Available online at www.sciencedirect.com
ScienceDirect
Advanced Engineering Informatics 22 (2008) 71–89
ADVANCED ENGINEERING INFORMATICS
www.elsevier.com/locate/aei

Constructing design representations using a sortal approach

Rudi Stouffs *

Faculty of Architecture, Delft University of Technology, P.O. Box 5043, 2600 GA Delft, The Netherlands

Received 29 November 2006; received in revised form 5 July 2007; accepted 17 August 2007

Abstract

Supporting the early phases of design requires, among others, support for the specification and use of multiple and evolving representations, and for the exchange of information between these representations. We consider a complex adaptive system as a model for the development of design representations, and present a semi-constructive algebraic formalism for design representations, termed *sorts*, as a candidate for supporting this approach. We analyze *sorts* with respect to the requirements of a complex adaptive system and compare it to other representational formalisms that consider a constructive approach to representations. We demonstrate the advantages of *sorts* in various examples, illustrate its use to support the specification of design queries and the recognition of emergent information, and consider *sorts* in relationship to integrated product models.

Keywords: Design representations; Design communication; Complex adaptive system; Design queries; Emergence

1. Introduction

Building design is a multi-disciplinary process, involving participants, knowledge and information from various domains. Building design problems, therefore, require a multiplicity of views, each distinguished by particular interests and emphases. Each actor in the design process takes his or her own professionally oriented view—derived from an understanding of current problem solution techniques in the respective domain. Each view, in turn, requires a different representation of the same (abstract) entity; a building may be considered in its entirety, as a shape, a collection of parts or some grouping of properties. As such, different views—or different representations—may derive from different design stages, and may also support different persons or applications within the same design stage. Even within the same task, or by the same person, various representations may serve different purposes defined within the problem context and the selected approach. This is certainly true in architecture, where the design process, by its exploratory and dynamic nature, invites a variety of approaches and representations (see, for example, [1]). Any man-machine system to aid the designer must recognize his reliance on multiple representations [2].

There has been concerted effort in developing integrated product models that span multiple disciplines, multiple methodologies and support different views (e.g., [3]). Various modeling schemes for defining product models and ontologies also exist (e.g., [4,5]). These allow for the development of representations in support of different disciplines or methodologies, and enable information exchange between representations and collaboration across disciplines. Such efforts tend to characterize an a priori top-down approach: an attempt is made at establishing an agreement on concepts and relationships, which offer a complete and uniform description of the project data, mainly independent of any project specifics [6]. We are concerned how data can be effectively structured a posteriori. Research in cognitive science and design cognition has shown that expertise in both problem solving and design often relates to having access to more and better representations [7,8]. This is especially true in architecture, Akin refers to architecture in this respect as a "representation

* Tel.: +31 15 278 1295; fax: +31 15 278 4178.
E-mail address: r.m.f.stouffs@tudelft.nl

1474-0346/$ - see front matter © 2007 Elsevier Ltd. All rights reserved.
doi:10.1016/j.aei.2007.08.007

saturated problem domain'' [9]. Importantly, the outcome of the design process relates to the representation that is used. Furthermore, architectural design differs from other forms of problem solving in that the problems in architecture are generally ill-structured; defining the problem space is an intricate part of the design activity [8]. As the problem shifts during the design process, so should the representation adapt. Design representations may be as much an outcome of as a means to the design process.

To clarify our aims we present an example. Within the conceptual phase of design, architects and designers study relevant precedents and collect and look at design and other documents as sources of knowledge and inspiration [10]. Generally, electronic design document libraries or image archives serve to collect this information, separately from the CAD or modeling environment that is used to shape the design. In this paper, we present a formalism for constructing design representations that can assist in specifying and maintaining relationships between design-related documents and the elements within a CAD model. Specifying such relationships helps to organize the information contained within these documents in relation to the CAD model. Consider a representational structure that reflects on (part of) the CAD model, for example composed of element IDs and descriptions. A corresponding data construct can easily be generated, automatically, from the CAD data. This representational structure can then be extended to allow for document references to be associated with the CAD elements. Using a graphical interface, the user can specify both the references and their associations to CAD elements. When the CAD model is subsequently changed, the data reflecting on the CAD model can be regenerated, while the associated data can be retrieved from the original representational structure using an automatic conversion based on the matching of both representational structures. The formalism here considered supports such matching of representational structures, by means of a subsumption relationship over representational structures and a behavioral specification for data constructs. Merging both data constructs re-associates the document references to the CAD elements, on condition that the respective element IDs have not changed.

Let us denote the representational structures constructed by means of this formalism as *sortal* representations. Consider the construction of sortal representations as compositions of other sortal representations through the use of two compositional operators, an attribute operator specifying an object–attribute relationship between two sortal representations and an addition operator specifying a disjunctive relationship between sortal representations. For instance, the representational structure that reflects on part of the CAD model can be expressed as a composition of element IDs and element descriptions under the attribute relationship (denoted '∧'):

$$element\_ids \wedge element\_descriptions$$

The user can associate document references to CAD elements through a sortal representation that is similarly composed of element IDs and document references using the attribute relationship:

$$element\_ids \wedge document\_references$$

Both sortal structures can subsequently be added, yielding the following sortal representation ('∧' distributes over '+'):

$$element\_ids \wedge (element\_descriptions + document\_references)$$

The resulting sortal structure can be matched back to the sortal representation *element_ids* ∧ *document_references* in order to extract the associations from the CAD-derived data. These can be updated, if necessary, through the user interface. At the same time, the CAD-derived data can be regenerated from the CAD model. Repeating the sortal addition of both structures yields once again the entire sortal structure, relating design documents to CAD elements. Fig. 1 offers a schematic overview of the entire process.

A similar application can be considered for associating cost data to CAD elements and calculating production or construction costs. A corresponding user interface can allow the user to associate component or material types

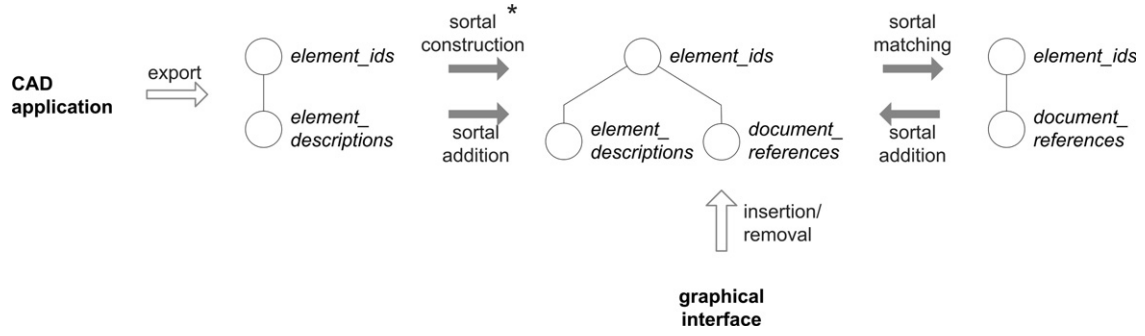

Fig. 1. Schematic overview of the process of relating design documents to CAD elements using *sorts*. Filled arrows denote automatic sortal conversions; * sortal construction applies only once, initially.

with related cost information to selected CAD elements. If the cost information is specified per unit length, area or volume, such information can be extracted from the CAD data to support the automatic calculation of overall costs. For this purpose, the formalism here presented enables the specification and integration of data functions into sortal structures, applying to selected property attributes of sortal representational components and automatically calculating the result of the function. Other examples can be similarly constructed. An application framework that offers the flexibility to associate various kinds of external data (e.g., images, documents, material data, cost data) in various compositions to CAD data within a commercial CAD application is one example of the kind of representational flexibility we aim at.

In the subsequent sections, we consider in detail how we can provide the user access to the specification of a design representation, and the means to create and adapt design representations according to the designer's intentions in the task at hand. We consider the specification of a design representation as a continuous process of adaptation and refinement, and consider a complex adaptive system as a model for such specification in the context of complex design representations. We present a formal approach, named *sorts*, for specifying design representations that offers support for the exchange of information between these representations. We analyze *sorts* with respect to the requirements of a complex adaptive system and compare it to other representational formalisms that consider a constructive approach to representations. We demonstrate the advantages of *sorts* in various examples, illustrate its use to support the specification of design queries and the recognition of emergent information, and consider *sorts* in relationship to integrated product models.

2. A complex adaptive system

A Complex Adaptive System (CAS) is a multi-agent system with certain typical characteristics, such as complexity, self-similarity, emergence and self-organization [11–13]. For example, Dooley considers three key principles of a CAS: "order is emergent as opposed to predetermined, and the state of the system is irreversible and often unpredictable" [12]. We consider a CAS here as a model for the development of design representations and apply these three key principles to the continuous process of adaptation and refinement of design representations in the context of the design activity. The outcome of the design process is generally *unpredictable*, as it is indeterminately related to the—initially, ill-defined—design requirements and the design process. Under the assumption that the design representation is intricately related to the design outcome, this representation is also unpredictable. Furthermore, the state of a design and its representation is in principle *irreversible* as changes to the representational structure can result in data loss. Finally, with respect to the *emergence of order* in the development of a design representation, we refer to Prigogine and Stengers [14]: "Order arises from complexity through self-organization." In the context of constructing a design representation, the process of self-organization takes on the form of human communication or correspondence, leading to an agreement on the representation that prevails in the system (see also [15]). This communication may be considered among different users or between the user and the design application (correspondence between the user's mental model and the application's design representation).

The basic building blocks of a CAS are agents. Depending on the application domain, these agents may constitute biological cells, human individuals, or software processes. "Agents are semi-autonomous units which seek to maximize some measure of goodness, or fitness, by evolving over time. Agents scan their environment—which includes both other agents within the CAS as well as the external environment—and develop schema representing interpretive and action rules" [12]. In the context of constructing design representations, we consider designers as agents, but confine their objective to the maximization of the fitness of their representational view. We consider design representations as the schema they develop, and consider a constructive approach to defining design representations, such that they may evolve from smaller, more basic representations. "These schema are rational bounded: they are potentially indeterminate because of incomplete and/or biased information; they are observer dependent because it is often difficult to separate a phenomenon from its context, thereby identifying contingencies; and they can be contradictory" [12]. Design representations reflect on the designers' views of the design task at hand. These views may depend on past experiences and new insights, on the design process and expectations of how this process will evolve, and on communication and exchange with other designers. Changes to the design representation offer opportunities for changes to the design, on the other hand, changes to the design may necessitate changes to the design representation. Though a design representation must always be precisely defined, its definition may be based on partial and biased information. "Schemas exist in multitudes and compete for survival" [12]. Designers rely on multiple representations that may change over time.

3. Constructing design representations

We consider the use of multiple and evolving design representations, where changes in these representations result from communication among designers and between the designer and the design application. For this purpose, we consider a constructive approach to design representations where representations may evolve from smaller, more basic representations. Typically, a representation is a complex structure of properties and constructors, and a representation may be a construction of another [16]. As such, an approach to constructing representations in terms of other representations can be considered, in which correspondence

can be achieved through the adaptation of the representational structure of properties and constructors and by agreement on the naming of representations, or parts thereof. For example, van Leeuwen and Fridqvist [17] describe a property-oriented data modeling approach in which design concepts are represented as flexible networks of objects and properties. In contrast to traditional modeling approaches, an object has no predefined set of properties and the composition of properties defining an object can be changed at any time. Under this property-modeling approach, correspondence can be achieved through the development (over time) of the network of objects and properties and by agreement on the naming of objects.

Such a constructive approach can greatly benefit from a formal framework that allows for representations of the same entity to be compared and related with respect to scope and coverage, in order to support translation and identify where exact translation is possible. Considering a representation as a complex structure of properties and constructors, comparing alternative representations requires a comparison of their respective properties and their mutual relationships, and of the overall construction. Such a comparison will not only yield a possible mapping in support of information exchange, but also uncover potential data loss when moving data from less-restrictive to more-restrictive representations. For example, Stouffs et al. [16] were able to show, using a subsumption relation defined on well-known solid models, that information loss between some of these solid models is inevitable. Subsumption is a powerful mechanism for comparing alternative representations of the same entity. When a representation subsumes another, the entities represented by the latter can also be represented by the former representation, without any data loss.

There are many representational formalisms that consider the subsumption relationship in order to achieve partially ordered representational structures; most are based on first-order logic. Applied to building design, a good example is Woodbury et al. [18], who adopt typed feature structures as a model for design space exploration. Like many other formalisms, typed feature structures consider a record-like data structure for representing data types. Record-like data structures facilitate the encapsulation of property information in (a variation of) attribute/value pairs [19]. Furthermore, the properties may themselves be typed feature structures, i.e., expressed in terms of record-like data structures, containing (sub)properties. Then, the subsumption relationship defines a partial ordering on feature structures. Furthermore, the algebraic operations of intersection and union (or others similar) may be defined on feature structures so that the intersection of two feature structures is subsumed by either structure, and the union of two feature structures subsumes either structure.

Key to typed feature structures is the notion of partial information structures and the existence of a unification procedure that determines if two partial information structures are consistent and, if so, combines them into a single, new (partial) information structure. Typed feature structures further consider a type hierarchy and a description language, where each type defines a corresponding description. The subsumption relation between feature structures extends the subsumption ordering on types inherent to the type hierarchy. Woodbury et al. [18] also specify a generating procedure that relates feature structures with a description (or type) that they satisfy, and that incrementally generates more complete design structures. This fact—that the generating procedure monotonically generates more complete information structures—could be interpreted as excluding the possibility for information loss and thus making design states reversible. However, the inclusion of an information removal operator is possible providing more flexibility at the cost of limiting search strategies [18]. Datta [20] also presents a visual notation for representing design correspondence between designer and typed feature structures, using the concepts of mixed initiative and rational conversation.

4. Sorts

We consider a semi-constructive algebraic formalism for design representations, termed *sorts* [21], that enables representations to be compared with respect to scope and coverage and that presents a uniform approach for dealing with and manipulating data constructs. Similarly to the property-oriented modeling approach [17], *sorts* can be considered as hierarchical structures of properties, where each property specifies a data type. Properties can be collected and a collection of one or more properties can be assigned as an attribute to another property. Alternatively, *sorts* can be considered as class structures, specifying either a single data type or a composition of other class structures. Like a primitive data type, a *primitive sort* is the basic entity, conceptually, specifying a set of similar data entities, e.g., a class of objects, equivalently, a set of tuples solving a system of equations. For example, points and lines are *sorts*, as are triangles and squares. *Sorts* are not limited to geometrical objects; colors are *sortal*, as are other data types.

A *sort* is typically a composition of other *sorts*. We consider two constructive operators on *sorts*. An *attribute* operator (denoted '$\wedge$') specifies a (conjunctively) subordinate composition of *sorts*. For example, consider a plan of a room, represented as a set of line segments, with the line segments distinguished by type, i.e., each segment represents a wall, door or window. A corresponding *sort*, denoted *plan*, may be a composition under the attribute operator of a *sort* of line segments, *segments*, and a *sort* of labels, *types*:

$$plan\text{: } (segments\text{: } [LineSegment]) \wedge (types\text{: } [Label])$$

A resulting data construct, or data *form*, consists of line segments with associated type labels, such that each line segment has one or more type labels, such as 'wall', 'door'

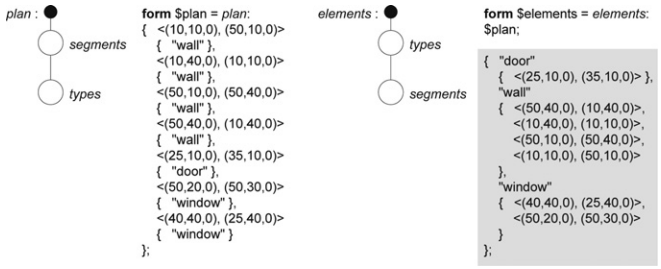

Fig. 2. Two sortal structures, including a graphical depiction of the respective *sort*, consisting of line segments and type labels: (left) segments with types as attributes and (right) types with segments as attributes. The grey box contains the result of the description above it (through automatic conversion).

or 'window', assigned as attribute (Fig. 2 (left)). Obviously, the ordering of the operands is of importance; by switching the ordering of the *sorts segments* and *types* under the attribute operator, we group the line segments by their type:

elements: *types* ∧ *segments*

Converting the data form of *plan* to the new *sort elements* results in a collection of (three) type labels, where each label has one or more line segments as attribute (Fig. 2 (right)). These are the respective line segments that have this label as their type.

We also define a disjunctive operation of *sum* (denoted '+'). For example, we can define a *sort* of points and line segments from a *sort* of points and a *sort* of line segments under the operation of sum:

points_and_segments: (*points*: [*Point*]) + *segments*

A resulting data entity can be either a point or a line segment (Fig. 3). The ordering of the operands under the operation of sum is not important; the operator is commutative and associative.

Similarly to the property-oriented modeling approach [17], the collection of properties of a *sort* is not predefined. This allows sortal structures to be modified easily, both by adding and removing properties, and by altering the constructive relationships. For this purpose, we consider property relationships and data functions (see Section 7.2) too as sortal properties, such that these can be dealt with in

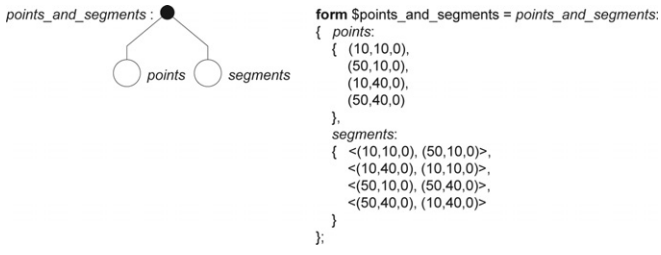

Fig. 3. A sortal structure of a composition of points and line segments under the operation of sum.

the same way. While *sorts* are similarly conceived, there is, at least, one exception: the association of properties to a *sort* occurs through sortal attribution—that is, each property of a *sort* is itself a *sort*. Primitive *sorts* are the exceptions to this rule. Like primitive data types, primitive *sorts* are the smallest building blocks for constructing sortal representational structures. Primitive *sorts* combine to form composite *sorts* under the constructive operators on *sorts*. A primitive *sort* defines the domain of possible values, e.g., a primitive *sort* of weights specifies the domain of positive real numbers, and a primitive *sort* of line segments specifies the domain of intervals on linear carriers. Primitive *sorts* may be constrained over the extent of their domain, for example, limiting weights to values between 0 and 1.

5. A subsumption relationship over *sorts*

Under the disjunction operator, any entity of the resulting *sort* is necessarily an entity of one of the constituent *sorts*. Sortal disjunction consequently defines a subsumption relationship on *sorts* (denoted '⩽'), as follows:

$$a \leqslant b \iff a + b = b;$$

a disjunctive *sort* subsumes each constituent *sort*.

Most logic-based formalisms, including typed feature structures, link subsumption directly to information specificity, that is, a structure is subsumed by another, if this structure contains strictly more information than the other. The subsumption relationship on *sorts* can also be considered in terms of information specificity, however, there is a distinction to be drawn in the way in which subsumption is treated in *sorts* and in first-order logic-based representational formalisms as exemplified by type feature structures. First-order logic formalisms generally consider a relation of inclusion (hyponymy relation), commonly denoted as an *is-a* relationship. *Sorts*, on the other hand, consider a *part-of* relationship (meronymy relation).

Two simple examples illustrate this distinction. Consider a disjunction of a *sort* of points and a *sort* of line segments; this allows for the representation of both points and line segments. We can say that the *sort* of points forms part of the *sort* of points and line segments—note the part-of relationship (Fig. 4). In first-order logic, this corresponds to the union of points and line segments. We can say that both are bounded geometrical entities of zero or one dimensions—note the is-a relationship.

This distinction becomes even more important when we consider an extension of sortal subsumption to attribution. Consider a *sort* of cost types as a *sort* of type labels with cost values associated under the attribute relationship:

cost_types: *types* ∧ (*costs*: [*Weight*])

For example, these cost values may be specified per unit length or surface area for building components. If we lessen the conjunctive character of the attribute operator by

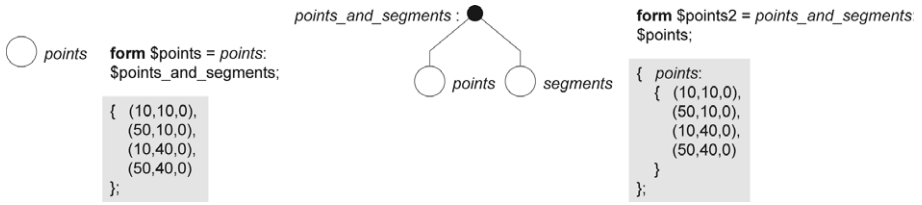

Fig. 4. Two sortal structures: (left) points and (right) points and line segments under the operation of sum, with the first *sort*/structure being part of the second *sort*/structure. The grey boxes contain the results of the respective descriptions above them.

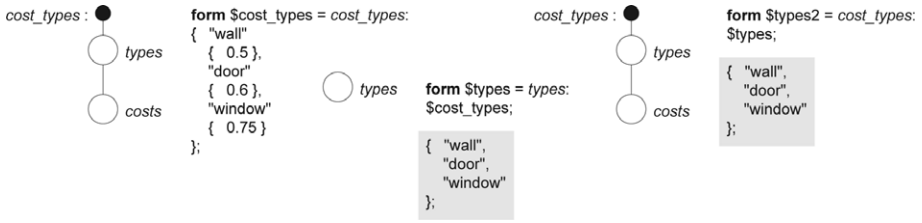

Fig. 5. Three sortal structures: (left) type labels and cost values under the attribute operator, (middle) only type labels and, again, (right) type labels and cost values, where the second *sort*/structure is part of the third *sort*/structure. The grey boxes contain the results of the respective descriptions above them.

making the cost attribute entity optional, then, we can consider a type label to be a cost type without an associated cost value or, preferably, a type label to be part of a cost type, that is, the *sort* of types is part of the *sort* of cost types (Fig. 5). Vice versa, the *sort* of cost types subsumes the *sort* of types or, in general:

$$a \leqslant a \wedge b.$$

In logic formalisms, a relational construct is used to represent such associations. For example, in description logic [22], roles are defined as binary relationships between concepts. Consider a concept Label and a concept Color; the concept of colored labels can then be represented as $\text{Label} \cap \exists \text{hasAttribute.Color}$,[1] denoting those labels that have an attribute that is a color. Here, $\cap$ denotes intersection and $\exists R.C$ denotes full existential quantification with respect to role R and concept C. It follows then that $\text{Label} \cap \exists \text{hasAttribute.Color} \subseteq \text{Label}$; that is, the concept of labels subsumes the concept of colored labels—this is quite the reverse of how it is considered in *sorts*.

Another important distinction is that first-order logic-based representations generally make for an open world assumption—that is, nothing is excluded unless it is done so explicitly. For example, polygon objects may have an assigned color. When looking for a yellow square, logically, every square is considered a potential solution—unless, it has an explicitly specified color, or it is otherwise known not to have the yellow color. The fact that a color is not specified does not exclude an object from potentially being yellow. As such, logic-based representations are automatically considered to be incomplete. *Sorts*, on the other hand, hold to a closed world assumption. That is, we work with just the data we have. A polygon has a color only if one is explicitly assigned: when looking for a yellow square, any square will not do; it has to have the yellow color assigned. This restriction is used to constrain emergence (see Section 7.1). Another way of looking at this distinction between the open or closed world assumptions is to consider their applicability to knowledge representation. To reiterate, logic-based representations essentially represent knowledge; *sorts*, on the other hand, are intended to represent data—any reasoning is based purely on present (or emergent) data.

6. A behavioral specification for *sorts*

Each *sort* has a behavioral specification, governing how data entities combine and intersect, and what the result is of subtracting one data entity from another or from a collection of entities from the same *sort*. This behavioral specification is an important ingredient of *sorts*; it is a prerequisite for the uniform handling of different and a priori unknown data structures and the effective exchange of data between various representations. Consider the association of building performance data to design geometries. The behavior of these data, as a result of an alteration to the geometry, can be expressed through a number of operations chosen to match the expected behavior. When an application receives data along with its behavioral specification, the application can then correctly interpret, manipulate, and represent this information without unexpected data loss. For instance, at the representational level, operations that may otherwise seem trivial, such as adding or removing data entities, become resolutely non-trivial—for instance, the addition of two numbers when these represent cardinal values (e.g., a number of columns that is

[1] Note that this syntax differs slightly from the syntax adopted by Baader et al. [22], which, for example, differentiates the intersection constructor on concepts from the operation of intersection on interpretations. Interpretations do not play a role in this example.

increased) and when these represent 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), and similarly, additive versus subtractive colors, depending on whether these refer to the mixing of surface paints or colors of light, respectively. Fortunately, behavioral specification is reasonably limited to the common operations of addition (combining entities), subtraction, and product (intersecting entities). The common CAD operations of creation and deletion, and selection and deselection, can all be expressed as some combination of addition and subtraction from one design space (*sort*) to another. The complex operations of grouping and layering can be treated likewise [23].

We consider a behavioral specification based on a part relationship on the entities of a *sort*, with the sortal operations of addition, subtraction, and product defined in accordance to this part relationship. The specification of the part relationship is depended on the kind of data being considered. Before presenting any exemplar behavioral specifications, first we consider the interaction between individual data entities and collections of data entities under the part relationship. Consider a data form of a *sort* to be a collection of zero, one or more data entities of the same *sort*. Consider the part relationship on the entities of a *sort* to apply equally to the data forms of this *sort*: an entity is part of a data form, if it is part of one or more entities in this collection; a data form is part of another data form, if each entity in the former collection is part of the latter data form. Then, the addition of two data forms (or entities) of the same *sort* results in a new data form, such that both operand forms are part of the resulting form. Likewise, the product of two data forms is a data form, possibly empty, that is a part of both operand forms. Finally, the subtraction of one data form from another results in a third data form, possibly empty, that is part of the second data form.

The simplest specification of a part relationship corresponds to the subset relationship in mathematical sets. Such a part relationship applies to points and labels, e.g., a point is part of another point only if they are identical, and a label is a part of a collection of labels only if it is identical to one of the labels in the collection (Fig. 6). Here, sortal operations of addition, subtraction, and product correspond to set union, difference, and intersection, respectively. In other words, if x and y denote two data collections of a *sort* of points (or labels), and X and Y denote the corresponding sets of data elements, i.e., sets of points (or labels), then ($x : X$ specifies X as a representation of x)

$$x : X \wedge y : Y \Rightarrow x \leqslant y \iff X \subseteq Y$$
$$x + y : X \cup Y$$
$$x - y : X/Y$$
$$x \cdot y : X \cap Y$$

Another kind of part relationship corresponds to interval behavior. Consider, for example, the specification of a part

Fig. 6. Two sortal structures of type labels: the data form is constructed by adding (left) and subtracting (right), respectively, a reference data form to/from itself. The grey boxes contain the results of the respective descriptions above them.

relationship on line segments. A line segment may be considered as an interval on an infinite line (or *carrier*); in general, one-dimensional quantities, such as time, can be treated as intervals. An interval is a part of another interval if it is embedded in the latter; intervals on the same carrier that are adjacent or overlap combine into a single interval. Specifically, a behavior for intervals can be expressed in terms of the behavior of the boundaries of intervals. Let $B[x]$ denote the boundary of a collection x of intervals and, given two collections x and y, let I_x denote the collection of boundaries of x that lie within y, O_x denote the collection of boundaries of x that lie outside of y, M the collection of boundaries of both x and y where the respective intervals lie on the same side of the boundary, and N the collection of boundaries of both x and y where the respective intervals lie on opposite sides of the boundary (Figs. 7 and 8) [24]. Then

$$x : B[x] \wedge y : B[y] \Rightarrow x \leqslant y \iff I_x = 0 \wedge O_y = 0 \wedge N = 0$$
$$x + y : B[x + y] = O_x + O_y + M$$
$$x - y : B[x - y] = O_x + I_y + N$$
$$x \cdot y : B[x \cdot y] = I_x + I_y + M$$

This behavior applies to indefinite intervals too, providing that there is an appropriate representation of both (infinite) ends of its carrier. Likewise, behaviors can be specified for area intervals (plane segments) and volume intervals (polyhedral segments). The equations above still apply though the construction of I_x, O_x, I_y, O_y, M, and N is more complex (Fig. 9) [24].

6.1. Compositional behaviors

Behaviors apply to composite *sorts* as well, that is, the part relationship is defined on data entities belonging to a *sort* defined by attribution or disjunction. Specifically, a

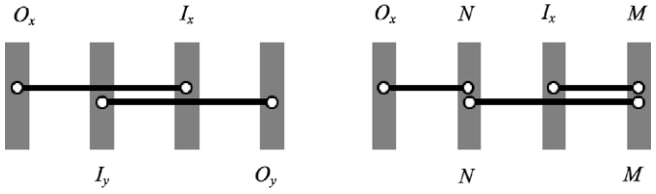

Fig. 7. The specification of the boundary collections I_x, O_x, I_y, O_y, M and N, given two collections of intervals x (above) and y (below).

segments

form $wall_segments = *segments*:
$points_and_segments;

{ <(50,40,0), (10,40,0)>,
<(10,40,0), (10,10,0)>,
<(50,10,0), (50,40,0)>,
<(10,10,0), (50,10,0)>
};

form $window_segments = *segments*:
{ <(50,20,0), (50,30,0)>,
<(40,40,0), (25,40,0)>
};

segments

form $added_segments = *segments*:
$wall_segments + $window_segments;

{ <(50,40,0), (10,40,0)>,
<(10,40,0), (10,10,0)>,
<(50,10,0), (50,40,0)>,
<(10,10,0), (50,10,0)>
};

segments

form $subtracted_segments = *segments*:
$wall_segments – $window_segments;

{ <(25,40,0), (10,40,0)>,
<(50,40,0), (40,40,0)>,
<(10,40,0), (10,10,0)>,
<(50,10,0), (50,20,0)>,
<(50,30,0), (50,40,0)>,
<(10,10,0), (50,10,0)>,
};

Fig. 8. Four sortal structures of line segments: two data forms are constructed by adding (middle) and subtracting (right), respectively, one reference data form (left above) to/from another (left below). The grey boxes contain the results of the respective descriptions above them.

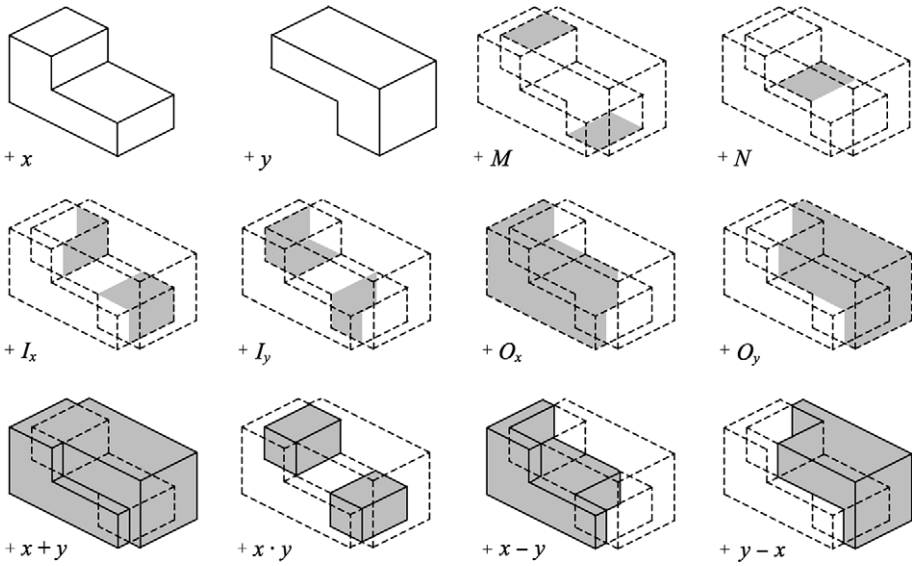

Fig. 9. The boundary collections I_x, O_x, I_y, O_y, M and N for two polyhedral segments x and y, and the collections of polyhedral segments resulting from the operations $x + y$, $x \cdot y$, $x - y$ and $y - x$.

composite *sort* inherits its behavior from its components in a manner that depends on the compositional relationship.

For example, consider the following situation, common in CAD applications, of classifying data entities, say line segments, into layers. Here, we define the *sort* of layered line segments as a composition under the attribute operator of a *sort* of line segments and a *sort* of layer labels. When layers are considered as attribute values, it may seem intuitive to allow just a single-layer label for each line segment (most CAD applications do not normally allow the same object to exist in multiple layers at the same time). However, there is no common notion of a layer ordering (or a corresponding part relationship) that can be used to define sortal operations on layer labels. Instead, if we were to allow a set of layer labels to be assigned as an attribute to a line segment, the result may be interpreted as a collection of identical copies of the same line segment existing in different layers, and could be presented as such to the user. In order to deal with or manipulate a single-layer copy, that is, a copy of the line segment associated with a single label (or singleton subset of labels), this copy needs to be differentiated from the other copies of the same line segment existing in the other layers. The sortal entity representing the single-layer line segment is a part of another sortal entity representing a multi-layer line segment if the first line segment is a part of the second (under the interval behavior for line segments) and if the single label of the first entity is a member of the set of labels for the second entity (Fig. 10). Consequently, by subtracting the first entity from the second, the single-layer line segment is distinguished (i.e., selected) from the remainder.

Thus, under the attribute operator, a data entity is part of a data form if it is a part of the data entities of the first component *sort*, and if it has an attribute form that is a part of the respective attribute form(s) of the data entity (or entities) of the first component *sort* it is a part of. When data forms of the same composite *sort* (under attribution) are pairwise summed (differenced or intersected), identical data entities merge, and their attribute forms combine, under this operation.

The disjunctive operator distinguishes, instead, all operand *sorts* such that each data entity belongs explicitly to one of these *sorts*. For example, a *sort* of points and lines distinguishes each data entity as either a point or a line.

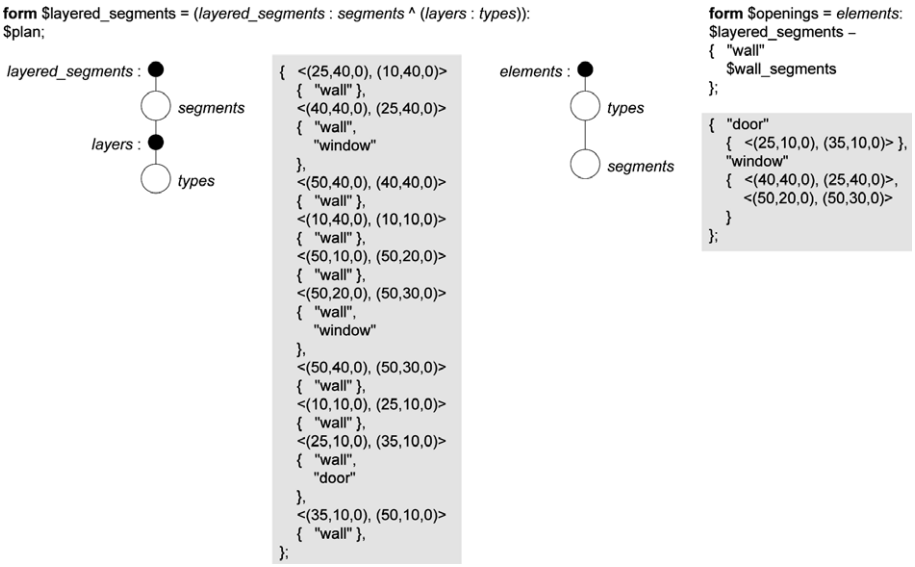

Fig. 10. Two sortal structures consisting of line segments and layer (type) labels: (left) segments with layers as attributes and (right) types with segments as attributes. The second data form is constructed by subtracting the wall-layer segments from the first data form, thereby distinguishing the window-layer and door-layer segments as single-layer elements. The grey boxes contain the results of the respective description above them.

Consequently, a data entity is part of a disjunctive data form if it is a part of the partial data form of entities from the same component *sort*. In other words, data forms from different component *sorts*, under disjunction, never interact; the resulting data form is the set of forms from all component *sorts*. When the operations of addition, subtraction or product are applied to two data forms of the same disjunctive *sort*, the operation applies to the respective component forms.

6.2. Behaviors and data loss

Behaviors play an important role when assessing data loss in data exchange between different *sorts*. When reorganizing the composition of *sorts* under the attribute operator, the corresponding behavior may be altered in such a way as to trigger data loss. Consider a behavior for weights (e.g., line thicknesses or surface tones) as is apparent from drawings on paper—a single line drawn multiple times, each time with a different thickness, appears as if it were drawn once with the largest thickness, even though it assumes the same line with other thicknesses (see also [25]). When using numeric values to represent weights, the part relation on weights corresponds to the less-than-or-equal relation on numeric values;

$$x:\{m\} \wedge y:\{n\} \Rightarrow x \leqslant y \Longleftrightarrow m \leqslant n$$
$$x+y:\{\max(m,n)\}$$
$$x-y:\{\} \text{ if } m \leqslant n, \text{else } \{m\}$$
$$x \cdot y:\{\min(m,n)\}$$

Thus, weights combine into a single weight, with its value as the least upper bound of the respective individual weights, i.e., their maximum value. Similarly, the common value (intersection) of a collection of weights is the greatest lower bound of the individual weights, i.e., their minimum value. The result of subtracting one weight from another is either a weight that equals the numeric difference of their values or zero (i.e., no weight), and this depends on their relative values.

Now consider a *sort* of weighted entities, say line segments and cost values per unit length, i.e., a *sort* of line segments with attribute cost values:

cost_segments: *segments* ∧ *costs*

and a *sort* of entity weights or segment costs, i.e., a *sort* of cost values with attribute line segments:

segment_costs: *costs* ∧ *segments*

A collection of weighted segments defines a set of disjoint line segments, each having a single weight assigned (possibly the maximum value of various weights assigned to the same line segment). These weights (cost values) may be different for different (disjoint) line segments. The behavior of the collection is, in the first instance, the behavior for line segments. On the other hand, a collection of segment weights, which is defined as a single weight (which is the maximum of all weights considered) with an attribute collection of line segments, adheres, in the first instance, to the behavior for weights. In both cases, line segments are associated with weights, or, cost values. However, in the first case, different (disjoint) line segments may be associated with different cost values, whereas, in the second case, all line segments are associated with the same cost value. In a conversion from the first to the second *sort*, data loss is inevitable (Fig. 11).

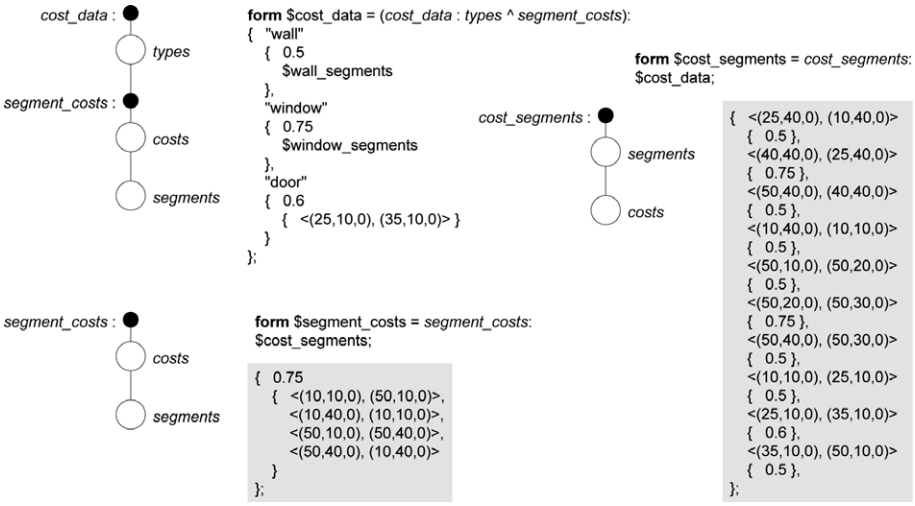

Fig. 11. Three sortal structures consisting of line segments and cost weights (and type labels): (left above) (types and) costs with segments as attributes, (right) segments with costs as attributes and (left below). In each conversion from one sortal structure to the next, data loss occurs due to the behavioral specification for weights. The grey boxes contain the results of the respective descriptions above them.

An understanding of when and where exact translation of data between different *sorts* or representations is or is not possible is important for assessing data integrity and controlling data flow [16], especially when considering a constructive approach to data representations. Data loss can easily be assessed under the subsumption relationship. If one *sort* subsumes another, exact translation is trivial from the part to the whole. If both *sorts* subsume a third, exact translation only applies to the data that can be said to belong to the third *sort*. When the subsumption relationship does not apply—as is the case in the example above—*sorts* can still be compared, based on their compositional structures, their behavioral specifications, and their naming [26]. If two *sorts* are structurally identical, exact translation of data is guaranteed, except for a possible loss of semantic identity. Otherwise, data loss depends also on their behavioral specifications, as in the example above.

7. Representational design queries

Computational design relies on effective data representations, not just for the specification of design artifacts, but also for querying the characteristics of such artifacts. With respect to geometry, Mäntylä [27] remarked that (geometric) representations must adequately answer "arbitrary geometric questions algorithmically". Even without the emphasis on geometric information, this remains as important today. However, computational design applications tend to focus on tools, operations and representations for the creation and manipulation of design artifacts. Techniques for querying receive less attention and are often constrained by the data representation that is adopted. Nevertheless, querying a design is as much an intricate aspect of the design process as is creation and manipulation.

Querying design information generally requires the analysis of existing information in order to derive new information that is not explicitly available in the information structure. Furthermore, a viable query language has to be based on a model for representing the different kinds of information that adhere to a consistent logic, providing access to information in a uniform and consistent manner, so that new queries can be easily constructed and posed, based on intent instead of availability. Stouffs and Krishnamurti [28] show how a query language for querying graphical design information can be built from basic operations of addition, subtraction and product (or intersection) and geometric relations—all defined as part of a representation for geometric entities augmented with attributes—and operations that are derived from techniques of data recognition and counting. For example, by augmenting networks of (utility) lines that are represented as volumes (or plane segments) with labels as attributes, and by combining these augmented geometries under the operation of addition, as defined for the representational model, colliding lines specifically result in geometries that have more than one label as attribute (for example, in Fig. 10(left), overlapping layered line segments result in multi-layer line segments, i.e., segments that have more than one layer label). These collisions can easily be counted, while the labels associated with each geometry identify the colliding lines, and the geometry itself specifies the location of the collision [28].

7.1. Emergence

Data recognition, or pattern matching, plays an important role in the specification of design queries. The part relationship underlying the behavioral specifications for a *sort* enables data recognition to be implemented for this *sort*. Since composite *sorts* inherit their behavior and part

relationship from their component *sorts*, any technical difficulties in implementing data recognition apply just once, for each primitive *sort*.

Under the part relationship that underlies the behavioral specification for a *sort*, any part of a data entity is itself a data entity of the same *sort*. Depending on the specific part relationship—thus, on the behavioral specification—an entity potentially specifies an indefinite number of (sub)entities, each part of the original. For example, considering the interval behavior for line segments, any part of a line segment is a line segment and we can deal with line segments in indeterminate ways—that is, line segments emerge (Fig. 12; also, in Fig. 10 (left), wall segments emerge that lie on either side of a window or door segment). Emergent information is the result of a new interpretation of existing information and relies on a restructuring of the information that is not yet captured in the current information structure. Creativity, as an activity in the design process, relies on emergent information, for example, when the design provides new insights that lead to a new interpretation of constituent design entities. Supporting emergence computationally provides the designer with the freedom to reinterpret a design in any way, and have this interpretation sustained by the system.

Recognizing emergent information structures requires determining a transformation under which a specified structure or pattern is similar to a part of the original information structure [29]. Which kinds of transformations are permitted depends on the kind of information being recognized. For example, in spatial recognition, the transformations are, commonly, Euclidean: a square must be computationally recognized as a square, irrespective of scale, orientation or location. Similarly, transformations can be considered for other kinds of information, for example, search-and-replace in text editing allows for case transformations of the constituent letters.

Recognizing emergent information is useful, especially, when the emergent information is subsequently the subject of an operation or manipulation, as in situations that rely on a restructuring of emergent information. Data recognition and subsequent manipulation can be considered part of a single computation $s - f(a) + f(b)$, where s is a data form, a is a representation of the data pattern, f is a transformation under which a is a part of s, and $f(b)$ is the data replacing $f(a)$ in s. Krishnamurti and Stouffs [30] discuss $s - f(a) + f(b)$ as a computational expression of spatial change, derived from a *shape rule* $a \rightarrow b$. Rule application, then, consists of replacing the emergent data corresponding to a, under some allowable transformation, by b, under the same transformation.

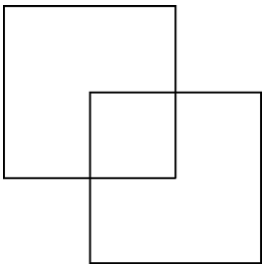

Fig. 12. Two or three squares: if the figure is constructed from two squares specified as a collection of eight line segments, then the third square can be recognized and its line segments will emerge under the part relationship.

Formally and technically, rules can be grouped as a device for specifying a language, namely, the set of all designs generated from an initial design, employing the rules to arrive at the design. In spatial design, the specification of rules leads naturally to the generation and exploration of possible spatial designs. Both Mitchell [31] and Stiny [32], and more recently, in a special issue on design spaces [33], some authors have remarked on the importance of emergent spatial elements and/or rules in design search. Stouffs and Krishnamurti [34] present a few examples of generational design formalisms that can be expressed using *sorts*. These are all spatial formalisms, emphasizing spatial compositions of geometric and attribute information. Stiny [35] perceptively notes that the spatial rules within these formalisms, used to compose spatial elements in designs, often go hand in hand with informal, verbal descriptions providing main details of the functional elements comprising designs. He suggests the specification of a descriptive formalism that, explicitly, recursively constructs the intended descriptions of designs, thereby augmenting the spatial formalism. Duarte [36] presents such a descriptive formalism for generating housing briefs. Such developments not only broaden the application domain for rule-based formalisms in design, but also accentuate the role *sorts* can play in offering a component-based approach for building rule-based design systems, utilizing a uniform characterization of such formalisms.

7.2. Data functions

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 the data constructs. Data functions are assigned to apply to one or more selected property attributes[2] of selected *sorts*, each of which may, itself, be a data function. The resulting value of the data function is then computed from the values of the respective property attributes of valid compositions of data entities of these *sorts*. A composition of data entities is a valid composition if the data entities are encountered along the same relational path, with some restrictions. Therefore, the target *sorts* must be related to the data function's *sorts* within the representational structure under a sequence of one or more constructive relationships. The resulting value is automatically recomputed each time the data structure is

[2] When *sorts* are considered as class structures, the property attributes correspond to class methods.

traversed, e.g., when visualizing the structure. As a data kind, data functions specify a functional description, a result value, and one or more *sorts* and their respective property attributes.

Data functions can introduce specific behaviors and functionalities into representational structures, for the purpose of counting or other numerical or geometric operations. Consider, for example, a *sort* of building elements, e.g., a composition of line segments and type labels as illustrated in Fig. 2 (left). By augmenting the corresponding data form with a counting function applied to any property attribute of the line segments, the value of this function is then automatically computed as the number of all line segments. For example, the function "count(*segments*.length)" applies to the length property attribute of the *sort segments* and adds one to the result value for each encounter of a valid data entity, i.e., a line segment (Fig. 13 (left)). The resulting *sort*, denoted *total_count*, is a composition of a *sort* of functions, *functions*, and the *sort elements* under the attribute operator:

total_count: (*functions*: [*Function*]) ∧ *elements*

The scope of the function—that is, the *sorts'* data entities that relate to the function under a sequence of one or more constructive relationships—is dependent on the position of the function, or its *sort*, in the representational structure. Moving the data function in the data construct—e.g., by altering the compositional structure of the representation—may alter the scope of the function and, thereby, its result. In this way, altering the representational structure can be used as a technique for altering design queries.

In the example above, the result is only an overall count; if we wish to count the number of segments per type, we only need to switch the order of the *sorts functions* and *types* under the attribute operator:

count_per_type: *types* ∧ *functions* ∧ *segments*

The specified function now becomes an attribute to each type label, so that the number of segments is calculated not once but three times, once for each type. The scope of each function is limited to the line segments that are part of the attribute form of this function. Thus, the resulting data form consists of a collection of (three) type labels, each of which has as attribute the function "count(*segments*.length)" with as value the number of line segments that are an attribute to this function (Fig. 13 (right)).

Rather than counting the respective numbers of line segments, we can also calculate an overall cost for the walls, doors and windows, based on a specification of the cost per unit length. Two modifications are required for this. Firstly, a cost value needs to be added as an attribute to each type:

total_cost: *functions* ∧ *types* ∧ *costs* ∧ *segments*

Secondly, the count function needs to be replaced by an inner product function applying to the numeric value property attribute of the *sort costs* and the length property attribute of the *sort segments*: "sum|product(*costs*.value, *segments*.length)". The data form that results consists of this function and an attribute form containing the three labels. In turn, each label has a cost value as attribute and this cost value has the respective line segments as an attribute form (Fig. 14(left)). Again, by switching the order of the *sorts functions* and *types*, we calculate the cost per type (Fig. 14 (right)):

cost_per_type: *types* ∧ *functions* ∧ *costs* ∧ *segments*

Finally, we consider other functions for other purposes. For example, we can calculate the distance from a certain point within the room to each of the elements. First, we calculate the average distance from this point to each of the elements, using an average over distance function that is applied to the position vector of this point and the end position vectors of each line segment, "avg|dist2lnseg(*points*.

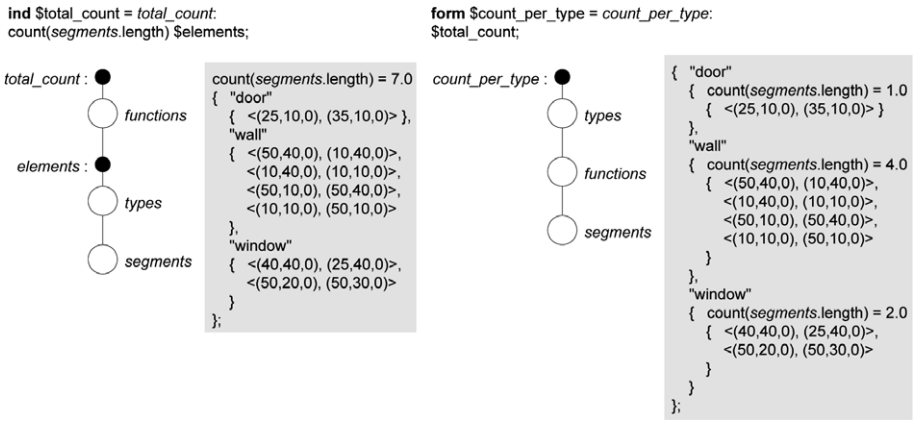

Fig. 13. Two sortal structures consisting of line segments, type labels and counting function(s): (left) one overall function and (right) one function per type. The grey boxes contain the results of the respective descriptions above them.

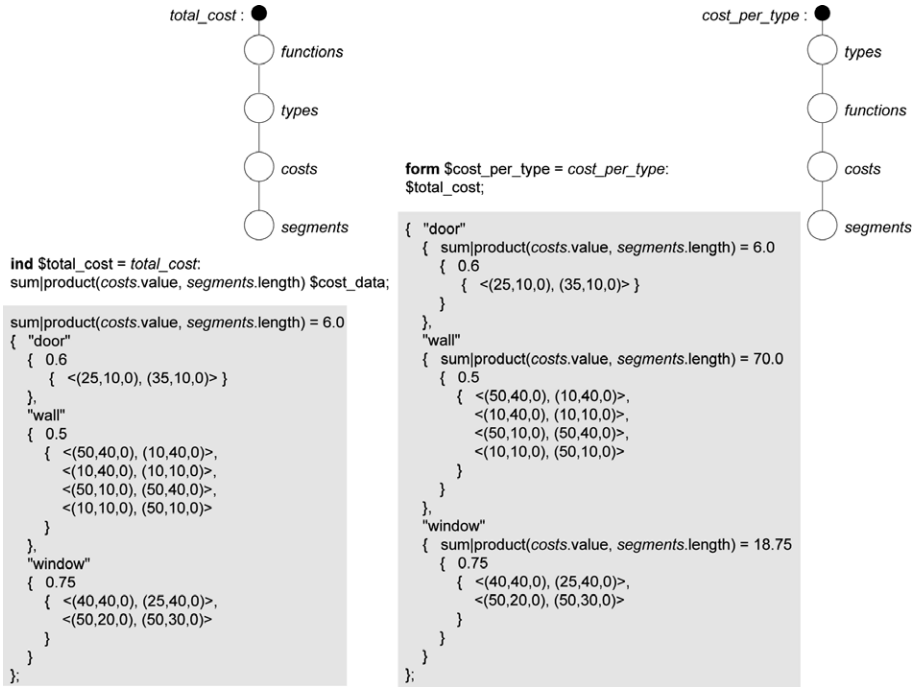

Fig. 14. Two sortal structures consisting of line segments, type labels, cost values and inner product function(s): (left) one overall function and (right) one function per type. The grey boxes contain the results of the respective descriptions above them.

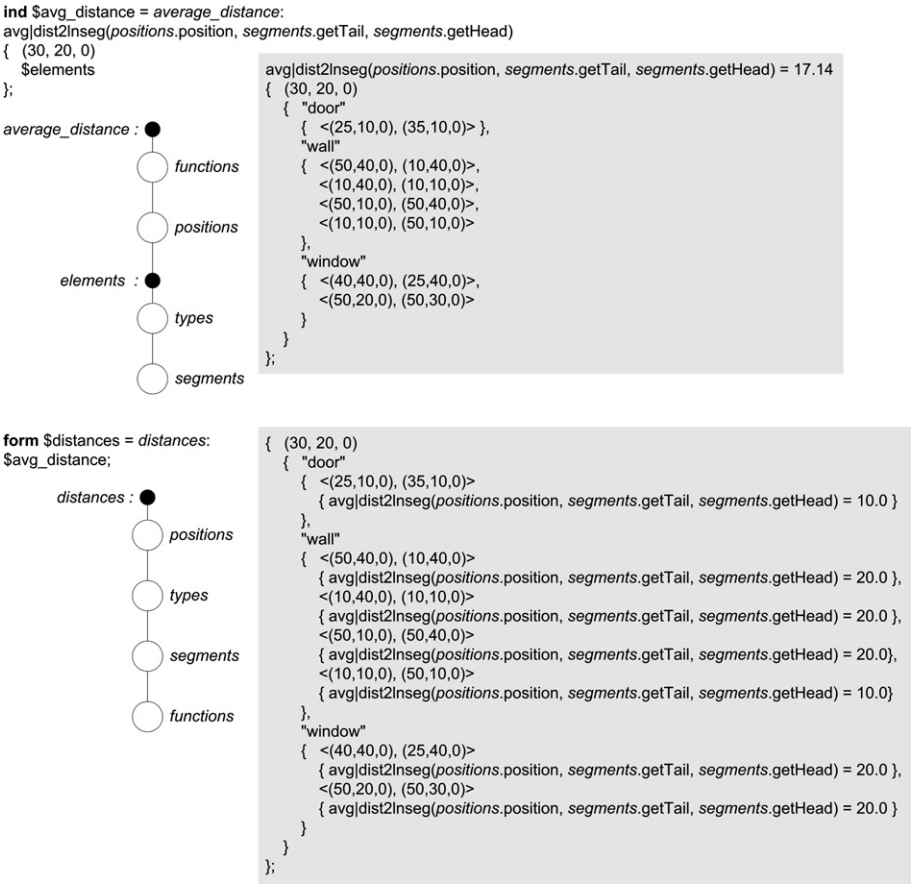

Fig. 15. Two sortal structures consisting of line segments, type labels, a position and average distance function(s): (above) one overall function and (below) one function per type. The grey boxes contain the results of the respective descriptions on the left.

position, *segments*.getTail, *segments*.getHead)'' (Fig. 15 (above)):

average_distance: *functions* ∧ (*positions*: [*Point*]) ∧ *elements*

Then, in order to determine the distance to each element individually, we make the *sort functions* an attribute to the *sort segments*:

distances: *positions* ∧ *types* ∧ *segments* ∧ *functions*

When the *sort functions* becomes an attribute of the *sort segments*, then, each segment has the function as an attribute, such that the scope of each function is limited to this one line segment, that is, it only calculates the distance to this one line segment (Fig. 15 (below)).

If we not only want the individual distances, but also the minimum overall distance, we can add another function, ''min(*functions*.value)'', that applies to the numeric value property attribute of the other functions and calculates the minimum value (Fig. 16 (above)):

minimum_distance: *functions* ∧ *positions* ∧ *types* ∧ *segments* ∧ *functions*

In this *sort*, the *sort functions* appears twice, once to hold the minimum value function and once to hold the (average over) distance function. Alternatively, we can calculate the minimum distance per type, e.g., for emergency safety reasons, by reordering the attribute relationships such that the minimum value function appears once per type (Fig. 16 (below)):

safety_distance: *positions* ∧ *types* ∧ *functions* ∧ *segments* ∧ *functions*

8. Discussion

Sorts present an algebraic formalism for constructing design representations. In this paper, we consider *sorts* to support the specification and use of multiple and evolving representations in the context of building design. There-

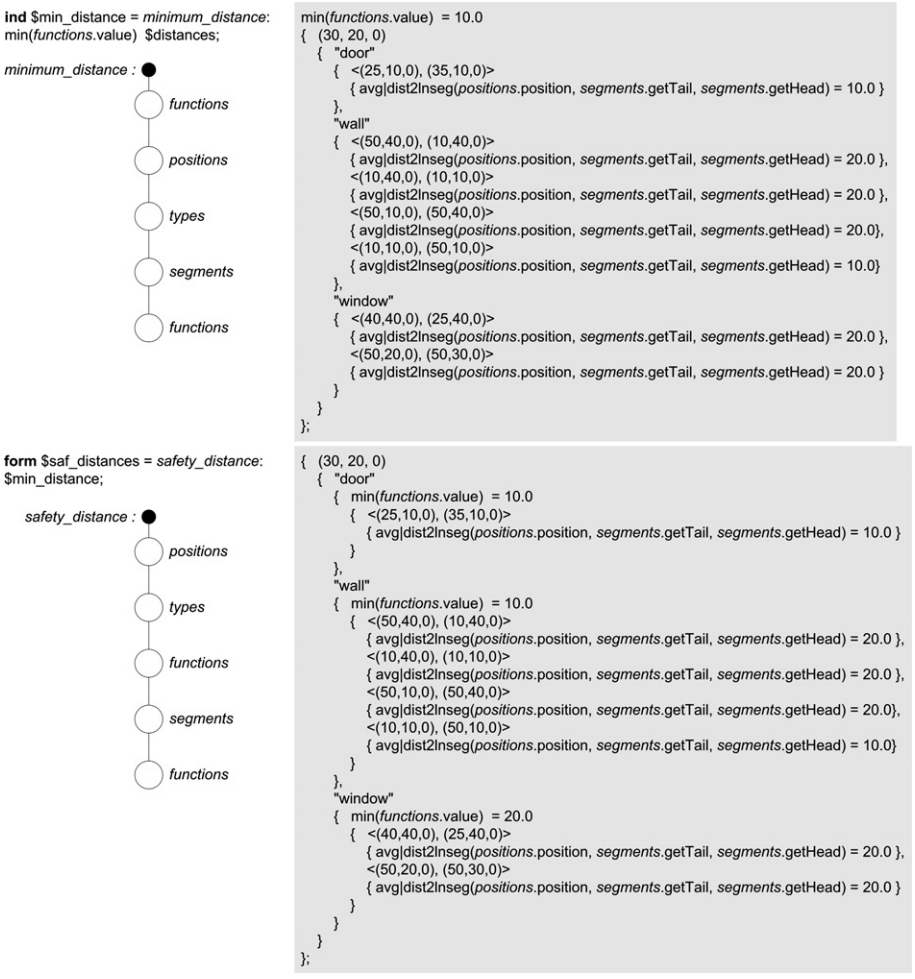

Fig. 16. Two sortal structures consisting of line segments, type labels, a position and average distance function(s): (above) one overall function and (below) one function per type. The grey boxes contain the results of the respective descriptions on the left.

fore, it is natural to compare *sorts* with existing efforts in developing integrated product models. In the process of establishing a product model, two steps—*semantic* and *syntactic*—can be considered. The former refers to the conceptual development, the latter to the translation of this conceptualization into a representational structure for practical use. For example, the ISO STEP [37] standardization effort includes both the definition of semantic information models and the development of the EXPRESS modeling language for generating product models. Some efforts or developments focus primarily on one of these steps. For example, the LexiCon [38] effort is mainly concerned with a formal vocabulary for the storage and exchange of information in the construction industry. On the other hand, the Product Information Specification (PIS) [39] framework solely defines a grammar for constructing product information. In its application to the precast concrete industry, the authors consider a basic lexicon of 2500 distinct terms that through application of the framework can lead to over 30,000 information items.

A sortal representation is a composition of *sorts* that can easily be modified by adding and removing *sorts* or by altering the constructive relationships, and that can be given a name. As such, *sorts* consider a purely syntactic approach. Whereas the PIS framework considers the criterion of consistency between terms to prevent lexical and structural ambiguities in the definition of terms [39], any *sort* can have any meaning (or name) assigned, without any constraints. Synonyms, i.e., the use of different terms, or *sorts*, to refer to the same concept, are absolutely possible. To some extent (through context/author identification), *sorts* even allow for homonyms, i.e., the use of a single term to refer to different concepts. The notion behind this is that ambiguity can be useful in order to express varying views, for example, mirroring reality, or because of changing views over time. Obviously, ambiguity is only useful if it can be dealt with, that is, if the means are available to identify and resolve ambiguity when necessary. The subsumption relationship over *sorts*, in combination with a behavioral specification of *sorts*, allows sortal representations to be compared and related with respect to scope and coverage, sortal structures to be converted automatically according to such relationships, and data loss to be assessed when converting data from one representation to another. In this way, ambiguity can be dealt with in *sorts*.

Product-modeling efforts can also be distinguished by their objectives. In principle, the major purposes for developing product models are exchanging product data and integrating project/product lifecycle information [39]. However, considering the various phases in building and construction processes and the various disciplines and knowledge domains that are involved, product-modeling initiatives that attempt to describe the entire building throughout its entire lifecycle, from conception to demolition, are huge endeavors that require immense resources. Even if partially successful [3], they leave room for alternative efforts that target specific phases, knowledge domains and/or interactions. As an example, Object Trees [40] constitute an approach primarily aimed at the construction-planning phase of large-scale construction projects, and offer a methodology for developing representational object trees that correspond to concept hierarchies of construction aspects, elements and their attributes.

Sorts emphasize the flexibility to construct various representational structures for varying purposes, over time or considering alternative point of views. Throughout this paper, we have presented numerous examples of *sorts* and sortal structures that were derived one from another, for representing, querying and deriving different yet related data sets. Fig. 17 shows a subset of these sortal structures, and corresponding *sorts*, with their derivational relationships.

Sorts are not aimed at replacing integrated product models, but only at complementing them. Whereas product models primarily pertain to technical product data, building design information also concerns abstract concepts such as space and functions, user and user activities, design intent and rationale, and other related information and documents, including images, movies, sounds, reports, etc. In this respect, the strength of *sorts*, firstly,

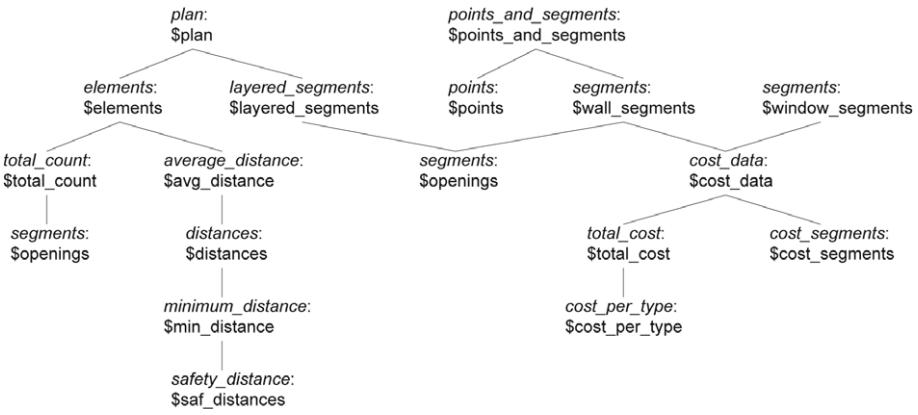

Fig. 17. A network of sortal structures, and corresponding *sorts*, and their derivations (from top to bottom).

lies in the uniform model for dealing with all kinds of information and data, based on a behavioral specification for *sorts*. Secondly, the formal framework for constructing sorts facilitates the comparison and relation of different *sorts* with respect to scope and coverage, and the assessment of data loss when converting data from one sortal representation to another. The behavioral specification for *sorts* also plays a role in assessing data loss. That the part relationship underlying this behavioral specification also facilitates data recognition (or emergence) is an added bonus. Together, the uniform model, the part relationship underlying the behavioral specification, and the additional integration of functional behavior into sortal constructs, support the specification of a query language and the development and implementation of generational formalisms.

A first practical application of *sorts* was developed in the context of a project building a toolset for the virtual AEC industry, in particular, integrating a numerical constraint solver in a document-centric collaboration environment [41]. The application considered design information for a steel-framed building project, especially, the dimensioning of holes in steel beams (Fig. 18), in the form of design constraints and related information [41,42]. This information consists, minimally, of a set of authors, a set of constraints for each author, and a common set of variables with each variable linked to the constraints defined over this variable. An organization of the design information by kind, i.e., constraints, variables, authors, and other data, with entities linked as appropriate, presents a straightforward and efficient way of storing this information into a relational database (Fig. 19(above)). In order to effectively 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 (Fig. 19(below)). Other information entities can be made accessible from these, thereby clarifying each constraint's context and role in the design. Specifically, each constraint specifies the variables affected; each variable, in turn, specifies the constraints from other authors that are defined over this variable; and each of these constraints specifies its author. Links to other data can be additionally provided. A VRML visualization of these sortal structures 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 [42].

Krishnamurti and Park [21] investigate the use of sorts in the context of building construction, within a larger project that "investigates ways of integrating suites of emerging evaluation technologies to help find, record, manage, and limit the impact of construction defects." The project considers an Integrated Project Model (IPM) that is continuously updated to reflect on both the as-designed and as-built building models. The as-designed model is obtained as an IFC [4] file from a commercial parametric design software, and contains geometric information and some attribute data. Laser scanners provide accurate 3D geometric as-built information. The laser scanned data is processed into collections of surfaces that are matched onto surfaces in the as-design model. Inconsistencies are recorded as defect entities, connecting to both as-built and as-designed entities. Embedded sensors provide frequent quality related information. Sensor data are also attached to related as-designed entities.

Sorts are adopted to provide the flexibility to generate multiple views that serve the purposes of the different collaborators in the project. These views include both pre-defined and user-defined views. From a general view of the IPM, that is, a dynamic graph representing the IPM and its component connectivity, predefined sortal description views of the model can be generated that are participant-specific. In addition, dynamically defined views can be generated by combining components in the general view with specific functions (for example, volume calculation, face generation, etc.) [21]. As an example, Park and Krishnamurti consider the use of *sorts* to generate different information views of a target object. "The embedded sensor planner needs geometric information, material type, and construction method of the target object. On the other hand, the laser scan technician needs the target region information, and target object geometry and location" [21] (Fig. 20). In the view of the embedded sensor planner, an object has a shape (specifying the geometric information) and a collection of material types, and each material type has one or more location values, e.g.

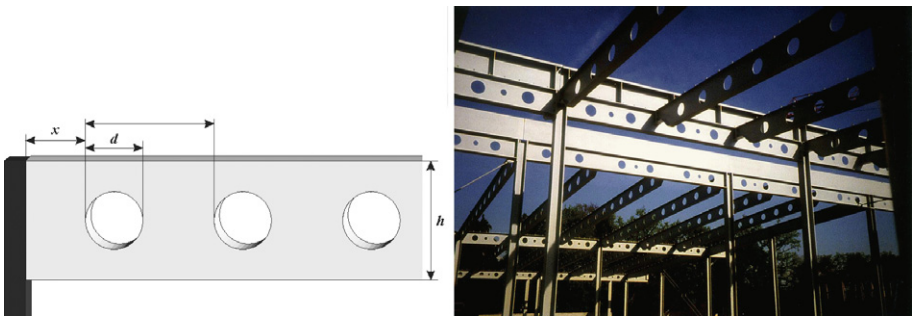

Fig. 18. Design problem from a building project: the dimensioning of holes in steel beams.

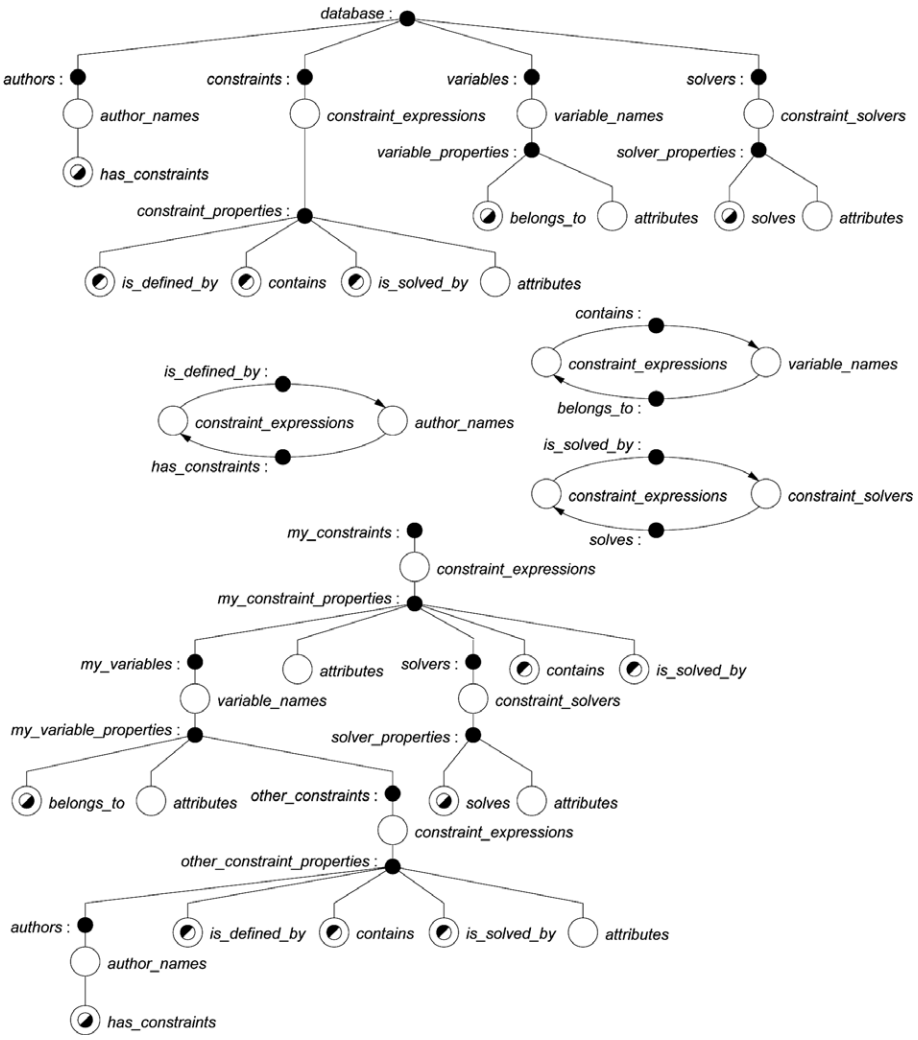

Fig. 19. *Sorts* depicting a database view (above) and a user's view (below), representing design information in the form of design constraints and related information. *Property sorts* (middle) serve to represent the various links between the constraint expressions and, respectively, the author names, variable names, and constraint solvers.

embeddedSensor_target_slabs:
slabs ∧ (*shapes* + *material_types* ∧ *locations*)

The laser scanning technician's view, on the other hand, considers a single location for the target object instead of multiple material locations, e.g.

laserScan_target_slabs:
slabs ∧ (*shapes* + *locations* ∧ *material_types*)

We have stated that the construction of a design representation can be the result of correspondence that forms part of the design process. This naturally raises the question how such correspondence can be facilitated, especially when representational structures become large. First, we consider the ability to conceptualize representational structures. Specifically, we are considering the definition of *sorts* as the specification of a concept hierarchy that, subsequently, can be detailed into a representational structure consisting of primitive *sorts* and con- structive relationships. By separating the specification of the representational semantics (the names of the structures and their hierarchical relationships) from the specification of the nuts and bolts (the data types and their behaviors, and the distinction between disjunctively compositional and attribute relationships) we aim to ease a conceptualization of the intended representational structures that facilitates their development (as reflected in the graphical representation of *sorts*, e.g., depicted in Fig. 19). Secondly, we consider the need for effective (graphical) visualizations of (parts of) sortal structures that can offer the user insight into these structures. In particular, we are implementing dynamic visualizations of these structures with variable focus and level of detail [21]. Lastly, facilitating correspondence can only be achieved when taking into account the practical context, e.g., the design process, in which it is expected to take place. Additional, larger-scale examples are being considered. These will need to prove the scalability of the correspondence approach.

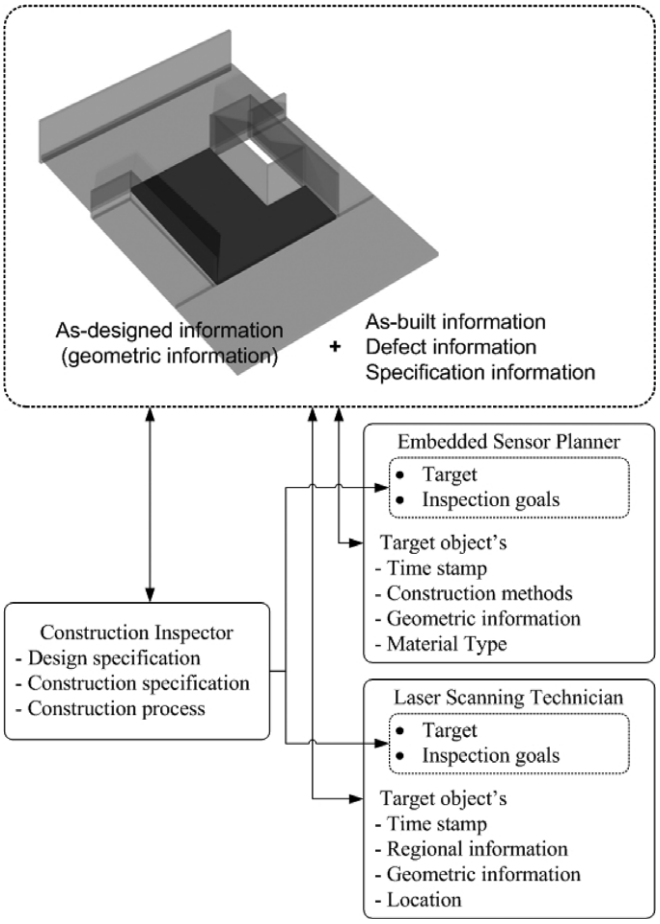

Fig. 20. Different representational needs on a slab (Ramesh Krishnamurti and Kuhn Park).

9. Conclusion

Supporting the early phases of design requires, among others, support for the specification and use of multiple and evolving representations, and for the exchange of information between these representations. A complex adaptive system can serve as a model for the development of design representations, considering design representations as complex structures of properties and constructors. Agreement on design representations can be achieved by correspondence on the construction of these structures and the naming of properties and substructures. *Sorts* present a semi-constructive algebraic formalism for design representations that supports this approach. Like many other representational formalisms, *sorts* consider a subsumption relationship that enables *sorts* to be compared with respect to scope and coverage. Unlike most other representational formalism that consider a subsumption relationship, *sorts* are concerned with the representation of data (present and emergent), not knowledge. As such, the definition of a *sort* includes a behavioral specification, governing the operational behavior of data entities and data forms under the common operations of addition, subtraction and product (or intersection). This behavioral specification plays an important role when assessing data loss in data exchange between different *sorts*. The part relationship underlying this behavioral specification also enables emergence and the implementation of data recognition. *Sorts* further consider data functions as representational entities, offering functional behavior integrated into data constructs. Data recognition, data functions and common operations (of addition, subtraction and product) and relationships can form a basis for the specification of a query language for design information. Altering the representational structure additionally serves as a technique for altering design queries.

Acknowledgements

The author wishes to thank Ramesh Krishnamurti for his collaboration on this research. This paper extends on a paper presented at the 13th International Workshop of EG-ICE on Intelligent Computing in Engineering and Architecture [43], at Ascona, Switzerland, 25–30 June 2006. The author thanks the reviewers and audience of this workshop, as well as the journal reviewers, for their constructive comments.

References

[1] B. Kolarevic, Digital morphogenesis, in: B. Kolarevic (Ed.), Architecture in the Digital Age: Designing and Manufacturing, Taylor and Francis, Oxon, UK, 2003, pp. 12–28.
[2] C.M. Eastman, On the analysis of intuitive design processes, in: G.T. Moore (Ed.), Emerging Methods in Environmental Design and Planning, MIT, Cambridge, MA, 1970, pp. 21–37.
[3] A. Kiviniemi, Ten years of IFC development—why are we not yet there? in: Keynote lecture at the Joint International Conference on Computing and Decision Making in Civil and Building Engineering, Montreal, 14–16 June 2006.
[4] AIA Model Support Group, IFC2x Edition 3. International Alliance for Interoperability, 2006, Available from: <http://www.iai-international.org/Model/R2x3_final/index.htm>.
[5] F. Manola, E. Miller, (Eds.), RDF Primer. W3C World Wide Web Consortium, 2004, Available from: <http://www.w3.org/TR/rdf-primer/>.
[6] R. Stouffs, R. Krishnamurti, On the road to standardization, in: B. de Vries, J. van Leeuwen, H. Achten (Eds.), Computer Aided Architectural Design Futures 2001, Kluwer Academic, Dordrecht, The Netherlands, 2001, pp. 75–88.
[7] B. Latour, Drawing things together, in: M. Lynch, S. Woolgar (Eds.), Representation in Scientific Practice, MIT, Cambridge, MA, 1990, pp. 19–68.
[8] C. Eastman, New directions in design cognition: studies of representation and recall, in: C. Eastman, M. McCracken, W. Newstetter (Eds.), Design Knowing and Learning: Cognition in Design Education, Elsevier, Amsterdam, 2001, pp. 147–198.
[9] Ö. Akın, "Simon says": design is representation, Arredamento, July 2001, Available from: <http://www.andrew.cmu.edu/user/oa04/Papers/AradSimon.pdf>.
[10] B. Tunçer, Z. Al-Ars, E. Akar, J. Beintema, S. Sariyildiz, DesignMap: capturing design knowledge in architectural practice, in: M. Dulaimi, (Ed.), Conference Proceedings of the Joint International Conference on Construction Culture, Innovation and Management, BUiD, Dubai, pp. 352–361.
[11] J.H. Holland, Complex adaptive systems, Daedalus 121 (1) (1992) 17–30.

[12] K.J. Dooley, A complex adaptive systems model of organization change, Nonlinear Dynamics, Psychology, and Life Sciences 1 (1997) 69–97.
[13] L. Monostori, K. Ueda, Design of complex adaptive systems: introduction, Advanced Engineering Informatics 20 (3) (2006) 223–225.
[14] I. Prigogine, I. Stengers, Order out of Chaos, Bantam Books, New York, 1984.
[15] J. Kooistra, Flowing, Systems Research and Behavioral Science 19 (2002) 123–127.
[16] R. Stouffs, R. Krishnamurti, C.M. Eastman, A formal structure for nonequivalent solid representations, in: S. Finger, M. Mäntylä, T. Tomiyama (Eds.), Proceedings of IFIP WG 5.2 Workshop on Knowledge Intensive CAD II. IFIP WG 5.2, Pittsburgh, PA, 1996, pp. 269–289.
[17] J.P. van Leeuwen, S. Fridqvist, Supporting collaborative design by type recognition and knowledge sharing, ITcon 7 (2002) 167–181, Available from: <http://www.itcon.org/2002/11/>.
[18] R. Woodbury, A. Burrow, S. Datta, T. Chang, Typed feature structures and design space exploration, Artificial Intelligence in Design, Engineering and Manufacturing 13 (1999) 287–302.
[19] H. Äit-Kaci, A lattice theoretic approach to computation based on a calculus of partially ordered type structures (property inheritance, semantic nets, graph unification), Ph.D. Diss., University of Pennsylvania, Philadelphia, PA, 1984.
[20] S. Datta, Modeling dialogue with mixed initiative in design space exploration, Artificial Intelligence for Engineering Design, Analysis and Manufacturing 20 (2006) 129–142.
[21] R. Stouffs, R. Krishnamurti, K. Park, Sortal structures: supporting representational flexibility for building domain processes, Computer-Aided Civil and Infrastructure Engineering 22 (2007) 98–116.
[22] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, P. Patel-Schneider, The Description Logic Handbook: Theory, Implementation and Applications, Cambridge University Press, Cambridge, 2003.
[23] R. Stouffs, R. Krishnamurti, The extensibility and applicability of geometric representations, in: Architecture Proceedings of 3rd Design and Decision Support Systems in Architecture and Urban Planning Conference, Eindhoven University of Technology, Eindhoven, The Netherlands, 1996, pp. 436–452.
[24] R. Krishnamurti, R. Stouffs, The boundary of a shape and its classification, Journal of Design Research 4 (1) (2004).
[25] G. Stiny, Weights, Environment and Planning B: Planning and Design 19 (1992) 413–430.
[26] R. Stouffs, R. Krishnamurti, M. Cumming, Mapping design information by manipulating representational structures, in: Ö. Akın, R. Krishnamurti, K.P. Lam (Eds.), Generative CAD Systems, School of Architecture, Carnegie Mellon University, Pittsburgh, PA, 2004, pp. 387–400.
[27] M. Mäntylä, An Introduction to Solid Modeling, Computer Science Press, Rockville, MD, 1988.
[28] R. Stouffs, R. Krishnamurti, On a query language for weighted geometries, in: O. Moselhi, C. Bedard, S. Alkass (Eds.), Third Canadian Conference on Computing in Civil and Building Engineering, Canadian Society for Civil Engineering, Montreal, 1996, pp. 783–793.
[29] R. Krishnamurti, C.F. Earl, Shape recognition in three dimensions, Environment and Planning B: Planning and Design 19 (5) (1992) 585–603.
[30] R. Krishnamurti, R. Stouffs, Spatial change: continuity, reversibility and emergent shapes, Environment and Planning B: Planning and Design 24 (3) (1997) 359–384.
[31] W.J. Mitchell, A computational view of design creativity, in: J.S. Gero, M.L. Maher (Eds.), Modeling Creativity and Knowledge-Based Creative Design, Lawrence Erlbaum Associates, Hillsdale, NJ, 1993, pp. 25–42.
[32] G. Stiny, Shape rules: closure, continuity, and emergence, Environment and Planning B: Planning and Design 21 (1994) s49–s78.
[33] R. Stouffs (Ed.), Design Spaces: the Explicit Representation of Spaces of Alternatives, Special Issue of AIEDAM 20(2) (2006).
[34] R. Stouffs, R. Krishnamurti, Sortal grammars as a framework for exploring grammar formalisms, in: M. Burry, S. Datta, A. Dawson, J. Rollo (Eds.), Mathematics and Design 2001, The School of Architecture and Building, Deakin University, Geelong, Australia, 2001, pp. 261–269.
[35] G. Stiny, A note on the description of designs, Environment and Planning B: Planning and Design 8 (1981) 257–267.
[36] J. Duarte, A discursive grammar for customizing mass housing: the case of Siza's houses at Malagueira, Automation in Construction 14 (2) (2005) 265–275.
[37] ISO TC 184/SC 4, ISO 10303, Industrial automation systems and integration—product data representation and exchange, International Organization for Standardization, 1994.
[38] K. Woestenenk, A common construction vocabulary, in: R. Amor (Ed.), Product and Process Modelling in the Building Industry, Building Research Establishment, Watford, England, 1998, pp. 561–568.
[39] G. Lee, C.M. Eastman, R. Sacks, S.B. Navathe, Grammatical rules for specifying information for automated product data modeling, Advanced Engineering Informatics 20 (2006) 155–170.
[40] G.A. van Nederveen, Object trees: improving electronic communication between participants of different disciplines in large-scale construction projects, Ph.D. Diss., Delft University of Technology, Delft, The Netherlands, 2000.
[41] C. Lottaz, R. Stouffs, I. Smith, Increasing understanding during collaboration through advanced representations, ITcon 5 (2000) 1–25. Available from: http://www.itcon.org/2000/1/.
[42] R. Stouffs, R. Krishnamurti, Representational flexibility for design, in: J.S. Gero (Ed.), Artificial Intelligence in Design'02, Kluwer Academic, Dordrecht, The Netherlands, 2002, pp. 105–128.
[43] I.F.C. Smith (Ed.), Intelligent Computing in Engineering and Architecture, Lecture Notes in Computer Science 4200, Springer, 2006.

Rudi Stouffs is associate professor of Design Informatics at the Department of Building Technology, Faculty of Architecture, Delft University of Technology. He holds an M.Sc. in architectural engineering from the Vrije Universiteit Brussel, an M.Sc. in computational design and a Ph.D. in architecture from Carnegie Mellon University (CMU). He has been assistant professor at the Department of Architecture at CMU and research coordinator at the Chair for Architecture and CAAD at ETH Zurich. His research interests include computational issues of description, modeling and representation for design in the areas of information exchange, collaboration, shape recognition and generation, geometric modeling, and visualization. He has been principal investigator of the Dynamic Digital Design Representations project funded by the Netherlands Organization for Scientific Research (NWO) and has participated in research projects in the areas of document management funded by the Swiss National Science Foundation and robotic manipulation funded by the Japan Research Institute.