NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:
The copyright law of the United States (title 17, U.S. Code) governs the making of photocopies or other reproductions of copyrighted material. Any copying of this document without permission of its author may be prohibited by law.

Variations in Solids: A Declarative Treatment

by

Robert F. Woodbury

48-17-90 *C,P)*

Variations in Solids: A Declarative Treatment

Robert F. Woodbury

Department of Architecture / Engineering Design Research Center
Carnegie Mellon University,
Pittsburgh, Pennsylvania, USA, 15213.

To appear in Computers and Graphics, Special Issue on Features and Geometric Reasoning, Vol. 14, No. 2, April 1990.

Abstract: Underlying the notions of variational geometry, design prototypes, features, and representation of assemblies seems to be a common concept of *variations.* This paper develops the core of a monotonically declarative system for variations on solids. It introduces a set of language constructs that are the basis for ASCEND, an object oriented equation solving language. It presents equations for representing certain spatial relationships between primitive geometric elements. Using plex grammar notation it develops a set of Euler operators that are monotonic in the strict sense required by the ASCEND language. These operators are collectively shown to generate representations for all plane models of 2-manifold objects and to generate only such representations. Finally it presents the core of a system for variations living the ASCEND language to implement both equation models and the new Euler operators.

This work has been supported by the Engineering Design Research Center, an NSi-Engineering Research Center, and by NSF grant MSM-8717307.

Variations in Solids: A Declarative Treatment

Robert F. Woodbury

Department of Architecture / Engineering Design Research Center
Carnegie Mellon University,
Pittsburgh, Pennsylvania, USA, 15213.

1. Introduction

Recently, much attention has been given in the research community to the creation of systems that perform automated design. These systems, regardless of their underlying computational methods and like all computer programs, ultimately operate on data of very simple form (numbers, text strings, characters; bits in the limit). These primitive parts are composed, through various strategies, into aggregations of data that denote a design artifact. These aggregations are often called *design representations*. An unfortunate characteristic of many current systems is that their "so-called" design representations are not complete in the face of significant reasoning; it is difficult or impossible to deduce much about the object being designed from the information in the representation. An extreme, though frequently seen, design representation is a purely parametric one, a set of parameters, with ad-hoc relations between them. Another common form of representation adds very generic relationships, typically variants of is_a and part_of, to compose more complex structures. In only a very few systems is there a rigorous semantics associated with a representation and in these the benefits of of having a known and explicit relationship between the data and the modeled world is striking [1].

The problem of developing suitable design representations is a deep one, admitting as yet no solutions of a general nature, but several ideas have emerged that make progress on this issue. The notion of *features*, often taken to mean a part of a design representation that has properties

UNIVERSITY LIBRARIES
CARNEGIE-MELLON UNIVERSITY
PITTSBURGH, PENNSYLVANIA 15213

of interest to the performance of the whole, is one of these. *Design by features* is accomplished by a grammar, which has such features as its alphabet. Designs, or members of the language of the grammar, are produced by a series of additions (and deletions) of features. Features are often conceived as (possibly) *parametric objects* that represent not just one form but a possibly infinite family of forms. Another idea is that of *variational geometry* which is the expression of classes of designed objects through *constraints*† that operate on some representational framework and that permit the description and manipulation of an entire class of objects through the use of a set of parameters. The types of these parameters include dimensions, spatial relationships between parts, and parameters related to other aspects of a design (for example, load in a beam, floor area of a room, ratios between height and width in a window). Relating these to deeper models of physical or design artifacts is the promise of variational geometry. A third idea is that of *design prototypes* which are partial and generic descriptions of designs that store generalized knowledge about known designs or artifacts. Prototypes are stored design knowledge, they represent configurations (most often those found useful in prior design situations) and means to adapt those configurations to a present design situation. Encompassing the notion of variations is the *search paradigm*, particularly in layout problems, that seeks to build a representation scheme in which each member of the range of the scheme models an entire class of similar designs. The variations within the class are typically modeled as conjunctive sets of algebraic relations.

I believe that all of these ideas are on common ground in the notion of *variations*; they all depend upon being able to state what remains constant and what changes in an organized manner. They also depend on a mechanism for explicating how things change, and for exploring the implications of such changes. It seems to me that integrated, elegant techniques for expressing both the what and the how would be useful to much of current design research. Further, it seems to me that these techniques should be of a wide application and should be, to the greatest extent possible, declarative (rather than procedural). The need for a widely applicable capability seems evident and is met by the choice of geometry as a domain‡; the call for declarative representation perhaps less so. A declarative representation permits deductive reasoning and explanation

---

†These constraints are typically expressed as either a constraint network, solved by propagation, or as a set of simultaneous equations, solved through some numerical technique.

‡Of all the types of information required to describe design artifacts geometry stands out in its generality and absolute necessity and is ideal as the object of this inquiry.

directly upon the representation. A more procedural representation must first be acted upon by some usually quite specific (and impenetrable by current automated reasoning methods) set of procedures before its meaning becomes explicit. Declarative representations are naturally descriptive; they are created by describing truths that must exist, rather than by specifying procedures to obtain these truths. There is a drawback to declarative representation though; efficiency appears to require that statements be *monotonic*\ that they only add information to a description and never retract it. As will be shown, this condition can require changes to conventional ways of expressing models.

This paper presents:

- A language for building complex declarative equation-based descriptions.
- The ASCEND programming language as an implementation of this language.
- Equations for modeling spatial relationships between simple geometric elements.
- A declarative form of boundary representation as a representation scheme for designed objects.
- A demonstration of the core of a system for *variations.*

2. Background

Several authors have presented methods for the specification and solution of algebraic spatial relationships. Two main differences distinguish the present work from all of these: the complete self-contained inclusion of a solid representation and the exclusive use of a terse declarative language. Other differences are discussed below. Light and Gossard [2] describe a system for two dimensional variational geometry that operates on the *characteristic points* of an object. They use a Newton-Raphson method for equation solution and present formulations for a variety of equations on point objects. The work here is distinguished from theirs by its development in three dimensions, its principle orientation to language rather than solution technique and its inclusion of a representation for solids. Lee and Andrews [3] present the mathematics for expression of the *fits* and *against* conditions on solid objects as well as a report on solution of the equations. Here their equations are extended to planes, and by use-of a terse language the algebraic development is made much simpler. Since the language used provides some explicit

mechanisms for redundancy control some of their reported solving problems do not arise. Popplestone et. al. [4, 5] present an elegant means of specifying spatial relationships between bodies with half-space features (and in the laterpaper, edges and vertices). Their solution method, symbolic algebra, permits them to retain explicit degrees of freedom. They develop a large number of spatial relationships between their feature components. In addition to the differences above the present work is distinguished from theirs by its reliance on numeric solution techniques. Rossignac presents a method of specifying and solving spatial constraints between solid objects by the independent evaluation of constraints and a user-controlled sequential satisfaction process. The primary advantage of his approach is that it avoids the solution of large sets of simultaneous equations. The approach here fundamentally differs in its strict adherence to declarative specification. In earlier work [6, 7, 8] I present a core set of concepts and an example implementation for algebraic spatial relationships and feature based reasoning. I specify constraints with a declarative language over objects called *features* and *abstractions* and use constraint satisfaction as a solution technique. The work reported here is a significant reformulation and generalization.

Other authors have focused on the language issues of describing variational models. In some of earliest of these Dixon [9, 10, 11], and his students advocate the use of domain specific geometry representations through what are called *features.* For Dixon a feature is *"any geometric form or entity, whose presence or dimensions are relevant to one or more CIM^ functions; or, whose availability to designers as a primitive facilitates the design process."* The implementations reported in these papers show features as a combination of a pure primitive instancing scheme and a boundary representation solid modeler. Providing hierarchical structure to the feature models is.the categorization of features into types: macro-features, primitive features and co-features. I maintain a stronger notion of a rigorous underlying representation scheme and give a greater emphasis to declarative specification.

The following two authors report pioneering work on the specification of geometry with what have become known as object oriented techniques. Both use a generalized cylinder representation, whereas I use boundary representation, a more complex, more evaluated representation

"^Computer Integrated Manufacturing.

based on simpler primitives. Agin [12] presents a formalism for a hierarchical modeler based on spatial relationships between generalized cylinders. Assemblies are defined hierarchically using any of a fixed set of ways of specifying attachment between primitive objects, represented as generalized cylinders. Brooks [13] presents a geometric modeler as part of the ACRONYM vision system. In this modeler complex objects are represented by compositions of generalized cones. Classes of composite objects are represented through the use of algebraic inequality constraints on the profile and the sweeping rule of parts of composite objects, on the kinematic relations between parts, and on the number of attached parts. Constraint solution is symbolic.

3. Language elements for equation models

In this Section I introduce language constructs that enable the declarative expression o# geometric relations between primitive elements and organization of those elements into boundary representations (B-reps) of 2-manifolds. These constructs (along with a small number of others) underlie the ASCEND modeling language [14]. They are reported here in the abstract and apart from ASCEND and their presentation is informal; for a more formal treatment see [14]. ASCEND shares much in common with many of the object oriented languages and frame based systems that have emerged in recent years yet is distinguished from these by two features. First, it is strongly typed. The use of strong types and their separation from instances makes merge (are_the_same) operations extremely clear and makes crisp definition of rich type hierarchies possible. Second, the core of the language is tiny. There are two types of objects and six relations in ASCEND (I have omitted two relations unnecessary to the exposition in this paper). All else is built from these.

The two types of objects are *atoms* and *models.* The four model building relations are: *isji%* *refines, isjefxnedjo,* and *arejhejame.* Both the objects and the relations are monotonically declarative (thus our language provides no sense of procedure). Underlying the language are the concepts of *type* and *instance* and the primitive types, *string, integer, real* and *boolean* of which only the concept *type* needs an explanation here. A *type* is template for data that describes a number of *slots,* each of which has a name and can, in an instance of the type, carry an object. Slots are themselves typed, that is, they may be restricted to carry only objects of a particular type.

3.1. Objects in the language

Atom — An atom is a type. An atom has a special slot, called its *value,* that must be of a primitive type. In addition to its value, an atom's template describes a fixed number of slots for (possibly) different primitive types. Atom names are unique.

Model — A model is a type. A model consists of a possibly variable number of slots for (possibly) different existing atoms and models. Some of the slots of a model may be indexed as *arrays;* the number of these slots in a model is established by assigning a value to the array index. Array indexes must be of the primitive types integer or string. All elements of an array must have the same base type*!". Slots (and thus array elements) are typed. A model specifies a naming scheme for all *internal atoms* of its instances. Tlie set of names, relative to a model, in this scheme is formed by recursively concatenating (with periods in between names) slot names of the model with the slot names of the types of the respective slots, until an atom type is reached. As part of a model definition, mathematical relations on this set of names may also be included. The *type structure* of a model is the hierarchy of types recursively described by its slots along with the mathematical relations on the internal atoms of the model. Model names are unique.

Programs in the language are descriptions of atoms and models. A program specifies a set of variables and mathematical relations between those variables. Computation on programs is achieved by instantiating programs and submitting their instances to a mathematical programming facility that finds values of the variables that are consistent with the specified mathematical relations. An instantiation of a program creates two classes of computational objects:

Instance of atom — An instance of an atom is an object containing an actual *value,* an instance of the primitive type declared for the atom, as well as instances of conformal primitive types for each of the slots in the atom template. The value of an instance of an atom is obtained by naming the atom. All other slots are obtained by concatenating the atom name, a period and the slot name.

1"Base types are explained in ihc discussion of the relation *refines(Sccuion* 3.2).

Instance of a model An instance of a model is an object containing instances of appropriate type for each of its atoms, arrays and models. Instances of models automatically contain instances of the mathematical relations described by the model of which they are an instance.

3.2. Relations in the Language

The relations in the language describe structure and content relations both within and between the objects of the language. They are declarative in all but one sense; all relation arguments with the exception of the first argument sets of *ISJJ* and *refines* must have been *established* prior to being used. An argument is established if it appears as the first argument set in an *is_a* or *refines* relation. Put simply, an object must be created before it is used.

Is_a Is_a is a relation of type instantiation. It is used in a type definition to restrict the type of its first argument, a set of slots, to be at least as restricted as the type described by its second argument.

Refines Refines describes an inheritance relation between a set of inheriting types and an ancestor type. Each member of a type set (the first argument of refines) that is a refinement of another type (the second argument) contains all of its typed slots and mathematical relations. It may in addition contain other slots and other mathematical relations among its internal atoms. The refines relation establishes an inheritance hierarchy of model types. The *immediate ancestor* of a model is the model from which it was refined. The *ancestry* of a model is the ordered set, including the model itself, of all models that are an immediate ancestor of any model in the set. Two ancestries are *conformal* if one is a subset of the other. Two models have the same *base type* if their ancestries are conformal.

Is_refined_to Is_refined_to specifies that the type of a slot (the first argument) be a more restrictive type (the second argument). The two arguments must have conformal ancestries and the ancestry of the first argument must be shorter than that of the second.

Are_the_same Are_the_same specifies that all of its arguments will, on instantiation, refer to precisely the same object. This object will be of a type, perhaps unestablished, whose slots are typed selecting the most refined type for that slot

among all of the arguments. This slot typing is recursive, that is, it applies to the entire type structure of the models that are are_the_same. All of the arguments and recursively all of the slots of the arguments to are_the_same must have conformal ancestries. Are_jhe_same is often referred to as ***equivalencing.***

3.3. Monotonicity

It is useful to introduce a concept of *direct-monotonicity* to describe the types of changes that our language makes to a structure. Informally, a relation is direct-monotonic with respect to a data structure if its assertion can only add (and never retract) information stored directly in the data structure. For example, adding elements to the end of a linked list is direct-monotonic on the linked list (treating nils as voids); adding elements in the middle of a linked list is not, as pointers in the linked list must be reassigned. Our language supports only direct-monotonic relations on any type in the language, and this can be shown by examination of each of the relations. Is_a simply creates a new type that is a copy of an old type; the old type remains unchanged. Refines graduates a type down the inheritance hierarchy; the more refined type contains at least the same structure and data as the initial type. Is_refined_to is merely a delayed binding for refines. Finally, are_the_same only works if the ancestries of the two types are conformal. Therefore are_the_same preserves structure and data in the same manner as refines.

Again informally, a model is direct-monotonic if its internal statements do not subtract information already existing in the model. Models can contain only: 1 Statements about the types of the contents of slots and 2) mathematical relations between their internal atoms. The former are accomplished with the relations already argued to be direct-monotonic; the latter are simply statements about atoms and cannot access the model except as a means of reference. Therefore models are direct-monotonic.

4. The ASCEND modeling language

The ASCEND modeling language, designed and developed at Carnegie Mellon University, implements these (and other) constructs as a programming language for building (primarily) equation models of engineering systems. ASCEND directly supports the object types and rela-

tions discussed in the previous Section. In this section I introduce a subset of the ASCEND syntax required to express models for spatial variations. I use examples of ASCEND code to explain the syntax of the language as I feel that a more formal treatment, for example BNF specification, would obscure the exposition.

ASCEND programs are written as sets of statements describing atoms and models. On instantiation, the mathematical relations on the internal atoms of these sets form an equation model that is submitted for solution to a general, non-linear equation solver. Only two restrictions on the order of the statements are imposed:

1. All types that are used as arguments to relations must have been *established* in a previous statement. A type is established by being used in the first argument set of either is_a or refines. If the type is a model or an atom, its establishing statement is preceded by ATOM or MODEL respectively.

2. All mathematical relations must be written on the internal atoms of established types.

An ATOM in ASCEND may be a refinement of or identical to an established atom or primitive type. An typical ATOM declaration is:

```
ATOM direction REFINES generic_real
END direction;
```

Generic_real is an atom type, built into the ASCEND implementation, that has no dimensions (units) associated with it. Atoms may have units, for example, mass, length, time, temperature, charge, and moles and compositions of these under the operators *, /, and ^. Units are not used in the descriptions in this paper, but are of potential interest in building variational models.

A MODEL may be a refinement of or identical to an established model. In the following MODEL declarations the relations is_a, refines, and are_the_same are demonstrated as well as the specification of mathematical relations:

```
MODEL vector3;
    x, y, z, magnitude is_a generic_real;
    mageqn : x^2 + y^2 + z^2 = magnitude^2;

    (*default values*)
    x := 0.5;
    y := 0.5;
```

```
    z := 0.5;
END vector3;

MODEL generic_j>lane REFINES vector3;
    a, b, c, d IS_A generic_real;
    a, x ARE_THE_SAME;
    b, y ARE_THE_SAME;
    c, z ARE_THE_SAME;

    magnitude = 1;
    magnitude.fixed := true;
END generic_plane;
```

A vector3 is a type consisting of four quantities related by a single equation. The first three quantities are the usual vector coefficients, the founh its magnitude. In an unrestricted vector this magnitude may take on any value. The values of a vector are given a default value by tfie assignment (:=) syntax. This default value is used as an initial condition for the solving process. Note that this has a different effect than the equality (=) syntax that makes a statement about an condition that must be satisfied in an instance of the model. A plane can be considered as a vector3 with an additional quantity, the negative of its distance from the origin. Since a plane refines a vector3, it inherits all of the slots of vector3 as well as its single equation. The components of a plane are conventionally referred to as a, *b,* c, and *d* so the are_jhe_same relation is used to create an equivalence between the components of the vector3 and the plane values. The vector3 equation, although written in terms of of x, y, and z, now applies identically to *a, b,* and *c.* Setting the magnitude of the plane to 1 (one) ensures the condition, necessary to our representation of a plane, that its direction vector be normalized.

Arrays are ordered sets of slots of the same base type; since slots can occur only in models, so can arrays. Two example definitions using arrays are:

```
vector[integer]  IS_A generic_jreal;

matrix[integer] IS_A vector;
```

The FOR construct provides a means to succinctly specify mathematical relations among members of an array. FOR may appear procedural but it is not. It rather is a terse means to specify an indeterminate number of equations. Only when vector Jength is specified in some later statment does the actual number of created equations become known.

```
a,b,c is_a vector;

vector^length is_a integer;

FOR i : 1..vector_length CREATE
  a[i] = b[i] + c[i];
END;
```

The relation is_refined_to can be used to define arrays that contain slot of different types (although the types must be of the same base type). To create an array of objects, some of which are vector3 types and some of which are generic__plane types the following statements are used:

```
geo_objects[integer] IS_A vector3;
num_objects IS_A integer;

FOR i : 4..8 CREATE
  geo_objects[i] is_refined_to generic_jplane;
END;
```

5. The Equations of Spatial Relationships

Spatial relationships can be expressed as algebraic conditions on simple geometric elements. A few of the many possible relationships are developed here with the intent of demonstrating the direct relation between mathematics and computer code made possible by ASCEND. The following notational conventions are used:

Scalar quantities	Italic, lower case, e.g., *x, y, z.*
Column vectors	Bold, lower case, e.g., n, o, a, p.
Row vectors	Greek, lower case, e.g., X, y, *x*>.
Matrices	Bold, upper case, e.g., T, R.
Indices	According to the type of the object referenced. e.g., a row vector of scalars would have indices written as *j, 2j-*

The fundamental structure used in developing the equations that describe spatial relationships is the *vector,* or *row vector,* used in the usual sense of a list of numbers. A vector of length *n* is called an *n-vector* and is written as *x*> = [$v_p v_2$, • • • vj. A *column vector* is a transposed row vector. From vectors are developed three generic types of geometric objects: *generic points, generic*

planes and *generic transformations.* These types and the types to be developed from them display dependencies among their component elements; these are referred to as *constraints* and are represented as equations.

A *generic point* is a column 3-vector with no internal constraints on the values of its elements. A generic point is written as:

$$'P = [x,y,z]^T \quad (1)$$

A *generic plane* is a row 4-vector. The first three elements of a generic plane represent a vector, \), originating at the origin and normal to the plane. By convention this vector is constrained to be normal (of unit length). Under this convention, the fourth element has an interpretation as the negative of the distance along the direction vector and between the origin and the plane. Two distinct representations exist for any one plane, with the unit vector pointing from the origin towards and away from the plane respectively. An interpretation of planes as having sides removes this ambiguity. A generic plane is written as:

$$\gamma = [\upsilon \mid d\backslash = [a,b_9c_9d] \quad (2)$$

The normality constraint on the direction is written:

$$\mathbb{M} = 1 \quad (3)$$

that expands to a single equation on atomic components:

$$a^2+b^2+c^2 = 1 \quad (4)$$

A generic *transformation*^ is a collection of four column 3-vectors, n,o,a, p. The first three vectors form a rotation matrix T^ and are unit vectors representing the axes of a right handed coordinate system as they are oriented relative to a reference coordinate system. The fourth vector T^ represents a position for T_R relative to the same reference coordinate system. A generic transformation is written as*:

"tThe mathematics presented here is equivalent to that implemented in the more usual form of homogeneous coordinates. The form used here permits a reduction in the number of variables as the *w* coordinates arc not needed for rigid body motions.

***The following development of transformation constraints follows [3] in its algebra, but differs in its use of object decompositions.**

$$\mathbf{T} = [\mathbf{T_R} \mid \mathbf{T_p}] = [\mathbf{n}, \mathbf{o}, \mathbf{a}, \mathbf{p}] = \begin{bmatrix} n_x & o_x & a_x & p_x \\ n_y & o_y & a_y & p_y \\ n_z & o_z & a_z & p_z \end{bmatrix} \quad (5)$$

The first three vectors are orthonormal (mutually orthogonal and unit length). The fourth vector is unrestricted. The orthogonality constraints are written as:

$$\mathbf{n} \bullet \mathbf{o} = 0 \quad (6)$$

that expands to:

$$n_x o_x + n_y o_y + n_z o_z = 0 \quad (7)$$

and

$$\mathbf{n} \otimes \mathbf{o} = \mathbf{a} \quad (8)$$

that expands to:

$$n_y o_z - n_z o_y = a_x \quad (9)$$

$$o_x n_z - n_x o_z = a_y \quad (10)$$

$$n_x o_y - n_y o_x = a_z \quad (11)$$

where $\bullet$ denotes vector dot product and $\otimes$ denotes vector cross product.

Only two normality constraints are required. The third is implied by the cross product from Equation 8 being written between orthonormal vectors.

$\|\mathbf{n}\| = 1$

expanding to:

$$n_x^2 + n_y^2 + n_z^2 = 1 \quad (12)$$

$\|\mathbf{o}\| = 1$

expanding to:

$$o_x^2 + o_y^2 + o_z^2 = 1 \quad (13)$$

From these three generic geometric types are developed the two composite types used to express all of the spatial relations in this paper: *points* and *planes*.

A *point* consists of a generic transform, **T**, and two generic points, **l** and **g**. **T** represents a local coordinate system for the point. **l** represents the position of the point w.r.t. the coordinate system specified by **T**. **g** represents the position of the point w.r.t. a single global coordinate system. **T**, **l**, and **g** are related by the constraint:

$$g = T_R x l + T_p \quad (14)$$

where x denotes matrix multiplication and + denotes vector addition.

A *plane* consists of a generic transform, T, and two generic planes, *X* and y. T represents a local coordinate system for the plane. *X* represents the position of the plane w.r.t. the coordinate systems specified by T. y represents the position of the plane w.r.t. a single global coordinate system. T, A, and y are related by the constraints:

$$\gamma_v = \lambda_v \times \mathbf{T}_R^T \quad (15)$$

$$\gamma_d = \lambda_v \times (-\mathbf{T}_R^T \mathbf{T}_p) + \lambda_d \quad (16)$$

where T denotes the transpose of a matrix'.

Points and planes are used to develop spatial relations of which three are demonstrated here: plane parallelism, plane perpendicularity, and point-plane coincidence. For all of these relations, the choice of using the local or global description vectors of points and planes depends on the intention of the equation. For expressing relationships between parts of the same rigid body, the local coordinates alone are used. For expressing relationships between different rigid bodies, the global coordinates are used.

Plane parallelism takes different forms depending on the orientation of the sides of the planes. The simplest occurs when two planes are parallel and facing the same direction and is expressed by equating the components of the planes' normals:

$$yl_a = y2_{fl} \quad (17)$$

$$YU = Y2 \quad US)$$

$$\gamma 1_c = \gamma 2_c \quad (19)$$

When the planes face in opposite directions one side of the above equations must be negated:

$$Yl_{*a} = -Y2_{*a} \quad (20)$$

---

^These expressions are equivalent to the solution for matrix inversion that exists when isomctry transformations are expressed in homogeneous form.

$$YI^{*}=-Y2> \quad (2D$$

$$yl_c = -72_C \quad (22)$$

When the planes lie a given signed distance, *dist*, apart the fourth components, yl^ and y2^ are related by:

Planes facing in the same direction:

$$dist = \gamma 1_d - \gamma 2_d \quad (23)$$

Planes facing in opposite directions:

$$dist = y\backslash_d + y2_d \quad (24)$$

The distance, *dist*, is positive when yl^, taken as a vector fixed on yl, points toward y2, negative when yl_v, taken as a vector fixed on yl, points away from y2, and zero when yl and y2 are coincident.

The condition that a point be on a plane is expressed as:

$$^{\bullet} Y_v + Y<^{*}=0 \quad (25)$$

Two perpendicular planes yield:

$$WY^O \quad (26)$$

6. B-reps as a Representation Scheme

Models of solid objects are, in a sense, a *lingua franca*, of engineering design. They capture completely an important set of properties of engineered objects, namely nominal geometry or form. They are theoretically well advanced and are widely understood. For these reasons I use solid models, particularly boundary representation models, as a framework with which to express notions of variations, features and prototypes. Other frameworks are possible [1] but tend to be specific to particular, limited domains. I informally sketch here the basis for the boundary model representation; readers are referred to [15, 16, 17, 18] for a more complete discussion.

Solid objects have boundaries that are *orientable 2-manifolds. A* 2-manifold is a surface for which every point on the surface is topologically equivalent to an open disk. A manifold is orientable iff it is "two sided". In illustration, consider a fly that lives on one side of the surface; such a creature can never arrive at any point on the other side by moving in contact with its home surface.

Boundary representation schemes (B-rep schemes) model solid objects via a subdivision of their ***fundamental topologies* into *topological polygons,* most commonly expressed *as plane models.*** The fundamental topology of an object is merely its genus, the number of "holes" in the object, or alternatively, the maximum number of slices that may be cut through the object without separating it into distinct parts. A topological polygon of *n* sides informally corresponds to the notion of a regular n-sided polygon defined on an infinitely deformable sheet of infinitely <hin material. A plane model is a collection of these polygons with the operations of *edge identification* and *vertex identification* possibly applied respectively to subsets of edges and vertices from the collection. When the following three conditions are met in the edge and vertex identification a plane model represents an orientable 2-manifold [18]:

1. Every edge is identified with one and only one other edge.

2. For each collection of identified vertices, the polygons identified at that collection can be arranged in a cycle such that each consecutive pair of polygons in the cycle is identified at an edge adjacent to a vertex from that collection.

3. Directions for each of its polygons (and thus for the edges of each polygon) can be chosen so that for each pair of identified edges, the edge directions are opposite.

Plane models are represented inside a computer by a set of instances of various types (usually *shell, face, loop, edge,* and *vertex*^, and a set of relations that capture the adjacency relations between the components of a plane model. A well-formed set of such instances is an instance of a boundary representation (B-rep). It is usual for the surface of the solid to be decomposed by the plane model so that the parts of the surface represented by the topological polygons of the plane model have relatively simple mathematical descriptions, for example planes, quadric or parametric surfaces.

---

^Shells and loops may.be formally represented as *artifact faces* and *artifact edges* respectively. A complete theory of plane models can be developed in their absence. For the sake of simplicity, that is what I have done here.

At the bottom of every set of operators that act on B-reps is usually found a set of operators called the Euler operators. These incrementally act on the components of a B-rep such that the 2-manifold and orientability conditions on the corresponding plane model are maintained. Euler operators conventionally assume that these properties are valid before operator application and under this condition guarantee that the same properties are valid after application. Euler operators are based on the plane model operations of cutting and pasting polygons applied to both the plane model and its dual. Polygon cutting applied to the primary plane model and to its dual (also known as vertex splitting) is considered a *constructive Euler operator* as it does not involve the deletion of any existing plane model elements; it only adds new elements and changes relations. It is easy to convince oneself that any such operators necessarily alter some of the adjacency relations between plane model components. Since our abstract language supports no capability for altering relationships once they are stated, it would seem that the usual Euler operators cannot be programmed with it.

Not every plane model adjacency relation is usually captured directly in B-reps. Many are left implicit in the data structure to be extracted, when needed, by some algorithm. In our language all of the operators must be direct-monotonic with respect to the explicitly represented adjacency relations. The implicitly represented relations are free to change under operator application. If a complete representation scheme could be found that used only relations with respect to which the constructive Euler operators were direct-monotonic it would be possible for the usual Euler operators to be programmed. This quandry is not addressed here and remains an open question.

Due to speed limitations of the current ASCEND compiler (not limitations in the language definition) an almost strictly hierarchical data B-rep structure (call it the *FE data structure*), shown in Figure 6-1, is used here to represent plane models. At the heart of this data structure is the **F<E>** adjacency relation, a cyclic ordered list of edges around a face†. **F<E>** is individually sufficient as a representation of plane model topology in curved surface domains [19]; though alone it is not very efficient for typical applications. To gain access to vertex elements the **E{V}** relation is also captured. The FE data structure inter-element relations are changed by the usual Euler operations, so a different set of operators is required.

---

†When taken to actual implementation, **F<E>** is actually represented as an array; the cycle is implicit but can be made explicit by joining the head and tail of the array.

```
type location: array [1.3] of real;

type plane: array [1..4] of real;

type vpointer: pointer to vertex;

type ehpointer: pointer to edgejialfr

type fpointer: pointer to face;

type vertex : location;

type edge_half = record
                   eh_vl, eh_v2: vpointer;
                   eh_oth:ehpointer;
                 end;                                   7

type face =      record
                   f_eh: array [l..n] of[ehpointer
                   gamma: plane;
                 end;
```

Figure 6-1: The FE data structure

Two classes of plane model operations are used: operations that create the topological polygons of a plane model, and operations that perform edge and vertex identification.

The first class contains two operations, corresponding to the creation of:

1. An isolated vertex.
2. A polygon from an isolated vertex.

The second class contains four operations corresponding to the identification of:

1. Two edge-halves.
2. Unidentified vertices to form a polygon sequence.
3. Vertices to extend a polygon sequence.
4. Vertices to close a polygon sequence to a cycle (or to extend it).

is

An unrestricted *plex grammar* G_{plex} defines these operations. G_{pUx}= *(N, T, /0, /, 5, P)* where:

N the *non-terminal vocabulary,* = {•, *»*»•t^»S}.

T the *terminal vocabulary,* = {•, >, f}. *N n* 7= 0. I, the *vocabulary,* = /Vu7. Each ae Z is a *NAPE* type. The attaching points of each a are referenced by an *index set* of *identifiers I(a)* given below:

/(S)=0

/(•)= 0 /(*) = *[v_eh\, v_eh2)*

/(*)= {v_e/ri, v^/22, v_Ml} /(•)= {v_e/il, v_e/z2, *vjdl)*

/(#)= {v^/zl, v_e^2, vjdl, *vjdl}*

I(>)= *[eh_vl9eh_v2, ehj]* / (» = {e/z_vl, e/z_v2, e/zj*, e/z_or/z) ^

I(f)={/Le/E''|'=l ...*n9neN)*

i_0 the null identifier, = -

/ the set of attaching point identifiers, = *ijul(j)je* Z

5 the initial NAPE, = S

P the (finite) set of *plex productions* (also called *rules)* of the form A —»B where A and B are plexes, given below:

Rule 1 creates a single "vertex" that will be used as the "seed" to generate a polygon.

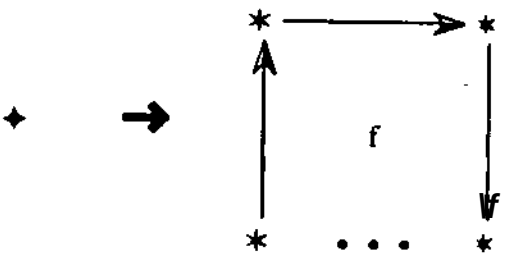
Rule 2 removes the initial symbol S so that no more "seed" vertices can be created.

S —•-»•

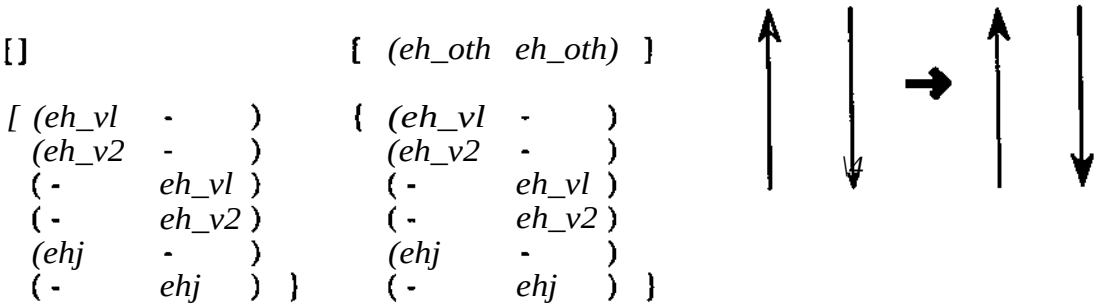
Rule 3 generates a complete polygon from a "seed" vertex. The seed vertex is removed.

•- > (* ^)i...n f

[(*vjehl; eh_v\i* -) / = 1 .. . *n*
(*v_eh2j+xeh_v2j* -) *j* = 1 . . . *n* - 1
(*v_e/i2 eh_vl n* -)
(- *ehjt f_eh<*)]



Rule 4 identifies edges from two polygons.



```
[]                                  [  (eh_oth   eh_oth)  ]

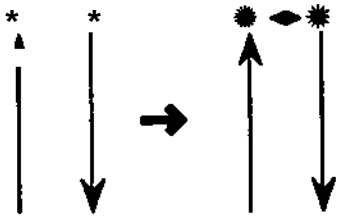
[ (eh_vl    -      )                {  (eh_vl    -      )
  (eh_v2    -      )                   (eh_v2    -      )
  (-        eh_vl  )                   (-        eh_vl  )
  (-        eh_v2  )                   (-        eh_v2  )
  (ehj      -      )                   (ehj      -      )
  (-        ehj    ) ]                 (-        ehj    ) ]
```

Rules 5_{abc} identify polygon vertices that lie at corresponding ends of identified edges.

- direct vertex identification

continued sequence of vertex identification

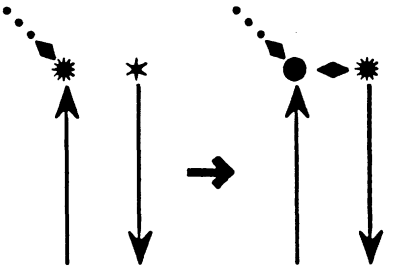
Rule 5_a



```
[ (v__ehl   eh_vl   -        -        )      [  (v_ehl   ehj/l   -        -        )
  (-        -       eh_v\    v_eh\    )         (-        -       ehjsl    v_eh\    )
  (-        eh_oth  eh_oth   -        ) ]       (-        eh_oth  eh_oth   -        )
                                                (v_idl    -       -        vjidl    ) ]

{ (v_ehl    -       -        -        )      {  (vjehl   -       -        -        )
  (-        ehj     -        -        )         (-        ehj     -        -        )
  (-        eh_vl   -        -        )         (-        eh_vl   -        -        )
  (-        -       ehj      -        )         (-        -       ehj      -        )
  (-        -       eh_vl    -        )         (-        -       eh_v2    -        )
  (-        -       -        v_ehl    ) }       (-        -       -        v_e/?2   ) }
```

Rule 5_b





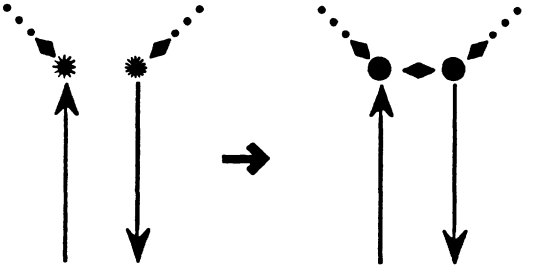
[(*v_eh2* *eh_v2* - -)
(- - *eh_v1* *v_eh1*)
(- *eh_oth* *eh_oth* -)]

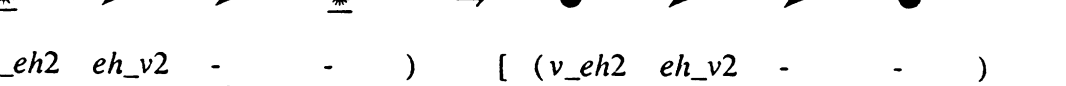
[(*v_eh2* *eh_v2* - -)
(- - *eh_v1* *v_eh1*)
(- *eh_oth* *eh_oth* -)
(*v_id2* - - *v_id1*)]

{ (*v_eh1* - - -)
(*v_id1* - - -)
(- *eh_f* - -)
(- *eh_v1* - -)
(- - *eh_f* -)
(- - *eh_v2* -)
(- - - *v_eh2*) }

{ (*v_eh1* - - -)
(*v_id1* - - -)
(- *eh_f* - -)
(- *eh_v1* - -)
(- - *eh_f* -)
(- - *eh_v2* -)
(- - - *v_eh2*) }

Rule 5_c





[(*v_eh2* *eh_v2* - -)
(- - *eh_v1* *v_eh1*)
(- *eh_oth* *eh_oth* -)]

[(*v_eh2* *eh_v2* - -)
(- - *eh_v1* *v_eh1*)
(- *eh_oth* *eh_oth* -)
(*v_id2* - - *v_id1*)]

{ (*v_eh1* - - -)
(*v_id1* - - -)
(- *eh_f* - -)
(- *eh_v1* - -)
(- - *eh_f* -)
(- - *eh_v2* -)
(- - - *v_eh2*)
(- - - *v_id2*) }

{ (*v_eh1* - - -)
(*v_id1* - - -)
(- *eh_f* - -)
(- *eh_v1* - -)
(- - *eh_f* -)
(- - *eh_v2* -)
(- - - *v_eh2*)
(- - - *v_id2*) }

By the above index set definitions, the elements of I form the sub-type hierarchy shown in Figure 6-2. By examination, the rules of G_{plex} are all M4P£-monotonic1" and can therefore be implemented by the direct-monotonic operators of ASCEND.

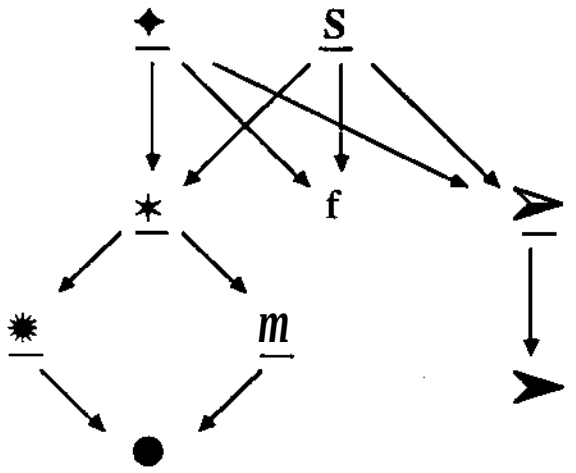


Figure 6-2: The sub-type hierarchy of the plex grammar G_{plex}

A correspondence exists between the plexes produced by the grammar and plane models. *NAPEs* of types •, *, #, •, and • correspond to vertices. *NAPEs* of types ≥: and > correspond to edges and *NAPEs* of type f correspond to faces. The index sets of vertex *NAPE* types describe vertex-edge (within a polygon) and immediate vertex identification relations. The index sets of edge *NAPE* types describe edge-vertex (within a polygon) and edge identification relations. The index set of the face *NAPE* type describe the F<E> relation within a polygon. Thus there is a direct relation between plexes produced by the grammar and plane models, and therefore between plexes and the FE data structure.

The language *L* generated by G_{plex} is:

$$L(G) = \{A\ 15 \overset{\bullet}{\underset{P}{=>}}\ A\ A\ V\ Xf\ r(Xf)\ e\ 7\backslash = 1\ \ldots\ /(X^{A})\}$$

†See Appendix A.

that is, the set of plexes, containing only terminal *NAPES,* that can be derived from the initial structure in **zero or more steps.**

Expressing the operations **as a** grammar permits proof of these theorems:

Lemma 1: Condition 1 is guaranteed by G_{pUr}

Proof: Rule 4 transforms a pair of unidentified (≥>edges into a pair of jointly identified (>)-edges. Since no rule can identify an already identified edge and since no plex is a member of *L(G)* unless all of its (≥^-edges are identified, condition 1 is met by every member of *L(G).*

Lemma 2: Condition 2 is guaranteed by G_{plex}.

Proof: For the purposes of this lemma, consider that a cycle is a sequence joined head to tail. Each vertex \) has an associated polygon, *p(x>).* Each unidentified vertex \) has an *associated polygon sequence* S_v consisting only of pCu). The *tail* of S_v is p(i)).

By Rule 5_a two unidentified vertices **a** and **p** can be *directly identified* if they exist respectively at the *eh_v2* and *eh_yl* ends of an identified pair of edges (call the edge identification conditions the *ID-conditions).* Thus Rule 5_a creates a length 2 sequence $S_a\$$ of associated polygons and can be considered to modify S_a and Sp to both be equal to 5_ap. p(p) is at the tail of S_{afi}.

Under the same ED-conditions as Rule 5_a, Rule 5_b extends by one polygon an associated polygon sequence S_a by directly identifying a vertex **a** whose polygon is the tail of S_a with an unidentified vertex **p** to form a sequence $5_{\ldots ap}$. As in Rule 5_a, $5_{\ldots ap}$ replaces S(oc) and S(p). p(p) becomes the tail of $S_{\ldots ap}$.

Under the same ID-conditions as Rule 5_a, Rule 5_C joins two (non-necessarily distinct) associated polygon sequences to form a single sequence $S_{\ldots a}p_{\ldots}$. A vertex **a** whose polygon is the tail of S_a is directly identified with an identified vertex **p** of type r(#). $\$_{\ldots afi\ldots}$ replaces S(oc) and S({3). The tail of 5(p) becomes the tail of $S_{\ldots a}p_{\ldots}$.

The associated polygon sequence of any identified vertex defines a collection of identified vertices. Since Rules 5_{abc} *are* the only grammar rules that perform vertex identification, all such collections in any member of *L(G)* must be sequences. The ID-conditions guarantee that adjacent polygons in the sequence are adjacent through edge identification.

From examining the NAPE types of the grammar, each vertex can be directly identified with at most two other vertices. Any member of *L(G)* contains only vertices that are directly identified with exactly two other vertices. Since each of Rules $5_{a,b,c}$ perfoim exactly one direct identification of a pair of vertices, if there are *n* vertices identified in a vertex collection within a member of *L(G)* then they must have been joined with *n* rule applications.

If *n* objects are arranged with *n* pairwise connections, the connections must form a set of cycles. Thus the direct vertex identifications and thus the associated polygon sequences in any vertex collection in a member of L(G) form a set of cycles. But each associated polygon sequence is, by construction, a single sequence, and therefore must be a single cycle.

Lemma 3: Condition 3 is guaranteed by G_{plex}.

Proof: The ED-conditions of Rules 5_{qbc} guarantee that in every pair of identified edges, the edges occur in opposing directions.

Theorem 4: Only data structures that model non-empty sets of plane models of orientable 2-manifolds are produced by the grammar.

Proof: Lemma 1, Lemma 2, and Lemma 3 collectively prove Theorem 4.

Theorem 5: A structure that represents an arbitrarily chosen plane model (with a non-zero number of edges) of an orientable 2-manifold is generated by the grammar.[1]

Proof: Consider the set of rules that is obtained from the rules of the grammar by reversing the directions of the production arrows. Denote the rule so obtained from Rule *n* by *Rule* $n^{\sim l}$. By inspection of the rules it is easily seen that Rules *n* and rc^{x} are inverses, that is, Rule *n* derives plex **a** from plex P iff Rule $n^{\sim x}$ derives plex (3 from a. Let a plane model of an arbitrary orientable 2-manifold be given. The following algorithm reduces the plane model to the initial object S.

while there exists an identified pair of vertices of type *t(%)*
    apply Rule 5
    while there exists an identified pair of vertices of types r(#) and *t(*)*
        apply Rule 5^
        while there exists an identified pair of vertices of types r(j*) and /(<*)
            apply Rule $5''^{1}$
assert no identified vertices remain .

[1] **"hfhe grammar docs not produce the primitive plane model with: a single vertex, no edges and a single face. This is a trivial limitation that could be overcome by a separate grammar rule for this model.**

for every identified edge pair
    apply Rule 4^{-1}
assert no identified edges remain

for every polygon
    apply Rule 3^{-1} to produce a vertex of type $t(\underline{\spadesuit})$
assert no polygons remain

apply Rule 2^{-1} once to any vertex of type $t(\underline{\spadesuit})$
assert an initial symbol of type $t(\underline{S})$ exists

for every vertex of type $t(\underline{\spadesuit})$
    apply Rule 1^{-1}
assert only $\underline{S}$ remains

Since rules n and n^{-1} are inverses, if the sequence of rules $R_{i1}^{-1} R_{i2}^{-1} \ldots R_{in}^{-1}$ reduces a plane model of an arbitrary orientable 2-manifold to the initial object $\underline{S}$, then the sequence $R_{in} R_{i(n-1)} \ldots R_{i1}$ constructs the plane model from $\underline{S}$. Therefore a plane model of every 2-manifold object is generated by the grammar.

With these proofs in place these operations can be used as the basis for building descriptions of sets of solid objects.

Many existing CAD systems have the capability to "paste together" instances of polygons to completely enclose a volume. Our grammar rules perform the same task, except that they operate at the level of the plane model representation of topology alone; they require no embedding into three dimensional space $\mathbf{R}^3$. We can thus use members of the language of the grammar together with a variable representation of their embedding into $\mathbf{R}^3$ (variable geometry) as a representation of a *variational class* of solid objects. We express this variable geometry using models built from the equations developed in Section 5. It is important to realize that embedding of a plane model into $\mathbf{R}^3$ creates constraints relating to the non-intersection of components of the plane model. The representation of these conditions within our chosen computation scheme (equation solving and mathematical programming) is an unsolved and current research problem that we do not address here. Our emphasis is on a self-contained language with which variational models may be precisely described.

7. A Variational Geometry System

In this section the equations of spatial relationships (Section 5) and the plane model grammar (Section 6) are coupled through the language constructs (Section 3) using the ASCEND language as an implementation vehicle. Although a complete listing of the ASCEND code required to implement a simple geometric modeler is only some eight pages in length, only examples that demonstrate key points are presented here (including all of the Euler operators). To date, several models of variational solid objects and assemblies have been created and tested. Figure 7-1 shows objects of typical complexity.

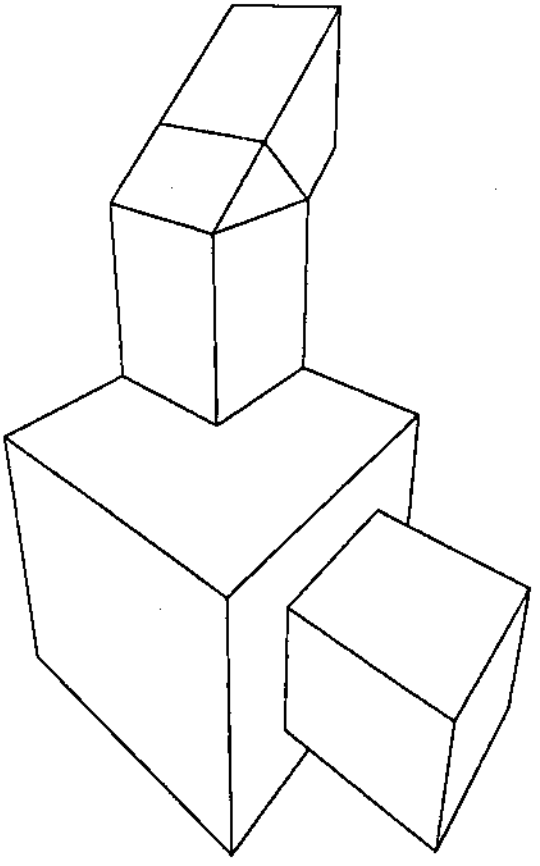


Figure 7-1: A "tinkertoy" set of solid models

7.1. Simple Geometric Types

The type **vector3** contains the three vector parameters plus an equation for its magnitude. Putting the equation **here** simplifies later expression of normality constraints and provides a facile means to control the proliferation of redundant equations.

```
MODEL vector3;
   x, y, z, magnitude IS_A generic_real;
   mageqn : x^2 + y^2 + z^2 = magnitude^2;

INITIALIZE
   x := 1;
   y := 1;

END vector3;
```

Dot and **cross products** operate on simple vectors.

```
MODEL dot3;
   vl, v2 IS_A vector3;
   prod IS_A generic^real;

   doteqn: vl.x*v2.x + vl.y*v2.y + vl.z*v2.z = prod;
END dot3;

MODEL cross REFINES vector3;
   vl, v2 IS_A vector3;

   xcomp : x = vl.y*v2.z - v2.y*vl.z;
   ycomp : y = vl.z*v2.x - v2.z*vl.x;
   zcomp : z = vl.x*v2.y - v2.x*vl.y;
END cross;
```

A generic_point removes the magnitude equation from the **vector3** model. This simplifies eventual equation models, while allowing points to inherit other useful properties of **vector3,** for example, defaults.

```
MODEL generic_point REFINES vector3;
   mageqn.included := FALSE;
END generic_jpoint;
```

A generic_plane provides the additional names *(a>b,c)* to those extant in vector3 and adds a fourth component *d.* Since the vector D is unit length, the magnitude quantity of the vector is constrained to be equal to 1.

```
MODEL generic_j>lane;
   a, b, c, d IS_A generic_real;
   uv IS_A vector3;

   a, uv.x ARE_THE_SAME;
   b, uv.y ARE_THE_SAME;
   c, uv.z ARE_THE_SAME;

   uv.magnitude := 1;
   uv.magnitude.fixed := TRUE;
END generic_jplane;
```

A transform uses the dot and cross products and the magnitude of its vector3 compo^eifts to describe its constraints. The magnitude equation of the cross product is released, removing a redundancy. Note the very simple form of the equations required.

```
MODEL transform;
   n, o, a, p IS_A vector3;
   d IS_A dot3;
   c IS_A cross;

   d.vl, n ARE_THE_SAME;
   d.v2, o ARE_THE_SAME;
   d.prod := 0;
   d.prod.fixed := TRUE;

   c.vl, n ARE_THE_SAME;
   C.v2, o ARE_THE_SAME;
   C, a ARE_THE_SAME;

   n.magnitude := 1;
   o.magnitude := 1;
   n .magnitude.fixed := TRUE;
   o.magnitude.fixed :=TRUE;

   a.mageqn.included := FALSE;

END transform;
```

:S

A **plane** refines a **generic_plane** and uses it as its global representation. A separate local representation is declared internally. The equations are the expansions of the vector and matrix operations of Equations 15 and 16. The redundant magnitude equation of normal vector of the local plane is eliminated. The **point** model is similar.

```
MODEL plane REFINES generic_plane;
   local IS_A generic_plane;
   t IS_A transform;

   a = t.n.x * local.a + t.o.x * local.b
                       + t.a.x * local.c;

   b = t.n.y * local.a + t.o.y * local.b
                       + t.a.y * local.c;

   c = t.n.z * local.a + t.o.z * local.b
                       + t.a.z * local.c;

   d = local.a * ( t.n.x * -t.p.x + t.n.y * -t.p.y
                                  + t.n.z * -t.p.z )
     + local.b * ( t.o.x * -t.p.x + t.o.y * -t.p.y
                                  + t.o.z * -t.p.z )
     + local.c * ( t.a.x * -t.p.x + t.a.y * -t.p.y
                                  + t.a.z * -t.p.z )
     + local.d;

   local.uv.mageqn.included := FALSE;
END plane;
```

7.2. Spatial Relationship Types

The following are some types that model the spatial relationships between simple geometric types.

Parallel (between two identically oriented planes) avoids creating three equations by simply equivalencing the components of the vector normals.

```
MODEL parallel;
   gamma1, gamma2 IS_A generic_plane;

   gamma1.uv, gamma2.uv ARE_THE_SAME;
END parallel;
```

The condition of two oppositely oriented planes being parallel implies that three simple equations be created. The magnitude equation of one of the vector normals is released.

```
MODEL invjparallel;
   gammal, gamma2 IS_A generic_plane;

   a_eqn: gammal.a = -gamxna2.a;
   b_eqn: gammal.b = -gamma2.b;
   c_eqn: gammal.c = -gamma2.c;

   gamma2.uv.mageqn.included := FALSE;
END invjparallel;
```

Constraining two oppositely oriented planes to be a certain distance apart adds a single constraint.

```
MODEL invjpar_dist REFINES invjparallel;
   distance IS_A generic_real;

   gammal.d + gamma2.d = distance;
END invjpar_dist;
```

7.3. Boundary Representation Types

Notice that these types have no internal definitions of simple "geometry"; all such information is inherited.

A vertex points to no other type, so the type vertex is merely a place holder.

```
MODEL vertex REFINES point;
END vertex;
```

A generic_edge_half is used to permit the inclusion of a slot of type edge in the model of an edge.

```
MODEL generic_edge_half;
END generic_edge_half;
```

An edgejialf includes a slot for the **edge_half** that it is identified with in a complete 2-manifold model and slots for **vertices** at each of its ends. The **transforms** of the two vertices are equivalenced.

```
MODEL edge_half REFINES generic_edge_half;
   ehjvl, eh_v2 IS_A vertex;
   eh_other IS_A generic_edge_half;

   eh_vl.t, eh_v2,t ARE_THE_S AME;
END edge_half;
```

A face contains slots for its plane, for each of its **edge_halves,** and for an uninstantiated parameter for its number of **edgejialves.**

```
MODEL face;
   gamma IS_A plane;
   num_sides IS_A integer;
   f_eh[integer] IS_A edge_half;
END face;
```

A **shell** is a holder for a **transform.**

```
MODEL shell;
      t IS_A transform;
END shell;
```

7.4, Grammar Rule Types

The rules of the grammar are represented as models. "Programs" are built by using these models to restrict the types of slots in larger models. The initial symbol in the grammar has no counterpart in the implementation; it is used in the grammar as a device to assist the proofs of well-formedness and completeness. Rule 2 of the grammar, whose purpose is only to remove the initial symbol thus does not exist here, nor are any of Rules 5_{abc} explicit, as these deal with relations not captured in the FE data structure (but see the vertex equivalencing in the implementation of Rule 4).

Rule 1. Make_vertex describes a **shell, a vertex,** and equivalences the **transform** of the two. They are thus joined in space a "rigid" unit, bound to move together.

```
MODEL make_yertex;

(•topology part*)

   s IS_A shell;
   v IS_A vertex;

(*geometry part*)

   S.t, v.t ARE_THE_SAME;

END make_vertex;
```

Rule 3. Make_face starts with a **make_vertex. It** establishes a **face** and then equivalen^e^the initial **vertex** with the first **vertex of** the **face. It** then links adjacent **edge_halves** and **vertices** on the face. The "geometry" is established by linking the **transform of** the **face** to that of the start **vertex;** the vertex-transform equivalencing in the **edge_half** model implicitly guarantees that all **vertices** will share a common **transform.** Each point is constrained to lie on the plane of the face.

```
MODEL make_face;

(•topology part*)

   mf_lhs IS_A make_vertex;
   mf_rhs IS_A face;
   mf_lhs.v, mf_rhs.f_eh[1].eh_vl ARE_THE_SAME;
     for i : 2..mf_rhs.num__sides create
       mf_rhs.f_eh[i-1].eh_v2,
         mf_rhs.f_eh[i].eh_vl ARE_THE_SAME;
    end;
   mf_rhs.f_eh[mf^rhs.num_sides].eh_v2,
                  mf_rhs.f_eh[l].eh_vl ARE_THE_SAME;

(*geometry part*)

   mf_rhs.gaLmma.t, mf^lhs.v.t ARE_THE_SAME;
   pop[integer] IS_A point_on_j>lane;
    for i : 1..mf__rhs.num^sides create
     mf^rhs.f__eh[i].eh_vl.local, pop.pt ARE_THE_SAME;
     mf^rhs.gamma.local, pop[i].gamma ARE_JTHE_SAME;
   end;
END make_face;
```

Rule 4. Identify_edge_halves equivalences the eh_other slots of the **edge_halves** to be identified. It also achieves the 2-manifold orientability condition by equivalencing respective vertices.

```
MODEL identify_edge_halves;

(*topology part*)

   e1, e2 IS_A edge_half;
   e1.eh_other, e2 ARE_THE_SAME;
   e2.eh_other, e1 ARE_THE_SAME;
   e1.eh_v1, e2.eh_v2 ARE_THE_SAME;
   e1.eh_v2, e2.eh_v1 ARE_THE_SAME;

(*there is no geometry part*)

END identify_edge_halves;
```

7.5. Using the Types

The types above are the core of a system for creating variational models of solid objects. Using these types demands primitives similar to those created as *primitives* in most boundary representation modelers.

Pgon4 simply sets the integer atom in num_sides slot of a **make_face** to the value 4. On instantiation of this model a quadrilateral will be created.

```
MODEL pgon4 REFINES make_face;
   f.num_sides := 4;
END pgon4;
```

Generic_extrusion_sides builds a ring **of pgon4** models using **identify_edge_halves** models. It establishes an integer parameter that will, on instantiation, determine the number of sides.

```
MODEL generic_extrusion_sides;
   ext_num_sides IS_A integer-
   ex^joins [integer] IS_A identify_edge_halves;
   ext_sides[integer] IS_A pgon4;
   for i:2..ext_num_sides create
     ext_sides[i-l].f.f_eh[3], ext_joins[i].el ARE_THE_SAME;
     ext_sides[i].f.f_eh[l], ext_joins[i].e2 ARE_THE_SAME;
   end;
   ext_sides[ext_num_sides].f.f_eh[3],
                        ext_joins[l] .el ARE_THE_SAME;
   ext_sides[l].f.f_eh[l],
   ext^joins [ext_n\im_sides] .e2 ARE_THE_SAME;
END generic_extrusion_sides;
```

Generic_extruded_solid places a top and bottom face onto a generic_extrusion_sidesf.

```
MODEL generic_extruded_solid REFINES generic_extrusion_sides;
   top, bottom IS_A make_face;
   top_join[integer],
    bottoin^join [integer] IS_A identify^edge^halves;
   top.f.num_sides, bottom.f.num_sides,
                          ext^num^sides ARE_THE_SAME;

   for i:1..ext_num_sides create
     bottom.f.f_eh[i], bottom_join[i].el ARE_THE_SAME;
     ext_sides[i].f.f_eh[4], bottom_join[i].e2 ARE_THE_SAME;
     top.f.f_eh[i], top_join[i] .el ARE_THE_SAME;
     ext_sides[i].f.f_eh[2], top_join[i].e2 ARE_THE_SAME;
   end;
END generic_extruded_solid;
```

Building extruded solids with different numbers of sides reduces to setting the value of the ext_num_sides slot of a **generic_extruded_solid.**

```
MODEL prism3 REFINES generic_extruded_solid;
   ext_jnum__sides := 3;
END prism3;

MODEL prism4 REFINES generic__extruded_solid;
   ext_num_sides := 4;
END prism4;
```

8. Summary and Future Work

I have developed and demonstrated the core of a declarative system for variations. Included in this core is a complete boundary representation solid modeling data structure, created through the use of an unconventional set of Euler operators. A system built on these foundations can be quite terse; this is evidenced by our inclusion of the entire ASCEND code required to build the topology of solid models of planar faced objects. In addition, the necessary constructs are in concept simple and seem to be amenable to crisp formalization. A fully developed variational system would go far towards solving the class of so-called *routine* design problems. It would also provide a sound target representation for search based design systems, whose products are, in essence, variational models.

Several challenges and questions remain in front of this approach. It must be further refold, formalized and extended. For me, obvious relations exist to notions of features, variational geometry, assemblies, and prototypes, and these should be explored. It currently lacks a clear paradigm for graphic interaction, so is unlikely to be widely used in design practice. A particular issue is the development of more graceful means of controlling initialization and redundancy of large equation models. A large portion of the time spent programming variational solids was spent tracking down over- (and worse, under-) specified models. I conjecture that some simple language constructs could greatly assist here.

Variations have long been an important (though implicit) component of automated design systems. That they have not fulfilled their promise can be attributed to many issues, including language for problem formulation, interaction, solving robustness, and speed. The lesson of this work and its future promise is, for me, the simplicity that arises from the discovery of the "right" set of primitive constructs. A few pieces of machinery can go a long way.

9. Acknowledgements

This work has been partially funded by the National Science Foundation through Grant #MSM-8717307, and by an internal grant from the Department of Architecture, Carnegie Mellon University. ASCEND has been partially funded by the National Science Foundation through its Engineering Research Centers program. The author would like thank Peter Piela, the principal

author of ASCEND, for his generosity and patience in providing assistance with ASCEND, Art Westerberg, for his help with ASCEND programming style, Chris Carlson for his outline of one of the proofs, and Linda Gates who assisted with programming the modeler and examples reported here.

Appendix A. Plexes and Plex Productions

The plex grammar in this paper follows [20] and its derivative [21]. The notation includes some minor changes to the syntax, some addition and reordering of concepts, and particularly a different means of visual presentation of plexes. These changes must be presented if the grammar in Section 6 is to be easily understood.

A *NAPE* is a symbol with a set of attaching points for joining to other symbols. *NAPEs* are typed and the set of attaching points is fixed by the type. A *NAPE* type may denoted by any literal, for example **a** or •; and an instance of a *NAPE* type is denoted by the same literal. The set of attaching points of a *NAPE* type a is given by the function /(a). The type of a *NAPE* instance a may also be expressed as r(a).

NAPE types are organized into a *sub-type hierarchy* by the sub-type relation *Q:A* -» A where A is a set of *NAPE* types.

Definition 6: A *NAPE-iype* a is a sub-type of another *NAPE*-type b if/(a)3/(b)+.

A *NAPE-string* is an ordered collection *of NAPEs* formed by concatenating *NAPEs* together.

A *plex* A is a collection of connected *NAPEs.*

$A = X^A F^A A^A$, where:

X^A is the *component list,* an ordered list, delimited by "<" ">", of *NAPEs* and/or *NAPE-strings. NAPE-strings* in X^A expand to their component *NAPEs.* The length of X^A is $/(X^A)$.

F^A is the *joint list,* a list, delimited by "[" "J", of $/(X^A)$ length lists of attaching point identifiers. Each element F_i^* in the joint list describes *one joint* in the plex. Each

†It follows that *Q:A* -> A is reflexive and transitive, but not symmetric.

element γ_j, $j = 1 \ldots l(X^A)$ of Γ_i^A is an attaching point identifier from $I(X_j^A) \cup i_0$, and specifies the attaching point through which X_j^A is connected to the joint. Each Γ_i^A must contain at least two non-null identifiers. The length of Γ^A is $l(\Gamma^A)$.

Δ^A is the *tie-point list*, a list, delimited by "{" "}", of lists of attaching point identifiers. Each element Δ_i^A in the tie-point list describes a joint (called a tie-point) that may be connected to other plex structures. Each element δ_j, $j = 1 \ldots l(X^A)$ of Δ_i^A is an attaching point identifier from $I(X_j^A) \cup i_0$, and specifies the attaching point through which X_j^A is connected to the joint. Each Δ_i^A must contain at least one non-null identifier. The length of Δ^A is $l(\Delta^A)$.

The following restrictions apply to plexes:

1. A *NAPE* cannot connect to itself. In other words, duplicate *NAPE* types appearing in a plex component list refer to distinct *NAPE* instances.

2. A joint list specifies all of the interconnections that occur between members of its associated component list.

3. Every attaching point of every *NAPE* in a plex specification must connect with either a joint in the joint-list or a tie-point in the tie-point list. It follows that every attaching point of every *NAPE* in the plex will be referenced in a plex specification.

Since each component of a plex A is composed from $l(X^A)$ length lists, a plex may be written as an array, the first row being X^A, the next $l(\Gamma^A)$ rows being Γ^A, and the last $l(\Delta^A)$ rows being Δ^A. The columns of the array provide a quick visual check for condition 3 above. Each column below a *NAPE* $\mathbf{a} \in X^A$ must contain all members of $I(\mathbf{a})$.

If, in a plex A, each member $\Gamma_i^A \in \Gamma^A$ has exactly two non-null identifiers and each member $\Delta_i^A \in \Delta^A$ has exactly one non-null identifier then the graph described by the plex has no hyper-arcs (the joints are taken to be arcs and the tie-points are not considered in the graph).

A *plex production* (or *plex rule*) *P* consists of two plexes A (the *LHS*) and B (the *RHS*) with tie-point lists of identical length ($l(\Delta^A)=l(\Delta^B)$). The component and joint lists of A describe the *NAPE*s and their internal interconnection pattern that will be removed from a plex by the plex production. The component and joint lists of B similarly describe what will be added to a plex by

the plex production. The tie-point lists A^A and A^B describe connections to tie-points; for A these are connections that will be recognized in application of the plex production, for B these describe how to connect the newly added *NAPEs* to the rest of the structure. A correspondence exists between members of A^A and A^B; the 1^{th} members each of specify connections to the same tie-point.

The following restriction applies to plex productions:

1. Separate tie-points of either the LHS or the RHS cannot refer to the same joint.

When a plex production from a set of productions *P* applies to a plex A to yield a plex B, then A is said to *directly derive* B in *P,* or symbolically $A \underset{P}{\Rightarrow} B$. If there exists a sequence of zero or more direct derivations using operators from P_9 such that $A \underset{p}{\Rightarrow} \ldots \underset{p}{\Rightarrow} B$ then A *derives* B in *P,* $A \underset{p}{\Rightarrow} B$.

A useful form of monotonicity can be defined for plex productions:

Definition 7: A plex production A —> B is *NAPE-monotonic*^ if:

1. For each *NAPE* a *e* X^A the corresponding *NAPE be* X^B is a sub-type of a,
2. The first $l(X^A)$ elements of F^B form a list identical to f^A, and
3. $A^A = A^B$

A *NAPE-monotonic* rule can be implemented as a model in ASCEND. To see this simply compare the definition of/VAPE-monotonicity with the discussion of direct-monotonicity in ASCEND. It can be seen that M4P£-monotonicity is at least as restrictive as direct-monotonicity.

^**This definition is considerably more restrictive than is really required. Its restrictions make the visual checking of plex rules easy.**

References

1. U. Hemming, "More on the representation and generation of loosely packed arrangements of rectangles", *Planning and Design,* Vol. 16, (1989), pp. 327-359.
2. R. Light, D. Gossard, "Modification of geometric models through variational geometry", *CAD • Computer Aided Design,* Vol. 14, No. 4, July (1982), pp. 209-214.
3. K. Lee, G. Andrews, "Inference of the positions of components in an assembly: pan 2", *Computer-Aided Design,* Vol. 17, No. 1, January (1985), pp. 20-24.
4. A.P. Ambler and R.J. Popplestone, "Inferring the Positions of Bodies from Specified Spatial Relations", *Artificial Intelligence,* Vol. 6, (1975), pp. 175-208.
5. R.J. Popplestone, A.P. Ambler, and I. Bellos, "An Interpreter for a Language for Describing Assemblies", *Artificial Intelligence,* Vol. 14, No. 1, (.1980), pp. 79-107.
6. R.F. Woodbury, IJ. Oppenheim, *An Approach to Geometric Reasoning,* North Holland, IFIP WG 5.2 Workshop Series (1988), ch. 4-7.
7. R.F. Woodbury, IJ. Oppenheim, "An Approach to Geometric Reasoning in Robotics", *IEEE Transactions on Aerospace and Electronic Systems,* Vol. 24, No. 5, (1988), pp. 630-646.
8. R.F. Woodbury, *The Knowledge Based Representation and Manipulation of Geometry,* PhD dissertation, Department of Architecture, Carnegie-Mellon University, December (1987), Also published as EDRC-02-08-88.
9. E.C. Libardi, J.R, Dixon, M.K. Simmons, "Designing with Features: Design and Analysis of Extrusions as an Example", *Spring National Design Engineering Conference,* American Society of Mechanical Engineers, Chicago, II., March 24-27 (1986).
10. E.H. Neilson, J.R. Dixon, M.K. Simmons, "How Shall We Represent the Geometry of Designed Objects?", Tech. report, Department of Mechanical Engineering, University of Massachusetts, (1986).

11. S.C. Luby, J.R. Dixon, M.K. Simmons, "Creating And Using A Features Data Base", *Computers in Mechanical Engineering*, November (1986), pp. 25-33.

12. G.J. Agin, "Hierarchical Representation of Three-Dimensional Objects Using Verbal Models", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. PAMI-3, No. 2, March (1981), pp. 197-204.

13. R.A. Brooks, "Symbolic Reasoning Among 3-D Models and 2-D Images", *Artificial Intelligence*, Vol. 17, (1981), pp. 285-348.

14. P. Piela, *ASCEND, An Object-Oriented Computer Environment for Modeling and Analysis*, PhD dissertation, Department of Chemical Engineering, Carnegie-Mellon University, April (1989).

15. A.A.G. Requicha, "Representation of Rigid Solid Objects", Tech. report TM-29, Production Automation Project, College of Engineering and Applied Science, University of Rochester, June (1980).

16. A.A.G. Requicha, "Representation for Rigid Solids: Theory, Methods and Systems", *Computing Surveys*, Vol. 12, No. 4, December (1980), pp. 437-464.

17. M. Mantyla, "A Note on the Modeling Space of Euler Operators", *Computer Vision, Graphics and Image Processing*, Vol. 26, (1984), pp. 45-60.

18. M. Mantyla, *An Introduction to Solid Modeling*, Computer Science Press, Rockville, Md., (1988).

19. K.J. Weiler, *Topological Structures for Geometric Modeling*, PhD dissertation, Computer and Systems Engineering, Rensselaer Polytechnic Institute, August (1986).

20. J. Feder, "Plex Languages", *Information Sciences*, Vol. 3, (1971), pp. 225-241.

21. R.C. Gonzalez, M.G. Thomason, *Plex Grammars*, Addison-Wesley Publishing Co., Reading, MA., Applied Mathematics and Computation No. 14, (1978), pp. 82-91, ch. 3.4.