First International Conference on Design Computing and Cognition

Workshop 3

Implementation Issues in Generative Design Systems

Saturday 17 July 2004, 14:00 - 17:45

GRAMMARS, *SORTS* AND IMPLEMENTATION

RUDI STOUFFS
Delft University of Technology, Faculty of Architecture, BT/TO&I, PO Box 5043, 2600GA Delft, The Netherlands
r.stouffs@bk.tudelft.nl

and

RAMESH KRISHNAMURTI
Carnegie Mellon University, College of Fine Arts, School of Architecture, Pittsburgh, 15213-3890 PA, USA
ramesh@cmu.edu

Abstract. We consider an extension of the algebraic framework for shape grammars to various information types. We denote this framework *sorts*. We describe an implementation of *sorts* in Java.

1. Introduction

Grammar formalisms have been around for over 40 years and have found application in a wide variety of disciplines and domains, to name a few, natural language, architectural design, mechanical design, and syntactic pattern recognition. Their implementations, however, have been mostly narrowly focused and sparse. In design, in particular, the expectation of grammar formalisms or similar rule-based systems to pervade design software has so far remained only an illusion. There are three main reasons for this. The first relates to the difficulty stemming from technical considerations of implementing grammars. The second difficulty pertains to ways of enabling designers to employ grammatical rules in a manner that does not impede their act of designing. The third difficulty affects the rapid development, adaptation, and maintenance of grammar-based systems.

Grammar formalisms come in a large variety, requiring different representations of the objects being generated, and different interpretative mechanisms for this generation. At the same time, all grammars share certain definitions and characteristics. Grammars are defined over an algebra of objects, U, that is closed under the operations of addition, +, and subtraction, –, and a set of transformations, F. In other words, if u and v are members of

U, so too are $u + f(v)$ and $u - f(v)$ where f is a member of F. In addition, a match relation, $\leq$, on the algebra governs when an object occurs in another object under some transformation, that is, $f(u) \leq v$ whenever u occurs in v for some member f of F, if u and v are members of U.

Computer implementation of shape grammars has been of interest for some considerable time. Most implementations of grammar interpreters apply to two dimensional shapes (Gips 1975; Krishnamurti 1982; Krishnamurti and Giraud 1986; Chase 1989; Tapia 1996; 1999; McCormack and Cagan 2002). Few have been implemented for three dimensional shapes and, mostly, these are restricted to certain kinds of shapes (Piazzalunga and Fitzhorn 1998), or based on representations that do not readily provide for 'emergent' or unanticipated subshape relationships (Longenecker and Fitzhorn 1991; Heisserman 1991; 1994), or were developed for specific purposes (see for example, Flemming 1987; Agarwal and Cagan 1998).

The technical machinery required for shape grammar implementations has, to a large extent, been established. Krishnamurti and Stouffs (2004) present a unified approach for arithmetic in any shape algebra, including curves and surfaces. Krishnamurti and Earl (1992) investigate shape recognition in U_{13}; Krishnamurti and Stouffs (1993; 1997) consider, respectively, shape recognition in U_{23} and in the cartesian product $U_0 \times U_1 \times U_2 \times U_3$. The notation U_{ij} refers to linear shapes made up of i-dimensional elements in j-dimensional Euclidean space, and U_i is shorthand when the dimensionality of the space is known (Stiny 1991).

However, most practical problems of generative design are not limited to geometry only. Part of the attractiveness of the algebraic model underlying shape grammars is its ability to include non-geometric attributes, such as labels (Stiny 1980; 1990), weights (Stiny 1992) and colors (Knight 1989). These augmented shapes are derived from shapes of spatial elements by associating symbols, labels or properties to the elements. Consequently, their algebraic operations are redefined in order to deal correctly with the associated symbols. The result is a different grammar formalism each time the attribute type is altered, which does not support the rapid development, adaptation, and maintenance of grammar-based systems.

Instead, by considering algebras not only for shapes but for many different types of information and by considering compositional operations on algebras, a framework can be established that enables the exploration of different grammar formalisms, based on a variety of algebras and interpretative mechanisms. *Sorts* (Stouffs and Krishnamurti 2002) implements such a framework. *Sorts* constitute a model for representations that defines formal operations on *sorts* and recognizes formal relationships between *sorts*. Each *sort* defines an algebra over its elements; formal compositions of *sorts* derive their algebraic properties from their component

sorts. As such, *sorts* enable the development of alternative representations of a same entity or design, the comparison of representations with respect to scope and coverage, and the mapping of data between representations, as well as data recognition and the specification of design rules.

Sorts can be considered as class structures identified by compositions of named data entities (Stouffs et al. 1996). These data entities are identified by a type specifying the set of possible values. Exemplar types are labels and numeric values or weights, and spatial types such as points, line segments, plane segments and volumes. Data entities are composed or grouped using one or more constructors, these are devices for relating entities together. At this time, we consider two constructors, resulting in either a subordinate composition of properties or a disjunctively co-ordinate composition. Others can be defined.

We are implementing *sorts* using an object-oriented approach (in Java). This modular approach enables the inclusion of new entity types (and constructors) without any modifications to the rest of the code. We distinguish three major types of object classes (figures 1 and 2). *Individuals* define the data entities (The term individual refers to Stiny's treatise of shapes as individuals (1982)). *Forms* are collections of *individuals* from the same sort; the object class defines the respective constructor. *Sorts* define the class structures.

2. Individuals and data operations

Grammars rely on a match relation to be defined on each algebra, which governs when an object occurs in another object under some transformation. This partial order relation is crucial to all data, both individuals and forms of individuals. The ability to compare individuals as to whether one is less than, greater than, less than or equal to, or greater than or equal to another is encoded in the *Element* class which is an abstract super class to both the classes *Individual* and *Form* (figure 1).

Each algebra must further specify the operations of sum and difference (and product). These operations on forms (or collections) of individuals can be expressed in terms of operations of combine and complement of one individual with respect to another. Corresponding methods are defined in the abstract *Individual* class and inherited or overwritten by each subclass, where each subclass specifies a particular data entity type. Currently, the following entity types are implemented: alphanumeric (labels) and numeric values and (numeric) weights, points, (infinite) lines, line segments and (infinite) planes, circles and (circular) arcs, URLs for texts and images, and unique IDs, (positive or negative) signs and property relationships. Below, we present a few exemplar entity types in detail.

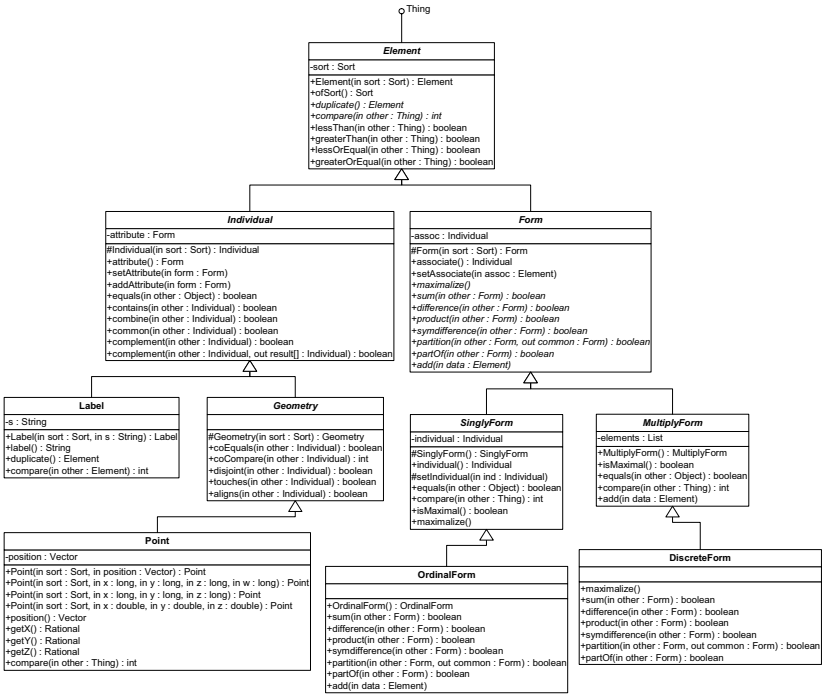

Figure 1. UML diagram depicting the abstract classes Element, Individual and Form and a number of exemplar subclasses, such as Label, Point, OrdinalForm and DiscreteForm.

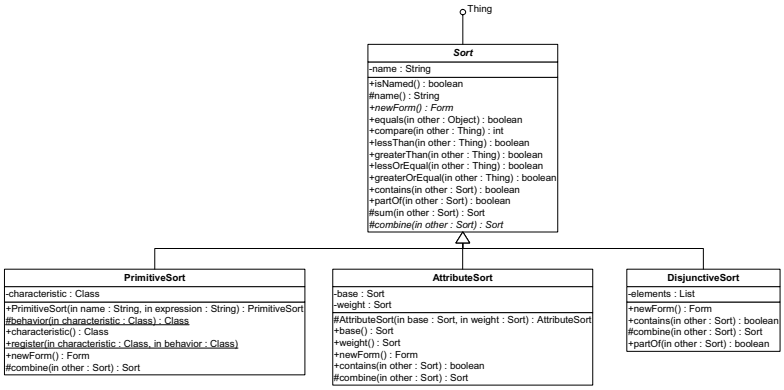

Figure 2. UML diagram depicting the abstract class Sort and three exemplar subclasses, PrimitiveSort, AttributeSort and DisjunctiveSort.

A *sort* of labels defines a label as a string of characters. In the algebra of labels, the empty label (string) defines the neutral element. Then, two labels combine if and only if both labels are identical, and the complement of a label with respect to another label is the empty label, if both labels are identical, and is the first label, otherwise.

A *sort* of (numeric) weights defines a weight as a positive (floating-point) value. 0 defines the neutral element in this algebra. Two weights always combine to form a single weight, the value of which is the maximal value of both weights. The complement of one weight with respect to another is 0, if this weight's value is less than or equal to the other, and is the weight itself, otherwise.

In the case of geometric data entities, an individual is always represented by a carrier, or co-descriptor, and, possibly, a boundary. Two individuals of the same geometric algebra combine only if they have the same co-descriptor and are not disjoint. Two individuals with equal co-descriptor are said to be disjoint if they do not overlap (do not share a common part) and they do not touch (do not share a common boundary). The abstract class *Geometry* defines methods to compare co-descriptors and to determine whether two individuals are disjoint, touch or are aligned (if they share a common part and a common boundary) (figure 1). This class is the super class for all geometric individual classes.

A *sort* of points defines a point as a vector of (rational) coordinates. A *nil* individual is defined to be the neutral element for this algebra. The co-descriptor of a point is the point itself, a point has no boundary. Similarly to labels, two points combine if and only if both points are identical, and the complement of a point with respect to another point is *nil*, if both points are identical, and is the first point, otherwise.

A *sort* of (infinite) lines defines a line as a couple of vectors. The first vector defines the direction of the line, the second the root point of the line (the intersection point of the line and a perpendicular line through the origin). Again, a *nil* individual is defined to be the neutral element for this algebra. The co-descriptor of a line is the line itself, an infinite line has no boundary. Similarly again, two lines combine if and only if both lines are identical, and the complement of a line with respect to another line is *nil*, if both lines are identical, and is the first line, otherwise. The class *Line* serves as a super class for the class *LineSegment*. A *sort* of (bounded) line segments defines a line segment as a line with, additionally, a couple of position vectors. This line defines the co-descriptor of the line segment, the position vectors define the boundary positions of the segment. Two line segments combine if and only if their co-descriptors are identical and they are not disjoint. The complement of a line segment with respect to another line segment is a new line segment, if both co-descriptors are identical and the

line segments are not disjoint and do not touch, and is the first line segment, otherwise. The new line segment is defined in terms of the boundary positions of both line segments (see below).

3. Forms and data behaviors

The abstract class *Form* defines the operations of sum and difference (and product), and if one form is part of another form. These operations can be defined in terms of (among others) the operations combine and complement. However, this relationship is dependent on the way the operations of combine and complement are defined and thus on the type of individuals. For example, the operations of combine and complement are similarly defined for *sorts* of labels, points and (infinite) lines (see above). We denote this relationship between the algebraic operations of sum and difference and the operations of combine and complement on individuals the (operational) *behavior* of the *sort*.

The behavior of the *sorts* of points and labels is a discrete behavior, corresponding to mathematical set operations. In this case, the part relation is defined as a subset relation, and the operations of sum, difference and product correspond to set union, difference and intersection, respectively. In other words, if a and b denote two forms of a *sort* with discrete behavior, and A and B denote the corresponding sets of data entities (e.g., points or labels), then ($a : A$ specifies A as a representation of a)

$$\begin{aligned} a : A \wedge b : B \;\Rightarrow\; & a \leq b \Leftrightarrow A \subseteq B \\ & a + b : A \cup B \\ & a - b : A / B \\ & a \cdot b : A \cap B \end{aligned} \tag{1}$$

Weights adhere to a different behavior. Considered weights to denote thickness for points and lines (or tones for surfaces and volumes). Then, a behavior for weights becomes apparent from drawings: a single line drawn multiple times, every time with different thickness, appears as it was drawn once with the largest thickness, even though it assumes the same line with other thickness (Stiny 1992). This behavior is termed *ordinal*; the part relation on weights corresponds to the less-than-or-equal relation on numbers;

$$\begin{aligned} a : \{x\} \wedge b : \{y\} \;\Rightarrow\; & a \leq b \Leftrightarrow x \leq y \\ & a + b : \{\max(x, y)\} \\ & a - b : \{\} \text{ if } x \leq y, \text{ else } \{x\} \\ & a \cdot b : \{\min(x, y)\} \end{aligned} \tag{2}$$

An *interval* behavior applies to line segments (as well as intervals of time or other one-dimensional quantities). A specification of the interval behavior

can be expressed in terms of the behavior of the boundary (positions) of the interval. Let $B[a]$ denote the boundary of a form a. In the case of a form of line segments, the boundary of this form is the collection of boundary positions from all line segments. This boundary can be partitioned with respect to another form b of line segments by distinguishing those boundary positions that lie within a line segment from b, those that lie outside of all line segments from b, and those that belong to the boundary of b. Let I_a denote the boundary of a that lies within b and O_a denote the boundary of a that lies outside of b. Let M denote the shared boundary of a and b where the respective line segments are aligned, and N the shared boundary of a and b where the respective line segments touch (Stouffs and Krishnamurti 2004; figure 3). Then,

$$\begin{aligned} a : \mathrm{B}[a] \wedge b : \mathrm{B}[b] \;\Rightarrow\; & a \leq b \Leftrightarrow I_a = 0 \wedge O_b = 0 \wedge N = 0 \\ & a + b : \mathrm{B}[a + b] = O_a + O_b + M \\ & a - b : \mathrm{B}[a - b] = O_a + I_b + N \\ & a \cdot b : \mathrm{B}[a \cdot b] = I_a + I_b + M \end{aligned} \tag{3}$$

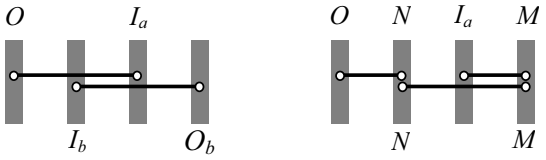

Figure 3. The specification of the boundary collections I_a, O_a, I_b, O_b, M and N, given two collections of intervals a (above) and b (below).

Similar behaviors can be specified for plane segments and volumes (Stouffs and Krishnamurti 2004) as well as hypersegments of higher dimension; (3) still applies though the construction of I_a, O_a, I_b, O_b, M, and N is correspondingly more complex (figure 4).

4. Sorts and data compositions

So far, we have considered only *sorts* that are made up of a single type of data entities, such as *sorts* of labels, points or line segments. We denote these *primitive sorts*. Primitive *sorts* can be combined into *composite sorts* under formal compositional operations. Examples of composite *sorts* are labeled points and weighted line segments, but also the combination thereof. At this time, we consider two formal operations for composing *sorts*. The *attribute* operator specifies a subordinate composition of *sorts.* Under the attribute operator, an individual of the resulting *sort* is an individual of the first operand *sort* that has as attribute a form of the second operand *sort*. For

example, a form of the *sort* of labeled points specifies a collection of points, where each (individual) point has a form of labels assigned as attribute.

$$labeled\_points : points \wedge labels \tag{4}$$

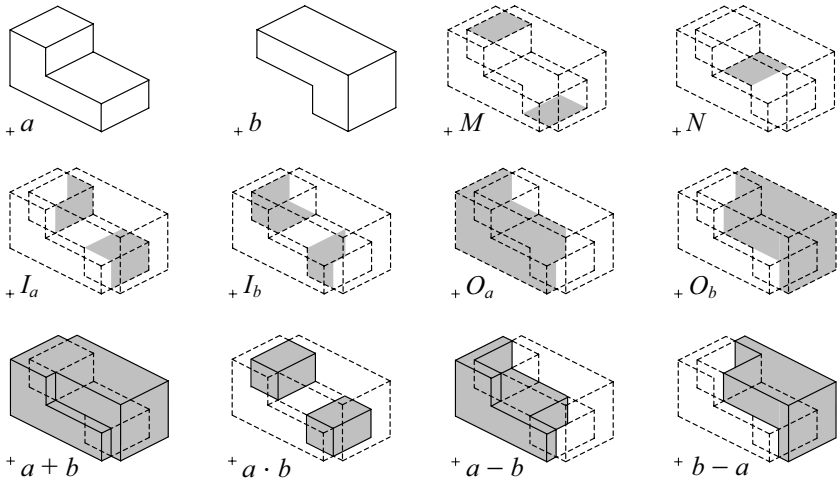

Figure 4. The boundary collections I_a, O_a, I_b, O_b, M and N for two volumes a and b, and the collections of volumes resulting from the operations $a + b$, $a \cdot b$, $a - b$ and $b - a$.

The operation of *sum* allows for disjunctively co-ordinate compositions of multiple *sorts*. Under the operation of sum, a form of the resulting *sort* is a combination of forms of each of the operand *sorts*. For example, a form of the *sort* of labeled points and line segments specifies a collection of labeled points and/or line segments. Such a form may contain only labeled points, only line segments, or both labeled points and line segments.

$$shapes : linesegments + labeled\_points \tag{5}$$

4.1. COMPOSITE BEHAVIORS

A composite *sort* defines its own algebra, which is a composition of the operand algebras in relationship to the formal compositional operator that defines the composition. As such, a composite *sort* can also be considered to define a behavior. This behavior is inherited from the component *sorts* in a manner that depends on the compositional relationship. Under the operation of sum, the behavior is that of the component *sort* for each component. Forms from different component *sorts* never interact, the resulting form,

corresponding the composite *sort*, is the collection of forms from all component *sorts*. When an operation applies to two forms of the same composite *sort*, the operation instead applies to the respective component forms.

The attribute operation on *sorts* specifies a dependency relation on the *sorts* in a composition, where each component, except the first, defines an attribute *sort* to the previous component. That is, a corresponding form consists of individuals of the first component *sort*, each element of which has, as attribute, another form corresponding to the *sort* as a composition of all but the first component, in a recursive manner. Thus, the behavior of such a *sort* is defined by the behavior of its first component *sort*. Specifically, when an operation applies to two forms of the same composite *sort* (under the attribute relationship), identical individuals combine and their attribute forms are composed under the same operation. Any resulting individuals that have an empty attribute form are removed.

4.2. *SORT* CLASSES

The abstract class *Sort* defines a generalized *sort* (figure 2). For efficiency purposes, a partial order is defined on *sorts* that allows these to be compared as to whether one is less than, greater than, less than or equal to, or greater than or equal to the other. A *sort* can be provided a name, *sorts* can be combined under the attribute operation or the operation of sum, and *sorts* can be compared as to whether one contains another, or vice versa, one is part of another.

Primitive *sorts* are distinguished by their name and their type (figure 2). This type is denoted the *characteristic* individual of the *sort* and is represented by the respective class. Primitive *sorts* have their behavior specified as part of their characteristic individual. A static method in the class *PrimitiveSort* allows each class of individuals to register itself and specify a behavior (or class of forms) at the same time. In this way, classes of individuals can easily be added without any need to alter or recompile the rest of the class package. Programming wise, they are only ever identified by their class (characteristic individual) and their behavioral class.

Attribute *sorts* are distinguished by their base and weight *sorts*. The base *sort* specifies the *sort* of individuals, the weight *sort* specifies the *sort* of attribute forms. Disjunctive *sorts* are distinguished by a list of (disjunctive) component *sorts*. Both attribute *sorts* and disjunctive *sorts* may be specified a name. The representational composition of *sorts* is subject to rules of reduction, in order to ensure a semi-canonical form (Stouffs and Krishanmurti 2002). For example, the operation of sum on *sorts* distributes over the attribute operation, such that a base *sort* is never a disjunctive *sort*:

$$(a + b) \wedge c = (a \wedge c) + (b \wedge c) \tag{6}$$

We are currently developing a prototype interface to build and edit definitions of *sorts*, compare and match *sorts* and to construct corresponding forms, in order to investigate the interaction with *sorts*, especially when these become large structures.

Acknowledgements

The first author is funded by a grant from the Netherlands Organization for Scientific Research (NWO), 016.007.007, support for which is gratefully acknowledged. The second author is funded by a grant from the National Science Foundation, CMS #0121549, support for which is gratefully acknowledged. Any opinions, findings, conclusions or recommendations presented in this paper are those of the authors and do not necessarily reflect the views of the Netherlands Organization for Scientific Research or the National Science Foundation. The authors would like to thank Michael Cumming for his work on the development of a prototype interface to build and manipulate *sorts*.

References

Agarwal, M and Cagan, J: 1998, A Blend of Different Tastes: The Language of Coffee Makers, *Environment and Planning B: Planning and Design* **25**(2): 205–226

Chase, SC: 1989, Shapes and shape grammars: from mathematical model to computer implementation, *Environment and Planning B: Planning and Design* **16**(2): 215–242.

Flemming, U: 1987, More than the sum of parts: the grammar of Queen Anne houses, *Environment and Planning B: Planning and Design* **14**(3): 323–350.

Gips, J: 1975, *Shape Grammars and Their Uses*, Birkhaüser, Basel.

Heisserman, J: 1991, *Generative Geometric Design and Boundary Solid Grammars,* Ph.D. Thesis, Department of Architecture, Carnegie Mellon University, Pittsburgh, Pa.

Heisserman, J: 1994, Generative Geometric Design, *IEEE Computer Graphics and Applications* **14**(2): 37–45

Knight, TW: 1989, Color grammars: designing with lines and colors, *Environment and Planning B: Planning and Design* **16**: 417–449.

Krishnamurti, R: 1982, *SGI: A Shape Grammar Interpreter*, Research report, Centre for Configurational Studies, The Open University, Milton Keynes.

Krishnamurti, R and Earl CF: 1992, Shape recognition in three dimensions, *Environment and Planning B: Planning and Design* **19**: 585–603.

Krishnamurti, R and Giraud, C: 1986, Towards a shape editor: the implementation of a shape generation system, *Environment and Planning B: Planning and Design* **13**: 391–404.

Krishnamurti, R and Stouffs, R: 1993, Spatial grammars: motivation, comparison and new results, in U Flemming and S Van Wyk (eds), *CAAD Futures '93*, North-Holland, Amsterdam, pp. 57–74.

Krishnamurti, R and Stouffs, R: 1997, Spatial change: continuity, reversibility and emergent shapes, *Environment and Planning B: Planning and Design* **24**: 359–384.

Krishnamurti, R and Stouffs, R: 2004, The boundary of a shape and its classification, *The Journal of Design Research* **4**(1).

Longenecker, SN and Fitzhorn, PA: 1991, A shape grammar for non-manifold modeling, *Research in Engineering Design* **2**: 159–170.

McCormack, JP and Cagan, J: 2002, Supporting designers' hierarchies through parametric shape recognition, *Environment and Planning B: Planning and Design* **29**: 913–931.

Piazzalunga, U and Fitzhorn, PA: 1998, Note on a three-dimensional shape grammar interpreter, *Environment and Planning B: Planning and Design* **25**: 11–33.

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

Stiny, G: 1982, Shapes are Individuals, *Environment and Planning B: Planning and Design* **9**: 359-367.

Stiny, G: 1990, What designers do that computers should, *in The Electronic Design Studio: Architectural Knowledge and Media in the Computer Era*, MIT Press, Cambridge, Mass., pp. 17–30.

Stiny, G: 1991, The algebras of design, *Research in Engineering Design* **2**: 171–181.

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

Stouffs, R and Krishnamurti, R: 2004, The boundary of a shape and its classification, *The Journal of Design Research* **4**(1).

Stouffs, R and Krishnamurti, R: 2002, Representational flexibility for design, *in* JS Gero (ed), *Artificial Intelligence in Design '02*, Kluwer Academic, Dordrecht, The Netherlands, pp. 105–128.

Stouffs, R, Krishnamurti, R and Eastman CM: 1996, A formal structure for nonequivalent solid representations, *in* S Finger, M Mäntylä and T Tomiyama (eds), Proceedings of *IFIP WG 5.2 Workshop on Knowledge Intensive CAD II*, International Federation for Information Processing, Working Group 5.2, pp. 269–289.

Tapia, MA: 1996, *From Shape to Style: Issues in Representation and Computation, Presentation and Selection,* Ph.D. Thesis, Department of Computer Science, University of Toronto, Toronto.

Tapia, MA: 1999, A visual implementation of a shape grammar system, *Environment and Planning B: Planning and Design* **26**: 59–73.