Generating Languages of Solid Models

Jeff Heisserman*
Research and Technology
Boeing Computer Services

Robert Woodbury
Engineering Design Research Center
& Department of Architecture
Carnegie Mellon University

Abstract

This paper explores the automatic generation of solid models based on a grammatical paradigm. It presents a formalism, boundary solid grammars, for this purpose. Boundary solid grammars allow complex geometric conditions and operations to be expressed using a logical reasoning mechanism, the construction of powerful rules, and the description of grammars for generation of solid models that is appropriate for a variety of design domains. The formalism is well suited to computer implementation, and we demonstrate a prototype interpreter.

Keywords: spatial grammars, solid modeling, boundary representations, design systems, geometric reasoning.

Introduction

The increasing ubiquity of computer graphics and three-dimensional images of astonishing realism have artfully concealed the fact that important aspects of three-dimensional computer graphics are hard, and in spots the frontier of what is easy has advanced little in 20 years.

Robert Sproull (1990)

Geometric design is central to many areas of engineering, computer graphics, and architecture. Examples of geometric design problems encountered by practitioners in these areas are listed below:

- Compose the rooms of a house according to a particular style and construct a massing model. From this model, construct the roof, floors, ceilings, and interior and exterior walls, and locate doors and windows.
- Given a volumetric model of the bays of a high-rise office building, place the structural columns, beams, shear walls and floor slabs.
- Given a set of components for a computer, lay out the components and design a housing to enclose them.
- Given a model of a part, generate support structures for prototyping the part using a stereo lithography process.
- Construct a model of a set of dies to manufacture a given part.

Current solid modeling systems provide users with little assistance in accomplishing these tasks. They generally provide operations to create, manipulate, and modify models. To construct the models previously mentioned, a user must examine the model and consider what parts should be added. The user must then determine where and how to apply the solid modeling operations to create the new components. Programming facilities are often added, however they provide little more than the ability to create designs with parametric variations.

Robert Sproull expressed this in a recent address [30], continuing the quotation above:

What remains hard is modeling. The structure inherent in three-dimensional models is difficult for people to grasp and difficult too for user interfaces to reveal and manipulate. Only the determined model three-dimensional objects, and they rarely invent a shape at a computer, but only record a shape so that analysis or manufacturing can proceed. The grand challenges in three-dimensional graphics are to make simple modeling easy and to make complex modeling accessible to far more people.

This paper attempts to simplify these modeling tasks through three mechanisms:

1. logical reasoning about solids;
2. rules that match and operate on solids; and
3. grammars that generate languages of solids.

First, the paper reports a method for reasoning about solids using first order logic. The graph based boundary representation of solids determines the basic axioms for reasoning about a set of solids. Clauses are then constructed to express complex conditions. A goal clause may be used to express a condition, and a theorem proving mechanism is then used to determine the satisfiability of the condition.

Second, the paper introduces rules that match on conditions of the solid models and perform modifications on them. These solid rules provide the ability to build "smart" operations that use the logical reasoning mechanism to locate where operations should be applied and how to apply them.

Finally, the paper introduces boundary solid grammars. Informally, a boundary solid grammar consists of an initial solid and a set of rules. The rules of the grammar are applied to the initial solid to generate new configurations of solids, which are in turn modified by additional rules. By applying the rules in this fashion, a grammar generates a multitude of designs, i.e., the members of the language of the grammar.

![Figure 1: A Queen Anne house.](image)

We have produced a prototype implementation of the boundary solid grammar formalism, the Genesis solid grammar interpreter, and have demonstrated its usefulness with a variety of grammars. For the purpose of illustration, we present a grammar that generates Queen Anne houses that is based on an architectural study of Queen Anne houses in Pittsburgh's Shadyside area [11]. A Queen Anne house generated using Genesis and the Queen Anne grammar is shown in Figure 1.

Background

Formal grammars of several forms have been used to generate geometric designs and objects. These formalisms populate a spectrum where, at one end, representations correspond directly to geometry and, at the other end, representations have no correspondence to geometry but are mapped to one or more geometric realizations.

The shape grammar formalism [32] occupies the first end of the spectrum, and has been particularly influential in architectural design. Shape grammars have been developed for various corpora of design, including Palladian villas [33], Frank Lloyd Wright's Prairie Houses [20], Queen Anne houses [10, 11], bungalows [8], modern Italian apartments (after the Casa Giuliani Frigerio) [9], and Japanese tea houses [19]. Additional shape grammars have been written to generate Chinese ice ray designs [31], Moghul gardens [34], and chair-back designs [18].

The work with shape grammars has been limited by the availability of effective shape grammar interpreters. The shape grammar formalism does not lend itself well to computer implementation due to the computational complexity of the matching operations. The available implementations by Krishnamurti [21, 22, 23] and Chase [4] are restricted to 2D arrangements of lines and do not allow parametric matching. The implementation developed by Flemming and Coyne [10] allowed 3D arrangements of lines and parametric matching, but suffered severe computational limitations. Because of these problems, the shape grammars mentioned above have been primarily pencil and paper exercises.

Graph grammars and L-systems, although not geometric representations, have been used to generate graphs or arrays that are mapped to polygons, lines, or primitive solids. Much of this work has been directed at describing biological growth, especially of plants and trees [25, 27, 12, 26, 13, 7]. A similar formalism [15] has been used to generate a variety of fractal-like objects. Schema grammars [14] provide a representation that allows rules to access some of the properties of a geometric realization, while the representation is not inherently geometric.

Carlson's structure grammars [3] define objects with well defined geometric transformations between them. A geometric realization of one of these "structures" requires a mapping between the primitive symbolic objects and geometric primitives. Since a mapping and a structure are decoupled, the mapping can be varied to produce different realizations (this is in addition to varying the rules and grammars that generate the structures). In addition, Carlson has developed an elegant formal mechanism for composing rules and grammars.

Some of the recent feature recognition research uses methods related to those used in boundary solid grammars for reasoning about solids. De Floriani used a face-based boundary representation and features described as topology graphs [6]. Pinilla used an edge-based representation augmented with arcs representing primitive geometric relations (parallel and perpendicular), and located subgraphs generated from a graph grammar description of features [24].

104
Representation of Solids

A boundary solid grammar uses a boundary representation of solid objects. The topology is represented as a graph composed of nodes and arcs. The nodes of these graphs are topological elements, and the arcs represent the adjacencies between elements. The geometry presented here consists of vertex coordinates for polyhedral solids. The topology and geometry together define a boundary representation solid model.

The boundary solid grammar detailed here uses a representation consisting of:

- a topology graph with vertex, edge-half, loop, face, shell and solid nodes, corresponding to the generalized split-edge data structure [16], and arcs representing their adjacencies;
- coordinate geometry in $\mathbb{R}^3$ associated with each vertex; and
- sets of labels associated with topology nodes, as desired; and
- a state associated with each instance of the representation.

![Figure 2: A simple solid.](image)

In order to explain our notation for our boundary representation, we will present the representation in four steps. The first will define the graphs of topological elements and their adjacencies. The second will define the coordinate geometry associated with vertices, providing a complete definition of the boundary representation. The third step will define how to associate non-geometric data to topological elements. The final step will define how to associate a current state to the representation.

To illustrate this representation on a concrete example, we will show the four incremental representations of the tetrahedron shown in Figure 2. In the figures, the vertices are labeled $v_1$ to $v_4$, the edge-halves are labeled by the vertices at either end (the "from" vertex listed first), the loops and faces are labeled by the cycle of vertices about it (in clockwise order as viewed from outside), the shell is labeled $sh_1$, and finally the solid is labeled $s_1$. The visible vertices, edge-halves, and faces are labeled in this fashion in Figure 2.

Definition 0.1 A topology graph over the alphabet $\Sigma_{\text{node}} \cup \Sigma_{\text{arc}}$ is a tuple $T_G = (K, (\rho_a)_{a \in \Sigma_{\text{arc}}})$, where

1. $\Sigma_{\text{node}} = \{\text{vertex, edgehalf, loop, face, shell, solid}\}$ is the alphabet of topological element types;
2. The following specifies the alphabet of relations between topological elements:

   $$\Sigma_{\text{arc}} = \{\text{vertex}, \text{loop}, \text{shell}, \text{solid}, \text{outside}, \text{inside}, \text{connected}, \text{nested}, \text{presolid}, \text{face}, \text{edge}\}$$

3. $K_t$ with $t \in \Sigma_{\text{node}}$ specifies a finite set of nodes of element type $t$. The sets $K_t$ are disjoint. $K = \bigcup_{t \in \Sigma_{\text{node}}} K_t$ is the set of all nodes; and
4. $\rho_a \subseteq K \times K$ are relations (arcs) over $K$, one for each $a$ in $\Sigma_{\text{arc}}$.

![Figure 3: A tetrahedron represented with a b-graph.](image)

The topology graph defines a graph of topological elements and their adjacencies. The nodes are the topological elements of our generalized split-edge representation. The arcs are the topology graph are the topological adjacencies between these elements.

The nodes of the topology graph of the tetrahedron in Figure 2 are shown in Figure 3. They include the four vertices (the black nodes), twelve edge-halves (two for each of the six edges), four loops and four faces (each face has a single loop of edges and vertices), one shell (surface), and one solid.

The topological adjacencies are shown as directed arcs in Figure 3. Although the arcs are not labeled, it is not difficult to infer the adjacency relation from the node types. For example, the arc from the shell $sh_1$ to the solid $s_1$ containing it is a $\text{shell}_{\text{solid}}$ arc, and the arc from $s_1$ to the first shell $sh_1$ is a $\text{solid}_{\text{shell}}$ arc.
Definition 0.2 A boundary graph over the alphabet \( \Sigma_{node} \cup \Sigma_{arc} \cup \mathbb{R}^3 \) is a tuple \( B = (K, (\rho_a)_{a \in \Sigma_{arc}}, \gamma) \), where

1. \( (K, (\rho_a)_{a \in \Sigma_{arc}}, \gamma) \) is a topology graph; and
2. \( \gamma : K_{vertex} \rightarrow \mathbb{R}^3 \) is a function that maps vertex nodes to vertex coordinates.

A boundary graph is a topology graph with coordinate geometry associated with each vertex. The mapping from vertices to vertex coordinates is shown in the upper right corner of Figure 3.

Definition 0.3 A labeled boundary graph over the alphabet \( \Sigma_{node} \cup \Sigma_{arc} \cup \mathbb{R}^3 \cup \Sigma_{label} \) is a tuple \( L = (K, (\rho_a)_{a \in \Sigma_{arc}}, \gamma, \lambda) \), where

1. \( (K, (\rho_a)_{a \in \Sigma_{arc}}, \gamma) \) is a boundary graph;
2. \( \Sigma_{attribute} \) is an alphabet of label attributes; \( \Sigma_{value} \) is an alphabet of label values; \( \Sigma_{label} = \Sigma_{attribute} \times (\Sigma_{value} \cup \mathbb{R}) \) is the set of labels;
3. \( \lambda : K \rightarrow 2^{\Sigma_{label}} \) is a function from nodes to sets of labels;

Labels provide a mechanism for associating non-geometric data with topological elements of solids. Labels may be required as conditions in rules to restrict rule application. Labels may also be used to reduce search by marking conditions needed in future operations.

In Figure 3, a "(mark,a)" label is associated with each of the four faces. This is an attribute-value pair to indicate that there is a "mark" of a type "a" on a face. These attribute-value pairs are also used, for example, to indicate the type of material from which a solid is composed. If we want to indicate that the tetrahedron in Figure 3 is made of steel, then we would use a label on the solid such that \( \lambda(s) = \{ \text{material}, \text{steel} \} \). This information could then be accessed for the computation of the mass of the solid. It could also be used in order to render the solid with the appropriate texture and color.

Finally, we associate a state, from a finite set of states, to the representation. We could refer to this final representation as a "finite state labeled boundary graph." However, we have chosen to use the shorter term, b-graph, to refer to this representation.

Definition 0.4 A b-graph over the alphabet \( \Sigma_s = \Sigma_{node} \cup \Sigma_{arc} \cup \mathbb{R}^3 \cup \Sigma_{label} \cup S \) is a tuple \( B = (K, (\rho_a)_{a \in \Sigma_{arc}}, \gamma, \lambda, \sigma) \), where

1. \( (K, (\rho_a)_{a \in \Sigma_{arc}}, \gamma, \lambda) \) is a labeled boundary graph;
2. \( S = \{ \text{start, done} \} \) is a finite set of states; and
3. \( \sigma \in S \) is the current state.

A state is associated with each labeled boundary graph. The state is used to determine if a given b-graph is a member of the language of the grammar. This will be discussed in greater detail in the discussion of the language of a grammar, later in this section. This finite state mechanism is similar to the state mechanism found in the programmed grammars of Rosenkrantz [28, 29], and the programmed graph grammars of Bunke [1, 2].

The state is also useful as a label on the graph indicating or restricting which rules may apply at the current time. It is also a useful and computationally efficient mechanism for determining if the b-graph is a member of the language of a grammar. Figure 3 illustrates the representation of a b-graph of a tetrahedron.

Let \( B(\Sigma_s) \) denote the set of all b-graphs over the alphabet \( \Sigma_s \), and \( B_e \) denote the empty b-graph. The empty b-graph has a state \( \sigma \in S \) as its current state, but contains no nodes, arcs, or labels.

Not all b-graphs represent rigid solid objects. A b-graph may have nodes and arcs that do not correspond to a physical solid. For example, we can construct a b-graph with an edgehalf node connected to more than one vertex node with edgehalf arcs. This is clearly invalid, since an edge connects exactly two vertices. Even if such simple discrepancies are avoided, it is still possible to generate b-graphs that do not correspond to valid plane models and have faces with inconsistent orientations. Euler operators are used in order to construct only topologically valid b-graphs.

### Reasoning About Solids

We express conditions or features of solids as clauses in first order logic. Explicit conditions of a given b-graph correspond to axioms about the boundary representation of a set of solids. Clauses (in the form of Horn clauses) allow deduction of complex conditions of the solids from simpler conditions. In this way, arbitrarily complex conditions may be specified using deductive reasoning on the solid representation. Locating a condition of a solid then becomes a matter of satisfying a goal clause that specifies the desired condition.

In order to express graph conditions as relations in first order logic, we present a mapping between the graph notation already presented and a relational notation, specified in Table 1. These relations are denoted primitive conditions. We use a logic programming notation [5] for these relations.

Determining the type of topological elements may be represented as a relation \( t(k) \), which denotes \( k \in K_t \). For example, a vertex node \( k \in K_{vertex} \) is denoted by the relation \( \text{vertex}(k) \). Similarly, adjacency relations between topological elements \( (k_1, k_2) \in \rho_s \) may be denoted by relations \( \text{a}(k_1, k_2) \). Then the relation \( \text{edgehalf}(e, v) \) denotes an edgearc arc from \( e \) to \( v \) in \( p_{edgearc} \). Coordinates of a vertex \( \gamma(v) = c \) are denoted by the relation \( \text{v.coord}(v, c) \). For labels, the relation \( \text{label}(k, a, v) \) denotes \( (a, v) \in \lambda(k) \). The current state of a b-graph, \( \sigma = s \), is denoted by the relation...
Table 1: Conversion from graph notation to clausal notation.

<table>
<thead>
<tr>
<th>Node</th>
<th>Graph Notation</th>
<th>Relational Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>( k \in K )</td>
<td>1(k)</td>
</tr>
<tr>
<td>Arcs</td>
<td>((k_1, k_2) \in \rho_a)</td>
<td>a(k_1, k_2)</td>
</tr>
<tr>
<td>Geometry</td>
<td>( \gamma(v) = c )</td>
<td>v.coord(v, c)</td>
</tr>
<tr>
<td>Labels</td>
<td>((a, v) \in \lambda(k))</td>
<td>label(k, a, v)</td>
</tr>
<tr>
<td>State</td>
<td>( \sigma = s )</td>
<td>state(s)</td>
</tr>
</tbody>
</table>

state(s). Relations that directly relate topological element types and adjacency relations, coordinates, and labels are considered primitive conditions.

Satisfying primitive conditions of b-graphs is a straightforward task of locating nodes and arcs, and accessing their type, labeling, and geometric functions. These serve as axioms for deducing additional conditions.

For example, we can specify that there exists a specific edge-half \( e \) in a given b-graph by edge_half(e, v) which corresponds to finding a node \( e \in K_{\text{EDGEEHALF}} \). We can find the vertex \( v \) associated with that edge-half by the clause edgeh_v(e, v). This corresponds to locating an edgeh_arc in the b-graph from \( e \) to \( v \). The coordinates of the vertex can be accessed with the clause v.coord(v, (x, y, z)) which corresponds to the geometric function \( \gamma \), where \( (x, y, z) = \gamma(v) \).

A label \( l \) on an edge-half \( e \) can be found by specifying label(e, l) and locating \( l \) in the set of labels associated with that edge-half, \( l \in \lambda(e) \). Finally, the state of a given b-graph may be found by specifying state(s).

Conditions may be combined using clauses in first order logic. A clause

\[ A \leftarrow B_1, \ldots, B_n \]

can be constructed to specify the conjunction of \( B_1, \ldots, B_n \).

For example, we express the length of an edge-half as the distance between the coordinates of the vertices associated with that edge-half and its other edge-half:

\[
\text{eh_length(Eh, Length):- edgeh_v(Eh, V1), other_ed(Eh, OtherEh), edgeh_v(OtherEh, V2), distance_v(V1, V2, Length).}
\]

where distance_v(V1, V2, Length) calculates the Euclidean distance between the coordinates of two vertices V1 and V2.

In this way, diverse graph conditions and conditions on the geometry can be combined to form higher level conditions. Some of these conditions include:

- the areas of faces, volumes and masses of solids;
- angles between edges and faces;
- orientations of faces and solids;
- coincident, colinear or coplanar vertices, edges and faces;
- the centers of edges, faces and solids;
- the moment of inertia of faces and solids;
- intersections of lines, planes, surfaces, edges, faces, and solids, including the intersection of two solids (Boolean intersection) and self-intersection of a solid [16].

A single condition may be used to express the conditions of an infinite set of graphs using recursive definitions. Alternately, a condition may match greatly varied topology graphs using several clauses with the same head.

**Operations on Solids**

Operations are used to transform one b-graph into another. Primitive operations modify the topology using Euler operations, modify the geometry using vertex coordinate assignment, add and remove labels, and modify the current state. Primitive operations are expressed as extra-logical relations, that directly modify the b-graph representation as a side-effect of being satisfied. Complex operations may be constructed from conditions and operations as clauses in first order logic. Complex operations may then produce a different effect depending on the context of their application.

Applying an operation to a solid then becomes a matter of satisfying a goal clause that specifies the desired operation.

Primitive operations have several forms. Euler operations modify boundary representation by adding and removing topological elements and relations, while maintaining valid topological adjacencies. They may be viewed as graph productions. Assigning coordinates \( c \) to a vertex \( v \) is equivalent to changing \( \gamma \) to \( \gamma' \) such that \( \gamma'(v) = c \). Adding a label \( l \) to an element \( e \) is accomplished by modifying the labeling function \( \lambda \) to \( \lambda' \) such that \( \lambda' = \lambda(e) \cup \{l\} \). Killing a label \( l \) modifies the labeling function \( \lambda \) to \( \lambda' \) such that \( \lambda' = \lambda(e) \setminus \{l\} \). Modifying the current state of the b-graph is equivalent to modifying \( \sigma \) to \( \sigma' \) such that \( \sigma' = \sigma \). The Euler operations, coordinate assignment, labeling operations, and state modification comprise the primitive operations.

Euler operations are specified as extra-logical relations that modify the topology graph of a solid. The relational form of these Euler operators are listed in Table 2.

For example,

\[
\text{new}(V, Eh, NewV, NewEh)
\]

is equivalent to the \text{new} Euler operation.
mssflv(NemS,NevSh,NeuF,NeuL,NeuV)
merge.solids(S1,S2)
mssflv(S,NevSh,NeuF,NeuL,NeuV)
mev(V,Eh,NeuV,NewEh)
esplit(Eh,NewEh,NeuV)
mefl(V1,PredSh,V2,SuccEh,NewEh,NewL,NewF)
kegl(Eh,NewL)
glu(F1,F2)
ksv(V1,V2)
kvmg(V1,V2)
keg(Eh1,Eh2)

Table 2: Euler operations as primitive operations.

The inversion operation and generalized unary intersection described in [16] are specified as
invert(S,NewS)
and
unary(N,S,NewS).

Vertex coordinates are assigned using
set.vertex(v,c),
which is equivalent to modifying γ to γ' such that γ'(v) = c.
Element labels are created using
make.label(e,a,v),
which is equivalent to changing λ to λ' such that λ'(e) = λ(e) ∪ {(a,v)}, or
kill.label(e,a,v)
which modifies λ to λ' such that λ'(e) = λ(e) − {(a,v)}. The current state of the b-graph is modified with
set.state(s),
which is equivalent to modifying σ to σ' such that σ' = s.

Primitive operations derive one b-graph from another according to the following definition:

Definition 0.5 A b-graph b' ∈ b(Σa) is directly derivable from another b ∈ b(Σa) by a primitive operator O (abbreviated b → b') iff

1. the arguments to O have valid types and adjacencies for that particular operation;
2. O applied to b produces b'.

Complex operations are specified by sequences of conditions and operations. Primitive operations modify the topology, geometry, labels, and state of a b-graph. Conditions, as described above, are used to control the use of operations in modifying the given b-graph. Clauses define high level operators from existing operators and conditions. The conditions allow an operator to respond to the context in which it is applied.

Conditions and operations may be combined using clauses in first order logic. A clause

\[ A \leftarrow B_1, \ldots, B_n. \]

can be constructed to specify the conjunction of B₁, ..., Bₙ. Operationally, the satisfaction of A results as the sequential satisfaction of B₁, ..., Bₙ.

For example, the point_face operator, illustrated in Figure 5, will pull any face out into a point, correctly modifying the topology while considering the number of edges and vertices of the face. The clauses for the point_face operator are listed below.

\[
\text{point_face}(\text{Face}, \text{Height}):- \\
\text{face_eh}(\text{Face}, \text{Eh}), \\
\text{ccw_eh}(\text{Eh}, \text{LastEh}), \\
\text{edgev_eh}(\text{Sh}, \text{V}), \\
\text{face_normal}(\text{Face}, \text{Normal}), \\
\text{face_center}(\text{Face}, \text{Center}), \\
\text{mev}(\text{V}, \text{LastEh}, \text{VTop}, \text{EhBt}), \\
\text{other_eh}(\text{EhBt}, \text{EhTb}), \\
\text{scalar}(\text{Height}, \text{Normal}, \text{Direct}), \\
\text{vecplus}(\text{Center}, \text{Direct}, \text{CTop}), \\
\text{set_vertex}(\text{VTop}, \text{CTop}), \\
\text{point_face_1}(\text{LastEh}, \text{Eh}, \text{VTop}, \text{EhTb}).
\]

\[
\text{point_face_1}(\text{EndEh}, \text{EndEh}, \_ , \_).
\]

\[
\text{point_face_1}(\text{EndEh}, \text{Eh}, \text{VTop}, \text{EhTb}):- \\
\text{cw_eh}(\text{Eh}, \text{NextEh}), \\
\text{edgev_eh}(\text{NextEh}, \text{V}), \\
\text{v.coord}(\text{V}, \text{C}), \\
\text{mev}(\text{V}, \text{Eh}, \text{VTop}, \text{EhBt}, \text{NewEhBt}, \_ , \_ ), \\
\text{other_eh}(\text{NewEhBt}, \text{NewEhTb}), \\
\text{point_face_1}(\text{EndEh}, \text{NextEh}, \text{VTop}, \text{NewEhTb}).
\]

General operators, such as offsetting, Boolean, and unary shape operators, may be constructed in this way. In addition, we can construct a single operation to be used in place of a set of rules.

**Solid Rules**

A solid rule is specified by a set of match conditions, and a sequence of operations. The match conditions of a rule determine when the rule may be applied to a given b-graph.
Each of the rule's conditions must be satisfied with respect to the given b-graph, and bindings for all the free variables must be found. When the rule is applied, the sequence of operations transform the b-graph, modifying the representation of the solid(s) or creating additional solids.

**Definition 0.6** A solid rule $R$ is a pair $\alpha \Rightarrow \beta$, where

- $\alpha$ is a goal clause $C_1, \ldots, C_n$, where each condition $C_i$ is a subgoal that must be satisfied in order to apply the rule; and
- $\beta$ is a goal clause $O_1, \ldots, O_n$, where each operation $O_i$ is considered a subgoal. If the operations $O_1, \ldots, O_n$ of $\beta$ are not satisfied with respect to the given b-graph then $\beta$ does not modify the b-graph (i.e. $\beta$ is equivalent to the identity transformation).

We can clarify the idea of a solid rule by looking at three of the rules of the Queen Anne grammar. The first rule we discuss locates and designates a room on the first floor to be the parlor. The second rule locates the dining room next to the kitchen. The third rule locates a second floor above a room on the first floor.

The solid rule definitions used in Genesis have three parts: the description, the lhs and the rhs. The description is a textual description of the rule that may be presented to the designer when the rule matches. The lhs specifies the match condition of the rule, where the first argument of lhs is the name of the solid rule, the second argument is a list of common variables between the lhs and rhs. The third argument of the rhs is a list of topological elements that will be highlighted in the graphical display of the solids when the rule matches, along with the description, to indicate which rule has matched and where. The rhs defines the operations which will be executed when the rule is applied.

**Figure 6:** A rule for locating the parlor in a Queen Anne house.

Rule qa-6 looks for a room at the front corner of the house and marks the room as the parlor. It will consider any solid labeled with a “name, room” label that has a vertex with a “mark, front” label. When the rule is applied, the “name, room” label is removed from the solid and a “name, parlor” label is added. The color of the solid is then set to the color designated for the parlor. The state is used here to control which set of rules may be applied during a particular stage of design. This is illustrated in Figure 6.

**Figure 7:** A rule for locating the dining room in a Queen Anne house.

Rule qa-11 selects a room to be the dining room. The rule matches on an unnamed room which is adjacent to the kitchen, and marks the room as a dining room. adjacent_solids determines whether two solids are adjacent by finding faces of the respective solids that are coplanar, overlap and have opposite orientations. The application of qa-11 is shown in Figure 7.

**Figure 8:** A rule for adding the second floor to a Queen Anne house.

Rule qa-16 creates a 2nd floor room on top of each 1st floor room. It looks for any first floor room (excluding the
stairway) and stacks a room on top of the first floor room.
Rule qa.16 is illustrated with Figure 8.

description(qa_16,
   'Add a second floor room above a existing room.').

lhs(qa_16, [Top, Height, Room], [Room]):-
state(second),
label(Room, name, S),
member(S, [room, parlor, kitchen, dining, hall, pantry]),
not(label(Room, mark, stacked)),
top(Top, Room),
room_height(Height).

rhs(qa_16, [Top, Height, NewRoom]):-
stack solid(Top, Height, NewRoom),
make label(NewRoom, floor, second),
set solid color(NewRoom, RColor).

---

Figure 9: Generating a Queen Anne house.

Solid rules derive one b-graph from another according to
the following definition:

Definition 0.7 A b-graph \( b' \in b(\Sigma_{b}) \) is directly derivable from another \( b \in b(\Sigma_{b}) \) by a solid rule \( R \) (abbreviated \( b \xrightarrow{R} b' \)) iff

1. the goal \( \beta_R \) is satisfiable with respect to the primitive conditions derivable from \( b \) and the clauses in \( P_C \);

2. one of two cases holds:
   a. the goal \( \beta_R \) is satisfiable with respect to the primitive conditions derivable from \( b \) and subsequent b-graphs as modified by the operations in \( \beta_R \), and with respect to the clauses in \( P_C \) and \( P_O \); and \( b' \) is directly derivable from \( b \) by the sequence of primitive operations applied in satisfying \( \beta_R \); or
   b. the goal \( \beta_R \) is unsatisfiable and \( b' = b \).

Grammars & Languages of Solids

The term "grammar" is used here as the generative specification of families of models. A grammar consists of a representation, an initial model in that representation, and a set of rules. A rule is applicable to a model if it can match on that model, in which case it transforms the model to a new form. Rules may be applied to an initial model, and to models produced thereof, to produce new models. Thus, applying different rules, and applying them in different ways, allows us to generate a family or language of different models.

With the representation, reasoning mechanism, and solid rule definitions in hand, we can now define boundary solid grammars.

Definition 0.8 A boundary solid grammar is a tuple \( \mathcal{G} = (\Sigma_{b}, I, P, R) \), where

1. \( \Sigma_{b} \) is as defined for b-graphs;
2. \( I \in b(\Sigma_{b}) \cup \{b_{r}\} \) is a topologically valid b-graph;
3. \( P = P_C \cup P_O \) is a finite set of clauses, where \( P_C \) is a set of clauses specifying conditions, and \( P_O \) is a set of clauses specifying operations; and
4. \( R \) is a finite set of solid rules, constructed with the clauses of \( P_C \) and \( P_O \).

An initial solid of the boundary solid grammar formalism is a valid b-graph. This b-graph may be the empty b-graph, \( B_r \), or it may be a b-graph consisting of an number of (arbitrarily complex) solids. Modifications of the initial solid, and subsequent solids, are accomplished by the application of the set of solid rules.

We can abbreviate \( \mathcal{G} = (\Sigma_{b}, I, P, R) \) to \( \mathcal{G} = (I, P, R) \) since \( \Sigma_{node}, \Sigma_{arc}, \mathcal{R}^{2} (\Sigma_{b}) \) are fixed for a given boundary representation, and \( \Sigma_{label} \) may be a fixed alphabet of labels.

The sentential set \( S(\mathcal{G}) \) of a boundary solid grammar \( \mathcal{G} \) is the set of b-graphs (sentential b-graphs) which contains the initial b-graph and all b-graphs which can be generated from the initial b-graph using the solid rules of the grammar.

A boundary solid grammar \( \mathcal{G} \) generates a language \( L(\mathcal{G}) \) of b-graphs. The language of a boundary solid grammar is the set of sentential b-graphs where the current state is done, i.e., \( L(\mathcal{G}) = \{b \in S(\mathcal{G}) \mid \sigma(b) = \text{done}\} \).
Conclusions

We have described the foundations for the automatic generation of solid models based on a grammatical paradigm. The boundary solid grammar formalism allows complex geometric conditions and operations to be expressed using a logical reasoning mechanism, the construction of powerful rules, and the description of grammars for generation of solid models that is appropriate for a variety of design domains.

Although we have primarily discussed the Queen Anne grammar, we have built a variety of additional grammars. These grammars generate various geometric forms, including a three dimensional variant of the Koch snowflake, Sierpinski sponges, fractal-like mountains, and spiral snail shells. More powerful grammars have been implemented that generate supports for models to be made by a stereo lithography (sla) process, computer housings for single board computers, structural designs of high rise buildings, and designs for firestations.

Our experience with the Genesis boundary solid grammar interpreter has shown that the methods presented help to “make simple modeling easy and to make complex modeling accessible to far more people” [30].

Our goal is to provide designers with tools for design in a variety of engineering and architectural domains. It is clear that the present formalism is not sufficient for these more general design tasks. Additional representations will be needed for reasoning and analysis, such as representations of electrical and hydraulic schematics, kinematics, and structural and aerodynamic meshes. Mechanisms will be needed to allow matching decisions to be based on combined analysis of conditions of several representations.

Acknowledgements

This work was supported in part by the National Science Foundation, and with equipment grants from Hewlett-Packard and Intel Corporation. We would like to thank Dana Scott, Ramesh Krishnamurti, Ulrich Flemming, Chris Carlson, Spiro Michaylov, and Nevin Heintze for reading and commenting on various versions of this paper. We would also like to thank Shih Hai Chiang and Ching Yin for their work on the Queen Anne grammar, and Clauss Strauch, Mikako Harada and Spiro Michaylov for their contributions to the development of Genesis.

References


Figure 10: Queen Anne houses.

The initial (empty) solid and solid rules define a boundary solid grammar, and produce a language of Queen Anne houses. The current Queen Anne grammar contains 136 rules, with rules that generate room layout, interior and exterior walls, window layout and details, fireplaces and chimneys, roofs, dormers and porches. A portion of this language of the grammar is shown in Figure 10. The designs generated by the grammar may have very complex forms, and that there is an huge number of members in this language. (Based on a conservative combinatorial analysis, the Queen Anne grammar generates more than five trillion different houses.)

Genesis is an implementation of the boundary solid grammar formalism, and closely follows the formalism as we have presented it. Genesis provides facilities for representation and display of solids, match conditions, solid modeling operations, rule and grammar definition, and searching through the language of a grammar. A detailed discussion of the implementation of Genesis is presented by Heisserman [16], and its facilities are described in the Genesis Reference Manual [17].

Genesis generates a detailed Queen Anne house, composed of about 135 solids and 1200 faces, in less than one minute (clock time), running on a Hewlett-Packard 835 workstation configured with 32 Megabytes of physical memory and 160 Megabytes of virtual memory. This time includes updating the graphical display of the model after each of approximately 70 rule applications (with hidden surface removal and three directional light sources plus ambient light).


SESSION 2: Boundary Evaluation and Robustness