JS Gero (ed), *Design Computing and Cognition'04,* 219-238
© 2004 Kluwer Academic Publishers, Dordrecht, Printed in the Netherlands

# DATA VIEWS, DATA RECOGNITION, DESIGN QUERIES AND DESIGN RULES

*Representational Flexibility for Design*

RUDI STOUFFS
*Delft University of Technology, The Netherlands*

and

RAMESH KRISHNAMURTI
*Carnegie Mellon University, USA*

**Abstract.** *Sorts* present a constructive approach to representational structures and provide a uniform approach to handling various design data. In this way, *sorts* offer support for multiple, alternative data views and for data exchange between these views. The representation of *sorts* extends on a maximal element representation for shapes that supports shape recognition and shape rules. In the same way, *sorts* offer support for data recognition, for querying design information and for expressing design rules. In this paper, we present an overview of the use of *sorts* to support these functionalities. Each of these relies on the ability to alter representational structures or *sorts*, and to manipulate the composition of data forms. In this regard, we briefly consider the user interaction aspect of utilizing *sorts*.

## 1. Introduction

Computational design relies on effective information models for design, for the creation of design artifacts and for the querying of the characteristics of such artifacts. Mäntylä stated in 1988 that these (geometric) representations must adequately answer “arbitrary geometric questions algorithmically.” Even without emphasis on the geometric aspects, this remains as important today. However, current computational design applications tend to focus on the representation of design artifacts, and on the tools and operations for their creation and manipulation. Techniques for querying receive less attention and are often constrained by the data representation system and methods. Nevertheless, querying a design is as much an intricate aspect of the design process as is creation and manipulation.

Design is also a multi-disciplinary process, involving participants, knowledge and information from various domains. As such, design problems require a multiplicity of viewpoints each distinguished by particular interests and emphases. For instance, an 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 building performance engineer is interested in 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—requires a different representation of the same (abstract) entity. Even within the same task and by the same person, various representations may serve different purposes defined within the problem context and the selected approach. Especially in architectural design, the exploratory nature of the design process invites a variety of approaches and representations.

Each view may rely on domain knowledge in order to provide a visualization that is particularly appropriate for the type of design object under investigation. In scientific visualizations, one can make use of the inherent dimensions of scientific data, connecting to three spatial and one temporal dimension, requiring only elementary linear algebra to lay out scientific data on a two-dimensional display (Groth and Robertson 1998). In architecture, designers commonly rely on a geometric visualization of the architectural object and its components, in both two and three dimensions, providing feedback on both aesthetic and configurational aspects of the design. A structural engineer, on the other hand, is less concerned with the geometry of the design components. Instead, a diagrammatic visualization of the design object presenting the structural characteristics of its components and their relationships is more appropriately used. Similarly, data visualizations in Geographic Information Systems (GIS) generally make use of map projections to visualize a variety of geographically related data.

Not all kinds of data structures can rely on specific domain knowledge in their visualization. For example, when exploring general information structures or databases, data may be collected from a large variety of domains and may not fit a single domain-specific visualization. In design, it can be said that there are as many design methods as there are designers. Different design methods may consider different data from different design domains and, therefore, require different visualizations. Furthermore, not all kinds of views can be envisioned a priori and specific support provided for. In such cases, the challenge is to achieve an effective mapping from data to display (Groth and Robertson 1998).

Effective visualizations enable a visual inspection of design data and information. Design queries, on the other hand, support the analysis of existing design information in order to derive new information that is not explicitly available in the information structure. Both effective

visualizations, in support of alternative design views, and an expression of arbitrary design questions require flexible design information models and representations that can be modified and geared to the kinds of visualizations and queries. Supporting arbitrary design questions also requires access to information in a uniform and consistent manner, so that new queries can be easily constructed and posed based on intent, instead of on availability.

*Sorts* (Stouffs and Krishnamurti 2002; 2001a) offer a framework for representational flexibility that provides support for developing alternative representations of a same entity or design, for comparing representations with respect to scope and coverage, and for mapping data between representations, even if their scopes are not original. *Sorts* support the specification of the operational behavior of data in a uniform way, based on a partial order relationship (Stouffs 1994; Stiny 1991). *Sorts* extend on a maximal element representation for shapes (Stouffs 1994; Krishnamurti 1992) that supports shape recognition and shape rules. Data views, data recognition, design queries and design rules all relate to the concept of emergence, i.e., the recognition of information components and structures that are not explicitly present in the information and its representation, and on the restructuring of information. The concept of emergence, in turn, supports creativity and novelty (Krishnamurti and Stouffs 1997; Stiny 1993).

In a previous paper (Stouffs and Krishnamurti 2002) we explored the mathematical properties of a constructive approach to *sorts* through an abstraction of representational structures to model *sorts*. We applied this approach to representational structures defined as compositions of primitive data types, and explored a comparison of representational structures with respect to scope and coverage. We considered a behavioral specification for *sorts* in order to empower these representational structures to support design activities effectively, and provided an example of the use of *sorts* to represent alternative views to a design problem. In this paper, we consider the application of *sorts* in a broader context and present an overview of the use of *sorts* to support data recognition, design queries and design rules, next to multiple data views. Each of these functionalities relies on the ability to alter representational structures or *sorts*, and to manipulate the composition of data forms. In this regard, we briefly consider the user interaction aspect of utilizing *sorts*.

## 2. Alternative Data Views

Integrated data models are under development that span multiple disciplines and support different views. Such models allow for various representations in support of different disciplines or methodologies and enable information

exchange between representations and collaboration across disciplines. Examples are, among others, the ISO STEP standard for the definition of product models (ISO 1994) and the Industry Foundation Classes (IFCs) of the International Alliance for Interoperability (IAI), an object-oriented data model for product information sharing (Bazjanac 1998). These efforts characterize an a priori and top-down approach: an attempt is made at establishing an agreement on the concepts and relationships which offer a complete and uniform description of the project data, independent of any project specifics (Stouffs and Krishnamurti 2001a).

Alternative modeling techniques that consider a bottom-up, constructive approach are also under investigation. These provide a more extensive degree of flexibility that allows for the development of information models that are context, and thus project, specific. We consider a few examples related to architectural design. Concept modeling (van Leeuwen and Fridqvist 2003; van Leeuwen 1999) allows for the extensibility of conceptual schemas and for flexibility in modeling information structures that differ from the conceptual schemas these derive from. 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. Woodbury et al. (1999) adopt typed feature structures in order to represent partial information models and use unification-based algorithms to support an incremental modeling approach.

*Sorts* (Stouffs and Krishnamurti 2002) offer a constructive approach to defining representational structures that enables these to be compared with respect to scope and coverage and that presents a uniform approach to dealing with and manipulating data constructs. Briefly, a *sort* is defined as a complex structure of elementary data types and compositional operators, and is typically a composition of other *sorts*. Comparing different *sorts*, therefore, requires a comparison of the respective data types, their mutual relationships, and the overall construction.

### 2.1. EXAMPLE A: A HIERARCHICAL STRUCTURE OF KEYWORDS

Figure 1 presents a simple example of a *sort* that represents a hierarchical structure of architectural concepts or keywords. The representation is conceived as a tree structure in which each keyword can have zero, one or more subordinate keywords. The *sort concepts*, a *sort* of labels, represents the individual keywords:

$$\textit{concepts} : [\text{Label}] \tag{1}$$

The subordinate relationship between keywords is expressed by the attribute

operator on *sorts* (‘^’). The resulting *sort*, named *conceptstree*, is defined recursively:

$$conceptstree : concepts + concepts \wedge conceptstree \quad (2)$$

The attribute operator relates to each individual keyword (*concepts*) a non-empty data form of subordinate keywords (*conceptstree*). The disjunctive composition operator (‘+’) allows the combination of keywords with (*concepts* ^ *conceptstree*) and without (*concepts*) attribute keywords. Thus, individual keywords are assigned either to the *sort concepts*, or with an attribute data form to the *sort concepts* ^ *conceptstree*.

```
sort concepts : [Label];
sort conceptstree : concepts + concepts ^ conceptstree;

form $concepts = conceptstree:
{ (concepts ^ conceptstree):
   { "theater"
      { concepts:
         { "infrastructure" },
        (concepts ^ conceptstree):
         { "construction"
            { concepts:
               { "load bearing structure",
                 "material" },
              (concepts ^ conceptstree):
               { "enclosure"
                  { concepts:
                     { "roof",
                       "facades" } } } },
          "format"
            { concepts:
               { "photo",
                 "scale model",
                 "text" },
              (concepts ^ conceptstree):
               { "view"
                  { concepts:
                     { "elevation",
                       "axonometric view",
                       "diagram",
                       "section",
                       "perspective",
                       "plan",
                       "site plan" } } } },
          ... } } } };
```

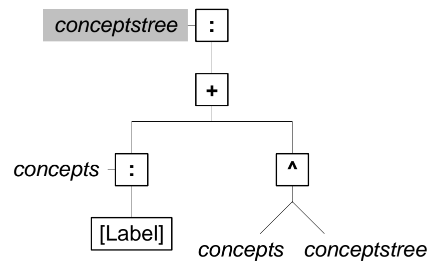



*Figure 1.* Textual and graphical definition of a recursive *sort* representing a hierarchical structure of architectural concepts, and the (partial) description of an exemplar data form (*Sorts* Description Language). In the definition of a *sort*, ‘+’ and ‘^’ denote the operations of disjunctive composition and attribute, respectively; ‘:’ denotes the naming of a *sort*; ‘[Label]’ is a primitive *sort* of labels.

### 2.2. EXAMPLE B: A NETWORK STRUCTURE OF KEYWORDS

An alternative view of a semantic structure (or an architectural typology) is in the form of a network or (semantic) map. A network structure distinguishes itself from a simple hierarchical structure in that a subordinate keyword may be shared by more than one keyword. Such a structure can be extended from the structure in Figure 1 by allowing references to be specified to keywords that are already defined elsewhere in the structure. Such references can be represented using a property relationship *sort* that is defined over the *sort concepts* and an equivalent *sort conceptrefs*:

*conceptrefs* : *concepts* (3)

The property relationship *sort* distinguishes two named *aspects*, *hasrefs* and *isrefs*, respectively corresponding to the relationship from *concepts* to *conceptrefs* and vice versa:

(*hasrefs*, *isrefs*) : [Property] (*concepts*, *conceptrefs*) (4)

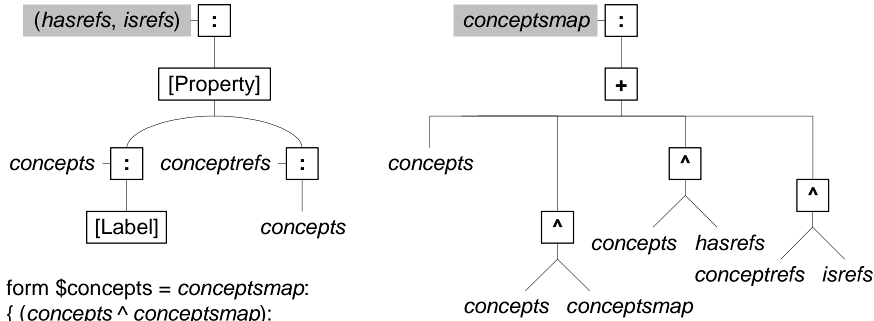
These two aspects can be considered as two different views of the same *sort*. Each aspect, however, is considered a distinct *sort* if used in the definition of other *sorts*. In order to maintain consistency, each aspect must be specified as an attribute to its respective *sort* of origin under the property relationship, e.g., *concepts* ^ *hasrefs* and *conceptrefs* ^ *isrefs*. The first attribute *sort*, *concepts* ^ *hasrefs*, allows for the specification of keywords with one or more references to (subordinate) keywords that are elsewhere defined. The second attribute *sort*, *conceptrefs* ^ *isrefs*, allows for the retrieval of all keywords this subordinate keyword is referenced from. Both attribute *sorts*, together with the *sorts concepts* and *concepts* ^ *conceptsmap*, recursively define the *sort conceptsmap* under the disjunctive composition operator, Figure 2:

*conceptsmap* : *concepts* + *concepts* ^ *conceptsmap* + *concepts* ^ *hasrefs* + *conceptrefs* ^ *isrefs* (5)

Thus, individual keywords are assigned to the *sort concepts*, with an attribute data form (that is recursively defined) to the *sort concepts* ^ *conceptsmap*, or with an attribute data form of references to the *sort concepts* ^ *hasrefs*. If a keyword has subordinate keywords of which some but not all are defined elsewhere (and thus referenced here), then, this keyword will be assigned to both the *sorts concepts* ^ *conceptsmap* and *concepts* ^ *hasrefs*.

Figure 2 also presents an exemplar data form considering an architectural typology for Ottoman mosques (Tunçer et al. 2002). Note that the data form does not specify any data to the *sort conceptrefs* ^ *isrefs*, these are automatically derived from the data to the *sort concepts* ^ *hasrefs*.

```
sort conceptrefs : (concepts : [Label]);
sort (hasrefs, isrefs) : [Property] (concepts, conceptrefs);
sort conceptsmap : concepts + concepts ^ conceptsmap + concepts ^ hasrefs +
        conceptrefs ^ isrefs;
```




```
form $concepts = conceptsmap:
{ (concepts ^ conceptsmap):
   { "physical"
      { (concepts ^ conceptsmap):
         { "mosque"
            { (concepts ^ conceptsmap):
               { "structural"
                  { (concepts ^ hasrefs):
                     { #om-concepts-26 "arcade"
                        { om-conceptrefs-5, om-conceptrefs-14, om-conceptrefs-19 } },
                    (concepts ^ conceptsmap):
                     { "arcade"
                        { concepts:
                           { "spandrel" },
                          (concepts ^ hasrefs):
                           { #om-concepts-11 "arch"
                              { om-conceptrefs-2, om-conceptrefs-6,
                                om-conceptrefs-16, om-conceptrefs-23 },
                             #om-concepts-13 "dome"
                              { om-conceptrefs-3, om-conceptrefs-7 } },
                          (concepts ^ conceptsmap):
                           { "arch"
                              { concepts:
                                 { "tympanum" } },
                             "column"
                              { concepts:
                                 { "column base",
                                   "column capital" } },
                             "dome"
                              { (concepts ^ hasrefs):
                                 { #om-concepts-5 "crescent"
                                    { om-conceptrefs-1, om-conceptrefs-4,
                                      om-conceptrefs-29 } } } } },
                  ... } } } } } } } };
```

*Figure 2.* Textual and graphical definition of a recursive *sort* representing a (semantic) map of architectural concepts, and the (partial) description of an exemplar data form (*Sorts* Description Language). In the definition of a *sort*, '+' and '^' denote the operations of disjunctive composition and attribute, respectively; ':' denotes the naming of a *sort*; '[Label]' and '[Property]' are primitive *sorts*, the latter defines a property relationship *sort* between two given *sorts*.

Both *sorts conceptstree* and *conceptsmap* present a possible representation of a semantic structure. The selection of any particular representation is dependent on the type of structure or semantic data, and also on the visualization or application of the semantic structure. Figure 3 illustrates three different visualizations of the data forms considered in Figures 1 and 2. For best results, data may need to be converted between different representations (in both directions). The effects of such conversion on the data can be deduced from a comparison of both *sorts*. The comparison of *conceptstree* and *conceptsmap* results in a partial match: the *sort concepts* is identical in both examples; when ignoring the attribute *sorts* involving the aspects *hasrefs* and *isrefs*, the *sort conceptstree* and (part of) the *sort conceptsmap* become similarly composed of identical and (recursively) similar *sorts*. Therefore, converting data from *conceptstree* to *conceptsmap* involves no data loss; obviously, a tree structure is a special instance of a network or map structure. However, converting data in the other direction may involve data loss; the data lost in this case is the identification of shared keywords.

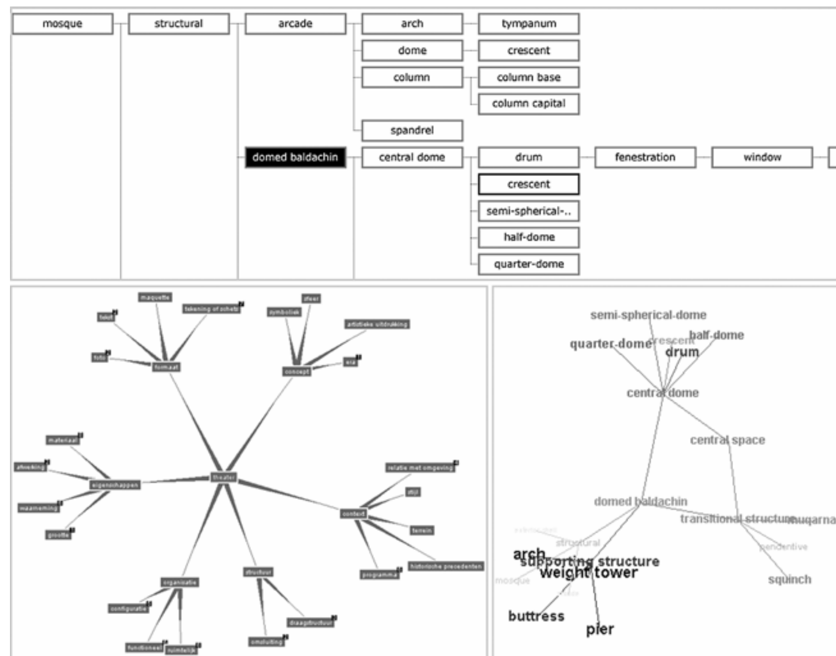



*Figure 3.* Three different visualizations of the data forms from Figures 1 and 2. The top-drawing shows a straightforward depth-first enumeration, albeit graphically enhanced; references to keywords already defined are marked with a different color border (e.g., “crescent”). The bottom left and right drawings show a 2D/3D graphical presentation of a hierarchical structure and a network structure, respectively. Image by Bige Tunçer.

### 2.3. EXAMPLE C: A COLLECTION OF KEYWORDS

Comparing *sorts* can become far more complex if one of the *sorts* is defined recursively, while the other is not. Consider another alternative representation of keywords without any (hierarchical or other) relationships. The *sort concepts* represents such a simple collection of keywords. Converting data from *concepts* to *conceptstree* (or *conceptsmap*) is fairly straightforward. Both *sorts conceptstree* and *conceptsmap* are defined in such a way that they allow for the representation of keywords without any subordinate relationships: the *sort concepts* is a part (or component) of the *sort conceptstree* under the disjunctive composition operator. The result is a partial match of identical *sorts*.

Converting data in the other direction is less obvious: the only keywords that can be converted are those that have no subordinate relationships. In the example of Figure 1 that leaves not a single keyword. However, this is not the only way that the *sort concepts* can be mapped onto a component of the *sort conceptstree*. First, the *sort concepts* also appears as a (parent) component under the attribute operator. Given such a partial match under the attribute relationship, the *sorts concepts* and *concepts ^ conceptstree* are said to be partially convertible. In the example of Figure 1, only the root keyword of the hierarchical structure, “theater,” will be retained upon conversion. Second, through the recursive definition of *conceptstree*, the *sort concepts* could also be mapped onto an attribute component of the *sort concepts ^ conceptstree*. However, such a mapping cannot be considered in the context of a comparison of the *sorts concepts* and *conceptstree* as mapping *concepts* onto the *conceptstree* component of *conceptstree* would create an infinitely recursive mapping.

The conversion of a hierarchical structure of keywords (as represented by a *sort* that is recursively defined) into a simple collection of keywords as represented by the *sort concepts*, such that all or most keywords are retained in the conversion, can only be achieved through the construction of an intermediate representation. Consider a *sort concepts2* that is a composition under the attribute operator of the *sort concepts* twice:

*concepts2* : *concepts ^ concepts* (6)

Comparing the *sorts concepts2* and *conceptstree* at best results in a partial match of similar *sorts*, allowing for the conversion of those keywords that are directly subordinate to the root keyword in the hierarchical structure of Figure 1 and do not have any subordinate keywords themselves. Comparing the *sorts concepts* and *concepts2* results in a partially convertible match where either component under the attribute operator can be mapped onto the *sort concepts*. In this way, the keywords that resulted from the conversion above can be further converted, in two steps, into a collection of keywords

(without hierarchical relationships). The intermediate representation can be extended in order to include more keywords in the conversion.

## 3. Shape and Data Recognition

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.

Computationally recognizing emergent shapes requires determining a geometric, commonly Euclidean, transformation under which a specified similar shape is a part of the original shape. For example, a square must be computationally recognized as a square irrespective of its scale, orientation or location. The same approach applies to other kinds of data. For example, search-and-replace functionalities in text editors generally consider case transformations of the constituent letters. Clearly, this matching problem depends on the representational structure adopted. The maximal element representation for shapes is a particularly appropriate representation as each element type specifies its own part or match relationship (Krishnamurti and Stouffs 1997; Krishnamurti and Earl 1992).

*Sorts* can be considered as an extension of the maximal element representation to other, non-geometric, data, without necessarily considering non-spatial information as attributes to shapes. The concept of *sorts* distinguishes various behaviors data can adhere to, all based on a part relationship (Stouffs and Krishnamurti 2002). Examples are a discrete behavior, corresponding to a mathematical set, for labels or points, an ordinal behavior for numeric weights, line thicknesses or shades of gray, an interval behavior for line segments, and similar behaviors for plane segments and volumes. Consider the *sort conceptstree* and the corresponding data form in Figure 1. Keywords are assigned to the *sort concepts*, a *sort* of labels, with discrete behavior. The recognition of keywords therefore requires the full keyword to be provided, though case transformations may apply such that the word “Infrastructure” can match the keyword “infrastructure”.

The behavior of a composite *sort* is derived from the behaviors of the component *sorts*, in a manner that depends on the compositional relationship (Stouffs and Krishnamurti 2002). That means that the keyword “Infrastructure” as instance of the *sort concepts* cannot be matched to the keyword “infrastructure” in the exemplar data form in Figure 1 without automatic conversion of data forms. The latter keyword, namely, is a

subordinate keyword to the keyword “theater” and together form an instance of the *sort conceptstree*.

However, transformations may apply not only to the data, corresponding to the individual data types, but also to the composite structure of the data. When looking for a yellow square, one does not necessarily need to be concerned with the fact that yellow is represented as an attribute to the square, as in a graphical visualization, or instead that the square is represented as an attribute to the color yellow, as in a sorting of the geometries by color. Thus, the matching problem may involve different data views and the conversion of data between these views, all of which can be supported using *sorts*. In this case, if an instance of the *sort concepts* ^ *concepts* (or *concepts2*) is constructed using “Theater” and “Infrastructure,” then the recognition of this instance in the exemplar data form will be successful because the *sorts concepts2* and *conceptstree* result in a partial match of similar sorts (see section 2.3). Similarly, the keyword “Theater” as instance of the *sort concepts* can be matched to the keyword “theater” as partial instance of the *sort conceptstree*. However, the problem still remains how the keyword “Infrastructure” as instance of the *sort concepts* could be recognized in the data form as mapping *concepts* onto the *conceptstree* component of *conceptstree* creates an infinitely recursive mapping.

The part relationship underlying the various behaviors enables the matching problem to be implemented for each primitive *sort* or data type. Since composite *sorts* inherit their behavior, and part relationship, from their component *sorts*, the technical difficulties of implementing the matching problem apply only once for each primitive *sort* or data type. As the part relationship can be applied to all kinds of data types, recognition algorithms can easily be extended to deal with arbitrary data forms, even if a proper definition of what constitutes a transformation is still necessary.

## 4. Design Queries

Querying design information, as distinguished from visual inspection, generally requires the analysis of existing information in order to derive new information that is not explicitly available in the information structure. A viable query language has to be based on a model for representing different kinds of information that adheres to a consistent logic providing access to information in a uniform and consistent manner.

Stouffs and Krishnamurti (1996) indicate how a query language for querying graphical design information can be built from basic operations and geometric relations that are defined as part of a maximal element representation for weighted geometries. These operations and relations are augmented with operations that are derived from techniques of counting and pattern matching for the purpose of composing more complex and versatile

geometric and non-geometric queries. For example, by augmenting networks of lines that are represented as volumes (or plane segments) with labels as attributes, and by combining these augmented geometries under the operation of sum, as defined for the representational model, colliding lines specifically result in geometries that have more than one label as attribute. These collisions can easily be counted, while the labels on each geometry identify the colliding lines, and the geometry itself specifies the location of the collision (Stouffs and Krishnamurti 1996).

In order to consider counting and other functional behavior as part of the representational approach, *sorts* consider data functions as a data kind, offering functional behavior integrated into data constructs. Data functions are assigned to apply to a selected property attribute of a specific *sort*, which itself may be a data function. Then, the result value of the data function is computed from the values of the property attribute of the data entities of this *sort*. This result value is automatically recomputed each time the data structure is traversed, e.g., when visualizing the structure. For this purpose, this target *sort* must be related to the data function's *sort* within the representational structure under a sequence of one or more attribute relationships, with restrictions. As a data kind, data functions specify both a functional description, a result value, and a *sort* and its property attribute.

Data functions can introduce specific behaviors and functionalities into representational structures, for the purpose of counting or other numerical operations. Consider, for example, a data structure corresponding to a composition of two *sorts* where one *sort* specifies a cost to the other *sort*. Then, by augmenting the data structure with a sum function, applied to the numeric value attribute property of the cost *sort*, the value of this function is automatically computed as the sum of all cost values. Figure 4 illustrates a similar example in the context of lighting design for a stage or TV or movie studio. Consider a *sort lights*, of labels denoting spot lights or other movable lights, a *sort intensityvalues*, of numeric values representing light intensities or wattage values, and a *sort intensity,* of numeric functions:

*lights* : [Label]
*intensityvalues* : [Numeric]
*intensity* : [NumericFunction] (7)

Both labels and numeric functions adhere to a discrete behavior, while numeric values adhere to an ordinal behavior. Consider a composition of these three *sorts* under the attribute relationship, such that each intensity function has as attribute a collection of lights and each light has as attribute a single intensity value:

*lights_intensity* : *intensity* ^ *lights* ^ *intensityvalues* (8)

By instantiating the *sort intensity* with a sum function applied to the numeric value property attribute of the *sort intensityvalues*, the value of this function is automatically computed as the sum of all intensity values of the lights that are assigned as attribute to this sum function.

Next, consider an extension of this composition of *sorts* with another *sort* providing type or clustering information, e.g., a *sort beams*, of labels denoting the beams that serve to hold the lights above the stage:

$$beams : [\text{Label}] \tag{9}$$

Then, the relative position of this new *sort* with respect to the *sort* of functions has important consequences considering the number of instances of the selected function and, therefore, the result of each of these functions. Figure 5 presents two alternative data views of the same data. In the first data view (left side of Figure 5), the *sort beams* is considered as an attribute to the *sort intensity*, such that the intensity function has as attribute a collection of beams, each of which has as attribute a collection of lights and each light has as attribute a single intensity value:

$$lights_intensity1 : intensity \wedge beams \wedge lights \wedge intensityvalues \tag{10}$$

In this case, the result of the sum function is still the total intensity of all lights, irrespective of the beams these lights are assigned to.

In the second data view (right side of Figure 5), the *sort intensity* is instead considered as an attribute to the *sort beams*, such that each beam has as attribute an intensity function, which itself has as attribute a collection of lights and each light has as attribute a single intensity value:

$$lights_intensity2 : beams \wedge intensity \wedge lights \wedge intensityvalues \tag{11}$$

In this case, each beam specifies its own sum function and the respective results are the total intensity of only the lights on this beam.

Thus, moving data functions within the data structure by altering the compositional structure of the representation, automatically alters the scope of the function and thus the result. In this way, data functions can be used as a technique for querying design information, and moving the data function alters the query. Functions that apply simultaneously to two property attributes of two different *sorts*, or compositions thereof, can be used to compute more complex derivations. Consider cost values for linear building elements such as beams, with the cost expressed per meter. If the beam element has a property attribute specifying the length of the element, in the case of a line segment representing the beam, a function might be applied that sums the product of the length of each beam element with the respective cost per unit length. A similar approach could be considered for non-numeric functions, for example, applying to strings or vectors.

```
sort lights : [Label];
sort intensityvalues : [Numeric];
sort intensity : [NumericFunction];
sort lights_intensity : intensity ^ lights ^ intensityvalues;

form $lights = lights_intensity:
{ sum(intensityvalues.value)
   { "light1"
      { 100 },
     "light2"
      { 150 },
     "light3"
      { 70 } } };
```

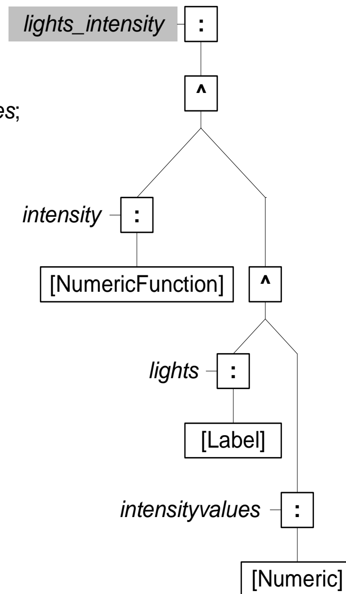


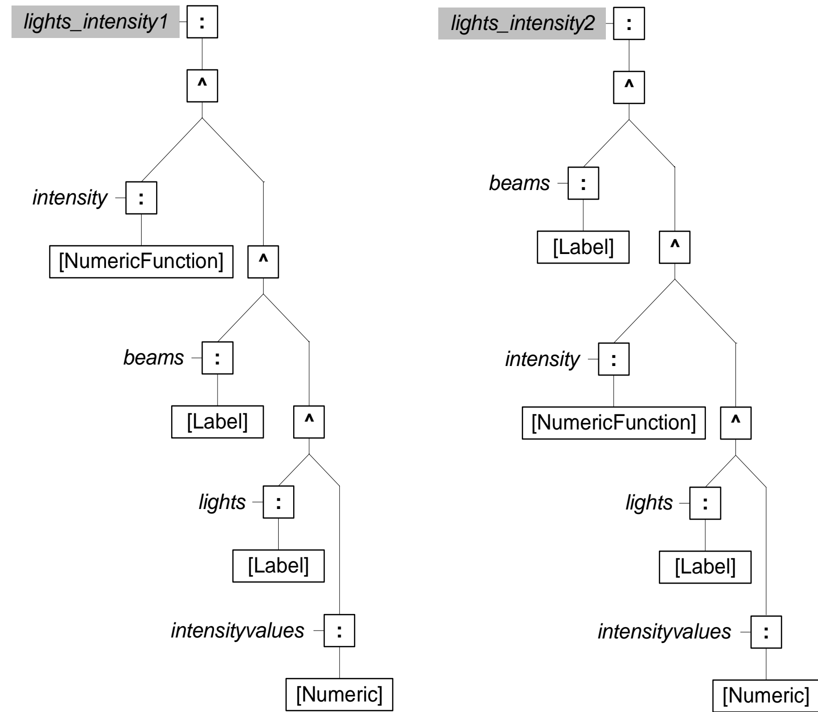

*Figure 4.* Textual and graphical definition of a *sort* representing the intensity values of lights and including a numeric function, and the description of an exemplar data form with a sum function applied to the numeric *value* attribute property of the *sort intensityvalues* (*Sorts* Description Language). In the definition of a *sort*, '^' denotes the operation of attribute; ':' denotes the naming of a *sort*; '[Label]', '[Numeric]' and '[NumericFunction]' are primitive *sorts*, the latter defines a *sort* of numeric functions applied to a single attribute property of another *sort*.

## 5. Design Rules and Grammars

Spatial change can be viewed as a computation $s - f(a) + f(b)$, where $s$ is a shape, and $f(a)$ is a representation of the emergent part (shape) that is altered by replacing it with the shape $f(b)$ (Krishnamurti and Stouffs 1997). This computation subsumes both spatial recognition and subsequent manipulation. It can also be expressed in the form of a spatial rule $a \rightarrow b$. Rule application, then, consists of replacing the emergent shape corresponding to $a$, under some allowable transformation, by $b$, under the same transformation.

Rules can further be grouped into grammars. A grammar is a formal device for the specification of a language; it defines a language as the set of all structures generated by the grammar, where each generation starts with an initial structure and uses rules to achieve a structure that contains only elements from a terminal vocabulary. The specification of spatial rules and grammars leads naturally to the generation and exploration of possible

sort *lights* : [Label];
sort *intensityvalues* : [Numeric];
sort *beams* : [Label]
sort *intensity* : [NumericFunction];
sort *lights_intensity1* : *intensity* ^ *beams* ^ *lights* ^ *intensityvalues*;
sort *lights_intensity2* : *beams* ^ *intensity* ^ *lights* ^ *intensityvalues*;




```
form $lights = lights_intensity1:
{ sum(intensityvalues.value)
   { "beam1"
      { "light1"
         { 100 },
        "light2"
         { 150 } },
     "beam2"
      { "light3"
         { 70 } } } };
```

```
form $lights = lights_intensity2:
{ "beam1"
   { sum(intensityvalues.value)
      { "light1"
         { 100 },
        "light2"
         { 150 } } },
  "beam2"
   { sum(intensityvalues.value)
      { "light3"
         { 70 } } } };
```

*Figure 5.* Textual and graphical definition of two alternative *sorts* representing the intensity values of lights (attached to beams) and including a numeric function, and the description of exemplar data forms with the sum function applied to the numeric *value* attribute property of the *sort intensityvalues* (*Sorts* Description Language). In the definition of a *sort*, '^' denotes the operation of attribute; ':' denotes the naming of a *sort*; '[Label]', '[Numeric]' and '[NumericFunction]' are primitive *sorts*.

spatial designs; the concept of spatial elements or shapes emerging under a part relation is highly enticing to design search (Mitchell 1993; Stiny 1993).

The concept of search is more fundamental to design than its generational form alone might imply. Furthermore, there is no need to restrict it to spatial structures. In fact, any mutation of an information structure into another one, or parts thereof, can constitute an action of search. As such, a design rule may be considered to specify a particular composition of design operations and/or transformations that is recognized as a new, single, operation and applied as such. Design rules can serve to facilitate common operations, e.g., for changing one design element into another or for creating new design information based on existing information in combination with a rule. Similarly, a grammar is more than a framework for generation; it is a tool that permits a structuring of a collection of rules or operations that have proven their applicability to the creation of a certain set (or language) of designs.

Applied to *sorts*, rules and grammars can be considered as a means to contain and facilitate the flexibility and dynamism that *sorts* provides. The specification of design queries through data functions, and the transformation of *sorts* to support alternative data views, can also play a role in the application of a design rule. The central problem in implementing design rules and grammars is the *matching problem*, that of determining the transformation under which the emergent part is recognized in the data. The implementation of the matching problem for *sorts* relies on the part relationship underlying the behavioral specification of *sorts* and only applies to each primitive *sort* or data type (section 3). Rule application then results in a subtraction operation followed by an addition operation. Both operations are also defined as part of the behavioral specification of *sorts*. Stouffs and Krishnamurti (2001b) present a few examples of grammar formalisms that can be expressed with *sorts*.

## 6. User Interaction

Exploring alternative design representations requires the ability to alter representational structures or *sorts*, e.g., by adding or removing components, or by modifying the compositional relationships. Integrating data functions into design data forms similarly necessitates the ability to intervene into the data form and manipulate its composition of data entities and constructive relationships. Utilizing data recognition and design rules also benefits from the same ability to alter and build *sorts* and corresponding data forms. This, furthermore, necessitates a degree of understanding of the representational and data structures that can only be achieved using visual (graphical) means. Practical representational structures for design however may become very large and achieving a visual understanding of the representational structure

may be hard to achieve. Furthermore, manipulating large representational structures by editing the individual components and relationships is far from straightforward. Achieving a desired result may require detailed knowledge or investigation of the structures and painstakingly specific manipulations. Therefore, we argue for an incremental modeling approach (see also Woodbury et al. 1999) and a user interaction to support this.

We consider *sorts* as a complex adaptive system; such systems possess the distinguishing characteristics of robustness and flexibility (Dooley 1997; Kooistra 2002). In the context of building representational structures, robustness can be considered to mean that the system offers the possibility for "correspondence" (communication) leading to an agreement on the representation that prevails in the system. At the same time, the system must offer the possibility for representations to change and in such a way that, in principle, claims on this representation generate quality improvement. With respect to *sorts*, assigning a name to a *sort* can be considered as laying claim to this *sort* with the purpose of improving quality. Correspondence on *sorts* can be achieved through incremental changes on *sorts* and by agreement on the naming of *sorts*. This implies that the incremental modeling of *sorts* in the form of defining *sorts* in terms of other *sorts* can play an important role in achieving agreement and thus in containing the "chaos" to which the construction of *sorts* can lead.

While *sorts* consider a finite vocabulary of primitive *sorts* or data types and compositional operators, practical representational structures for design can be very large and, therefore, the variety in *sorts* that can be constructed is by any practical means immeasurable. Constructing *sorts* could therefore result in a seemingly infinite series of questions or choices on which component to add when and where in the representational structure and, thus, result in "chaos." The complexity paradigm implies "systemic inquiry to build fuzzy, multivalent, multilevel and multidisciplinary representations of reality" (Dooley 1997). *Sorts* can be considered, to a certain extent, as a means to build such representations. "Order arises from complexity through self-organization" (Prigogine and Stengers 1984). In the context of building representational structures or *sorts*, the process of self-organization can take on the form of human communication or correspondence.

Correspondence on *sorts* must be facilitated through the user interaction with *sorts*. We have already referred to the ability to model *sorts* incrementally. We are also investigating kinds of actions that can be perceived purposeful in an exploratory process. These are, for example, the specification of a focus onto the structure expressing a particular interest and the selection of a part of the structure as extent of our interest. Each of these actions results in a transformation of the structure.

The expression of a focus onto a representational or data structure can be directly related to the (hierarchical) composition of the structure's entities

under the compositional relationships. Entities that are considered more important are commonly found at a higher level in the structure's composition. The attribute relationship serves as a prime example, leading the focus onto the object of the relationship, while the attribute expresses a qualifier with respect to this object. For example, in an architectural design description, spatial information is commonly considered more important such that other information entities are assigned as attributes to the relevant spatial entities. Similarly, object-oriented models often adopt a hierarchical structure of functional objects at various levels of detail, reflecting upon an increasingly narrower information focus. For example, architectural design models are commonly organized by a hierarchical classification of functional areas, such as buildings, floors and zones, in that order.

Thus, expressing a focus onto the representational or data structure can result in a transformation of the hierarchical structure that raises the entity under focus towards the top of the structure. Such a transformation can be achieved automatically by reversing attribute relationships and by modifying other compositional relationships. This transformation may take place under the objective to maximize compatibility with the original representation and minimize data loss. Selecting a part of the structure can similarly lead to the breakup of compositional relationships attempting to maintain maximal compatibility with respect to the selection.

When considering that every change to a representational structure or *sort* constitutes a different data view, it can be argued that advanced support for exploring different data views at the same time facilitates the investigation and manipulation of representational structures. We are currently developing a prototype interface to build and edit definitions of *sorts*, to compare and match *sorts* and to construct corresponding data forms.

## 7. Conclusion

Representational flexibility for design cannot be simply realized by providing the user access to the representational and data structures and enabling the modification of these structures through the addition of attributes or the manipulation of the structures' entities and compositional relationships. It also has to facilitate the exploration of these structures through searching and querying the structures. Furthermore, it can be desirable to be offered the ability to identify and store common actions and manipulations for later reuse. Through support for data views, data recognition, design queries and design rules, the theory of *sorts* is a more than viable candidate for achieving representational flexibility for design. The success of this or other approach is as much dependent on the accessibility of the approach and its techniques to the user. For example,

powerful query languages do not as such serve the end user (or designer) who is only interested in having easy access to the information, not in learning a new language. A visual approach can offer a solution. "Visual query languages […] allow the user to express arbitrary queries without having to master the syntax of a rigid textual query language" (Erwig 2002). Further research and developments into *sorts* will focus onto the user interaction aspect of utilizing *sorts* for exploring alternative data views, data recognition, design queries and design rules. This should also enable us to consider and investigate more complex and practical examples.

## Acknowledgements

This work is partly funded by the Netherlands Organization for Scientific Research (NWO), grant nr. 016.007.007. 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 Bige Tunçer for the development of the semantic structures presented in Figures 1 and 2, and Michael Cumming for his work on the development of a prototype interface to build and manipulate *sorts*. The first author benefited from communication with Jan Kooistra concerning complex adaptive systems.

## References

Bazjanac, V: 1998, Industry foundation classes: Bringing software interoperability to the building industry, *The Construction Specifier* **6/98**: 47–54.

Dooley, KJ: 1997, A complex adaptive systems model of organization change, *Nonlinear Dynamics, Psychology, and Life Sciences* **1**(1): 69–97.

Erwig, M: 2002, Design of spatio-temporal query languages, position paper presented at the *Workshop on Spatio-temporal Data Models for Biogeophysical Fields*, San Diego Supercomputer Center, La Jolla, California, <www.calmit.unl.edu/BDEI/papers/erwig_position.pdf>(12 February 2004).

Groth, DP and Robertson, EL: 1998, Architectural support for database visualization, *Proceedings of the 1998 Workshop on New Paradigms in Information Visualization and Manipulation*, ACM Press, New York, NY, pp. 53–55.

ISO: 1994, *ISO 10303-1, Overview and Fundamental Principles*, International Standardization Organization, Geneva.

Kooistra, J: 2002, Flowing, *Systems Research and Behavioral Science* **19**(2): 123–127.

Krishnamurti, R: 1992, The maximal representation of a shape, *Environment and Planning B: Planning and Design* **19**: 267–288.

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

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

Mäntylä, M: 1988, *An Introduction to Solid Modeling*, Computer Science Press, Rockville, MD.

Mitchell, WJ: 1993, A computational view of design creativity, *in* JS Gero and ML Maher (eds), *Modeling Creativity and Knowledge-Based Creative Design*, Lawrence Erlbaum Associates, Hillsdale, NJ, pp. 25-42.

Prigogine, I and Stengers, I: 1984, *Order Out of Chaos*, Bantam Books, New York.

Snyder, JD: 1998, *Conceptual Modeling and Application Integration in CAD: The Essential Elements*, PhD 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: 1991, The algebras of design, *Research in Engineering Design* **2**: 171–181.

Stiny, G: 1993, Emergence and continuity in shape grammars, *in* U Flemming and S Van Wyk (eds), *CAAD Futures '93*, North-Holland, Amsterdam, pp. 37–54.

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

Stouffs, R and Krishnamurti, R: 1996, On a query language for weighted geometries, *in* O Moselhi, C Bedard and S Alkass (eds), *Third Canadian Conference on Computing in Civil and Building Engineering*, Canadian Society for Civil Engineering, Montreal, pp. 783–793.

Stouffs, R and Krishnamurti, R: 2001a, On the road to standardization, *in* B de Vries, J van Leeuwen and H Achten (eds), *Computer Aided Architectural Design Futures 2001*, Kluwer Academic, Dordrecht, The Netherlands, pp. 75–88.

Stouffs, R and Krishnamurti, R: 2001b, Sortal grammars as a framework for exploring grammar formalisms, *in* M Burry, S Datta, A Dawson and J. Rollo (eds), *Mathematics and Design 2001*, The School of Architecture & Building, Deakin University, Geelong, Australia, pp. 261–269.

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.

Tunçer, B, Stouffs, R and Sariyildiz S: 2002, Document decomposition by content as a means for structuring building project information, *Construction Innovation* **2**(4): 229–248.

van Leeuwen, JP: 1999, *Modelling Architectural Design Information by Features*, PhD dissertation, Eindhoven University of Technology, The Netherlands.

van Leeuwen, JP and Fridqvist, S: 2003, Object version control for collaborative design, *in* B Tunçer, S Özsariyildiz and S Sariyildiz (eds), *E-Activities in Building Design and Construction*, Europia Productions, Paris, pp. 129–139.

Woodbury, R, Burrow, A, Datta, S and Chang, T, 1999, Typed feature structures and design space exploration, *Artificial Intelligence in Design, Engineering and Manufacturing* **13**(4): 287-302.