Constructing Design Representations

Rudi Stouffs and Albert ter Haar

Design Informatics Chair, Dept. of Building Technology, Faculty of Architecture, Delft University of Technology, Berlageweg 1, 2628 CR Delft, The Netherlands
{r.m.f.stouffs, a.terhaar}@tudelft.nl

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.

1 Introduction

Design is 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, and each of these views, in turn, requires a different representation of the design entity. Even within the same task and for the same person, various representations may serve different purposes defined within the problem context and the selected approach. Especially in the early phases of design, the exploratory and dynamic nature of the design process invites a variety of approaches and representations, and any particular representation may be as much an outcome of as a means to the design process. Therefore, supporting the designer in these early phases requires, among others, support for the specification and use of multiple and evolving representations, and for the exchange of information between these representations.

Various modeling schemes for defining product models and ontologies exist (e.g., [1], [8]). These allow for the development of representations in support of different disciplines or methodologies, and enable information exchange between representations and collaboration across disciplines. However, they still require an a priori effort 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. We are particularly interested in providing 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, in support of creativity. Creativity, as an activity in the design process, relies on a restructuring of information that is not yet captured in a current information structure — that is, emergent information — for example, when the design provides new insights that lead to a new interpretation of constituent design entities.

I.F.C. Smith (Ed.): EG-ICE 2006, LNAI 4200, pp. 653 – 662, 2006.

This flexibility in using design representations necessitates a solution for dealing with the complexity of representations. For this purpose, we consider a complex adaptive system as a model for the development of design representations, in particular, its three key principles [5]. First, the outcome of the design process is generally *unpredictable*, as it is indeterminately related to the design requirements and the design process. Under the assumption that the design representation is an intricate part of the design outcome, this representation is necessarily also unpredictable. Secondly, the state or history of a design 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 [11]: "Order arises from complexity through self-organization." In the context of constructing a design representation, the process of self-organization can take on the form of human communication or correspondence, leading to an agreement on the representation that prevails in the system (see also [6]). 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).

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. Such an approach can benefit from a formal framework that allows for alternative 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. At the same time, the vocabulary of available properties and constructors defines the expressive power of the representational framework.

In this paper, we consider a semi-constructive algebraic formalism for design representations, termed *sorts* [15], as a candidate for supporting a constructive approach to design representations, and analyze *sorts* with respect to the requirements of a complex adaptive system. In particular, we consider the ability of *sorts* to support correspondence on design representations, to compare representations with respect to scope and coverage and detect data loss, and consider its potential to support the design process and the design outcome. Critical in this respect is the formal specification of *sortal* representations in terms of other *sortal* representations under formal compositional operations, the behavioral specification of *sortal* representations enabling the comparison of alternative representations and the detection of data loss, and the ability to integrate data functions into *sortal* representations. *Sorts* are also compared to other representational formalisms that consider a constructive approach to representations.

2 Incrementally Constructing Design Representations

Van Leeuwen et al. [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 the 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 an approach can greatly benefit from a formal framework that allows for representations to be compared and related, formally, in order to support translation and identify where exact translation is possible. 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 [2]. 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 [4] also presents a visual notation for representing design correspondence between designer and typed feature structures, using the concepts of mixed initiative and rational conversation.

3 A Subsumption Relationship over *sorts*

Sorts [15] provide a semi-constructive algebraic formulation of design representations 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 [16], where 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. 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*. We denote a *sort* identified by a single property as primitive and all other *sorts* as composite. A primitive *sort* necessarily has a name; a composite *sort* can also have a name assigned. Named *sorts* can be conceived to define object classes. Similarly to the property-oriented modeling approach [17], 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. For this purpose, we consider even property relationships and data functions (see Section 5) as properties, such that these can be dealt with in the same way.

Properties are composed using one or more constructors. We consider an *attribute* operator (denoted '^'), resulting in a conjunctively subordinate composition of properties, and an operation of *sum* (denoted '+'), resulting in a disjunctively co-ordinate composition. For example, a *sort* of colored labels may be defined as a composition of labels and colors under the attribute operator, such that each label has one (or more) colors assigned as attribute. On the other hand, a *sort* of points and line segments may be defined as a composition of points and line segments under the operation of sum; a resulting data entity can be either a point or a line segment. The operation of sum defines a subsumption relationship (denoted '≤') over *sorts*, as follows:

$$a \leq b \Leftrightarrow a + b = b \,. \tag{1}$$

The typed feature structures formalism, like most logic-based formalisms, links subsumption directly to information specificity, that is, a structure is subsumed by another, if this structure contains strictly more information than the other. One consequence of subsumption is that the absence of information in a design representation does not necessarily imply the absence of this information in the design, that is, representations are automatically considered to be incomplete. As a result, when searching for a design (representation) that satisfies certain information, less specific representations cannot automatically be excluded (e.g., [3]).

The subsumption relationship over *sorts* does not formally apply over the attribute operator. Though $a \wedge b$ is more information specific than either a or b, $a \wedge b$ is not subsumed by a or b (nor $a + b$), i.e., $(a \wedge b) + a + b \neq a + b$ — algebraically, the attribute operator corresponds to the Cartesian product operator; $a \wedge b$ is the *sort* of all proper 2-tuples of which the first member belongs to a and the second to b. In logic formalisms, a relational construct is used to represent such tuples. For example, in description logic [3], roles are defined as binary relationships between individuals. Consider a concept Label and a concept Color; the concept of colored labels can then

be represented as Label ∩ ∃hasAttribute.Color[1], denoting those labels that have an attribute that is a color. Here, ∩ denotes intersection and ∃R.C denotes full existential quantification with respect to role R and concept C. It follows that Label ∩ ∃hasAttribute.Color ⊆ Label; the concept of labels subsumes the concept of colored labels. A similar construct is not considered with respect to *sorts* — e.g., when looking for a yellow square, any square will not do, unless it has the yellow color assigned. In other words, logic-based models adhere to an open world — that is, nothing can be excluded unless it is explicitly excluded — *sorts*, on the other hand, adhere to a closed world, any reasoning is based purely on present or emergent (under a part relationship, see Section 4) information. *Sorts* only represent data; logic-based models essentially represent knowledge.

4 A Behavioral Specification for *sorts*

An important ingredient of *sorts* is behavioral specification. Behavioral specification is a prerequisite for the effective exchange of data between various representations. When an application receives data along with its behavioral specification, the application can 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 elements, become resolutely non-trivial — for instance, the addition of two numbers when these represent cardinal values (e.g., a number of columns that is 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 arithmetic operations of addition, subtraction, and product. It turns out that the more 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 [14].

The simplest specification of a part relationship corresponds to the subset relationship on mathematical sets. This part relationship particularly applies to points and labels, e.g., a point is part of another point only if the two 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. Then, operations of addition (combining elements), subtraction, and product (intersecting elements) correspond to set union, difference, and intersection, respectively.

Another kind of behavior arises when we consider the part relationship on line segments. A line segment is an interval on an infinite line carrier; in general, one-dimensional quantities such as time may be considered as intervals. An interval is a part of another interval if it is embedded in this interval; intervals on the same carrier that are adjacent or overlap combine into a single interval. Specifically, interval behavior can be expressed in terms of the behavior of the boundaries of intervals [7].

---

[1] Note that this syntax differs slightly from the syntax adopted by Baader et al. [3], 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.

Behaviors also apply to composite *sorts*, that is, a part relationship can be defined for its component data elements belonging to a composite *sort* defined under a conjunction (attribute operator) or disjunction. The composite *sort* inherits its behavior from its components in a manner that depends on the compositional relationship.

The disjunctive operator distinguishes all operand *sorts* such that each data element belongs explicitly to one of these *sorts*. Consequently, a data element is part of a disjunctive data collection if it is a part of the partial data collection of elements from the same component *sort*. In other words, data collections from different component *sorts*, under the disjunctive operator, never interact; the resulting data collection is the set of collections from all component *sorts*. When the operation of addition, subtraction or product is applied to two data collections of the same disjunctive *sort*, the operation instead applies to the respective component collections.

Under the attribute operator a data element is part of a data collection if it is a part of the data elements of the first component *sort*, and if it has an attribute collection that is a part of the respective attribute collection(s) of the data element(s) of the first component *sort* it is a part of. When data collections of the same composite *sort* (under the attribute operator) are pairwise summed (differenced or intersected), identical data elements merge, and their attribute collections combine, under this operation. Elements with empty attributes are removed.

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 [12] (e.g., line thickness or surface tones) as becomes 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 thickness. When using numeric values to represent weights, the part relation on weights corresponds to the less-than-or-equal relation on numeric values. Thus, weights can combine into a single weight, which has as its value the least upper bound of all the respective weight values, i.e., their maximum value. Similarly, the common value (intersection) of a collection of weights is the greatest lower bound of all 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 points, i.e., a *sort* of points with attribute weights, and a *sort* of pointed weights, i.e., a *sort* of weights with attribute points. A collection of weighted points defines a set of non-identical points, each having a single weight assigned (possibly the maximum value of various weights assigned to the same point). These weights may be different for different points. On the other hand, a collection of pointed weights is defined as a single weight (which is the maximum of all weights considered) with an attribute collection of points. In both cases, points are associated with weights. However, in the first case, different points may be associated with different weights, whereas, in the second case, all points are associated with the same weight. In a conversion from the first to the second *sort*, data loss is inevitable.

An understanding of when and where exact translation of data between different *sorts*, or representations, is possible, becomes important for assessing data integrity and controlling data flow [16]. 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 two *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 doesn't apply, such as under the attribute operator — as is the case in the examples above — *sorts* can still be compared, roughly, as equivalent, similar, convertible and incongruent [15]. Two *sorts* are said to be equivalent if these are semantically derived from the same *sort* — through renaming. Equivalent *sorts* are syntactically identical; this guarantees exact translation of data, except for a loss of semantic identity. Two *sorts* are denoted similar if these are similarly constructed from equivalent *sorts*. The similarity of *sorts* relies on the existence of a semi-canonical form of a composite *sort* as a disjunctive composition over *sorts*, each of which is either a primitive *sort* or composed of primitive *sorts* under the attribute operator [15]. Associative and distributive rules with respect to both compositional operators allow for a syntactical reduction of *sorts* to this semi-canonical form. If two *sorts* reduce to the same semi-canonical form, then these *sorts* are considered similar, and exact translation, except for a loss of semantic identity, applies. Otherwise, two *sorts* are either convertible or incongruent. If two *sorts* are convertible, data loss depends also on their behavioral specification, as in the examples above.

5 Functional Descriptions

The part relationship that underlies the behavioral specification 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*. Data recognition plays an important role in the specification of design queries. So does counting. Stouffs and Krishnamurti [13] indicate how a query language for querying graphical design information can be built from basic operations and geometric relations that are defined as part of a maximal element representation for weighted geometries, augmented with operations that are derived from techniques of counting and data recognition. For example, by augmenting networks of utility pipes, represented as volumes (or plane segments) with appropriate behavioral specification, with labels as attributes, and by combining these augmented geometries under the operation of sum, colliding pipes specifically result in geometries that have more than one label as attribute. These collisions can easily be counted, while the labels on each geometry identify the colliding pipes, and each geometry itself specifies the location of the collision [13].

In order to consider counting and other functional behavior as part of the representational approach, *sorts* consider data functions as a data kind, offering functional behavior integrated into data constructs. Data functions are assigned to apply to one or more selected *sorts* — specifically, they apply over tuples of data entities, one from each selected *sort*, where these data entities relate to the function under a sequence of one or more compositional relationships. Then, the result value of the data function is computed (iteratively) from the values of these tuples of data entities. The value of a data entity used in the computation is the actual value of the entity, such as its numeric value, or the position vector for a point, but may also be a derived value, such as the length of a line segment, or its direction vector. The data function's result value is automatically recomputed each time the data structure is traversed, e.g., when

visualizing the structure. As a data kind, data functions specify both a functional description, a result value, and one or more *sorts* and their respective value methods.

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 linear building elements, represented as line segments, with an attribute *sort* specifying the cost of each element per unit length. Then, by augmenting the corresponding data construct with a sum-over-product function applied to the numeric value of the cost entities and the length value of the linear elements, the value of this function is automatically computed as the total cost of all the building elements. As another example, consider a composite *sort* specifying both a reference point and a number of emergency exits represented as line segments. Then, a minimum-value function in combination with a function that computes the distance between a position vector and a line segment, specified by two end vectors, will yield the minimum distance from the reference point to any emergency exit.

Moving data functions in the data construct, by altering the compositional structure of the representation, alters the scope of the function — that is, the sorts' data entities that relate to the function under a sequence of one or more compositional relationships — and thereby its result. In this way, data functions can be used as a technique for querying design information, where moving the data function alters the query.

6 Discussion

Sorts present an algebraic formalism for constructing design representations. 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. A 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, and data loss to be assessed when converting data from one representation to another. Data functions can be integrated into data constructs in order to query design information. In this way, the design representation is an intricate part of the design outcome, and 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 through an application interface.

In developing such an application interface, we consider three aspects in particular; these are the ability to conceptualize representational structures, the need for effective visualizations of these structures and the embedding of the application in a practical context. First, we're 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 constructive 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. Secondly, we're exploring effective (graphical) visualizations of (parts of) the representational and data structures that can offer the user insight into these structures. In particular, we're implementing a dynamic visualization of these structures with variable focus and level of detail.

Thirdly, we're investigating practical applications of *sorts* in order to illustrate their strengths in practical contexts. Specifically, we're looking into the context of collaborative building design projects where a CAD program is used to express the design but where the design process also involves other information that is collected and stored in the form of documents. We're investigating the use of *sorts* to specify relationships between these documents and elements within the CAD model that help to organize this information. Given a *sort element_ids ^ element_descriptions* that reflects on (part of) the CAD model, the data for this *sort* can be automatically generated from the CAD data. This representation can then be extended using the *sort element_ids ^ (element_descriptions + document_references)* to represent CAD elements with associated document references. Using a graphical interface, the user can specify both the references and their associations to CAD elements. When the CAD model is changed, the data for *element_ids ^ element_descriptions* can be regenerated, while the data for *element_ids ^ document_references* can be retrieved from *element_ids ^ (element_descriptions + document_references)* using an automatic conversion based on the matching of both *sorts*. Since the first *sort* is subsumed by the second ('^' distributes over '+'), exact translation applies. Merging both data forms re-associates the document references to the CAD elements, on condition that the respective element IDs have not changed.

Park and Krishnamurti 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" [10]. 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 an IFC file obtained from a commercial parametric design software. Laser scanning provides accurate 3D geometric as-built information (e.g., component identity); similarly, embedded sensors provide frequent quality related information (e.g., thermal expansion)" [10]. *Sorts* are adopted to provide the flexibility to generate both pre-defined and user-defined views. For example, Park and Krishnamurti consider the use of sorts to generate different information views of a target object. "The embedded sensor planner needs the geometric information, location, material type, and construction method of the target object. On the other hand, the laser scanning technician needs two-dimensional geometric information of the target region, geometric information, and location of the target object." [9]

Sortal representations can complement a Building Information Model (BIM). Kuhn and Krishnamurti argue that "current methods of obtaining precise information from a BIM are cumbersome. Furthermore, it is computationally expensive to produce a representation from a BIM" [10]. We aim to put forward the concept of a *sortal* building model as an extension to a BIM, offering the user the means to build up design representations, in support of common or interdisciplinary views, and to use such representations for querying building information.

Acknowledgments

The first author wishes to thank Ramesh Krishnamurti for his collaboration on this research.

References

1. AIA Model Support Group: IFC2x Edition 3. International Alliance for Interoperability (2006). http://www.iai-international.org/Model/R2x3_final/index.htm (1 May 2006)
2. Aït-Kaci, H.: 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)
3. Baader, F., Calvanese, D., McGuinness, D., Nardi, D., Patel-Schneider, P.: The Description Logic Handbook: Theory, Implementation and Applications. Cambridge University Press, Cambridge (2003)
4. Datta, S.: Modeling dialogue with mixed initiative in design space exploration. Artificial Intelligence for Engineering Design, Analysis and Manufacturing 20 (2006) 129-142
5. Dooley, K.J.: A complex adaptive systems model of organization change. Nonlinear Dynamics, Psychology, and Life Sciences 1 (1997) 69-97
6. Kooistra, J.: Flowing. Systems Research and Behavioral Science 19 (2002) 123-127
7. Krishnamurti, R., Stouffs, R.: The boundary of a shape and its classification. The Journal of Design Research 4(1) (2004)
8. Manola, F., Miller, E. (eds.): RDF Primer. W3C World Wide Web Consortium (2004). http://www.w3.org/TR/rdf-primer/ (1 May 2006)
9. Park, K., Krishnamurti, R.: Flexible design representation for construction. In: Lee, H.S., Choi, J.W. (eds.): CAADRIA 2004. Yonsei University Press, Seoul, South Korea (2004) 671-680
10. Park, K., Krishnamurti, R.: The digital diary of a building. In: Bhatt, A. (ed.): CAADRIA'05, Vol 2. TVB School of Habitat Studies, New Delhi (2005) 15-25
11. Prigogine, I., Stengers, I.: Order out of Chaos. Bantam Books, New York (1984)
12. Stiny, G.: Weights. Environment and Planning B: Planning and Design 19 (1992) 413-430
13. Stouffs, R., Krishnamurti, R.: On a query language for weighted geometries. In: Moselhi, O., Bedard, C., Alkass, S. (eds.): Third Canadian Conference on Computing in Civil and Building Engineering. Canadian Society for Civil Engineering, Montreal (1996) 783-793
14. Stouffs, R., Krishnamurti, R.: 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) 436-452
15. Stouffs, R. Krishnamurti, R., Cumming, M.: Mapping design information by manipulating representational structures. In: Akın, Ö., Krishnamurti, R., Lam, K.P. (eds.): Generative CAD Systems. School of Architecture, Carnegie Mellon University, Pittsburgh, PA (2004) 387-400
16. Stouffs, R., Krishnamurti, R., Eastman, C.M.: A Formal Structure for Nonequivalent Solid Representations. In: Finger, S., Mäntylä, M., Tomiyama, T. (eds.): Proc. IFIP WG 5.2 Workshop on Knowledge Intensive CAD II. IFIP WG 5.2, Pittsburgh, PA (1996) 269-289
17. van Leeuwen, J.P., Hendrickx A., Fridqvist, S.: Towards dynamic information modelling in architectural design. Proc. CIB-W78 International Conference IT in Construction in Africa. CSIR, Pretoria (2001) 19.1-19.14
18. Woodbury, R., Burrow, A., Datta, S., Chang, T.: Typed feature structures and design space exploration. Artificial Intelligence in Design, Engineering and Manufacturing 13 (1999) 287-302