JS Gero (ed.), *Artificial Intelligence in Design'02,* 105-128

REPRESENTATIONAL FLEXIBILITY FOR DESIGN

RUDI STOUFFS
Delft University of Technology
The Netherlands

AND

RAMESH KRISHNAMURTI
Carnegie Mellon University
USA

Abstract. We present an abstraction of representational schema to model *sorts* that allows us to explore the mathematical properties of a constructive approach to *sorts*. We apply this approach to representational schema defined as compositions of primitive data types, and explore a comparison of representational structures with respect to scope and coverage. We consider a behavioral specification for *sorts* in order to empower these representational structures to support design activities effectively. We provide an example of the use of *sorts* to represent alternative views to a design problem. We conclude with a comparison with other approaches for flexibility of design representations.

1. Introduction

A variety of design problems requires a multiplicity of viewpoints each distinguished by particular interests and emphases. For instance, the architect is concerned with aesthetic and configurational aspects of a design, a structural engineer is engaged by the structural members and their relationships, and a performance engineer is engaged by the thermal, lighting, or acoustical performance of the eventual design. Each of these views – derived from an understanding of current problem solution techniques in these respective domains - require different representations of the same entity. The work described in this paper is based on the recognition that there will always be a need for different representations of the same entity, albeit a building or building part, a shape or other complex attribute.

This exigency ensues, formally, to define the relations between alternative representations, in order to support translation and identify where exact translation is possible, and to define coverage of different representations.

In order to support representational flexibility for design, a framework must be conceived that provides support for exploring alternative design representations, for comparing design representations with respect to scope and coverage, and for mapping design information between representations, even if their scopes are not identical. Typically, a representation is a complex structure of attributes and constructors, and a representation may be a construction of another (Stouffs, Krishnamurti and Eastman 1996). Comparing different representations, therefore, requires a comparison of the respective attributes, their mutual relationships, and the overall construction. On the other hand, the expressive power of a representational framework is defined by its vocabulary of primitive attributes (or data types) and constructors. A proper definition of this representational framework and its vocabulary can give designers the freedom and flexibility to develop or adopt representations that serve their intentions and needs, while at the same time these representations can be formally compared with respect to scope and coverage in order to support information exchange. Such a comparison will not only yield a possible mapping, but also uncover potential data loss when moving data from less-restrictive to more-restrictive representations.

Requicha (1980) defines a representational schema as a relationship between, on one hand, representations as concrete descriptions and, on the other hand, models as the abstract entities described. Models are considered mathematical abstractions of real-world or designed entities; representations are symbolic descriptions that can be conceived and manipulated using a computer. Each particular representation describes a single model but generally adheres to a global descriptive convention as expressed by the representational schema. We seek to develop a formalism that is based on a general description and, at the same time, applies to the particular representational schema that we are ultimately interested in. Hereto, we consider an abstraction of representational schema to model *sorts* that allows us to explore the mathematical properties of a constructive approach to *sorts*. We then apply this constructive approach to representational schema defined as compositions of primitive data types, and explore a comparison of representational structures with respect to scope and coverage. We consider a behavioral specification for *sorts* in order to empower these representational structures to support design activities effectively. We provide an example of the use of *sorts* to represent alternative views to a design problem. We conclude with a comparison with other approaches for flexibility of design representations.

2. A Conceptual Definition of *Sorts*

A *sort* constitutes the basic entity for our formalism. Conceptually, a *sort* may define a set of similar data elements, e.g., a class of objects or the set of tuples solving a system of equations. For example, points and lines each are a *sort* and so are triangles and squares. *Sorts* are not limited to geometrical objects, colors are a *sort* and other attributes can also define *sorts*. When described by a system of equations, the solutions to this system specify the data elements with respect to some chosen universe, e.g., lines may be expressed as infinite sets of points in a Euclidean space, colors as entities in an RGB (red-green-blue) or HSI (hue-saturation-intensity) space. Within such a system of equations, we can distinguish two types of parameters and equations. Characteristic parameters serve to define a generalized data element within its universe, e.g., a line in Euclidean space, or a color in RGB space. Instance parameters, on the other hand, identify each data element within the *sort*, e.g., a particular line in a *sort* of lines. For example, in the description of a (cartesian) coordinate *sort*, a characteristic parameter x specifies the coordinate as an entity on a cartesian axis, and an instance parameter x_c serves to represent each value x may take within this *sort*, $x = x_c$. Similarly, characteristic equations characterize a generalized data element, while instance equations bound the set of valid data elements. In the description of the coordinate *sort*, $x = x_c$ is the characteristic equation; an optional instance equation may take the form $x_c \geq 0$, limiting the coordinates to positive values.

If for a *sort* a, $\mathbf{A_c}$ denotes the system of characteristic equations, $\mathbf{A_i}$ the system of instance equations, $A^c = \{ a_1^c, \ldots, a_n^c \}$ the set of characteristic parameters, and $A^i = \{ a_1^i, \ldots, a_n^i \}$ the set of instance parameters, then, the representation of a can be written symbolically as follows,

$$\left\| \begin{matrix} \mathbf{A_c} & A^c \\ \mathbf{A_i} & A^i \end{matrix} \right\| \tag{1}$$

For example, a *sort* of positive coordinates may be specified as

$$\left\| \begin{matrix} x = x_c & \{x\} \\ x_c \geq 0 & \{x_c\} \end{matrix} \right\| \tag{2}$$

Similarly, a *sort* of infinite lines may be specified as

$$\left\| \begin{matrix} (x_2 - x_1)(y - y_1) = (y_2 - y_1)(x - x_1) & \{x, y\} \\ x_1 \neq x_2 \vee y_1 \neq y_2 & \{x_1, y_1, x_2, y_2\} \end{matrix} \right\| \tag{3}$$

Additional instance equations may limit which lines belong to the *sort*.

Sorts combine into new *sorts*. A *composite sort* can be defined as the result of an operation on two or more operand *sorts*; sometimes, an expression in terms of a single system of equations, as described above, may exist. Given two *sorts* a and b, the *sort* of all data elements that belong to a or b is the result of the operation of sum, $a + b$. All data elements that belong to both a and b define the result of the operation of product (or intersection), $a \cdot b$. The result is zero, $a \cdot b = 0$, if no data elements belong to both a and b. The result of the operation of difference, $a - b$, is the sort of all data elements that belong to a but not to b. Data elements from a and b can also be combined into 2-tuples under the operation of cartesian product: $a \times b$ contains all 2-tuples of which the first member belongs to a and the second to b.

For each operation, one or more rules can be declared that govern when the expression can be reduced to a single system of equations. For instance, the product $a \cdot b$ is non-zero if the following requirements hold (we refer to (Stouffs and Krishnamurti 1998) for an elaborate account, as well as for a proper definition of the terms *domains*, *identifiable*, and *equivalent* within their respective contexts):

- the characteristic parameters of a and b have identical *domains*, $A^c = B^c$
- the instance parameters of a and b are *identifiable*, $A^i \equiv B^i$
- the characteristic equations of a and b are *equivalent*, $\mathbf{A_c} \equiv \mathbf{B_c}$

The instance equations of $a \cdot b$ are a composition of the instance equations of a and b under the logical connective $\wedge$ (and). In the case that the instance equations of a and b exclude each other, then, $a \cdot b = 0$. The following rules apply:

$$\begin{vmatrix} \mathbf{A_c} & A^c \\ \mathbf{A_i} & A^i \end{vmatrix} \cdot \begin{vmatrix} \mathbf{B_c} & B^c \\ \mathbf{B_i} & B^i \end{vmatrix} \rightarrow \begin{vmatrix} \mathbf{A_c} \equiv \mathbf{B_c} & A^c = B^c \\ \mathbf{A_i} \wedge \mathbf{B_i} & A^i \equiv B^i \end{vmatrix};$$

$$\begin{vmatrix} \mathbf{A_c} & A^c \\ \mathbf{A_i} & A^i \end{vmatrix} \cdot \begin{vmatrix} \mathbf{B_c} & B^c \\ \mathbf{B_i} & B^i \end{vmatrix} \rightarrow 0, \text{otherwise} \qquad (4)$$

The difference of two *sorts* is dependent on the existence of another *sort* that is a common part of both *sorts*. Otherwise, if $a \cdot b = 0$, then, $a - b = a$. If a and b have identical domains and equivalent characteristic equations, then, the instance equations of $a - b$ are a composition of the instance equations of a and the negation ($\neg$) of the instance equations of b, under the logical connective $\wedge$. The following rules result:

$$\begin{Vmatrix} \mathbf{A}_c & A^c \\ \mathbf{A}_i & A^i \end{Vmatrix} - \begin{Vmatrix} \mathbf{B}_c & B^c \\ \mathbf{B}_i & B^i \end{Vmatrix} \rightarrow \begin{Vmatrix} \mathbf{A}_c \equiv \mathbf{B}_c & A^c = B^c \\ \mathbf{A}_i \wedge \neg \mathbf{B}_i & A^i \equiv B^i \end{Vmatrix};$$

$$\begin{Vmatrix} \mathbf{A}_c & A^c \\ \mathbf{A}_i & A^i \end{Vmatrix} - \begin{Vmatrix} \mathbf{B}_c & B^c \\ \mathbf{B}_i & B^i \end{Vmatrix} \rightarrow \begin{Vmatrix} \mathbf{A}_c & A^c \\ \mathbf{A}_i & A^i \end{Vmatrix}, \text{ otherwise} \tag{5}$$

Other rules can be specified for the operations of sum, and cartesian product (Stouffs and Krishnamurti 1998). However, these do not always reduce to a single system of equations; given two *sorts* a and b that do not have identical domains, their sum $a + b$ cannot be reduced. Instead, by using distributive rules, any expression of *sorts* over the operations of sum and cartesian product (as well as product and difference) can be reduced to a semi-canonical representational form using sum and cartesian product over at most two levels. The top level specifies a composition over sum of one or more *sorts*, each of which is a composition over cartesian product, at the second level. In this tree structure, the leaf nodes correspond to *sorts* that can be expressed through a single system of equations. Naturally, either or both levels may be absent depending on the particular *sort*. As an example, consider the *sorts* of points, colors and labels: the *sort* of all points with either a color or a label assigned is identical to the *sort* of all colored and all labeled points; *points* × (*colors* + *labels*) = *points* × *colors* + *points* × *labels*. The following distributive rules apply:

$$\begin{aligned} a \times (b + c) &\rightarrow a \times b + a \times c \\ (a + b) \times c &\rightarrow a \times c + b \times c \end{aligned} \tag{6}$$

Furthermore, a subsumption relationship, $\leq$, may be defined over *sorts*:

$$a \leq b \Leftrightarrow a \cdot b = a \tag{7}$$

From the semi-canonical form described above, we can derive some rules that govern when a *sort* may be a part of another *sort*. Consider the following classification on the universe of sorts: let D_0 be the set of *sorts* that can be expressed by a single system of equations (0 composition levels), let D_1 be the set of *sorts* that can be expressed using only the operation of cartesian product (1 composition level), and let D_2 be the set of *sorts* that include the operation of sum (1 or 2 composition levels). Then, for $a \in D_i$ to be a part of $b \in D_j$, j must either be equal to i or equal to 2 (Stouffs and Krishnamurti 1998). If i and j are 0, rules (4) specify when a is a part of b, or $a \cdot b = a$: provided the characteristic equations are identical, a *sort* subsumes another *sort* if its instance equations form a subsystem of the other *sort*'s instance equations.

The subsumption relationship facilitates the comparison of *sorts* in terms of scope and coverage. For example, the sum of two *sorts* has both operands as a part. Similarly, the product of two *sorts* forms a part of both *sorts*, while the operations of product and difference define a classification of a *sort* with respect to another *sort* into disjoint parts.

3. A Constructive Approach to Representational Structures

For all practical purposes, we consider elementary data types as building blocks for the construction of *sorts* as representational structures. The representational structure of a *primitive sort*, corresponding to an elementary data type, may be the tuple of values that correspond to the instance parameters of the mathematical expression of this same *sort*. The systems of characteristic and instance equations are encoded in the definition of the primitive *sort* in order to allow for an interpretation of the representational structure. Since instance equations constrain the set of valid data elements, the definition of a primitive *sort* may include a number of arguments that offer access to (some of) these constraints when instantiating a primitive *sort*. At instantiation, a primitive *sort* is also assigned a name, in order to semantically distinguish *sorts* that are, otherwise, syntactically identical.

3.1 FORMAL COMPOSITIONAL OPERATORS

Primitive *sorts* combine to *composite sorts* under formal compositional operations over *sorts*. These formal characteristics should derive from the conceptual operations considered above, defining a subsumption relationship that facilitates the comparison of sorts, and offering reduction rules and distribution rules that give access to a semi-canonical form. At the same time, these operations must reflect on semantic relationships that make sense in a representational definition of design data. For example, the operation of *sum* allows for disjunctively co-ordinate compositions of multiple *sorts*; a resulting data collection combines the different types of data elements from its operand *sorts* without imposing any hierarchical relationships. The resulting representational structure distinguishes all operand structures such that each data element belongs explicitly to one of the operand *sorts*. For example, a *sort* of points and lines distinguishes each data element as either a point or a line. Even if both operand *sorts* are syntactically identical, though semantically distinguished, all data elements are still recognized as belonging to one *sort* or the other. For example, an expression of a rule has both a *lhs* (left-hand-side) and *rhs* (right-hand-side)

of the same composite data type; any data element must belong either to the *lhs* or *rhs*. Conceptually, a rule element might instead be considered as a 2-tuple, consisting of a *lhs* and *rhs* element, in accordance to the operation of cartesian product. However, in a general expression of a rule, either of the *lhs* and *rhs* components may be omitted, as allowed by the operation of sum.

The *attribute* operator, instead, specifies a subordinate composition of *sorts*. The resulting representational structure is a combination of both operand structures under an object-attribute relationship. For example, a *sort* of labeled points is specified as a *sort* of points, with one or more labels assigned to each point in the data collection. Similar to the operation of cartesian product, the attribute operation is non-commutative. Other interpretations of the operation of cartesian product can also be considered. Conceptually, a composition of the operation of cartesian product to the n^{th} degree enables the definition of a *sort* of n-tuples of data elements from the respective operand *sorts*. Correspondingly, a *vector* operator can be defined that allows for conjunctively co-ordinate compositions of multiple *sorts*. More interestingly, a *grid* operator can allow for co-ordinate compositions of a single *sort*, with a specification of the grid size according to one, two, or more dimensions. Within the resulting representational structure, operand data elements are distinguished by the grid location these are assigned to. As an example, a grid operation may be useful in tiling design in order to distinguish the individual tiles in a larger composition.

Considering the attribute operation, denoted '^', instead of the operation of cartesian product, all associative and distributive rules that apply to *sorts* conceptually (Stouffs and Krishnamurti 1998), still apply to *sorts* as representational structures, e.g.:

$$\begin{aligned} a \wedge (b \wedge c) &\rightarrow a \wedge b \wedge c \\ (a \wedge b) \wedge c &\rightarrow a \wedge b \wedge c \\ a \wedge (b + c) &\rightarrow a \wedge b + a \wedge c \\ (a + b) \wedge c &\rightarrow a \wedge c + b \wedge c \end{aligned} \tag{8}$$

However, the reduction to a semi-canonical form is complicated by the ability to semantically identify a *sort*, that is, to assign a name to a *sort*. Let ':' denote the operation of semantic identification. Then, consider the following associative rule over the attribute operation:

$$a \wedge (d : b \wedge c) \rightarrow a \wedge b \wedge c \tag{9}$$

Both sides to the rule are syntactically equivalent, i.e., in both cases a data collection contains data elements from a, each of which has an attribute collection consisting of data elements from b, again, each of which has an attribute collection consisting of data elements from c. However,

semantically, the attribute collections of data elements of *a* are defined and identified as belonging to *d* or $b \wedge c$, respectively. When attempting to interpret and deal with a *sort*, such a distinction may be important, as it offers a simple way of limiting the depth at which a *sortal* specification must be minimally considered. Therefore, though rule (9) may be used when comparing *sorts* with respect to scope and coverage, this reduction is not automatically applied in the definition of a *sort*. Similarly, the following distributive rule only serves the comparison of *sorts*:

$$a \wedge (d : b + c) \rightarrow a \wedge b + a \wedge c \tag{10}$$

The situation is even more complex if, under an attribute operation, the first argument is a named composite *sort*, as in the case of $(d : a \wedge b) \wedge c$. Under the attribute operator, data collections are assigned as attribute to individual data elements. In the *sort* $d : a \wedge b$, these are the individual data elements of *b* within collections that are themselves assigned as attribute to elements of *a*. A representational structure that links the data collections of *c* to the individual data elements of *b* will obliterate the boundaries of the semantic identification as defined for *d*, resulting in the *sort* $a \wedge b \wedge c$. In order to maintain a reference to this semantic identification, a new, named *sort* $c'd : a \wedge b \wedge c$ may be constructed to replace $(d : a \wedge b) \wedge c$ as the result of this definition. Note that the apostrophe should not be mistaken as an operator, but instead serves as a binding element in the construction of a name out of two component names. Consider the *sorts* of *labeledpoints* : *points* ^ *labels* and *colors*. Then, the construction *labeledpoints* ^ *colors* is redefined as *colors'labeledpoints* : *points* ^ *labels* ^ *colors*, effectively assigning colors to labels that are assigned to points, while maintaining a reference to the original construction of the *sort* through its name. The following rules can be specified to apply automatically in the definition of a *sort*:

$$\begin{aligned} &(d : a \wedge b) \wedge c \rightarrow c'd : a \wedge b \wedge c \\ &(d : a + b) \wedge c \rightarrow c'd : a \wedge b + a \wedge c \end{aligned} \tag{11}$$

Thus, although a semi-canonical form can be arrived at to support the comparison of *sorts* with respect to scope and coverage, representational structures corresponding to *sorts* may contain many more levels of operations over attribute and sum that reflect on the historical definition of this *sort* as a hierarchical composition of previously defined *sorts*.

Operations of product and difference can also be defined on *sorts* as representational structures. However, their semantic significance to representational structures is doubtful. Consider the following distributive

rules involving either the operation of product or difference with the operation of sum or cartesian product (Stouffs and Krishnamurti 1998):

$$(a + b) \cdot c \rightarrow a \cdot c + b \cdot c$$
$$a \cdot (b + c) \rightarrow a \cdot b + a \cdot c$$
$$(a + b) - c \rightarrow (a - c) + (b - c)$$
$$a - (b + c) \rightarrow (a - b) \cdot (a - c) \quad (12)$$

$$(a \wedge b) \cdot (c \wedge d) \rightarrow (a \cdot c) \wedge (b \cdot d) \text{ if } a \cdot c \neq 0 \text{ and } b \cdot d \neq 0;$$
$$(a \wedge b) \cdot c \rightarrow 0 \text{ and}$$
$$a \cdot (b \wedge c) \rightarrow 0, \text{ otherwise} \quad (13)$$

$$(a \wedge b) - (c \wedge d) \rightarrow (a - c) \wedge b + a \wedge (b - d) \text{ if } a - c \neq 0 \text{ and } b - d \neq 0;$$
$$(a \wedge b) - c \rightarrow a \wedge b \text{ and}$$
$$c - (a \wedge b) \rightarrow c, \text{ otherwise} \quad (14)$$

Upon applying these distributive rules to any *sortal* composition involving any of these four operators, the only operations of product or difference that will result will be between primitive *sorts* that differ syntactically only in their arguments shaping the constraints that govern the resulting *sort*. According to rules (4) and (5), as specified for *sorts* conceptually, the resulting *sort* can be interpreted in terms of its overall constraints as a composition of the operand constraints (or instance equations) and defined as such. However, in this case, all references to the original construction of the *sort* will be lost, while the same effect can be achieved simply by altering the respective *sorts'* constraints. If instead, the original construction is maintained, a *sort* results that no longer offers a one-to-one relationship between its definition as an operational expression on primitive *sorts* and its representational structure. After all, the subtraction of a *sort* constrains the data elements that can be represented but the subtraction operation itself cannot be reflected in the representational structure. While we are interested in the definition of a language of expression of representational structures, ultimately, we are concerned with the representational structures themselves and the data that can be, and is, represented in these.

3.2 COMPARING REPRESENTATIONAL STRUCTURES

Sorts can be compared and matched as, roughly, equivalent, similar and convertible. Firstly, two *sorts* are *equivalent* if both are semantically derived from the same *sort*, where a semantic derivation consists of zero, one, or more renaming operations, $b : a$. Under equivalency, one *sort* may be

semantically derived from the other, or both may be semantically derived from a third *sort*. Thus, the equivalency relationship is reflexive, commutative, and transitive. Obviously, two *sorts* are *identical* only if these are also semantically identical. Secondly, two *sorts* are denoted *similar* if these are similarly constructed from the same primitive components, that is, the respective primitive components are syntactically identical and their respective constructions can be reduced to the same semi-canonical form using the associative and distributive rules specified above. Two primitive *sorts* are said to be syntactically identical if these have been identically defined, except for their name. A further distinction may be made between *strongly* similar *sorts*, which are constructed over equivalent *sorts*, and *weakly* similar *sorts*, which also contain (derivations from) primitive *sorts* that are only syntactically identical. For example, $a \wedge b$ and $a \wedge c$ are strongly similar if and only if b and c are equivalent. Also, $a + (d : b + c)$ and $(e : a + b) + c$ are strongly similar. The similarity relationship is also reflexive, commutative, and transitive.

Equivalent as well as similar *sorts* guarantee a correct exchange of data without data loss, except semantically. Furthermore, if one *sort* matches a part of another *sort*, under a composition over sum, data exchange without data loss applies from the first *sort* to the second. In the opposite direction, the occurrence of data loss is fully dependent on the actual data that is being exchanged. If two *sorts* are not similar, these may still be *convertible*, possibly without data loss. This is the case if two primitive *sorts* differ only in their arguments or constraints, similar to the instance equations of the mathematical approach. Data loss is then dependent on the relationships between these constraints and, possibly, on the actual data. This is also the case if, given a *sort*, another *sort* is constructed from this *sort* by reversing one or more attribute relationships, for instance, $a \wedge b$ and $b \wedge a$ are said to be convertible. (Whether data-loss occurs in such conversions is dependent on the behavioral categories of the constituent *sorts*; see section 4.2 "Composite Behaviors.") Naturally, *sorts* that are similarly constructed over convertible *sorts* are also convertible.

Given a partial match under the attribute relationship, *sorts* are still partially convertible. Again, data loss will be dependent on the actual match and on the direction of the data exchange. For example, two *sorts* a and $a \wedge b$ are partially convertible. Data loss occurs when exchanging data from $a \wedge b$ to a: all attribute information represented in b will be discarded. When exchanging data from a to $a \wedge b$, a default attribute must be assigned to each data element from a in order to ensure the validity of the resulting information. Finally, two *sorts* are incongruous if no match, even partial, applies.

While two *sorts* may be uniquely classified as equivalent, convertible, or otherwise, various alternative matches may still exist. Consider the matching of two compositions of *sorts* under the operation of sum. Each pairing of component *sorts*, one from each composition, may be a potential match. If one composition contains two equivalent *sorts*, neither of which has an identical counterpart in the other composition, though at least one equivalent match does apply, then, multiple combinations of *sorts* from either composition will be evenly matched. A decision from the user will be necessary to resolve this situation. This decision may be stored for later retrieval, such that subsequent occurrences of the same or similar match no longer require the user's interaction. In another case, a single best match may exist and be automatically selected for a component *sort*, even if the intention of the user may be otherwise. As an example, a primitive *sort* may have an equivalent as well as a convertible counterpart in the other composition, with the user's preference for the convertible match. In this case, a lesser match may be assigned a higher priority by the user such that this is considered first in the matching process.

In general, a quantification of the matching between *sorts* may be considered, with integral values defined by the identical, equivalent, strongly similar, weakly similar, convertible, partially convertible, or incongruent character of the match, increasingly in this order, and decimal values reflecting on the details of the match. For example, a decimal value for a strongly similar match may be derived from, amongst others, the number of equivalent versus identical component *sorts*. Then, the user may choose to override selective matches by assigning higher (or lower) matching values.

The comparison and matching of *sorts* for data exchange allows one to monitor data-integrity during the design process, at all times, for a large variety of data. Specifically, the coverage of *sorts* can be compared; data can always be moved from more-restrictive to less-restrictive *sorts* without data loss; and data loss can be measured when moving data in the opposite direction. Active control over which conversions should and should not be allowed or considered may be presented to the user in the form of a level tuner: three user-defined levels specify matching value intervals of predefined handling behavior Table 1.

4. Behavioral Specifications for *Sorts*

Most CAD applications adopt an object-oriented approach at the conceptual level, providing users with line segments, surfaces, or solids as objects with

attributes that maintain their properties at all times, unless explicitly altered by the user. While conceptually attractive and very understandable to the user, this approach is inimical to creative design. Creative design activities rely on a restructuring of information uncaptured in the current information structure, as when looking at a design provides new insights that lead to a new interpretation of the design elements. It can be proven that continuity of computational change requires an anticipation of the structures that are to be changed (Krishnamurti and Stouffs 1997). Creativity, on the other hand, is devoid of anticipation.

TABLE 1. Level tuner

level	**handling**
$< l_1$	*sorts* are considered equal, conversion is performed without notification
$< l_2$	user is notified of conversion
$< l_3$	user's approval is requested for conversion
$\geq l_3$	conversion is not allowed, unless upon user's specific initiation

Consider for example the composition of squares in figure 1. Specified as two square objects, each square can almost effortlessly be resized and moved. Instead, a manipulation of the individual line segments would require each square to be re-represented as a collection of four line segments. Visually, the composition in Figure 1 contains not two but three squares. Irrespective of whether the composition has been defined as a collection of two square objects or as a collection of twice four line segments, neither representation allows the third square easily to be distinguished and manipulated, unless it is additionally defined in the composition, possibly by drawing the shape over. Instead, if line segments would constitute dynamic data entities that can split themselves into any number of smaller line segments, the resulting composition of line segments would not only represent each of the three squares, but also an infinite number of other collections of line segments. Furthermore, representationally, each of these configurations has the same significance, and the designer may select any one as an interpretation of the design.

When dispensing with the object-oriented approach at a representational level, operations that may otherwise seem trivial, such as adding or removing elements or figures, become resolutely non-trivial. Consider the

addition of two numbers, in the case these represent cardinal values, e.g., a number of columns that is increased, and ordinal values, e.g., for a given space, determining the minimum distance to a fire exit or the (maximum) amount of ventilation required given a variety of activities. Similarly, consider additive versus subtractive colors, depending on whether these refer to the mixing of surface paints or light colors, respectively. A specification of such operational behavior needs to be included in the definition of a data type or structure. Fortunately, this behavioral specification may be reasonably limited to a few basic operations. Specifically, we consider the common arithmetic operations of addition and subtraction, and of product or intersection. The most common CAD operations of creation and deletion, and selection and deselection, can all be expressed as a combination of addition and subtraction operations from one *sort* or design space to another. More complex operations of grouping and layering can be similarly defined over more complex *sorts* (Stouffs and Krishnamurti 1996).

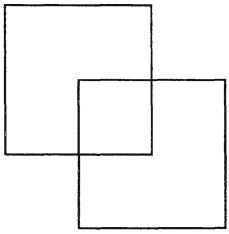

Figure 1. A simple, yet, ambiguous composition of (two or three) squares

A behavioral specification also is a prerequisite for an effective exchange of data between various representations and for a uniform handling of different and a priori unknown data structures. Consider the association of building performance data to design geometries. The behavior of these data as a result of alterations to the geometries can be expressed through a number of operations chosen to match the expected behavior. When an application receives the data together with its behavioral specification, the application can correctly interpret, manipulate, and represent this information without unexpected data loss.

For maximum flexibility, when composing *sorts* from other *sorts*, the operational behavior of the resulting *sort* should be automatically derived from the behavior of the component *sorts*. Hereto, we consider an algebraic framework for the specification of a *sorts'* operational behavior based on a partial order relationship (Stouffs 1994; Stiny 1991). This partial order relationship determines when an element can be considered a *part* of

another element. Fundamentally, any *part* of a data element defines in itself a valid data element of the same data type, and any combination of data elements under addition, subtraction, and product also constitutes a valid data element. Algebraically written, any collection of data elements of the same type is a member of an algebra that is ordered by a part relation and closed under the algebraic operations of addition, subtraction, and product.

4.1 TYPES OF BEHAVIORS

The simplest behavior that fits the requirements is a discrete behavior, corresponding to a mathematical set, where the part relation reduces to the subset relation, and the operations of addition, subtraction, and product correspond to set union, difference, and intersection, respectively. In other words, if a and b denote two data collections of a *sort* with discrete behavior, and A and B denote the corresponding sets of data elements ($a : A$ specifies A as a representation of a), then

$$\begin{aligned} a : A \wedge b : B \quad \Rightarrow \quad & a \leq b \Leftrightarrow A \subseteq B \\ & a + b : A \cup B \\ & a - b : A / B \\ & a \cdot b : A \cap B \end{aligned} \tag{15}$$

Under the discrete behavior, an explicit action is still required from the user in order to alter any data element. Only if two elements are identical do these combine into one. Stiny (1992) explores the application of the algebraic model to geometries with weights as attributes. Weights may be considered to denote thickness for points and lines, or tones for surfaces and volumes. 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. Thus, a collection of weights always combines into a single weight, which has as value the least upper bound of all the individual weight values, i.e., their maximum value. This behavior is termed *ordinal*; using numbers to represent weights, the part relation on weights corresponds to the less-than-or-equal relation on numbers;

$$\begin{aligned} a : \{x\} \wedge b : \{y\} \quad \Rightarrow \quad & a \leq b \Leftrightarrow x \leq y \\ & a + b : \{\max(x, y)\} \\ & a - b : \{\} \text{ if } x \leq y, \text{ else } \{x\} \\ & a \cdot b : \{\min(x, y)\} \end{aligned} \tag{16}$$

Line segments may be considered as intervals on infinite line carriers. An *interval* behavior applies to line segments as well as intervals of time or

other one-dimensional quantities: intervals on the same carrier that are adjacent or intersect combine into a single interval. An interval is a part of another interval if it is embedded in this interval. A specification of the interval behavior can be expressed in terms of the behavior of the interval boundaries. Let $B[a]$ denote the boundary of a collection a of intervals and, given two collections a and b, let I_a denote the collection of boundaries of a that lie within b, O_a denote the collection of boundaries of a that lie outside of b, M the collection of boundaries of both a and b where the respective intervals lie on the same side of the boundary, and N the collection of boundaries of both a and b where the respective intervals lie on opposite sides of the boundary (Stouffs 1994), Figure 2. Then,

$$\begin{aligned} a : \mathrm{B}[a] \wedge b : \mathrm{B}[b] \Rightarrow \quad & a \leq b \Leftrightarrow I_a = 0 \wedge O_b = 0 \wedge N = 0 \\ & a + b : \mathrm{B}[a + b] = O_a + O_b + M \\ & a - b : \mathrm{B}[a - b] = O_a + I_b + N \\ & a \cdot b : \mathrm{B}[a \cdot b] = I_a + I_b + M \end{aligned} \tag{17}$$

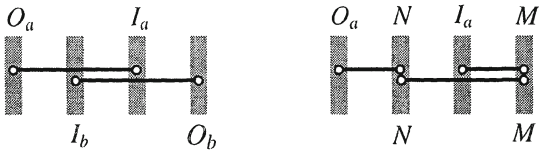

Figure 2. The specification of the boundary collections I_a, O_a, I_b, O_b, M and N, given two collections of intervals a (above) and b (below).

Note that the interval behavior also applies in the case of infinite intervals, provided an appropriate representation of both (infinite) ends of a carrier exists. Similar behaviors can be specified for plane segments and volumes (Stouffs 1994) as well as hypersegments of higher dimension; (17) still applies though the construction of I_a, O_a, I_b, O_b, M, and N is correspondingly more complex, Figure 3.

4.2 COMPOSITE BEHAVIORS

A composite *sort* inherits its behavior from its 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. Data collections from different component *sorts* never interact, the resulting data collection, corresponding the composite *sort*, is the group of collections from all component *sorts*. When an operation applies to two data collections

of the same composite *sort*, the operation instead applies to the respective component collections.

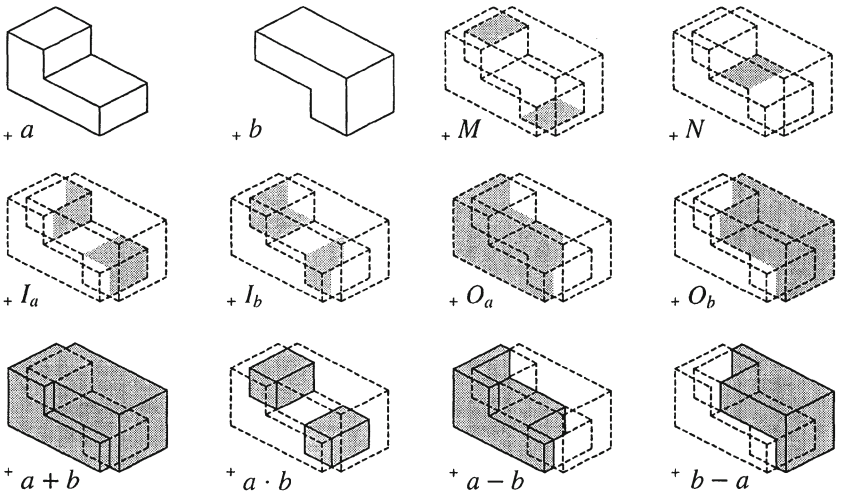

Figure 3. The boundary collections I_a, O_a, I_b, O_b, M and N for two volumes a and b, and the collections of volumes resulting from the operations $a + b$, $a \cdot b$, $a - b$ and $b - a$.

The 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 data collection consists of data elements of the first component *sort*, each element of which has, as attribute, another data collection 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 data collections of the same composite *sort* (under the attribute relationship), identical data elements merge and their attribute collections combine under the same operation. Any elements that have an empty attribute collection are removed. An object-oriented behavior can be achieved for any *sort* by combining this with a *sort* of (unique) identifiers under the attribute relationship. The resulting data form is akin to a database of individuals, where each individual has a unique key assigned.

Behaviors play an important role when assessing data-loss in information exchange between different *sorts*. Reorganizing component *sorts* under the attribute relationship into a different compositional hierarchy may alter the corresponding behavior and trigger data-loss. Consider a *sort* of weighted

points, i.e., a *sort* of points with attribute weights, and a *sort* of points of weights, i.e., a *sort* of weights with attribute points. A collection of weighted points defines a set of non-coincident points, each of which has a single weight assigned. These weights may be different for different points. The collection's behavior is discrete. Instead, a collection of points of weights is defined as a single weight with an attribute collection of points, and has an ordinal behavior. In both cases points are associated with weights. However, in the first case different points may be associated with different weights, whereas, in the second case all points are associated with the same weight. In a conversion from the first to the second *sort*, data-loss is inevitable.

5. Example

Consider design information in the form of design constraints and related information, e.g., for a steel-framed building project (Lottaz, Stouffs and Smith 2000), Figure 4. The information consists, minimally, of a set of authors, a set of constraints for each author, a common set of variables with each variable linked to the constraints defined over this variable, and a constraint solver (in the form of a URL) for each author (or constraint).

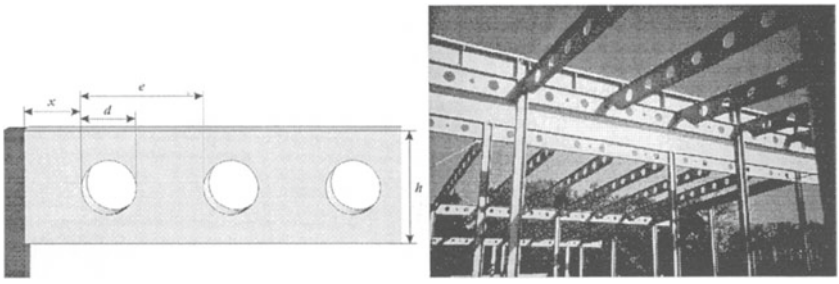

Figure 4. Design problem from a building project: the dimensioning of holes in steel beams

The author names, the constraint expressions and the variable names all define primitive *sorts* with characteristic individual 'Label'. The constraint solvers define a primitive *sort* with characteristic individual 'Url'. Three more primitive *sorts* with characteristic individual 'Property' serve to represent the various links between the constraint expressions and, respectively, the author names, variable names, and constraint solvers. Figure 5 presents a definition of these *sorts*, including a graphical depiction.

authornames : [Label]
constraintexpressions : [Label]
variablenames : [Label]
constraintsolvers : [Url]
(*isdefinedby, hasconstraints*) : [Property] (*constraintexpressions, authornames*)
(*contains, belongsto*) : [Property] (*constraintexpressions, variablenames*)
(*issolvedby, solves*) : [Property] (*constraintexpressions, constraintsolvers*)

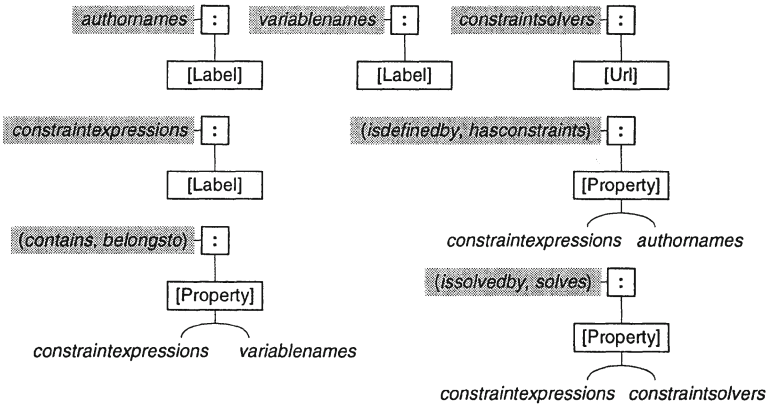

Figure 5. Definition and graphical depiction of primitive *sorts* for the constraints example

An organization of the design information by type, i.e., constraints, variables, authors, and solvers, with entities linked as appropriate, presents a straightforward and efficient way of storing this information into a relational database, Figure 6.

In order to support an actual design session, the author's design itself, i.e., his or her design constraints, should form the focus of the information organization. All other information entities can be made accessible from these, thereby clarifying each constraint's context and role in the design. A corresponding representational schema is presented in Figure 7. The author's constraints each specify the variables affected and provide access to the author's constraint solver. Each variable, in turn, specifies the constraints from other authors that are defined over this variable, and each of these constraints specifies its author. All information links are additionally provided. This representational schema supports the user in evaluating the effect of altering a constraint on the design and whether such a change may interfere with other constraints specified by the collaboration partners.

Figure 8 offers a VRML visualization of design data from the steel-framed building project represented in this schema.

database : *authors* + *constraints* + *variables* + *solvers*
authors : *authornames* ^ *hasconstraints*
constraints : *constraintexpressions* ^ *constraintproperties*
constraintproperties : *isdefinedby* + *contains* + *issolvedby* + *attributes*
variables : *variablenames* ^ *variableproperties*
variableproperties : *belongsto* + *attributes*
solvers : *constraintsolvers* + *solverproperties*
solverproperties : *solves* + *attributes*

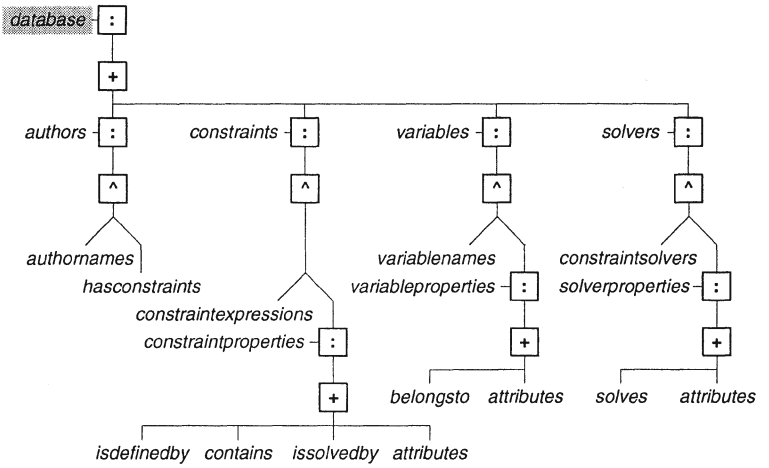

Figure 6. Database view of the design constraints example: definition and graphical depiction of the database sort. Note that a definition of the attributes sort is assumed, but not provided.

6. Discussion

In the definition and development of *sorts* as a concept and framework for representational flexibility, we let ourselves be guided as much by the algebraic strength of the adopted approach as by its practical benefits. This algebraic character offers a handle for the comparison and matching of representational structures and for the exchange of data accordingly. It also provides the ability to alter representational structures on the fly while maintaining their uniform approach of dealing with and manipulating data entities. The practical benefits must show in the description and building of representational structures as compositions of basic building blocks, that serve the design process in all its aspects. Ideally, a designer should be able to build or alter a representational structure to reflect on a particular need, in an almost intuitive way. Hereto, the behavioral specification serves as an attempt to bridge the desired simplicity of expressing representational structures and the necessary power of these structures to support design

activities effectively. In this paper, we have explored and argued the algebraic strength on both a conceptual and representational level and offered a simple example in the context of design constraints. In order to further illustrate the approach's practical benefits, other, more complex, examples of practical design cases are currently being investigated.

myconstraints : *constraintexpressions* ^ *myconstraintproperties*
myconstraintproperties : *myvariables* + *solvers* + *hasvariables* + *issolvedby* + *attributes*
myvariables : *variablenames* ^ *myvariableproperties*
myvariableproperties : *otherconstraints* + *belongsto* + *attributes*
otherconstraints : *constraintexpressions* ^ *otherconstraintproperties*
otherconstraintproperties : *authors* + *contains* + *issolvedby* + *isdefinedby* + *attributes*

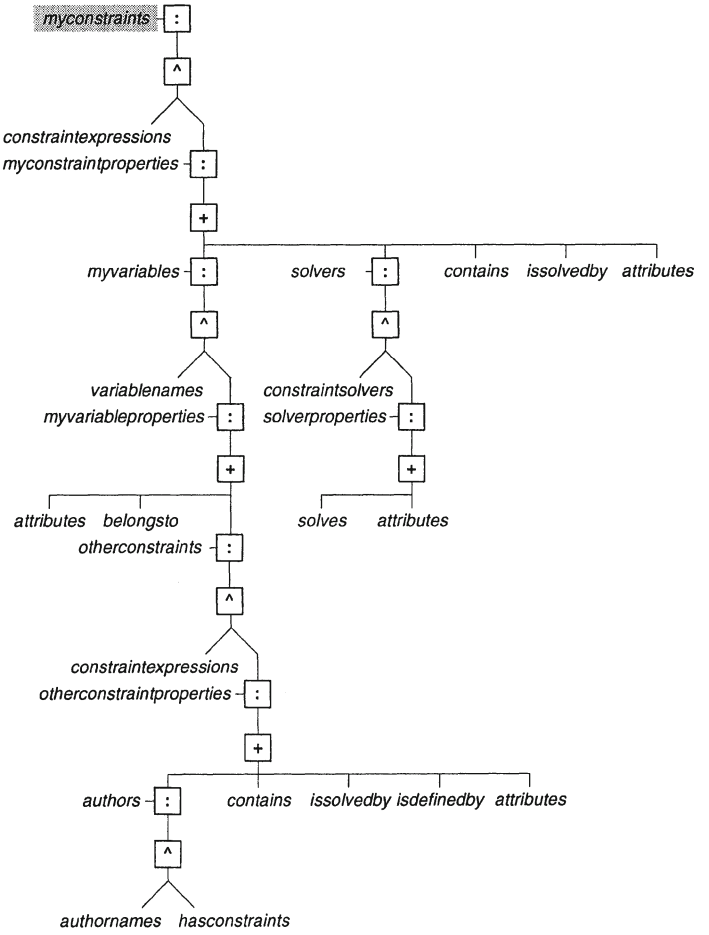

Figure 7. Design view of the design constraints example: definition and graphical depiction of the myconstraints sort

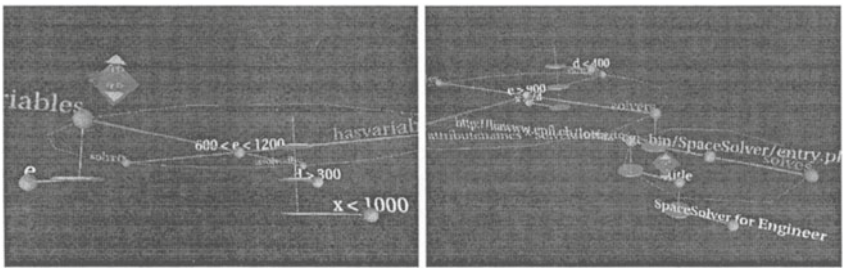

Figure 8. Snapshots of a VRML visualization for the steel-framed building project

While many other approaches exist and are being explored to support flexibility and extensibility of design models and representations, as well as the exchange of information between various representations or applications, few attempt to achieve this through a simple expressive language. Both van Leeuwen (1999) and Snyder (1998) adopt a schema approach. Feature-based modeling (van Leeuwen 1999) allows for the extensibility of conceptual schemas and supports these to be shared between different applications or designers. It also allows the designer to apply schemas to particular design situations by modifying or extending these with relationships and properties. Additionally, feature type recognition (van Leeuwen and de Vries 2000) enables the mapping of user-defined schemas to approved schema libraries, further supporting communication. In order to access and develop these schemas while modeling the design information, a 3D graphical tool is being developed that offers access to both the design model and the schema library (Coomans 2001). The SPROUT modeling language (Snyder and Flemming 1999; Snyder 1998) allows for the specification of schematic descriptions that can be used to generate computer programs that provably map data between different applications.

The LexiCon semantic model defines a formal vocabulary for the storage and exchange of information in the construction industry (Woestenenk 1998). It is defined in a semi-syntactic approach in which concepts are unambiguously defined by their constituent attributes. In this way, these attributes comprise the primitive concepts defining the semantic vocabulary of the model. If this descriptive approach is taken one step further, the attributes themselves can be described syntactically, leading to a purely syntactic description of concepts as compositions of primitive data types. Similar to *sorts*, a formal descriptive framework may then allow these syntactical descriptions to be compared independently of their conceptual meanings, allowing for synonym concepts, and for various degrees of similarity between alternative concepts.

XML defines such a formal framework. XML is a meta-language that serves to define markup languages for specific purposes. By specifying a grammatical structure of markup tags and their composition, a markup language is defined that can be shared with others. When an agreement can be reached on the tags, various markup languages can be developed based on these tags, and information exchanged between these, even if these languages differ in scope or composition. XML has the advantage that it is readable by both humans and the computer; markup languages based on XML can easily be adapted or extended to one's own specific purposes or needs. XML is particularly suited to structure otherwise unstructured information, such as textual data, and to organize information available over the Web. However, an XML based markup language does not provide any information on how to manipulate its data and, as such, is ill suited to represent detailed graphical or geometrical data.

From XML, our approach borrows a foundation consisting of an extensible vocabulary of data components that can be composed hierarchically into a representational language. In order to integrate the behavioral specification in the implementation, an object-oriented approach is applied, defining data elements as objects that encapsulate both the data structure and the operations defined on these structures. Also, the compositional operators are selected such that all representational structures offer the same operational functionality and derive this behavior from their component structures.

Then, a language specification is derived on two levels, similar to the approach used by the International Alliance for Interoperability (IAI) in the specification of its Industry Foundation Classes (IFCs) for a building object model (Bazjanac 1998). On a first syntactic level, the vocabulary of primitive object classes and their respective behaviors is defined. This behavior, in itself, does not provide any meaning to the object class. In fact, a same data structure may define two or more object classes if as many different behaviors can be said to apply, for different purposes. On a second level, a selection of object classes is defined and, individually, named in order to express a semantic concept. These named classes can, subsequently, be composed into a hierarchical structure in order to define an appropriate representational schema. In contrast to the IFC approach, this semantic concept can be specified by the user and the representational structure composed accordingly. Alternative representations can be defined by altering the compositional structure or the selection of component classes. As each representation defines the same common operations, these can be reasonably plugged into an applicative interface for manipulation.

Acknowledgements

This work is partly funded by the Netherlands Organization for Scientific Research (NWO), grant nr. 016.007.007. The second author is partially supported by the National Science Foundation, grant nr. CMS-0121549. The first author would like to thank Michael Cumming for his invaluable comments.

References

Bazjanac, V: 1998, Industry foundation classes: bringing software interoperability to the building industry, *The Construction Specifier* **6**: 47–54.

Coomans, MKD: 2001, DDDiver: 3D interactive visualization of entity relationships, *in* D. Ebert (ed.), *Data Visualization 2001: Proceedings of the Joint Eurographics and IEEE TCVG Symposium on Visualization*, Springer, Vienna, pp. 291-299.

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

Lottaz, C, Stouffs, R and Smith, I: 2000, Increasing understanding during collaboration through advanced representations, *Electronic Journal of Information Technology in Construction* **5**: 1-25. [itcon.org/2000/1/]

Requicha, A.A.G.: 1980, Representations for rigid solids: theory, methods and systems, *Computing Surveys* **12**: 437–464.

Snyder, J.D.: 1998, *Conceptual Modeling and Application Integration in CAD: The Essential Elements*, Ph.D. dissertation, School of Architecture, Carnegie Mellon University, Pittsburgh, PA.

Snyder, J and Flemming, U: 1999, Information sharing in building design, *in* G. Augenbroe and C. Eastman (eds), *Computers in Building*, Kluwer Academic, Boston, pp. 165–183.

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

Stouffs, R: 1994, *The Algebra of Shapes*, Ph.D. dissertation, Department of Architecture, Carnegie Mellon University, Pittsburgh, PA.

Stouffs, R and Krishnamurti, R: 1998, An algebraic approach to comparing representations, *in* J. Barallo (ed.), *Mathematics & Design 98*, The University of the Basque Country, San Sebastian, pp. 105–114.

Stouffs, R and Krishnamurti, R: 1996, The extensibility and applicability of geometric representations, *Architecture Proceedings of 3rd Design and Decision Support Systems in Architecture and Urban Planning Conference*, Eindhoven University of Technology, Eindhoven, pp. 436–452.

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.

van Leeuwen, JP: 1999, *Modeling Architectural Design Information by Features*, Ph.D. Dissertation, Eindhoven University of Technology, Eindhoven.

van Leeuwen, JP and de Vries, B: 2000, Modelling with features and the formalisation of early design knowledge, *in* R Gonçalves, A Steiger-Garçao and R Scherer (eds), *Product and Process Modeling in Building and Construction*, A.A. Balkema, Rotterdam, pp. 167–176.

Woestenenk, K: 1998, A common construction vocabulary, *in* R Amor (ed.), *Product and Process Modelling in the Building Industry*, Building Research Establishment, Watford, pp. 561–568.