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.

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, 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).

2.2. 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 \( \pi \)-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, the two typed feature structures are said to be inconsistent.

- \( \pi \)-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 \( \pi \)-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.

2.3. DESIGN SPACE EXPLORATION

Researchers at Adelaide have devised a new method for generative design they call incremental mixed-initiative \( \pi \)-resolution over typed feature structures (\( i\pi \)-resolution)(Woodbury et al., 1999, \( i\pi \)-resolution, Woodbury et al., 1999, Exploring, Burrow and Woodbury, 1999, Typed Feature Structures in Design Space Exploration, Woodbury et al., 2000, Erasure, Woodbury et al., 2000, Navigation). Under \( i\pi \)-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 \( i\pi \)-resolution algorithm is constructed using subsumption and unification.

2.4. 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 planning 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-comply parts. The performance part of a code is deliberately indeterminate—it specifies what a building is to achieve in functional terms. The deemed-to-comply parts give explicit, measurable conditions that, if met in a building, imply its compliance with the code. In practice, the deemed-to-comply provisions are used in the majority of cases. The Building Code of Australia is an exemplary modern performance code. In it
the deemed-to-comply 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. Methodology

We have two implementations of typed feature structures. ALE(Carpenter and Penn, 1997) is a free implementation of typed feature structures. KRYOS is an implementation of typed feature structures developed largely by Andrew Burrow. KRYOS was originally designed to support the exploration of building designs, but it also provides an implementation of typed feature structures more complete than ALE and is explicitly designed to accommodate extensions.

Our method has been to develop first, a uniform representation scheme in which to express and compare design information; second, a research tool for creating and comparing representations; and third a topical demonstration of representation comparison. Corresponding to these three goals are the following technical objectives.

1. extension of the typed feature structure mechanism to represent continuous information;
2. development of a representation comparison “shell”; and
3. case studies in representation comparison using the Building Code of Australia (BCA) and buildings represented in the new mechanism.
3.1. EXTENSION TO CONTINUOUS DOMAINS

Typed feature structure theory requires that the type hierarchy be a *bounded complete partial order* (a BCPO). This means that there must be a universal type and that every consistent set of types has a unique *most general subtype* (a join or least upper bound). The extant theory uses a finite set of types arrayed in an inheritance hierarchy, but this is convention and not a formal restriction. Our approach to continuous information is to treat each set of such information (integers, reals, intervals, point-sets, etc.) as being a type hierarchy whose elements have no features defined over them. We define the type hierarchy by choosing a useful relation of information specificity. For each of the sets listed above several such relations meet the required BCPO conditions and are useful in the building domain. For example, for three-dimensional point-sets, set inclusion and its converse both yield correct type hierarchies. Under the former, unification corresponds to set intersection. Under the latter, unification corresponds to set union. We call such types order types and the types from the usual finite case succession types. Chang’s Ph.D. thesis (Chang, 1999), presents a partial theory and demonstration implementation for this extension.

A feature structure system holding order types refers to a computation external to the system for all queries about the order type. It suffices that the external computation behaves as if it were a well-formed feature structure system. This indirection allows us to use existing commercial packages to represent order type information. For three-dimensional point-sets use the mature, commercial cell-complex-based non-manifold modeller SHAPES (XOX Corp.). The existing KRYOS interpreter for feature structures has an order type mechanism built in. We use this mechanism to link together KRYOS and SHAPES. This gives us the ability to bring three-dimensional information into the feature structure mechanism of KRYOS.

Given a hierarchy of order types, two further extensions to the typed feature structure formalism are needed. The first provides an *order type expression language*—a means, other than unification, of picking out the desired members of the type hierarchy as a computation proceeds. The second is the *subsumption constraint*, a new kind of constraint in the description language. The subsumption constraint permits the expression and maintenance of basic geometric relations such as containment.

The order type expression language provides a set of composable functions returning members of an order type. Such expressions fit into the part of the feature structure description language that calls types out by name. In effect, an order type expression stands as a name for a type. Operationally, there are two interpretations for an order type expression: either expressions are a commitment to what is known at the point in *iπ-resolution* at which they are resolved; or expressions are constraints on an object that must remain true throughout *iπ-resolution*. The order type expression language can be very small,
or arbitrarily rich, in the functions it provides. For this project we are using the
standard operators of a small set of primitive objects, union, intersection,
difference, similarity transformations, and adjacencies. All are supported in the
SHAPES modeller. Chang’s thesis(Chang, 1999) gives several example
languages in similar and simpler domains.

The subsumption constraint allows an object $A$ to remain more specific than
another object $B$. An attempt to specialise $B$ beyond the current state of $A$ will
result in $A$ and $B$ both being specialised to the new level of $B$ (but $A$ and $B$ will
remain as distinct objects). This constraint allows the persistent declaration of
specific numeric and geometric relations. It remains to show that the
subsumption constraint extension to the description language retains the
operational properties of typed feature structures. In particular, the new
satisfiability relation must be monotonic with respect to feature structure
specialisation, and must admit a function that maps extended descriptions to
feature structures. The subsumption constraint is closely related to path
inequations, and the same technique of augmenting the feature structure with
syntactic descriptions of the constraints is expected to suffice.

3.2. THE REPRESENTATION COMPARISON “SHELL”

The representation comparison shell provides a simple interface with which to
build case studies. It has two parts. The first is an interactive type hierarchy
editor. The second is a classification controller that allows detailed examination
of the process of comparing representations. In his thesis Datta(2000) describes
an implementation of a type hierarchy editor and $\pi$-resolution controller using
the QT user interface toolkit(Dalheimer, 1999). This work provided an initial
design for the representation comparison shell.

3.3. CASE STUDIES IN REPRESENTATION COMPARISON

The project’s case study in representation comparison has three parts
1. representing building codes as typed feature structures;
2. representing designs as typed feature structures; and
3. case studies of computing design compliance against building codes.

3.3.1. Representing codes

We are currently representing a portion (Part D-1.0 Provision for Escape) of the
Building Code of Australia (BCA) as typed feature structures. The method for
this work is similar to that used in grammatical study of a corpus of
design(Woodbury and Griffith, 1993; Woodbury and Chang, 1995; Datta and
Woodbury, 1997). In such studies a grammar (in our case a type hierarchy) is
developed and repeatedly tested by executing it, usually with an interpreter (in
our case with $\pi$-resolution). The analogue to the corpus of designs is the BCAider program itself, the behaviour of which we are duplicating.

3.3.2. Representing designs
We represent building configurations as typed feature structures using the same modelling technique described above. We follow the standard practices of (1) considering a building design as a collection of related assemblies and elements, where an element is a primitive object and (2) separately representing function and form. We employ the SEED knowledge-level (Flemming and Woodbury, 1995), a mature instance of these standard notions.

3.3.3. Code compliance
This part of the work remains in front of us. The tasks at this stage are to integrate the type hierarchies from the code and building representations, to specify and implement compliance algorithms as combinations of the fundamental comparison algorithms over feature structures, and to review the results with CSIRO code experts. We take advantage of the partial representation and recursive type constraint properties of feature structures to automatically create minimal design fragments to which provisions apply and to order these fragments into a provision classification hierarchy according to the subsumption ordering for designs. We again to use subsumption to classify designs against the hierarchy of provisions.

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


