean algebra is proved by Requicha and Tilove in [99]. This claim can be used to prove some of the results in Section 3.2.

The Minkowski sum and difference operations are presented in a formal mathematical manner by Matheron and Serra in [76, 108]. A more understandable introduction by Haralick *et al.*, along with some applications to image analysis, may be found in [49]. The formalism for tracings and the operations on them, introduced in Section 3.3, is presented in its entirety by Guibas, Ramshaw, and Stolfi in [48], although none of the theorems set forth in the paper are proved there. The Minkowski sum and difference are used to solve various problems by Ghosh and Mudur in [41, 42, 43, 44]. These papers also touch on the possibility of using Minkowski operations in production systems, but do not actually present any such systems. The vector space nature of some shapes, developed fully in Section 3.4, is hinted at, but never claimed or proved, by Ghosh in [43].

The Minkowski operations have been used in some interesting ways. In [105, 106], Rossignac and Requicha use them to automatically round corners and produce fillets on

solid models by alternately taking the Minkowski sum and difference of a solid model with a small sphere. Solid objects may be interpolated, or "morphed," using a weighted Minkowski sum, as discussed by Kaul and Rossignac in [59]. A polyhedron $\mathcal{A}$ can be smoothly deformed into polyhedron $\mathcal{B}$ as time parameter t moves from zero to one by defining the resulting shape at time t as $(1-t)\mathcal{A} \oplus t\mathcal{B}$, where multiplication of an object by the real number t is equivalent to scaling it by a factor of t.

The Minkowski sum was first used in the robot path planning domain by Lozano-Pérez and Wesley in [75]. In the configuration space of a robot, both the robot and obstacles appear as polygons. In order to move and avoid the obstacles in the physical world, the polygonal representation of the robot must move through the configuration space without touching the polygonal obstacles there. However, the problem of finding a path for a polygon to move through a space with polygonal obstacles is difficult to solve. Therefore, the robot's polygonal representation is Minkowski summed with the obstacles in the configuration space. The edges of the resulting larger polygons form the locus of points that would be swept by the coordinate frame of the robot if it moved through the configuration space in its original form while touching the obstacles. Thus the problem is reduced to one of finding a path for a point through the new configuration space with the larger obstacles. This problem is much easier to solve graphically or computationally than the original one.

7.7 Formal Language Theory

The standard reference work on language theory and computation is by Hopcroft and Ullman [53], although the book by Moll, Arbib, and Kfoury [84] can also serve as a good introduction. The Chomsky hierarchy was first proposed by Chomsky in [18], and the expressive powers of grammars at the various locations in that hierarchy is discussed by Grune and Jacobs in [47]. A general form for rewriting systems, termed a Post production system, is presented in [97], where Post also proves that many systems with other forms can be reduced to this form. A more understandable proof is presented by Minsky in [78]. Grammars and a number of other rule-based production systems, all of them subclasses of the family of Post production systems, are described and compared by Gips and Stiny in [46].

7.8 Natural Language Grammars

Much work has been done on creating parsers for natural language grammars. Many of the results of this work could be transferred to the domain of engineering grammars. Possible methods for parsing natural languages for syntax and semantics are discussed by Chomsky in [17]. Recently, advances have been made in principle-based parsing by Berwick and others. The philosophy behind this method of parsing natural languages holds that all languages have identical underlying principles and the semantics, or meaning of sentences, is a sort of universal language. Only the syntax, or words and sentence structure, varies between languages [5, 6]. Several parsers and translators have been constructed based on these assumptions, and they all work well on several different languages [23, 58]. These results parallel the observations made on the modification of design rules. Just as a small number of design rules can be modified to produce designs in a different style, so can a subset of syntax rules be modified to produce sentences in a different language.

7.9 Modular Robot Arms

Modular, reconfigurable robot manipulators are described in [20, 40, 141]. For the most part, these authors only address the design of the manipulators, not their use. In [60, 107], some attention is given to the derivation of the kinematics for any given configuration of a modular robot arm. The problem of generating non-isomorphic configurations of an arm is studied by Chen and Burdick in [15, 16].

Chapter 8

Conclusions

8.1 Search of Design Spaces

As discussed in the preceding chapters, grammars are particularly well-suited for exploring design spaces. A design tree defined by a grammar can fill a design space with its leaves and various search methods through the tree can converge to a good design relatively quickly. The choice of appropriate rules and transformations can provide a grammar with enormous generative power. By modifying some rules, a grammar can be made to search related areas of the design space. Rules can also limit the search of the space to those designs which satisfy certain criteria, speeding up the search for good designs.

8.2 Provability and Other Formal Properties

By using grammars to generate designs, the properties of formal languages are transferred to the designs. For example, certain conjectures about the expressibility of an engineering grammar may be proved or disproved based on the form of the grammar's rules. Other questions about the membership of a shape in a language of shapes might also be answered.

If shape or engineering grammars use only shapes and rules discussed in one of the sections of Chapter 3, then the languages have even more inherent structure. The most important property which stems from the use of such grammars is the closure property, guaranteeing that all the shapes in a language belong to a certain set.

8.3 Multiple Domains

Grammars can function well in multiple domains. Parallel grammars can use their rules to modify different aspects of a design simultaneously. This property is especially important with regard to the structure and functionality interrelationships in engineering designs. Grammars can easily represent most relations and can automatically change representations in all domains if only one parameter is changed by the designer. The ability to represent and manipulate data in multiple domains is one of the most powerful properties of grammars.

8.4 Suitability of Grammars for Engineering Design

Grammars can be very useful tools for designers. They are able to search large design spaces quickly. They have ample structure provided by their formal properties. They can represent multiple aspects of designs. For conceptual design, grammars easily outshine expert systems because of their ability to generate large numbers of alternatives which all satisfy some basic criteria. Because all members of a language must be "grammatical," the use of grammars in a design problem solution is an excellent choice.

8.5 Suggestions for Future Work

Many of the properties and uses of grammars for engineering design discussed in this dissertation could be extended or built upon. First and foremost, more grammars need to be constructed to solve real design problems. Grammars are suited to help designers come up with a large number of concepts for designs based on a relatively small library of components. For example, the layout of four-bar linkages and other simple mechanisms could be accomplished with a grammar. A parallel grammar could derive the kinematic and dynamic properties of the mechanisms. The rules would have to be parametric, with some values (*e.g.*, the sizes of some links) input by the designer. Mechanisms with desired layouts could be generated, or ones with user-defined link sizes. The grammar could even be directed to search for mechanisms which have a point which closely follows a user-defined path.

Grammars for engineering need not be defined only in the areas of robotics and kinematics, as the above example and the arm grammar of Chapter 6 might imply. Grammars for

gear trains [31, 32], bridge trusses [101], or solid models [35] might be extended to use larger sets of parts and operations. New domains should also be investigated. Researchers have built expert systems for many applications, but have encountered some problems which can not be efficiently solved using standard rule-based or knowledge-based systems. Grammars, with their different approach discussed in Chapter 4, might be used to solve some of these problems. Problems which are already solved by expert systems might also be solved using grammars. Tasks which are now performed using expert systems and their process-based rules should be examined to determine if they may also be accomplished using the more product-based rules of grammars. If so, then grammars could be created and the results produced by the two different formalisms could be compared.

Designers could make more use of the convolution operation discussed in Chapter 3. That operation and the Minkowski sum and difference are used in several graphical applications, but have found limited engineering use. Based on their usefulness in modeling typical material removal operations like milling and etching, as demonstrated in Section 3.4.3, the operations should receive more use in solid modeling and automated machining applications.

Designers can also more fully exploit the group-theoretic properties of shapes and shape operations. New operations and sets of shapes which satisfy the definitions of groups, rings, fields, or algebras might be discovered after further investigation. The ability to manipulate shapes in relatively complex ways and to be assured of obtaining only other shapes which satisfy certain requirements is quite powerful and should allow designers more room for experimentation with forms.

New ways of transforming data to match the left-hand sides of rules can also be investigated. Shapes might be made to match using skew transformations or some types of nonlinear mappings. Similarly, numeric data might be made to match by using some transformation of the set of real numbers into itself. New transformations can add power to rules and increase the expressiveness of languages.

Finally, new methods for searching the space of designs generated by a grammar are needed. As discussed in Chapter 5, languages can be quite large and it can be difficult to rank members of a language quickly. Methods of finding optima in the design space exist, but there are few ways to properly prune a design tree while it is being generated. More attention needs to be paid to this problem.

8.6 Contributions Made in This Dissertation

Several significant contributions to the field of design theory have been discussed in this dissertation. The group-theoretic properties of certain classes of shapes and shape operations have important implications for the use of those shapes and operations. While some of the more basic properties have been proven, it has never been shown that some shapes are members of rings or vector spaces.

Grammars and expert systems have many similarities and some very important differences. Until now, these two formalisms have never been compared or differentiated explicitly. The suitability of each formalism for solving particular design problems has also never been investigated.

The modular robot arm grammars of Chapter 6 are some of the first to operate on both the geometric and the kinematic properties of a system.

Finally, the appropriate uses of grammars were discussed in several contexts. While grammars have been used by others to address various engineering and architectural problems, little attention has been paid to the ways in which grammars may be used to search design spaces or to the proper choice of rules for grammars. Those and other issues have been considered in greater depth here than they have been before.

Appendix A

Definitions of Group-Theoretic and Other Formal Structures

A.1 Language-Theoretic Structures

These definitions were taken almost verbatim from *An Introduction to Formal Language Theory* [84]. They apply to all formal languages, whether they be textual, visual, mathematical, or in any other domain.

Definition A.1 (Alphabet) An *alphabet* X is a nonempty finite set of symbols.

Let X^* denote the set of all finite length strings formed by members of X. This set includes the empty string, denoted λ.

Definition A.2 (Grammar) A *phrase structure grammar* (or grammar, for short) G is a quadruple (T, N, S, P), where T and N are disjoint finite alphabets, and

1. T is the terminal alphabet for the grammar;
2. N is the nonterminal alphabet for the grammar, and $T \cap N = \emptyset$;
3. S is the start symbol for the grammar;
4. P is the set of productions for the grammar. P is a set of pairs (y, z), usually written $y \rightarrow z$, where y is a string in $(T \cup N)^*$ containing at least one nonterminal symbol and z is any string in $(T \cup N)^*$.

Definition A.3 (Derive) Let G be a grammar and let $y, z \in (T \cup N)^*$. Then y *directly derives* z (or z is directly derived from y), written $y \Rightarrow z$, if z can be obtained from y by replacing an occurrence in y of the left-hand side of some production by its right-hand side. Furthermore, y *derives* z (or z is derivable from y), written $y \stackrel{*}{\Rightarrow} z$, if $y = z$ or if there is some sequence of strings $w_1, w_2, \ldots, w_n$, with $w_1 = y$ and $w_n = z$ such that for all $i \in \{1, 2, \ldots, n-1\}$, w_i directly derives w_{i+1}.

Definition A.4 (Language) A *language* generated by grammar G, denoted $L(G)$, is the set of terminal strings which is derivable from the start symbol, S, of the grammar: $L(G) = \{z | z \in T^* \text{ and } S \stackrel{*}{\Rightarrow} z\}$.

A.2 Group-Theoretic Structures

In this section, definitions and examples of many group-theoretic structures are presented. The most basic structures are presented first and later definitions often depend on their predecessors. Most of the following definitions were obtained almost verbatim from *Algebra*, by Hungerford [56]. In order to provide an intuitive grasp of the properties of each structure, each definition is followed by a number of examples of sets and operations which satisfy the definition.

A.2.1 Groups

Definition A.5 (Binary operation) A *binary operation* on set S is a function $S \times S \rightarrow S$.

Examples:

- the addition or multiplication or subtraction or division operations on reals
- the tracing addition or convolution operations

Definition A.6 (Semigroup) A *semigroup* is a nonempty set G together with a binary operation, $+$, on G which is associative: $a + (b + c) = (a + b) + c$ for all $a, b, c \in G$.

Examples:

- N^*, the set of positive integers, along with addition or multiplication

- $\mathbb{R}^+$, the set of positive reals, along with addition or multiplication or division
- the set of tracings, along with the tracing addition or convolution operations

Definition A.7 (Monoid) A *monoid* is a semigroup G which contains a (two-sided) identity element $0 \in G$ such that $a + 0 = 0 + a = a$ for all $a \in G$.

Examples:

- N, the set of non-negative integers, along with addition
- $\mathbb{R} - \{0\}$, the set of reals minus zero, along with multiplication (identity is 1)
- the set of tracings and the tracing $\langle\langle 0 \rangle\rangle$, along with the convolution operation
- the set of tracings (including the null tracing), along with the tracing addition operation

Definition A.8 (Group) A *group* is a monoid G such that for every $a \in G$ there exists a (two-sided) inverse element $a^{-1} \in G$ such that $a^{-1} + a = a + a^{-1} = 0$.

Examples:

- $\mathbb{Z}$, the set of integers, along with addition
- $\mathbb{R}$, the set of reals, along with addition
- $\mathbb{R} - \{0\}$, the set of reals minus zero, along with multiplication (identity is 1)
- $\mathbb{Z}_p$, the integers modulo p, where p is prime, under multiplication mod p
- the set of tracings, along with convolution
- the set of $n \times n$ real matrices with nonzero determinants, along with matrix multiplication

Definition A.9 (Abelian group) A group is said to be *abelian* or *commutative* if its binary operation is commutative: $a + b = b + a$ for all $a, b \in G$.

Examples:

- all but the last group above are abelian
- the set of $n \times n$ real matrices with nonzero determinants along with matrix multiplication is *not* an abelian group because the order of multiplication matters

A.2.2 Rings

Definition A.10 (Ring) A *ring* is a nonempty set R together with two binary operations (usually denoted as addition, $+$, and multiplication, $*$) such that:

1. $(R, +)$ [the set R together with the operation $+$] is an abelian group, with additive identity element denoted 0;
2. $(a * b) * c = a * (b * c)$ for all $a, b, c \in R$ (associative multiplication);
3. $a * (b + c) = a * b + a * c$ and $(a + b) * c = a * c + b * c$ for all $a, b, c \in R$ (left and right distributive laws).

Examples:

- reals or integers along with addition and multiplication
- complex numbers along with complex addition and multiplication
- $n \times n$ real or integer or complex matrices along with matrix addition and multiplication
- vectors in $\mathbb{R}^n$ along with vector addition and cross product
- the set $\mathbb{Z}_n$, the integers modulo positive integer n, under addition and multiplication mod n

Definition A.11 (Commutative ring) A ring R is said to be a *commutative ring* if $a * b = b * a$ for all $a, b \in R$.

Examples:

- reals or integers along with addition and multiplication
- complex numbers along with complex addition and multiplication

Definition A.12 (Ring with identity) A ring R is said to be a *ring with identity* if R contains an element 1_R, the identity, such that $1_R * a = a * 1_R = a$ for all $a \in R$.

Examples:

- reals or integers along with addition and multiplication (identity is 1)
- complex numbers along with complex addition and multiplication (identity is $1 + 0i$)
- $n \times n$ real or integer or complex matrices along with matrix addition and multiplication (identity is $n \times n$ identity matrix)
- vectors in $\mathbb{R}^n$ along with vector addition and cross product does *not* have an identity
- the set $\mathbb{Z}_n$, the integers modulo positive integer n, under addition and multiplication mod n (identity is 1 mod n)

A.2.3 Fields

Definition A.13 (Inverse) An element a in a ring R with identity is said to be *left [resp. right] invertible* if there exists c in R [resp. b in R] such that $c * a = 1_R$ [resp. $a * b = 1_R$]. The element c [resp. b] is called the *left [resp. right] inverse* of a.

Examples:

- every element except 0 in the ring of reals is invertible
- only 1 and -1 are invertible in the ring of integers

Definition A.14 (Unit) An element a in ring with identity R is a *unit* if it is both left and right invertible.

Examples:

- every element except 0 in the ring of reals
- although $n \times n$ real or complex matrices are not commutative, every element with a nonzero determinant is a unit

Definition A.15 (Division ring) A ring D with multiplicative identity not equal to the additive identity ($1_D \neq 0$) in which every nonzero element is a unit is called a ***division ring***.

Examples:

- the reals and complex numbers form division rings

Definition A.16 (Field) A *field* is a division ring in which multiplication, $*$, is commutative.

Examples:

- real, rational, or complex numbers form fields
- the set $\mathbb{Z}_p$ of integers modulo prime p is a field

A.2.4 Boolean Rings

Definition A.17 (Boolean ring) A *Boolean ring* is a ring R such that $a * a = a$ for all $a \in R$.

Examples:

- the set $\mathbb{Z}_2$ of integers modulo 2, *e.g.*, the set $\{0, 1\}$, where $1 * 1 = 1$, $1 + 1 = 0$
- **Theorem A.1** *Every Boolean ring R is commutative and $a + a = 0$ for all $a \in R$.*

  Proof. Given a, b in Boolean ring R,

$$\begin{aligned}
(a + a^{-1}) * (a + a^{-1}) &= 0 \\
a * a + a^{-1} * a^{-1} + a * a^{-1} + a^{-1} * a &= 0 \\
a + a^{-1} + a * a^{-1} + a^{-1} * a &= 0 \\
(a + a^{-1}) + a * a^{-1} + a^{-1} * a &= 0 \\
0 + a * a^{-1} + a^{-1} * a &= 0 \\
a * a^{-1} + a^{-1} * a &= 0. \qquad \text{(A.1)}
\end{aligned}$$

Since $a^{-1} + a = 0$,

$$\begin{aligned} (a^{-1} + a) * a &= 0 \\ a^{-1} * a + a * a &= 0 \\ a^{-1} * a + a &= 0 \\ a + a^{-1} * a &= 0. \end{aligned} \tag{A.2}$$

So from Equations A.1 and A.2,

$$a = a * a^{-1}. \tag{A.3}$$

Also, since $a * (a^{-1} + a) = 0$

$$\begin{aligned} a * a^{-1} + a * a &= 0 \\ a * a^{-1} + a &= 0. \end{aligned} \tag{A.4}$$

Substituting from Equation A.3 into Equation A.4, we obtain:

$$a + a = 0$$

for all a in Boolean ring R.

To prove the second part of the claim, note that:

$$\begin{aligned} a + b &= (a + b) * (a + b) \\ a + b &= a * a + b * b + a * b + b * a \\ a + b &= a + b + a * b + b * a. \end{aligned}$$

So

$$a * b + b * a = 0$$

for all $a, b \in R$. But $a * b + a * b = 0$ (as proved above), so

$$a * b + b * a = a * b + a * b,$$

or

$$b * a = a * b$$

for all $a, b \in R$. So every Boolean ring is commutative. Note that the above derivations could be simplified considerably if the concept of the negative of an element was used. ■

- Shapes, taken as sets of maximal elements, under the sum and difference relation form a Boolean ring with zero given by the empty set and no unit.

- Sets satisfy the definition of a Boolean ring when the above operations are used:

  - Sets S form an abelian group under the operation $\diamond$ with identity $\emptyset$.

    * The operation $\diamond$ is commutative since for $a, b \in S$,

$$a \diamond b = (a \cap b') \cup (a' \cap b) = (b \cap a') \cup (b' \cap a) = b \diamond a.$$

    * The operation $\diamond$ is associative. For $a, b, c \in S$,

$$\begin{aligned}
a \diamond (b \diamond c) &= \{a \cap [(b \cap c') \cup (b' \cap c)]'\} \cup \{a' \cap [(b \cap c') \cup (b' \cap c)]\} \\
&= [a \cap (b \cap c')' \cap (b' \cap c)'] \cup [(a' \cap b \cap c') \cup (a' \cap b' \cap c)] \\
&= [a \cap (b' \cup c) \cap (b \cup c')] \cup (a' \cap b \cap c') \cup (a' \cap b' \cap c) \\
&= \{a \cap [((b' \cup c) \cap b) \cup ((b' \cup c) \cap c')]\} \\
&\quad \cup (a' \cap b \cap c') \cup (a' \cap b' \cap c) \\
&= \{a \cap [((b \cap b') \cup (b \cap c)) \cup ((c' \cap b') \cup (c' \cap c))]\} \\
&\quad \cup (a' \cap b \cap c') \cup (a' \cap b' \cap c) \\
&= \{a \cap [(b \cap c) \cup (c' \cap b')]\} \cup (a' \cap b \cap c') \cup (a' \cap b' \cap c) \\
&= (a \cap b \cap c) \cup (a \cap b' \cap c') \cup (a' \cap b \cap c') \cup (a' \cap b' \cap c) \\
&= (a \diamond b) \diamond c
\end{aligned}$$

    by permutation of the original members ($a \to c$, $b \to a$, $c \to b$).

⋆ The sets contain additive identity $\emptyset$ since for $a \in S$,

$$a \diamond \emptyset = \emptyset \diamond a = (a \cap \emptyset') \cup (a' \cap \emptyset) = (a \cap U) \cup \emptyset = a,$$

where U denotes the universal set.

⋆ The two-sided inverse of an element is just the element itself, *i.e.*, $a^{-1} = -a = a$ for all $a \in S$, since $a \diamond a = (a \cap a') \cup (a' \cap a) = \emptyset$.

– The operation $\cap$ is associative. For all $a, b, c \in S$, $(a \cap b) \cap c = a \cap (b \cap c)$ by definition.

– The operation $\cap$ is distributive over the operation $\diamond$.

⋆ Left distributive:

$$\begin{aligned} a \cap (b \diamond c) &= a \cap [(b \cap c') \cup (b' \cap c)] \\ &= (a \cap b \cap c') \cup (a \cap b' \cap c) \\ &= (a \cap b \cap a') \cup (a \cap b \cap c') \cup (a' \cap a \cap c) \cup (b' \cap a \cap c) \\ &= [(a \cap b) \cap (a' \cup c')] \cup [(a' \cup b') \cap (a \cap c)] \\ &= [(a \cap b) \cap (a \cap c)'] \cup [(a \cap b)' \cap (a \cap c)] \\ &= (a \cap b) \diamond (a \cap c). \end{aligned}$$

⋆ Right distributive:

$$\begin{aligned} (a \diamond b) \cap c &= [(a \cap b') \cup (a' \cap b)] \cap c \\ &= (a \cap b' \cap c) \cup (a' \cap b \cap c) \\ &= (a \cap b' \cap c) \cup (a \cap c \cap c') \cup (a' \cap b \cap c) \cup (b \cap c \cap c') \\ &= [(a \cap c) \cap (b' \cup c')] \cup [(a' \cup c') \cap (b \cap c)] \\ &= [(a \cap c) \cap (b \cap c)'] \cup [(a \cap c)' \cap (b \cap c)] \\ &= (a \cap c) \diamond (b \cap c). \end{aligned}$$

– The universal set, U, is the identity under $\cap$, since for any $a \in S$, $a \cap U = U \cap a = a$.

– No elements in the Boolean ring of sets are invertible except the universal set U.

A.2.5 Modules

Definition A.18 (R-module) A (left) *R-module* (or a module over a ring R) A is an additive abelian group A and a ring R together with a function $R \times A \to A$ (the image of (r, a) being denoted by ra) such that for all $r, s \in R$ and $a, b \in A$:

1. $r(a + b) = ra + rb$;
2. $(r + s)a = ra + sa$;
3. $r(sa) = (r * s)a$.

Examples:

- vectors in $\mathbb{R}^n$ multiplied by reals or integers
- real or complex matrices multiplied by reals or integers

Definition A.19 (Unitary R-module) A *unitary R-module* is an R-module A in which R has identity element 1_R and $1_R a = a$ for all $a \in A$.

Examples:

- vectors in $\mathbb{R}^n$ multiplied by non-negative reals or integers modulo an integer
- real or complex matrices multiplied by non-negative reals

Definition A.20 (Vector space) A (left) *vector space* is a unitary (left) R-module in which R is a division ring.

Examples:

- vectors in $\mathbb{R}^n$ multiplied by reals or integers modulo a prime
- real or complex matrices multiplied by reals

A.2.6 Algebras

Definition A.21 (K-algebra) A *K-algebra* (or algebra over a ring K) A is a ring A and a commutative ring with identity K such that:

1. $(A, +)$ is a unitary (left) K-module;
2. $k(a * b) = (ka) * b = a * (kb)$ for all $k \in K$ and $a, b \in A$.

Examples:

- every ring R is an R-algebra
- the rings of $n \times n$ matrices of real or complex numbers are algebras over both complex and real numbers (actually, they are division algebras)
- the ring of $n \times n$ real matrices is an algebra over integers modulo some integer s

Definition A.22 (Division algebra) A K-algebra A in which ring A is a division ring is called a *division algebra*.

Examples:

- vectors in $\mathbb{R}^n$ form a division algebra over the reals
- the ring of complex numbers is a real (or complex) division algebra

A.3 Properties of Maps

These definitions also come from [56].

Definition A.23 (Mapping) A *mapping* (also called a map or function), denoted by m, from class X to class Y, written $m : X \to Y$, assigns to each $x \in X$ an element $y \in Y$.

The element y is called the *value* of function m at x or the *image* of x, and is written $m(x)$. The class X is called the *domain* of the mapping and the class Y is called the mapping's *range*.

Definition A.24 (Surjection) A mapping $m : X \to Y$ is said to be *surjective*, or onto, if for each $y \in Y$, there exists some $x \in X$ such that $m(x) = y$ (or in a more convienient notation, $m(X) = Y$).

In other words, a surjection's range is the entire class Y.

Definition A.25 (Injection) A mapping $m : X \to Y$ is said to be *injective*, or one-to-one, if for all $w, x \in X$, if $w \neq x$, then $m(w) \neq m(x)$.

Alternatively, if $m(w) = m(x)$, then $w = x$ for all $w, x \in X$. Every unique element in an injection's domain maps to a unique element in the range.

Definition A.26 (Bijection) A mapping $m : X \to Y$ is said to be *bijective* if it is both surjective and injective.

A mapping which is bijective also has a two-sided *inverse*, a bijective mapping from the range of the original function to its domain.

Definition A.27 (Homomorphism) A mapping m from semigroup X to semigroup Y is a *homomorphism* if $m(wx) = m(w)m(x)$ for all $w, x \in X$.

If m is injective, then it is called a *monomorphism*. If m is surjective, then it is said to be an *endomorphism*. If m is bijective, then the mapping is called an *isomorphism* and X and Y are said to be isomorphic.

References

[1] J. L. Adams. *Conceptual Blockbusting.* W. H. Freeman and Company, San Francisco, California, 1974.

[2] H. Adeli. Artificial intelligence and expert systems. In H. Adeli, editor, *Expert Systems in Construction and Structural Engineering*, chapter 1, pages 1–12. Chapman and Hall, London, 1988.

[3] R. H. Allen, M. G. Boarnet, C. J. Culbert, and R. T. Savely. Using hybrid expert system approaches for engineering applications. *Engineering with Computers*, 2:95–110, 1987.

[4] V. R. Annadata, P. A. Fitzhorn, and W. O. Troxell. An assembly grammar for soma-n shapes. *Journal of Design and Manufacturing*, 1:35–45, 1991.

[5] R. C. Berwick. Principle-based parsing. In P. Sells, S. M. Shieber, and T. Wasow, editors, *Foundational Issues in Natural Language Processing*, chapter 4, pages 115–226. MIT Press, Cambridge, Massachusetts, 1991.

[6] R. C. Berwick. Principles of principle-based parsing. In R. C. Berwick, S. P. Abney, and C. Tenny, editors, *Principle-Based Parsing: Computation and Psycholinguistics*, pages 1–37. Kluwer Academic Publishers, 1991.

[7] R. J Brachman, S. Amarel, C. Engelman, R. S. Engelmore, E. A. Feigenbaum, and D. E. Wilkins. What are expert systems? In F. Hayes-Roth, D. A. Waterman, and D. B. Lenat, editors, *Building Expert Systems*, chapter 2, pages 31–57. Addison-Wesley Publishing Co., Reading, Massachusetts, 1983.

[8] D. M. Byrne and S. Taguchi. The Taguchi approach to parameter design. In *Quality Congress Transaction – Anaheim*, pages 168–177. ASQC, May 1986.

[9] J. Cagan. The constrained geometric knapsack problem and its shape annealing solution. Technical Report EDRC-24-86-92, Carnegie Mellon University, 1992.

[10] J. Cagan and W.J. Mitchell. Optimally directed shape generation by shape annealing. *Environment and Planning B*, 20(1):5–12, 1993.

[11] J. Cagan and G. Reddy. An improved shape annealing algorithm for optimally directed shape generation. In J.S. Gero and F. Sudweeks, editors, *Artificial Intelligence in Design '92*, pages 307–324, Dordrecht, Kluwer Academic Publishers, 1992.

[12] C. Carlson. Structure grammars and their application to design. Master's thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1988.

[13] C. Carlson, R. McKelvey, and R. F. Woodbury. An introduction to structure and structure grammars. EDRC Technical Report 48-20-90, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1990.

[14] C. Carlson and R. F. Woodbury. Structure grammars and their application to design. In P. Fitzhorn, editor, *First International Workshop on Formal Methods in Engineering Design*, pages 99–132, Fort Collins, Colorado, Colorado State University, January 1990.

[15] I.-M. Chen. *Theory and Applications of Modular Reconfigurable Robotic Systems.* Ph.D. thesis, California Institute of Technology, Pasadena, California, 1994.

[16] I.-M. Chen and J. W. Burdick. Enumerating non-isomorphic assembly configurations of a modular robotic system. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1985–1992, Yokohama, Japan, July 1993.

[17] N. Chomsky. *Syntactic Structures.* Mouton, The Hague, 1957.

[18] N. Chomsky. On certain formal properties of grammars. *Information and Control*, 2:137–167, 1959.

[19] S.-H. Chuang and M. R. Henderson. Compound feature recognition by web grammar parsing. *Research in Engineering Design*, 2(3):147–158, 1991.

[20] R. Cohen, M. G. Lipton, M. Q. Dai, and B. Benhabib. Conceptual design of a modular robot. *ASME Journal of Mechanical Design*, 114(1):117–125, March 1992.

[21] R. D. Coyne, M. A. Rosenman, A. D. Radford, and J. S. Gero. Innovation and creativity in knowledge-based CAD. In J. S. Gero, editor, *Expert Systems in Computer-Aided Design*, pages 435–465, Amsterdam, Elsevier Science Publishers B. V., 1987.

[22] T. G. Dietterich and D. G. Ullman. FORLOG: A logic-based architecture for design. In J. S. Gero, editor, *Expert Systems in Computer-Aided Design*, pages 1–17, Amsterdam, Elsevier Science Publishers B. V., 1987.

[23] B. J. Dorr. Principle-based parsing for machine translation. In R. C. Berwick, S. P. Abney, and C. Tenny, editors, *Principle-Based Parsing: Computation and Psycholinguistics*, pages 153–183. Kluwer Academic Publishers, 1991.

[24] F. Downing and U. Flemming. The bungalows of Buffalo. *Environment and Planning B*, 8:269–293, 1981.

[25] R. J. Doyle. *Hypothesizing Device Mechanisms: Opening Up the Black Box.* Ph.D. thesis, Massachusetts Institute of Technology, 1988.

[26] M. G. Dyer, M. Flowers, and J. Hodges. EDISON: An engineering design invention system operating naively. *Artificial Intelligence*, 1(1):36–44, 1986.

[27] C. L. Dym. EXPERT SYSTEMS: New approaches to computer-aided engineering. *Engineering with Computers*, 1:9–25, 1985.

[28] Clive L. Dym and R. E. Levitt. Toward the integration of knowledge for engineering modeling and computation. *Engineering with Computers*, 7:209–224, 1991.

[29] H. Ehrig. Introduction to the algebraic theory of graph grammars (a survey). In V. Claus, H. Ehrig, and G. Rozenberg, editors, *Graph-Grammars and Their Application to Computer Science and Biology*, pages 1–69, Berlin, Springer-Verlag, 1979.

[30] S. J. Fenves and N. C. Baker. Spatial and functional representation language for structural design. In J. S. Gero, editor, *Expert Systems in Computer-Aided Design*, pages 511–530, Amsterdam, Elsevier Science Publishers B. V., 1987.

[31] S. Finger and J. R. Rinderle. A transformational approach to mechanical design using a bond graph grammar. In W. H. ElMaraghy, W. P. Seering, and D. G. Ullman, editors, *Design Theory and Methodology - DTM '89*, pages 107–116. ASME, September 1989. First International ASME Design Theory and Methodology Conference, Montreal, Canada.

[32] S. Finger and J. R. Rinderle. Transforming behavioral and physical representations of mechanical designs. In P. Fitzhorn, editor, *First International Workshop on Formal Methods in Engineering Design*, pages 133–151, Fort Collins, Colorado, Colorado State University, January 1990.

[33] P. Fitzhorn. A linguistic formalism for engineering solid modeling. In H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, editors, *Graph-Grammars and Their Application to Computer Science*, Lecture Notes in Computer Science, # 291, pages 202–215, Berlin, Springer-Verlag, 1987. Proceedings of the Third International Workshop on Graph-Grammars and Their Application to Computer Science, Warrenton, VA, December 1986.

[34] P. A. Fitzhorn. A computational theory of design. In J. R. Dixon, editor, *NSF Engineering Design Research Conference Proceedings*, pages 221–233, University of Massachusetts, Amherst, NSF, June 1989.

[35] P. A. Fitzhorn. Formal graph languages of shape. *Artificial Intelligence in Engineering Design and Manufacturing*, 4(3):151–163, 1990.

[36] U. Flemming. The secret of the Casa Giuliani Frigerio. *Environment and Planning B*, 8:87–96, 1981.

[37] U. Flemming. Rule-based systems in computer-aided architectural design. In M. D. Rychener, editor, *Expert Systems for Engineering Design*, pages 93–112, Boston, Academic Press, Inc., 1988.

[38] S. Fong. The computational implementation of principle-based parsers. In R. C. Berwick, S. P. Abney, and C. Tenny, editors, *Principle-Based Parsing: Computation and Psycholinguistics*, pages 65–82. Kluwer Academic Publishers, 1991.

[39] M. Friedell and J.-L. Schulmann. Constrained, grammar-directed generation of landscapes. In *Graphics Interface '90*, pages 244–251, 1990.

[40] T. Fukuda and S. Nakagawa. Dynamically reconfigurable robot system. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1581–1586, 1988.

[41] P. K. Ghosh. A mathematical model for shape description using Minkowski operators. *Computer Vision, Graphics, and Image Processing*, 44:239–269, 1988.

[42] P. K. Ghosh. A solution of polygon containment, spatial planning, and other related problems using Minkowski operations. *Computer Vision, Graphics, and Image Processing*, 49:1–35, 1990.

[43] P. K. Ghosh. An algebra of polygons through the notion of negative shapes. *CVGIP Image Understanding*, 54(1):119–144, 1991.

[44] P. K. Ghosh and S. P. Mudur. The brush-trajectory approach to figure specification: Some algebraic solutions. *ACM Transactions on Graphics*, 3(2):110–134, April 1984.

[45] J. Gips. *Shape Grammars and Their Uses.* Ph.D. thesis, Stanford University, Stanford, California, 1974.

[46] J. Gips and G. Stiny. Production systems and grammars: A uniform characterization. *Environment and Planning B*, 7:399–408, 1980.

[47] D. Grune and C. Jacobs. *Parsing Techniques: A Practical Guide.* Ellis Horwood, New York, 1990.

[48] L. Guibas, L. Ramshaw, and J. Stolfi. A kinetic framework for computational geometry. In *Twenty-Fourth Annual Symposium on Foundations of Computer Science*, pages 100–111, Los Angeles, IEEE, IEEE Computer Society Press, 1983.

[49] R. M. Haralick, S. R. Sternberg, and X. Zhuang. Image analysis using mathematical morphology. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 9(4):532–550, 1987.

[50] F. Hayes-Roth, D. A. Waterman, and D. B. Lenat. An overview of expert systems. In F. Hayes-Roth, D. A. Waterman, and D. B. Lenat, editors, *Building Expert Systems*, chapter 1, pages 3–29. Addison-Wesley Publishing Co., Reading, Massachusetts, 1983.

[51] J. K. Hirschtick. Geometric feature extraction using production rules. Master's thesis, MIT, 1986.

[52] S. P. Hoover and J. R. Rinderle. A synthesis strategy for mechanical devices. *Research in Engineering Design*, 1(2):87–103, 1989.

[53] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, Massachusetts, 1979.

[54] T. J. Hubbard and E. K. Antonsson. Emergent Faces in Crystal Etching. Engineering Design Research Laboratory Report EDRL-TR 92c, California Institute of Technology, 1992.

[55] B. Humpert. Solving problems with automated reasoning, expert systems and neural networks. *Computer Physics Communications*, 61:58–75, 1990.

[56] T. W. Hungerford. *Algebra.* Springer-Verlag, New York, 1974.

[57] P. Jackson. *Introduction to Expert Systems.* Addison-Wesley, Wokingham, England, 1986.

[58] M. B. Kashket. Parsing Warlpiri—a free word order language. In R. C. Berwick, S. P. Abney, and C. Tenny, editors, *Principle-Based Parsing: Computation and Psycholinguistics*, pages 123–151. Kluwer Academic Publishers, 1991.

[59] A. Kaul and J. Rossignac. Solid-interpolating deformations: Construction and animation of PIPs. *Computers & Graphics*, 16(1):107–115, 1992.

[60] L. Kelmar and P. K. Khosla. Automatic generation of kinematics for a reconfigurable modular manipulator system. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 663–668, 1988.

[61] T. W. Knight. The generation of Hepplewhite-style chair-back designs. *Environment and Planning B*, 7:227–238, 1980.

[62] T. W. Knight. The forty-one steps. *Environment and Planning B*, 8:97–114, 1981.

[63] T. W. Knight. Languages of designs: From known to new. *Environment and Planning B*, 8:213–238, 1981.

[64] T. W. Knight. Transformations of languages of designs: Part 1. *Environment and Planning B: Planning and Design*, 10:125–128, 1983.

[65] T. W. Knight. Transformations of languages of designs: Part 2. *Environment and Planning B: Planning and Design*, 10:129–154, 1983.

[66] T. W. Knight. Transformations of languages of designs: Part 3. *Environment and Planning B: Planning and Design*, 10:155–177, 1983.

[67] T. W. Knight. Transformations of the meander motif on Greek Geometric pottery. *Design Computing*, 1:29–67, 1986.

[68] S. Kochhar. A prototype system for design automation via the browsing paradigm. In *Graphics Interface '90*, pages 156–166, 1990.

[69] H. Koning and J. Eizenberg. The language of the praire: Frank Lloyd Wright's prairie houses. *Environment and Planning B*, 8:295–323, 1981.

[70] R. Krishnamurti. The arithmetic of shapes. *Environment and Planning B*, 7:463–484, 1980.

[71] R. Krishnamurti. The construction of shapes. *Environment and Planning B*, 8:5–40, 1981.

[72] S. Longenecker and P. Fitzhorn. A shape grammar for modeling solids. In P. Fitzhorn, editor, *First International Workshop on Formal Methods in Engineering Design*, pages 70–98, Fort Collins, Colorado, Colorado State University, January 1990.

[73] S. N. Longenecker and P. A. Fitzhorn. Form + function + algebra = feature grammars. In S. L. Newsome, W. R. Spillers, and S. Finger, editors, *Design Theory '88*, pages 189–197, RPI, Troy, New York, NSF, June 1988. Proceedings of the 1988 NSF Grantee Workshop on Design Theory and Methodology.

[74] S. N. Longenecker and P. A. Fitzhorn. A shape grammar for non-manifold modeling. *Research in Engineering Design*, 2(3):159–170, 1991.

[75] T. Lozano-Pérez and M. A. Wesley. An algorithm for planning collision-free paths among polyhedral obstacles. *Communications of the ACM*, 22(10):560–570, October 1979.

[76] G. Matheron. *Random Sets and Integral Geometry.* John Wiley & Sons, New York, 1975.

[77] S. Mills, C. Erbas, Y. Hurmuzlu, and M. M. Tanik. Selection of expert system development tools for engineering applications. *ASME Journal of Energy Resources Technology*, 114:38–45, March 1992.

[78] M. L. Minsky. *Computation: Finite and Infinite Machines.* Prentice-Hall, Englewood Cliffs, New Jersey, 1967.

[79] W. J. Mitchell. *The Logic of Architecture.* The MIT Press, Cambridge, Massachusetts, 1990.

[80] S. Mittal and A. Araya. A knowledge-based framework for design. In *Proceedings AAAI-86*, pages 856–865, Los Altos, California, American Association of Artificial Intelligence, Morgan Kaufmann, August 1986.

[81] S. Mittal, C. L. Dym, and M. Morjaria. PRIDE: An expert system for the design of paper handling systems. In C. L. Dym, editor, *Applications of Knowledge-based Systems to Engineering Analysis and Design*, pages 99–115, New York, ASME, 1985.

[82] S. Mittal, C. L. Dym, and M. Morjaria. PRIDE: An expert system for the design of paper handling systems. *Computer*, 19(7):102–114, July 1986.

[83] S. Mittal, M. Morjaria, and C. L. Dym. Interactive graphics in expert systems for engineering applications. In *Proceedings of the 1985 International Computers in Engineering Conference and Exhibit*, pages 235–239, New York, ASME, 1985.

[84] R. N. Moll, M. A. Arbib, and A. J. Kfoury. *An Introduction to Formal Language Theory.* The AKM Series in Theoretical Computer Science. Springer-Verlag, New York, 1988.

[85] S. Mullins and J. R. Rinderle. Grammatical approaches to design. In P. Fitzhorn, editor, *First International Workshop on Formal Methods in Engineering Design*, pages 42–69, Fort Collins, Colorado, Colorado State University, January 1990.

[86] S. Mullins and J. R. Rinderle. Grammatical approaches to engineering design, part I: An introduction and commentary. *Research in Engineering Design*, 2(3):121–136, 1991.

[87] T. Murakami and N. Nakajima. Design-diagnosis using feature description. In H. Yoshikawa and D. Gossard, editors, *Intelligent CAD, I*, pages 169–185, Amsterdam, IFIP, North-Holland, 1989. Proceedings of the IFIP TC 5/WG 5.2 Workshop on Intelligent CAD, Boston, MA, 6–8 October, 1987.

[88] T. Murakami and N. Nakajima. Using features for machine design problems. *Computers & Graphics*, 14(2):201–210, 1990.

[89] T. Murakami and N. Nakajima. Diagnosing machine design through physical feature analysis. In H. Yoshikawa and F. Arbab, editors, *Intelligent CAD, III*, Amsterdam, IFIP, North-Holland, 1991. Proceedings of the IFIP WG 5.2 Third International Workshop on Intelligent CAD, Osaka, Japan, 26–29 September, 1989.

[90] S. S. Murthy and S. Addanki. PROMPT: An innovative design tool. In J. S. Gero, editor, *Expert Systems in Computer-Aided Design*, pages 323–341, Amsterdam, Elsevier Science Publishers B. V., 1987.

[91] M. Nagl. A tutorial and bibliographical survey on graph grammars. In V. Claus, H. Ehrig, and G. Rozenberg, editors, *Graph-Grammars and Their Application to Computer Science and Biology*, Lecture Notes in Computer Science, # 73, pages 70–126, Berlin, Springer-Verlag, 1979. International Workshop on Graph-Grammars and Their Application to Computer Science and Biology, 1978.

[92] R. Narasimhan. Picture languages. In S. Kaneff, editor, *Picture Language Machines*, pages 1–30, London, Academic Press, 1970.

[93] D. Navinchandra and D. H. Marks. Design exploration through constraint relaxation. In J. S. Gero, editor, *Expert Systems in Computer-Aided Design*, pages 481–509, Amsterdam, Elsevier Science Publishers B. V., 1987.

[94] B. Paden and S. Sastry. Optimal kinematic design of 6R manipulators. *International Journal of Robotics Research*, 7(2):43–61, 1988.

[95] G. Pahl and W. Beitz. *Engineering Design.* The Design Council, Springer-Verlag, New York, 1984.

[96] J. M. Pinilla, S. Finger, and F. B. Prinz. Shape feature description and recognition Using an augmented topology graph grammar. In J. R. Dixon, editor, *NSF Engineering Design Research Conference Proceedings*, pages 285–300, University of Massachusetts, Amherst, NSF, June 1989.

[97] E. L. Post. Formal reductions of the general combinatorial decision problem. *American Journal of Mathematics*, 65:197–215, 1943.

[98] A. A. G. Requicha. Representions for rigid solids: Theory, methods, and systems. *Computing Surveys*, 12(4):437–464, December 1980.

[99] A. A. G. Requicha and R. B. Tilove. Mathematical foundations of constructive solid geometry: General topology of regular closed sets. Technical Memorandum 27, Production Automation Project, University of Rochester, Rochester, NY, 1978.

[100] A. L. Ressler. A circuit grammar for operational amplifier design. Artificial Intelligence Laboratory Technical Report 807, MIT, 1984.

[101] J. R. Rinderle. Grammatical approaches to engineering design, part II: Melding configuration and parametric design using attribute grammars. *Research in Engineering Design*, 2(3):137–146, 1991.

[102] J. R. Rinderle, E. R. Colburn, S. P. Hoover, J. P. Paz-Soldan, and J. D. Watton. Form–function characteristics of electro-mechanical designs. In S. L. Newsome, W. R. Spillers, and S. Finger, editors, *Design Theory '88*, pages 132–147, RPI, Troy, New York, NSF, June 1988. Proceedings of the 1988 NSF Grantee Workshop on Design Theory and Methodology.

[103] C. Roads. Grammars as representations for music. In C. Roads and J. Strawn, editors, *Foundations of Computer Music*, chapter 23, pages 403–442. MIT Press, Cambridge, Massachusetts, 1985.

[104] D. W. Rosen and J. R. Dixon. Languages for feature-based design and manufacturability evaluation. *International Journal of Systems Automation: Research and Applications*, 2:353–373, 1992.

[105] J. R. Rossignac and A. A. G. Requicha. Constant-radius blending in solid modeling. *Computers in Mechanical Engineering*, 3(1):65–73, July 1984.

[106] J. R. Rossignac and A. A. G. Requicha. Offsetting operations in solid modeling. *Computer Aided Geometric Design*, 3(2):129–148, August 1986.

[107] D. Schmitz, P. Khosla, and T. Kanade. The CMU reconfigurable modular manipulator system. Technical Report CMU-RI-TR-88-7, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 1988.

[108] J. Serra. *Image Analysis and Mathematical Morphology.* Academic Press, London, 1982.

[109] J. E. Shigley and C. R. Mischke. *Mechanical Engineering Design.* McGraw-Hill Book Company, New York, fifth edition, 1989.

[110] M. Stefik, J. Aikins, R. Balzer, J. Benoit, L. Birnbaum, F. Hayes-Roth, and E. Sacerdoti. Basic concepts for building expert systems. In F. Hayes-Roth, D. A. Waterman, and D. B. Lenat, editors, *Building Expert Systems*, chapter 3, pages 59–86. Addison-Wesley Publishing Co., Reading, Massachusetts, 1983.

[111] M. Stefik, J. Aikins, R. Balzer, J. Benoit, L. Birnbaum, F. Hayes-Roth, and E. Sacerdoti. The architecture of expert systems. In F. Hayes-Roth, D. A. Waterman, and D. B. Lenat, editors, *Building Expert Systems*, chapter 4, pages 89–126. Addison-Wesley Publishing Co., Reading, Massachusetts, 1983.

[112] G. Stiny. *Pictorial and Formal Aspects of Shape and Shape Grammars.* Interdisciplinary Systems Research. Birkhäuser Verlag, Basel, 1975.

[113] G. Stiny. Two exercises in formal composition. *Environment and Planning B*, 3:187–210, 1976.

[114] G. Stiny. Ice-ray: A note on the generation of Chinese lattice designs. *Environment and Planning B*, 4:89–98, 1977.

[115] G. Stiny. Generating and measuring aesthetic forms. In E. C. Carterette and M. P. Friedman, editors, *Handbook of Perception, Volume X*, pages 133–152, New York, Academic Press, 1978.

[116] G. Stiny. The Palladian grammar. *Environment and Planning B*, 5:5–18, 1978.

[117] G. Stiny. Introduction to shape and shape grammars. *Environment and Planning B*, 7:343–351, 1980.

[118] G. Stiny. Kindergarten grammars: Designing with Froebel's building gifts. *Environment and Planning B*, 7:409–462, 1980.

[119] G. Stiny. A note on the description of designs. *Environment and Planning B*, 8:257–267, 1981.

[120] G. Stiny. Shapes are individuals. *Environment and Planning B*, 9:359–367, 1982.

[121] G. Stiny. A new line on drafting systems. *Design Computing*, 1:5–19, 1986.

[122] G. Stiny. Computing with form and meaning in architecture. *Journal of Architectural Education*, 39(1):7–19, 1988.

[123] G. Stiny. Formal devices for design. In S. L. Newsome, W. R. Spillers, and S. Finger, editors, *Design Theory '88*, pages 173–188, RPI, Troy, New York, NSF, June 1988. Proceedings of the 1988 NSF Grantee Workshop on Design Theory and Methodology.

[124] G. Stiny. Parallel grammars in design. In V. A. Tipnis and E. M. Patton, editors, *Computers in Engineering 1988*, pages 513–514, New York, ASME, 1988. Proceedings of the 1988 ASME Computers in Engineering Conference.

[125] G. Stiny. The algebras of design. *Research in Engineering Design*, 2(3):171–181, 1991.

[126] G. Stiny. Boolean algebras for shapes and individuals. Graduate School of Architecture and Urban Planning, UCLA, October 1992.

[127] G. Stiny. Weights. *Environment and Planning B: Planning and Design*, 19:413–430, 1992.

[128] G. Stiny and J. Gips. Shape grammars and the generative specification of painting and sculpture. In C. V. Freiman, editor, *Information Processing 71*, pages 1460–1465, Amsterdam, North-Holland, 1972.

[129] G. Stiny and J. Gips. An evaluation of Palladian plans. *Environment and Planning B*, 5:199–206, 1978.

[130] G. Stiny and L. March. Design machines. *Environment and Planning B*, 8:245–255, 1981.

[131] G. Stiny and W. J. Mitchell. Counting Palladian plans. *Environment and Planning B*, 5:189–198, 1978.

[132] G. Stiny and W. J. Mitchell. The grammar of paradise: on the generation of Mughul gardens. *Environment and Planning B*, 7:209–226, 1980.

[133] M. H. Stone. The theory of representations for Boolean algebras. *Transactions of the American Mathematical Society*, 40:37–111, 1936.

[134] G. Taguchi. *Introduction to Quality Engineering.* Asian Productivity Organization, Unipub, White Plains, NY, 1986.

[135] W. A. Taylor. *What Every Engineer Should Know About Artificial Intelligence.* MIT Press, Cambridge, Massachusetts, 1988.

[136] D. G. Ullman and T. A. Dietterich. Mechanical design methodology: Implications on future developments of computer-aided design and knowledge-based systems. *Engineering with Computers*, 2:21–29, 1987.

[137] A. C. Ward, T. Lozano-Pérez, and W. P. Seering. Extending the constraint propagation of intervals. *Artificial Intelligence in Engineering Design and Manufacturing*, 4(1):47–54, 1990.

[138] K. L. Wood, E. K. Antonsson, and J. L. Beck. Representing Imprecision in Engineering Design – Comparing Fuzzy and Probability Calculus. *Research in Engineering Design*, 1(3/4):187–203, 1990.

[139] R. F. Woodbury. Searching for designs: Paradigm and progress. EDRC Technical Report 48-16-90, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1990.

[140] R. F. Woodbury, C. N. Carlson, and J. A. Heisserman. Geometric design spaces. EDRC Technical Report 48-13-89, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1989.

[141] K.-H. Wurst. The conception and construction of a modular robot system. In *Proceedings of the 16th International Symposium on Industrial Robotics*, pages 37–44, 1986.