MAPPING DESIGN INFORMATION BY MANIPULATING REPRESENTATIONAL STRUCTURES

RUDI STOUFFS¹, RAMESH KRISHNAMURTI² AND MICHAEL CUMMING¹
¹Delft University of Technology, Faculty of Architecture, Delft, The Netherlands
²Carnegie Mellon University, School of Architecture, Pittsburgh, PA, USA

Abstract. Design problems require a multiplicity of viewpoints each distinguished by particular interests and emphases. Alternative viewpoints necessitate different representations of the same entity, albeit a building or building part, a shape or other complex attribute. We argue that the exploration of alternative design views can be supported by providing access to the representational structure and by allowing the structure to be manipulated through incremental changes. Hereto, we briefly describe the representational framework of sorts and present its support for comparing representational structures and mapping design information according to it. We illustrate the creation and manipulation of structures and their comparison. We consider the specification of design queries through the integration of data functions into representational structures. We conclude with a presentation of future work.

1. Introduction

Computational design relies on effective information models, 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.” Without emphasis on the geometric aspects, this remains as important today. However, current computational design applications tend to focus on the tools and operations for the creation and manipulation of design artifacts. 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.

Design views facilitate a visual inspection of design data and information. Design queries support the analysis of existing design information in order to derive new information that is not explicitly available in the information structure. Design views can be understood to be discrete and domain-specific; design queries on the other hand seem to indicate small incremental steps transcending common, domain-specific views in search of information that does not naturally form part of the design view. At the same time, the result of a design query, possibly presented in the context of other design information, may be seen to define a design view and, similarly, design views may be expressed through design queries.

The distinction between discrete and incremental views can also be related to the development of integrated data models. Integrated data models span multiple disciplines and support different views. These 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 2001). These efforts also mainly target software developers who can ensure compatibility of their own representation corresponding to a particular design view with the integrated data model.

Alternative modeling techniques consider a bottom-up, constructive approach. These provide a more extensive degree of flexibility that allows
for the development of information models that are context, and thus project specific. This flexibility may also enable incremental changes to existing representations supporting alternative design views. 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. Concept modeling (van Leeuwen and Fridqvist 2003) 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) allows for the specification of schematic descriptions that can be used to generate computer programs that provably map data between different applications.

This suggests that support can be provided for exploring alternative design views by providing access to the representational structure and by allowing the structure to be manipulated through incremental changes. In this paper, we briefly describe the representational framework of sorts (Stouffs and Krishnamurti 2002) and present its support for comparing representational structures and mapping design information according to it. We illustrate the creation and manipulation of structures and their comparison. We consider the specification of design queries through the integration of data functions into representational structures. We conclude with a presentation of future work.

2. Defining sorts

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

In the construction of sorts, every composition of properties is considered a sort. Even a single property defines a sort. Thus, a sort is typically a composition of other sorts. Comparing different sorts, therefore, requires a comparison of the respective properties and their constructive relationships. We denote a sort identified by a single property a primitive sort and all other
sorts composite sorts. A primitive sort necessarily has a name, that is the name assigned to the property. A composite sort can also have a name assigned. Named sorts can be conceived to define object classes. However, in contrast to the traditional product modeling approach, the collection of properties of a class is not predefined. This allows class structures easily to be modified, both by adding and removing properties, and by altering the constructive relationships (see also Van Leeuwen et al. 2001). For this purpose, we consider even property relationships and numeric data functions as properties, such that these can be dealt with in the same way.

The attribute constructor, denoted ‘^’, specifies a subordinate composition of sorts, under an attribute relationship. For example, a sort of labeled plane segments is specified as a sort of plane segments, with one or more labels assigned as attribute to each plane segment. Figure 1 illustrates three alternative sorts derived from the same sort by composing this sort with another sort under the attribute constructor. Consider a sort to represent a drawing, denoted drawings. Consider a new sort with the purpose of representing a collection of drawings where each drawing is distinguished by some attribute information. If the distinguishing aspect is a name, we can define the new sort named_drawings as follows:

\[
\begin{align*}
\text{labels} : & [\text{Label}] \\
\text{named\_drawings} : & \text{drawings} ^\text{\textquotesingle} \text{labels} 
\end{align*}
\]

(1)

Here, ‘\(^\text{}\)’ denotes the operation of semantic identification, i.e., assigning a name to a sort, and ‘[Label]’ defines the type of the primitive sort. If the distinguishing aspect is a point of reference for the respective drawing, the resulting sort layouts can be defined as follows:

\[
\begin{align*}
\text{points} : & [\text{Point}] \\
\text{layouts} : & \text{drawings} ^\text{\textquotesingle} \text{points} 
\end{align*}
\]

(2)

We may also combine both distinguishing aspects, for example, by assigning the labels as attribute to the respective reference point, as follows:

\[
\begin{align*}
\text{named\_layouts} : & \text{drawings} ^\text{\textquotesingle} \text{points} ^\text{\textquotesingle} \text{labels} 
\end{align*}
\]

(3)

The sum constructor, denoted ‘+’, allows for disjunctively co-ordinate compositions of sorts. For example, a sort of spatial elements may be defined as the sum of a sort of points, a sort of line segments, a sort of plane segments and a sort of volumes; then, a spatial element can be either a point, line segment, plane segment or volume. Figure 2 illustrates a sort representing a hierarchical tree 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:
MAPPING DESIGN INFORMATION BY MANIPULATING...

\[ \text{concepts : [Label]} \] (4)

\[ \text{named_drawings : [Label]} \]
\[ \text{drawings} \]
\[ \text{labels} \]
\[ \text{[Label]} \]

\[ \text{layouts : [Point]} \]
\[ \text{drawings} \]
\[ \text{points} \]
\[ \text{[Point]} \]

\[ \text{named_layouts : [Point]} \]
\[ \text{drawings} \]
\[ \text{labels} \]
\[ \text{points} \]

Figure 1: Diagrammatic definition of three sorts, named_drawings, layouts and named_layouts, each derived from the sort drawings by combining this sort with another sort under the attribute relationship.

The subordinate relationship between keywords is expressed by the attribute constructor on sorts. The resulting sort, named conceptstree, is defined recursively:

\[ \text{conceptstree : concepts + concepts ^ conceptstree} \] (5)

The attribute constructor relates to each individual keyword (concepts) a non-empty data form of subordinate keywords (conceptstree). The sum constructor (‘+’) allows for 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. Figure 2 also presents an exemplar data form corresponding to the sort conceptstree and expressed using the Sorts Description Language; individual concepts, i.e., labels, are assigned either to the sort concepts, or with an attribute form to the sort concepts ^ conceptstree.

An alternative view of a semantic structure (or 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 2 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:

\[ \text{conceptrefs : concepts} \] (6)
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 2: Diagrammatic definition of a recursive sort conceptstree representing a hierarchical structure of architectural concepts, and the (partial) description of an exemplar data form.

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

$$(\text{hasrefs, isrefs}) : [\text{Property}] (\text{concepts, conceptrefs})$$

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 sum constructor (figure 3):

$$\text{conceptsmap : concepts + concepts ^ conceptsmap + concepts ^ hasrefs + conceptrefs ^ isrefs}$$

(6)

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.

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 3: Diagrammatic definition of a recursive sort conceptsmap representing a (semantic) map of architectural concepts, and the (partial) description of an exemplar data form.
Figure 3 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.

3. Mapping sorts

Sorts can be compared by matching their primitive sorts and constructive relationships. Matches can be identified, roughly, as equivalent, similar and convertible (Stouffs and Krishnamurti 2002). This classification is considered from the perspective of possible data loss and on the basis of syntactic and semantic similarity. Two sorts are equivalent if these are related under semantic identification, e.g., one is semantically derived from the other. For example, the sorts concepts and conceptrefs (5) are equivalent. Equivalent sorts are syntactically identical; this guarantees the exchange of data without data loss, except for the loss of semantic identity. Two sorts are denoted similar if these are similarly constructed from equivalent sorts. For example, the sorts conceptstree (4) and concepts + concepts ^ conceptsmap would be considered similar, if only the sorts conceptstree and conceptsmap were similar. However, comparing the sorts conceptstree and conceptsmap only results in a partial match: the sort conceptstree matches a part of the sort conceptsmap, where the corresponding parts are similar under this partial match.

The similarity of sorts relies on the existence of a semi-canonical form, specifying a composition over sum of one or more sorts, each of which is a composition over the attribute constructor of one or more primitive sorts. Associative and distributive rules with respect to the constructors of sum and attribute allow for a syntactical reduction of sorts to this semi-canonical form (Stouffs and Krishnamurti 2002), e.g.:

\[
\begin{align*}
    a \land (b \land c) &= a \land b \land c = (a \land b) \land c \\
    a + (b + c) &= a + b + c = (a + b) + c \\
    a \land (b + c) &= a \land b + a \land c \\
    (a + b) \land c &= a \land c + b \land c
\end{align*}
\]

(7)

The rules above are automatically applied to any sort structure; the respective sorts are considered identical. However, these rules do not take into account the operation of semantic identification. Consider, for example, the following associative rules over the attribute constructor:

\[
\begin{align*}
    a \land (d : b \land c) &\rightarrow a \land b \land c \\
    (d : a \land b) \land c &\rightarrow a \land b \land c
\end{align*}
\]

(8)

Though syntactically identical, these sorts cannot be considered identical; converting the left-hand-side into the right-hand-side would induce a loss of
MAPPING DESIGN INFORMATION BY MANIPULATING …

semantic information. These rules are not automatically applied to the specification of a sort, but only when comparing sorts based on their semi-canonical form. Similarly, the following distributive rules serve the reduction of sorts to their semi-canonical form for the comparison of sorts:

\[
\begin{align*}
  a \land (d : b + c) & \rightarrow a \land b + a \land c \\
  (d : a + b) \land c & \rightarrow a \land b + a \land c
\end{align*}
\]

(9)

In general, if two sorts can be reduced to the same semi-canonical form, then these sorts are considered similar. No data loss, except for the loss of semantic identity, occurs when exchanging data between similar sorts. In the case of a partial match, data exchange without data loss will apply from the part to the whole if the parts are similar. In the opposite direction, the occurrence of data loss is dependent on the actual data that is exchanged. For example, converting data from conceptstree to conceptsmap involves no data loss; a tree structure is a special instance of a network or map structure. Converting data in the other way may involve data loss; the data lost in this case is the identification of shared concepts even if each copy of these concepts is fully expanded in a depth-first traversal.

If two sorts are constructed from the same equivalent sorts but cannot be reduced to the same semi-canonical form, then these sorts are considered convertible. For example, points \(^\land\) labels and labels \(^\land\) points are considered convertible. Whether data loss occurs when exchanging data between convertible sorts depends on the specifics of the primitive sorts, in particular, their behavioral specification (Stouffs and Krishnamurti 2002).

Figure 4 illustrates two sorts built from the same primitive sorts using only the attribute constructor, but considering the primitive sorts in a different order. These primitive sorts are the sorts lights and beams, both of labels, intensity values of numeric values, and intensity of numeric functions:

\[
\begin{align*}
  \text{lights} & : \text{[Label]} \\
  \text{beams} & : \text{[Label]} \\
  \text{intensity values} & : \text{[Numeric]} \\
  \text{intensity} & : \text{[NumericFunction]}
\end{align*}
\]

(10)

Consider lighting design for a stage or TV studio: a number of lights are selected and positioned, and placed on a stand or attached to a beam. Next, electrical cables are strung in order to power the lights. When laying out these cables, the intensity (wattage) of the lights on each beam has to be considered. The use of numeric functions as a data type enables numeric functional behavior to be integrated into data constructs. Numeric functions specify both a functional description, a sort's property attribute as argument and a result value; the result value is automatically recomputed using the functional description over the sort's property attribute each time the data form is traversed, e.g., when visualizing the data. In the lighting example,
this property attribute is the numeric value of the *intensityvalues* entities (see the exemplar data forms in figure 4).

The sort *lights_intensity1* represents numeric functions that apply to beams that have lights that have an intensity value; the sort *lights_intensity2* represents beams that consider numeric functions that apply to lights that have an intensity value:

\[
\begin{align*}
\text{lights_intensity1} & : \text{intensity} \wedge \text{beams} \wedge \text{lights} \wedge \text{intensityvalues} \\
\text{lights_intensity2} & : \text{beams} \wedge \text{intensity} \wedge \text{lights} \wedge \text{intensityvalues}
\end{align*}
\] (11)

The position of the numeric function in the structure defines its scope. Consider the two data forms presented in figure 4. On first inspection, these
seem to contain the same data, even if their organization is slightly different: the design consists of two beams, the first one has two lights with intensity values of 100 and 150 and the second has one light with an intensity value of 70. The numeric function is in both cases the sum function applied to the numeric value of the intensity values entities, i.e., 100, 150 and 70. However, in the first example, the scope of the single sum function is the entire design and the result of the function will be the sum of all three values, thus, 320. In the second example, each beam possesses a sum function, the scope of which is only the lights attached to this beam. The respective results will be 250 and 70.

Since the two sorts lights_intensity1 and lights_intensity2 are constructed from the same primitive sorts, they are considered convertible. Converting each data form according to the sort of the other data form will result in the other form. Therefore, when converting data between the sorts lights_intensity1 and lights_intensity2, data loss as such does not occur because reconverting the data to the first sort results in the original data form. At the same time, the data in both views is not identical.

4. Manipulating sorts

All examples considered above illustrate how incremental changes to representational structures can yield alternative design views that offer new functionality or answer design queries. Figure 1 presents three alternative examples of how adding a new sort to an existing sort can expand the functionality of this sort. In these examples, a collection of named drawings or a layout of drawings is represented using a sort that is a composition of a given sort for a single drawing and a sort of labels, points or labeled points, under the attribute constructor. The opposite action of removing a sort can similarly be considered to support a different design view, in the case of the examples, one drawing out of a collection of drawings. Figures 2 and 3 similarly illustrate two sorts where one can be considered as an extension of the other. These sorts, conceptstree and conceptsmap, support related data structures, the one hierarchical, the other a hierarchical structure with shared nodes. This extension of conceptstree to conceptsmap requires the specification of one new primitive sort, with the aspects hasrefs and isrefs, as well as the semantic derivation of conceptrefs from concepts. Finally, figure 4 shows how a small change in the compositional structure, in this case switching the position of two adjacent sorts in a series of attribute relationships, can yield two different design views that answer two different queries.

The same actions can also lead to more far-reaching changes. Merging two sorts together under a sum or attribute relationship, selecting and extracting a part of a sort as a new sort, and altering the compositional
structure by redefining the hierarchical order, all can be considered as incremental changes. The effect and reach of such change is very much dependent on the complexity of the sorts involved. We still consider the change incremental if it involves only a single action by the user. Merging a sort into another requires the sort and its intended location in the other sort to be identified. Depending on the compositional structure of both sorts near the point of merging, changes in these respective structures may be necessary in order to achieve a new structure that complies to the definition and representation of a sort. Extracting a part of a sort requires this part to be selected. The exact boundaries of this selection can be made dependent on the need to minimize data loss through maximal compatibility.

Altering the compositional structure by redefining the hierarchical order can be achieved by selecting an entity from the representational or data structure in order to bring this to the top of the structure. We consider this action to be an expression of focus. For example, object-oriented models often adopt a hierarchical structure of functional objects at various levels of detail, reflecting upon an increasingly narrower information focus. Similarly, architectural design models are commonly organized by a hierarchical classification of functional areas, such as buildings, floors and zones, in that order. 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 properties to the relevant spatial entities. 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. For example, consider a primitive sort \( b \) in a composition with sorts \( a \) and \( c \) under the attribute constructor. Then, the following rule specifies a transformation that raises the entity \( b \) to the focus:

\[
(a \wedge b) \wedge c \rightarrow (b \wedge a) \wedge c = b \wedge a \wedge c
\]

Since semantic identity cannot be maintained under such transformation, reduction rules for the syntactical reduction of sorts to their semi-canonical form can be used to assist in the above transformation.

From this, we can derive that incremental manipulations of representational structures can support the exploration of alternative design views. While each action may yield only a small step, a series of incremental changes may lead the user from one desired design view to another and enable design information to be mapped accordingly. Furthermore, these incremental changes may give the user insight into the composition of the representational structure and its data and into the potential for alternative
design views. We are currently developing a graphical interface to sorts that allows for the creation of representational structures as sorts and of corresponding data forms. We envision its extension to support the manipulations here described.

5. Conclusion

Both 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 views and queries. Then, exploring design views and design queries may be achieved by manipulating the representational structure through incremental changes. Such actions may also lead to a conceptual understanding of the representational structure. Effective visualizations of the data structure, combined with intuitive ways of manipulating this structure, can further support such understanding.

Sorts enable the development of alternative representations of a same entity or design, the comparison of representations with respect to scope and coverage, and the mapping of data between representations, even if their scopes are not identical. Alternative design representations can be defined as variations on a given sort, by altering the constructive entities or the composition. Comparing sorts not only yields a possible mapping, but also uncovers the potential for data loss when moving data from less restrictive to more restrictive representations. As such, reorganizations can be guided in order to maximize compatibility with the original representation and minimize data loss.

Acknowledgements

This work is partly funded by the Netherlands Organization for Scientific Research (NWO), grant nr. 016.007.007. The third 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 the reviewers for their constructive comments and Bige Tunçer for the development of the semantic structures presented in figures 1 and 2.

References

R. STOUFFS, R. KRISHNAMURTI, M. CUMMING


