AN INTRODUCTION TO REPRESENTATION COMPARISON

Robert Woodbury[1], Andrew Burrow[2], Robin Drogemuller[3], Sambit Datta[4]
[1]School of Architecture, Landscape Architecture and Urban Design, Adelaide University
[2]Department of Computer Science, Adelaide University
[3]Division of Building, Construction and Engineering, CSIRO Australia
[4]School of Architecture and Building, Deakin University

ABSTRACT: In current computational building design theory and practice, representation schemes depend upon a set of formal operations for creating, changing and querying a representation. With a few notable exceptions, these operations do not provide ways of comparing representations to determine how representations are alike and how they are different. We have developed a theory for and a formal representation scheme that supports representation comparison. *This theory opens new approaches to unsolved problems in computational building design, notably the long-standing issue of automated building code checking.*

KEYWORDS: design representation, standards processing, typed feature structures

1. INTRODUCTION

This paper outlines a theory and formal scheme for *representation comparison* in computational design. The essential idea of representation comparison is to cast problems directly in terms of fundamental comparison operators acting on suitable representations. Though the CAD literature contains several components of the representation comparison idea, the area of computational linguistics provides a class of mature formalisms that directly address the idea. We describe how we have extended one of these formalisms in the context of a project on automated building code checking.

1.1 What is representation comparison?

Consider two representations, *A* and *B*, of a building or a part thereof, each of a possibly different level of abstraction or completeness. A facility for representation comparison would efficiently answer the following questions:

- is A more specific than B? (information specificity)
- is *A* the same as *B*? (information equivalence)
- is A a part of B? (information inclusion)
- what is in common between *A* and *B*? (information commonality)
- what is different between *A* and *B*? (information difference)
- can *A* and *B* be combined to create *C*? (information consistency)

1.2 Why is representation comparison useful?

The idea of representation comparison arose by observing that several areas in computational design ask similar questions of their representations. *Rule based generative design* compares its rules to designs. *Building code checking* compares provisions of a building code with a building design. *Information exchange* translates between representations—understanding the similarities and differences between these forms is crucial. *Constraint modelling* typically constructs models through the combination of comparable partial models. The techniques used to answer these questions vary widely and most are not declarative—they cannot reveal their operation to a user. The hypothesis of representation comparison is that a uniform declarative representation and set of comparison operators are useful across these disparate domains.

2. BACKGROUND

Precursors to the idea of representation comparison occur throughout the computational design literature, but only hint at the generality to which we aspire. For example, the null-shape detection problem in solids modelling is a primitive representation comparison operation. The shape grammar field is based on a sophisticated notion of sub-part. This allows shapes to be compared for equality and sub-part(Stiny, 1980, Krishnamurti and Stouffs,

Construction Informatics Digital Library http://itc.scix.net/
paper w78-2000-1058.content

1997). This approach misses the notion of information specificity and provides only limited notions of information commonality, difference and consistency. The Genesis system(Heisserman and Woodbury, 1993, Heisserman, 1994) expresses the match part of its rules in logical notation. Formally a rule matches if the conditions of its Left Hand Side can be found in a part of a single design state. This combines information specificity and information inclusion.

The idea of comparing and combining symbol structures is not new. Many formalisms depend on the operations of unification and subsumption. Unification simultaneously checks that two partial objects are consistent and, if so, combines them into a single more complete result. Subsumption checks that one partial object is more general than another possibly partial object. These, and other, operations underlie many of the formalisms in computational design. The insight of representation comparison is that these basic comparison and combination operators may suffice for many tasks without the complexity of intervening formalisms such as rule systems. Other fields have learned this lesson. For example, in *theoretical linguistics* rules have been almost entirely supplanted by formalisms relying on unification(Chomsky 1988).

2.1 Typed feature structures

Feature structures are a class of such unification-based formalisms. All feature structure systems comprise sets of attribute-value pairs called features and values respectively. Carpenter's *typed feature structures*(Carpenter, 1992), is distinguished from others by several properties. It displays a strong type discipline organised around an unambiguous version of multiple inheritance; has an efficient, clear notion of information specificity; is specifically designed to deal with partial information gracefully; and can handle both intensional and extensional kinds of objects. Further, it has a well-defined resolution mechanism supporting the enumeration of sets of satisfying objects from statements in a textual description language. This gives it an ability to tersely express structures and a natural programming interface.

Typed feature structures comprise five components: a set of features, an inheritance hierarchy of types, the language of feature structures that these together define, a description language and a set of algorithms. On the first four of these are posed conditions that collectively admit the robustness and efficiency of the algorithms.

- *Features* are simply a set of identifiers.
- A *type hierarchy* has two special conditions. First it must be a bounded complete partial order (BCPO) (essentially implying unique joins). Second, each feature used is introduced in exactly one class in the hierarchy and inherited in all subclasses. The former avoids disjunctive search on classification; the latter removes ambiguity in multiple inheritance.
- A *typed feature structure* may be depicted as a rooted, directed, finite, node- and edge-labelled graph, where node labels are interpreted as types, edge labels name the functional role that the target fills with respect to the source, and structure sharing models identity.
- *Descriptions* are the well-formed formulae in a description logic of which typed feature structures are the models. Therefore, descriptions pick out typed feature structures as well as constrain existing typed feature structures. Descriptions enter the type system to play the role of constraining typed feature structures to particular configurations.

Subsumption, *unification* and *π-resolution* are the basic typed feature structure algorithms.

- Given two typed feature structures A and B, A *subsumes* B if and only if all information in A is also asserted in B; it may also be said that A generalises B or that B specialises A. Subsumption is a partial order: the ordering is reflexive, anti-symmetric and transitive.
- *Unification* computes the minimal (most general) feature structure subsumed by two typed feature structures. Unification is a partial function, and where undefined, there is no common specialisation and the two typed feature structures are said to be inconsistent.
- *π-resolution* generates typed feature structures satisfying a description and enforces arbitrary type constraints specified as descriptions. Given a function assigning a description to each type, a resolved typed feature structure of a given type is one in which every substructure satisfies the constraint on its type.

Subsumption, unification and π-resolution are fundamental in the sense that other algorithms specific to the domain at hand can be developed from them. The typed feature structure mechanism is, *inter alia*, structured to admit efficient instances of these algorithms.

Feature structures in general are alternative objects over which logic programming languages have been defined, for example LOGIN(Ait-Kaci and Nasr, 1986), LIFE(Ait-Kaci and Podelski, 1993) and ALE(Carpenter and Penn, 1997). Feature structures extend terms from the Herbrand universe by replacing subterm positions with feature labels, expressing a subsumption ordering over the term labels, allowing path cycles, and defining token identity at the level of nodes(Ait-Kaci and Podelski, 1993).

Present feature structure theory does not admit the representation of continuous domains such as the reals or point-sets. Representations of buildings require such domains and a theory of representation comparison for buildings must admit them. Chang(1999) gives a preliminary approach to the representation of continuous domains in typed feature structures.

2.2 Design space exploration

Researchers at Adelaide have devised a new method for generative design they call *incremental mixed-initiative π-resolution over typed feature structures* (*iπ-resolution*)(Woodbury *et al*., 1999, Burrow and Woodbury, 1999, Woodbury *et al*., 2000, *Erasure*, Woodbury *et al*., 2000, *Navigation*). Under *iπ-resolution* a design space is a strongly ordered object. New states in the space are constructed as refinements of existing states. Rules entirely disappear from the generative formalism. The needed operation of erasure moves from being a generative operator acting on design states to an exploration operator acting on the design space. The *iπ-resolution* algorithm is constructed using subsumption and unification.

2.3 Computer-aided building code checking

The area of *computer aided building code checking* (more generally called *standards processing*) is also concerned with the representation of designs and with rigorous computations over them. Viewed through the idea of representation comparison, building code checking becomes the comparison of the provisions of a building code with a building design and its parts. One of the top research groups in standards processing realised this some time ago. Hakim and Garrett(1993) showed that description logic based systems offer the needed automatic description and comparison techniques. In addition they correctly saw the need for partial descriptions of objects for both standards and designs. They used a general description logic package (LOOM)(Mac Gregor, 1990) and thus encountered the intractability of classification in such languages. They did not discover that there are approaches, similar to description logic, that admit efficient classification, for example, Carpenter's(1992) *typed feature structures*.

In computational design, any current proposal to do work in automated building code checking must attend to the past record. Though building code checking has an extensive literature, many researchers view it as a difficult problem. Those who count its successes cite current programs like BCAider(Sharpe and Oakes, 1995), the NBCC Classifier(Vanier, 1995). BCAider is a particular accomplishment in both research and commercial terms. It implements a mechanism for automatically classifying and checking a building design based on an interactive dialogue with a user. It is recognised by most councils in Australia (the relevant code compliance authorities) as being logically equivalent to the Code. Those who view building code checking as a problematic enterprise look to past efforts and make three basic arguments (Fenves *et al*., 1995, Kiliccote, 1997). The first is that codes are rife with indeterminate provisions, that is, provisions that require judgement and knowledge of context. The second is that codes are complex and include exceptions and higher-order provisions. The third is that codes and buildings have different computer-based representations.

Countering this retrospective criticism are four recent developments. The first is the use of modern performance codes, in which a code is divided into performance and deemed-to-satisfy parts. The performance part of a code is deliberately indeterminate—it specifies what a building is to achieve in functional terms. The deemed-to-satisfy parts give explicit, measurable conditions that, if met in a building, imply its compliance with the code. In practice, the deemed-to-satisfy provisions are used in the majority of cases. The Building Code of Australia is an exemplary modern performance code. In it the deemed-to-satisfy provisions contain very few indeterminate provisions, exceptions or higher-order provisions. The second development is in the use of *mixed initiative* in complex systems. It is now widely recognised that the appropriate questions to ask when researching computational support for complex tasks lie not in the full automation of processes, but in the appropriate division of a task between human and computer. *Mixed initiative* reduces the impact of indeterminate provisions in a computer-aided code checking process. The third development is the advent of object-oriented building modellers. These represent buildings as aggregations of building components instead of aggregations of graphical marks or abstract objects. All extant code representations are component based. Increasingly, the representation of buildings and codes is on common ground. The fourth and final development is that researchers have better representations. "Better" means more structured, more efficient and more relevant to a domain. In these terms, typed feature structures are a better representation than the logical terms of Genesis or the hybrid frame and production system of BCAider.

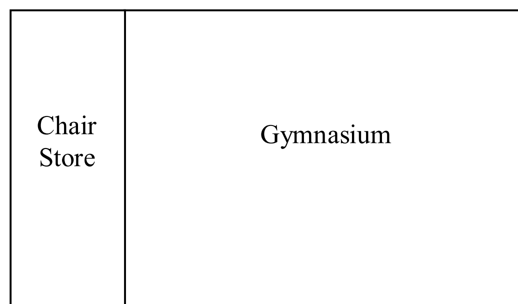
The special nature of representation of continuous information such as geometry is mostly ignored in the extant work on computer-aided building code checking. Most researchers relied on a belief that derived data such as volumes would be available to a code checking system when needed. When a processing system relies on its subject information having strong formal properties the system ceases to apply and ad-hoc methods have to be used.

3. CODE CHECKING WITH REPRESENTATION COMPARISON

3.1 Structure of the Building Code of Australia

In its most recent version the Building Code of Australia (BCA 1996) is a performance-based code. Such codes have the (not coincidental) feature that they isolate statements of intended performance from the specification of exemplary solutions that are deemed to satisfy the performance measures.

A simple example is a school gymnasium, which is to be used for basketball, other court games and as an assembly area. A chair store is required at one end. The required area for a basketball court (http://www.nba.com/basics/rules/) is 94 feet x 50 feet (29m x 15m). Allowing 10 foot (3m) run off areas all round gives a total size of 114 feet by 70 feet (35m x 21m).



Under the Building Code of Australia (BCA) this is a Class 9b, Assembly Building. The relevant objectives and functional statements are:

DO1 (Objective)

The Objective of this Section is to-

(a) provide, as far as is reasonable, people with safe, equitable and dignified access to-

(i) a building; and

(ii) the services and facilities within a building; and

(b) safeguard occupants from illness or injury while evacuating in an emergency.

DF2 (Functional requirement)

A building is to be provided with means of evacuation which allow occupants time to evacuate safely without being overcome by the effects of an emergency.

DP4 (Performance Requirement)

Exits must be provided from a building to allow occupants to evacuate safely, with their number, location and dimensions being appropriate to-

(a) the travel distance; and

(b) the number, mobility and other characteristics of occupants; and

(c) the function or use of the building; and

(d) the height of the building; and

(e) whether the exit is from above or below ground level.

These are obviously "open" clauses requiring human interpretation. However, the "Deemed-to-satisfy" clauses allow automated checking for most buildings.

Under Table D1.13, School multi-purpose hall, the area per person for occupancy purposes is given as one person per square metre. Hence, the total number of people for which exits need to be provided is 735.

Clause D1.6 requires:

In a required exit or path of travel to an exit-

...

(d) if the storey or mezzanine accommodates more than 200 persons, the aggregate width, except for doorways, must be increased to-

(i) 2 m plus 500 mm for every 60 persons (or part) in excess of 200 persons if egress involves a change in floor level by a stairway or ramp with a gradient steeper than 1 in 12; or

(ii) in any other case, 2 m plus 500 mm for every 75 persons (or part) in excess of 200;

Hence, the total width of exits needs to be 5.5 m. This could be achieved with 3 doors, each 1.9m wide.

Clause D1.5 additionally requires:

Exits that are required as alternative means of egress must be-

...

(b) not less than 9 m apart; and

(c) not more than-

...

(iii) in all other cases - 60 m apart ; and

This imposes the additional requirement that the doors be more than 9 m apart and less than 60 m apart.

3.2 Feature structure representation for code checking

We have designs expressed in a feature structure language in which the separation of function and form is explicit(Flemming and Woodbury, 1995). A pair of feature structures represents each design. One is called the *FU* for *functional unit*, and the other is called the *DU* for *design unit*. Since each is a feature structure, they are each recursively composed. The FU is a composition of *functional units*, and the DU is a composition of *design units*. The FU describes the decomposition of the design into functional assemblages where each edge in the feature structure describes a subpart relation concerning function. Likewise, the DU describes the decomposition of the design into physical assemblages where each edge in the feature structure describes a physical subpart relation. A design is a DU-FU pair held in a certain relation.

A design is such a DU-FU pair where there exists an allocation of DU components to FU components. This allocation attempts to describe the satisfaction of the functional requirements by the physical arrangements. To be *well formed* a design needs to make an allocation for every FU component. The *soundness* of a design must be tested by an analysis of the properties of the physical arrangements with respect to the functional requirements, and the allocation is meant to guide this analysis. Therefore, soundness refers only to the combination of FU, DU, and allocation.

One aspect of a design's soundness is its conformance to building codes. This is of particular interest where the building code contains deemed-to-satisfy rules. Such rules are explicitly designed to simplify the question of recognising soundness by rules. Therefore, we are interested in the algorithmic checking of a design's soundness with respect to the deemed-to-satisfy rules of a building code.

In the proposed system, the resolution of recursive type constraints is the vehicle for exploration in the space of well formed FU structures. The designer has access to constructive steps that advance the design by adding information to the functional decomposition, as well as navigational mechanisms that traverse the resultant ordered space of designs. These navigational mechanisms include analogues of typical editing steps such as erasure and backtracking. Therefore, we are interested in conformance testing in two distinct time scales: during constructive steps, and during traversal.

3.3 Interleaved Conformance Tests

One goal of computer aided design environments is to inform the designer of issues arising in the finished artefact at earlier stages of design. The idea is to shorten the cycle time in analysing alternatives. Design decisions that affect building code conformance are a pertinent example.

In the proposed system, the earliest point of intervention is during constructive steps. Each constructive step accepts two arguments. The first is the substructure of interest. In feature structure terms, this is indicated by a feature path extending from the root node to the substructure, hence the name π-resolution (for *path resolution*).

The designer indicates the substructure by a user interface gesture. The second argument is the type to which the substructure is being committed. Incremental π-resolution is so named because the set of types against which a substructure has been resolved is incrementally built up. To be fully resolved to its target type, a substructure must be resolved to all types subsuming the target. The subsuming types must be resolved in a linear extension of the inheritance ordering, that is, each type must be resolved only after the types it inherits. The pair of arguments in each incremental π-resolution step is also suitable to triggering conformance tests.

Conformance tests may be interleaved with π-resolution. Each resolution step involves unification with a satisfier of the resolved type constraint. As a by-product, the unification procedure maps the satisfier into the resulting FU. Therefore, the conformance tests may be interleaved by identifying those functional unit types associated with sufficient (feature) structural information to identify components coming under deemed-to-satisfy rules. The interleaved deemed-to-satisfy rules enjoy bindings to the relevant components in the FU and the DU in the resultant design. The mapping requires two steps: firstly, via the map from the type constraint satisfier to the resultant FU, and then, via the allocation map to the corresponding DU. An advantage to this approach is that the incremental type resolution mechanism ensures that an attached deemed-to-satisfy rule is applied at most once and only after all constructive steps associated with subsumed types have been applied. A disadvantage to this approach is the task of pairing deemed-to-satisfy rules with suitable type constraint satisfiers.

The more general the satisfiers to which deemed-to-apply rules are attached, the earlier the conformance tests intervene in the design process. However, the main consideration in the attachment of rules must be the completeness of the compliance testing. For a design to be compliant, every applicable deemed-to-satisfy rule must have been tested. Whether a particular association of deemed-to-satisfy rules with type constraint satisfiers guarantees this condition is a non-trivial question. Much of the power of recursive type constraints derives from the use of disjuncts, which give rise to finite collections of satisfiers. While this allows type constraints to provide collections of distinct prototypes, it also means that the completeness of an association between type constraint satisfiers and deemed-to-satisfy rules must take into consideration the act of selecting a disjunct.

One solution is to engineer types in the system simply to carry deemed-to-satisfy rules. This is certainly possible given that the type hierarchy is a lattice (under inclusion of the absurd type as a top element). The original types will be augmented by derived types that also inherit the deemed-to-satisfy types. The incremental π-resolution mechanism ensures that these deemed-to-satisfy types will be resolved against. In addition, the designer will be able to defer the resolution of deemed-to-satisfy types within the rules of incremental π-resolution mechanism. A design will be compliant if all substructures are fully resolved with respect to the deemed-to-satisfy types.

Another solution is to restrict the structure in the satisfiers associated with deemed-to-satisfy rules to trees of unit depth. In this case, we can utilise the underlying minimal structure constraints known as *appropriateness specifications*. Appropriateness specifications are a discipline associating types with features. Each feature must be associated with a minimum (most general) introducing type, so that every type subsumed by the introducing type carries the feature. The appropriateness specifications also constrain the types of feature values. Feature introduction provides efficient domain side type inferences when computing the satisfiers of a description. It can also be used to demand the existence of features according to the resolution state of a substructure. Therefore, if a deemed-to-satisfy rule can be written using only the immediate children of a substructure, then it can be associated with the most general type at which the required features have been introduced. The advantage of this mechanism is the clarified relationship between a type and the deemed-to-satisfy rules without the need to introduce new types.

The problem common to both solutions is the extent of the image of the deemed-to-satisfy rule in the design. Each deemed-to-satisfy rule makes use of a collection of values drawn from the design. In order to gain access to this information a feature structure is mapped onto a substructure of the design. The mechanism proposed in this section interleaves this mapping with the incremental π-resolution steps. This has the considerable advantage of applying the deemed-to-satisfy rules at the moment when the user is considering the particular substructure. However, it is somewhat artificial since the subjects of the deemed-to-satisfy rule need not be the same as any particular constraint resolution step. The first suggestion was to include these structures into the constraint store as satisfiers of special deemed-to-satisfy types. The second suggestion was to restrict the subjects to those accessible along a single feature. Unfortunately, the second is untenable.

As an example, consider the Clause~D1.6. This rule talks directly about the widths of the exits. However, the rule includes as subjects: a mezzanine or storey, exit way floor or stairs, and exits. Therefore, the rule must get its mappings from a satisfier in which each of these subjects is reachable along feature paths. This is a real restriction. Typed feature structures represent a description logic explicitly designed to ensure efficiency in the principal operations of subsumption, unification, and satisfaction. One result is that features are interpreted as

describing functions rather than relations; another is that each feature structure and substructure is a rooted graph. Therefore, the satisfier in the example is of a type that has as subparts a mezzanine and an exit, e.g., a building. The difficulty is that: whether or not a particular component of a design has been shown to satisfy the applicable deemed-to-satisfy rules depends not on the deemed-to-satisfy satisfiers applied at the substructure, but rather the deemed-to-satisfy satisfiers applied "upstream" at substructures from which the particular component is reachable.

In summary, there are several difficulties with interleaving conformance tests and incremental π-resolution.

It is difficult to argue the completeness of the tests since: they are distributed across the types of the FU system, and a component's compliance results from tests applied to those components that contain the component. However, this problem is largely remedied by associating deemed-to-satisfy rules with special deemed-to-satisfy types that are interwoven into the type hierarchy.

Programming effort is required to associate deemed-to-satisfy rules with appropriate type constraints, and where this effort concerns finding satisfiers to carry the rules it is less declarative of and more brittle to changes in the deemed to satisfy rules. Again, special deemed-to-satisfy types alleviate this problem at the cost of adding noise to the structure of the type hierarchy.

Where deemed-to-satisfy types are used the type hierarchy and therefore the design space is only applicable to designs under the particular building code.

Against these disadvantages, there is a principal advantage. Interleaved conformance tests apply the deemed-to-satisfy rules as the user considers the particular substructure. With the introduction of deemed-to-satisfy types, individual deemed-to-satisfy rules are applied at the discretion of the designer. This could be considered a distraction, but incremental π-resolution aims for the maximum freedom in the order of evaluation of fine grained, type constraints. It is envisaged that many chains of steps will be automatically composed into single user interactions under the control of user tunable heuristics.

3.4 Post-processing Conformance Tests

We can seek to quarantine the type hierarchy from the exact details of the building code by applying conformance tests as a post-processing step. The separation allows designs and parts of designs to be reused under different building codes, and the effect of changes to the building code may be the subject of experimentation. However, further advantages accrue in the respite from integrating the deemed-to-satisfy rules into the type hierarchy.

Post-processing can follow the same general approach as interleaved conformance testing in binding the subjects of the deemed-to-satisfy rules via a mapping from a feature structure into the design. Subsumption in feature structures is decided by a homomorphism from the subsuming feature structure into the subsumed feature structure. In the unification procedure, the two operands are combined to create the most general feature structure subsumed by both operands. The by-product of unification is the union of the homomorphisms from the operands to the result. It is this mapping that is referred to in the discussion of interleaved conformance tests. In post-processing, the mapping is generated directly by a subsumption test from the feature structure containing the deemed-to-satisfy provisions to a substructure of the result. The feature structures containing the deemed-to-satisfy provisions must themselves be generated by a separate π-resolution process. The mapping between resolved designs and resolved deemed-to-satisfy provisions must be computed outside of π-resolution and its algorithmic complexity could prove to be a problem.

Nonetheless, post-processing yields a distinct advantage. It separates the representation of designs and their parts from the representation of codes. However, the type hierarchies used by both must share some structure, at least the same feature set and possibly the same feature introduction and appropriateness specifications.

3.5 Summary

At the time of writing, we have conducted a few small experiments into code checking by representation comparison. These are sufficient for us to sketch the two fundamental approaches above but do not provide sufficient support to declare that one approach is better than the other, or indeed that the two approaches exhaust the range of possibilities. What we do have though is an approach distinctly different from any yet tried.

ACKNOWLEDGMENTS

The authors would like to acknowledge the support given by the Australian Research Council Large and Small Grants Schemes, The Department of Industry, Science and Tourism Major Grant Scheme, the Australian

Postgraduate Award Scheme and The University of Adelaide School of Architecture, Landscape Architecture and Urban Design.

REFERENCES

Ait-Kaci H. and Nasr. R. (1986). Login: A logical programming language with built-in inheritance. *Journal of Logic Programming*, 3:187-215.
Ait-Kaci H. and Podelski A.(1993). *Towards a meaning of life.* technical report PRL-RR-11, Digital Paris Research Laboratory.
Australian Building Codes Board. (1996). *Building Code of Australia*. CCH Australia Limited.
Burrow A. and Woodbury R. (1999). π-resolution in design space exploration. In Godfried Augenbroe and Charles Eastman, editors, *Computers in Building: Proceedings of the CAADFutures'99 Conference*, pp 291-308, Kluwer Academic Publishers, Atlanta, Georgia.
Carpenter B. and Penn G. (1997). *The Attribute Logic Engine User's Guide*. Computational Linguistics Program, Philosophy Department, Carnegie Mellon University, Pittsburgh, PA USA 15213, 2.0.3 edition.
Carpenter B. *The Logic of Typed Feature Structures with applications to unification grammars, logic programs and constraint resolution*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.
Chang T.W. (1999). *Geometric Typed Feature Structures: Towards Design Space Exploration,* Ph.D. Thesis, The University of Adelaide (Architecture), October 1999.
Chomsky N. (1988). *Languages and Problems of Knowledge*. Vol. 16 of *Current Studies in Linguistics*, MIT Press Cambridge, Massachusetts.
Fenves S. J., Garrett J. H., Kiliccote H., Law K. H. and Reed K. A. (1995). Computer representation of design standards and building codes: U.S. perspective. *The International Journal of Construction Information Technology*, 3(1):13-34.
Flemming U and Woodbury R. F. (1995) Software Environment to support Early phases in building Design SEED: Overview. ASCE Journal of Architectural Engineering, 1(4):147-152.
Hakim M and Garrett J. H. Jr. (1993). Using description logic for representing engineering design standards. Journal of Engineering with Computers, 9:108-124.
Heisserman J.A. (1994) Generative Geometric Design, *IEEE Computer Graphics and Applications*, 14(2):37-45.
Heisserman J.A. and Woodbury R. F. (1993).Generating languages of solid models. In *Proceedings of Second ACM/IEEE Symposium on Solid Modeling and Applications*, pages 103–112.
Kiliccote H. (1997). A Standards Processing Framework. PhD thesis, Department of Civil Engineering, Carnegie Mellon University, USA. URL: http://han.ices.cmu.edu
Mac Gregor R.M. (1990). *Loom users manual*. Technical report, ISI University of Southern California, La Jolla, California.
Krishnamurti R. and Stouffs R. (1997). Spatial change: Continuity, reversibility, and emergent shapes. *Environment and Planning B: Planning and Design*, 24:1-25.
Sharpe R. and Oakes S.(1995). Advanced IT processing of Australian standards and regulations. *The Intl. Journal of Construction Information Technology*, 3(1):73-89.
Stiny G. (1980). Introduction to shape and shape grammars. *Environment and Planning B: Planning and Design*, 7(3):343-352.
Vanier D. J. (1995). Canada and computer representations of design standards and building codes. *International Journal of Construction Information Technology*, 3(1):1-12.
Woodbury R. F., Burrow A. L., Datta S. and Chang T.W. (1999). Typed feature structures in design space exploration, *AIEDAM Special Issue on Generative Systems in Design*, 13(4):287-302.
Woodbury R. F. and Datta S and Burrow A. L. (2000). Erasure in Design Space Exploration, In John Gero, editor, *Artificial Intelligence in Design 2000*, Worcester, Mass., June 2000, to appear.
Woodbury R. F. and Datta S and Burrow A. L., (2000). Navigating subsumption-based design spaces, *CAADRIA2000*, National University of Singapore, Singapore, May 18-19 2000, to appear.
Woodbury R. F. Burrow A. L., Drogemuller R. M. and Datta S. (2000). Code Checking by Representation Comparison, *CAADRIA2000*, National University of Singapore, Singapore, May 18-19 2000, to appear.