UNIVERSITY OF ADELAIDE LIBRARY
10.9.99

Geometric Typed Feature Structures: Toward *design space exploration*

Teng-Wen Chang

Dissertation

Submitted in fulfilment of the requirements
for the degree of Doctor of Philosophy

School of Architecture, Landscape Architecture and Urban Design
The University of Adelaide
12 August 1999

copy right © 1999 by Teng-Wen Chang
All rights reserved

Table of Contents

0.1 Abstract 1

0.2 Statement 2

0.3 Acknowledgements 5

I Introduction and Background 7

1 Introduction 9

1.1 Overview of dissertation 11

1.2 Organisation of dissertation 12

2 Background: From design to exploration 15

2.1 Views on Design 16

2.2 *State space search* is a useful view 27

3 Directly Supporting exploration—design space explorers 35

3.1 Components of design space explorers 35

3.2 Several implementations of design space explorers 39

3.3 Summary 78

II Theories and Algorithms 81

4 Domain objects: the SEED knowledge level 83

4.1 The knowledge level concepts 84

4.2 Realisation of the SEED knowledge level 91

5 Representation of the knowledge level using Typed Feature Structures 93

5.1 Three Components 94

5.2 Three Algorithms 98

5.3 A representation using Typed Feature Structures 100

5.4 Toward representing geometric information 103

6 Geometric Typed Feature Structures 105

6.1 The characteristics of Order Types 108

6.2 Order Type Models 113

6.3 Representing Order Types 116

6.4 Order Types carrying geometric information 154

6.5 Summary 180

III Implementation and Examples 181

7 Implementation—Ordering Geometry System 183

7.1 System Integration 184

7.2 Order Type implementation 187

7.3 Two examples of Order Type implementation using C++ .190

7.4 How to specify geometric types in OGS 205

8 Examples 207

8.1 Single Fronted Cottages 207

8.2 Examples—Building Enclosures 213

8.3 Summary 221

IV Conclusion 223

9 Conclusion 225

10 Contributions and future work 227

10.1 Contributions 227

10.2 Prospect of future research work 228

Bibliography 231

Appendix 241

Appendix A: The syntax of Kryos descriptions 243

Appendix B: Examples of Order Type Implementation 245

Appendix C: Single-Fronted cottage 283

Appendix D: Index of Symbols and Expressions 287

List of Figures

FIGURE 1. Interrelationships in the Design Process from *Digital Design Media*[Mitchell and McCullough, 1995b]. 20

FIGURE 2. An Abstract Interpretation of three Phases in the Design Process[Radford and Gero, 1988]. 23

FIGURE 3. The Relationships Between Decisions with Constraints and Ideal Goals[Radford and Gero, 1988]. 24

FIGURE 4. Possibilities of a Derivation Graph in the Design States[Woodbury, 1991a]. 25

FIGURE 5. Woodbury's Informal Equation for the Exploration Model[Woodbury, 1996]. 32

FIGURE 6. A Partial Functional Unit Hierarchy for an Insulated Enclosure. 86

FIGURE 7. Abstract Enclosure Elements for (a) Wall Enclosure, (b) Enclosure for the Joint between the Wall and Roof, (c) Ground Enclosure and (d) Roof Enclosure. 87

FIGURE 8. An Example of a Technology for the Continuous-Support of a Wall Enclosure is shown on the Top. The Partial **FU** Hierarchy is on the Bottom. 88

FIGURE 9. Left: the Initial Status of Spaces Enclosure. Right: Wall Continuous Support (Membrane) Design Unit (After Applying Technology). 89

FIGURE 10. An Example of Design Spaces Demonstrates the Transformation of Geometry from one State to another. 90

FIGURE 11. Mapping between a Generic Abstract Knowledge System Concept and the Corresponding Representation Scheme[Woodbury et al., 1998]. 91

FIGURE 12. Mapping Between the SEED Knowledge Level and its Realisation Concept. 92

FIGURE 13. A Graph Notation of an Inheritance Hierarchy. 94

FIGURE 14. Definition of Appropriateness Specification[Carpenter, 1992]. 95

FIGURE 15. Definition of a Feature Structure Over Types and Feats[Carpenter, 1992]. 96

FIGURE 16. An Example Feature Structure in Graph Notion. 96

FIGURE 17. Definition of Path Value[Carpenter, 1992]. 97

FIGURE 18. Definition of Description[Carpenter, 1992]. 97

FIGURE 19. Definition of Satisfaction[Carpenter, 1992]. 98
FIGURE 20. Definition of Subsumption[Carpenter, 1992]. 99
FIGURE 21. Definition of Unification[Carpenter, 1992]. 100
FIGURE 22. Mapping Between the Abstract System Concepts and the Corresponding Typed Feature Structures(TFS) Concepts. 101
FIGURE 23. A Complete Mapping from the Knowledge Level to its Symbol Level Representation. 102
FIGURE 24. A Description of Atomic Values in a Formal System. 103
FIGURE 25. Example of a Complete Partial Order of Three Real Intervals and a Bottom. 106
FIGURE 26. Example of a Succession *InheritanceHierarchy.* 107
FIGURE 27. Example of an Order Type Hierarchy. 109
FIGURE 28. An Example Represents the Properties of a Cube with a Higher Level Structure (in AVM Notation). 110
FIGURE 29. The Descriptions of Order Types have no Path Values. 111
FIGURE 30. Definition of a Geometric Feature Structure Over Order Types. 111
FIGURE 31. Comparison of Satisfaction of Typed Feature Structures with Satisfaction of Geometric Typed Feature Structures. 112
FIGURE 32. Order Types Need no Constraints. 112
FIGURE 33. Requirements of a Partial Order Set[Davey and Priestley, 1994]. 115
FIGURE 34. A Part of 'Lifted' Reals as an *InheritanceHierarchy.* 120
FIGURE 35. A Part of 'Lifted' Integers in an *InheritanceHierarchy.* 123
FIGURE 36. A Part of 'Lifted' Whole Numbers as an *InheritanceHierarchy.* 126
FIGURE 37. A Part of Ascending Reals as an *InheritanceHierarchy.* 129
FIGURE 38. A Part of Descending Reals as an *InheritanceHierarchy.* 132
FIGURE 39. A Part of Ascending Integers as an *InheritanceHierarchy.* 135
FIGURE 40. A Part of Descending Integers as an *InheritanceHierarchy.* 138
FIGURE 41. A Part of Ascending Whole Numbers as an *InheritanceHierarchy.* 141
FIGURE 42. A Part of Descending Whole Numbers as an *InheritanceHierarchy.* 144
FIGURE 43. A Part of Real Intersection Intervals as an *InheritanceHierarchy.* 147
FIGURE 44. A Part of Integer Intersection Intervals as an *InheritanceHierarchy.* 150
FIGURE 45. A Part of Whole Number Intersection Intervals as an *InheritanceHierarchy.* 154
FIGURE 46. Examples of Cell Complex: a Cell Complex Comprises Multi-Dimensional Cells. 156
FIGURE 47. The Boundary of a Region is Comprised by Cells from Different Dimensions, e.g. the Vertices of the Region Above are 0D Cells. 157
FIGURE 48. The Boundary of an nD Cell Comprises the (n-k)D Cells that Bound it. 157
FIGURE 49. Two Regions: A and B are Part of a Cell Complex C. Each Region Comprises a Set of Cells and two Regions might Share a Set of Cells. 158
FIGURE 50. The Boundary of a Region is Represented as a Region Containing Cells of Lower Dimensionality Than the Given Region. 158
FIGURE 51. The Cells of a Region do not Maintain their Identities Through the Merge Operation. 159
FIGURE 52. The Point-Set of a Region Remain Invariant Across a Merge Operation, but the Cells it Comprises might Change, e.g. the Cells that

Region A Comprises Change After Merge Operation, but the Boundary of Region A still Remains Invariant. 159
FIGURE 53. Boolean Ops over Regions are the Set Boolean Ops (After Merging). 160
FIGURE 54. Regions Need not Model Closed Sets. Use two Regions: A and B from Figure 52, the Results of Boolean Ops Over these two Regions are not Closed. 160
FIGURE 55. Example of Point-Set Containing Relation Among Geometry Types. 161
FIGURE 56. A Part of 'Lifted' Point-Set as an *InheritanceHierarchy.* 165
FIGURE 57. A Part of *IPSet* as an *InheritanceHierarchy.* 168
FIGURE 58. A Part of *OPSet* as an *InheritanceHierarchy.* 171
FIGURE 59. Example of Two IOPSets *a* and *b*, Such that . 174
FIGURE 60. A Part of *IOPSet* as an *InheritanceHierarchy.* 175
FIGURE 61. Example of Two OIPSets *a* and *b*, Such That . 178
FIGURE 62. A Part of *OIPSet* as an *InheritanceHierarchy.* 179
FIGURE 63. System for a Design Space Explorer Needs to Communicate with Both Typed Feature Structures and Non-Manifold Geometry. 183
FIGURE 64. Implementation of Geometric Typed Feature Structures Comprises Both Kryos and SHAPES (and its Display Library: Viewer). 184
FIGURE 65. A Generic System Design for a Design Space Explorer Requires a Design Representation. 185
FIGURE 66. An Implementation of Representation: Geometric Typed Feature Structures which Contains Typed Feature Structures and Order Types. 185
FIGURE 67. A Proposed System Comprises two External Components: Kryos and SHAPES. 187
FIGURE 68. A 2D Display of the Schematic Layout Generated by OGS. 211
FIGURE 69. A 3D View of the Abstract Layout: Each Room is Represented as a 3D Region. 212
FIGURE 70. Each Room is a Sub-Cell of a Cell-Complex: $Cell_{sfc_house}$. 213
FIGURE 71. The Wall Continuous Support (Membrane Layer). Outfaces of this Wall from the Geometry of $\mathbf{DU}_{we}$. 217
FIGURE 72. A 3D Display of Four Layers of Wall Enclosures: Sun/Rainscreen, Insulation, Air-Barrier, and Wall Continuous Support (from Left to Right). 219

List of Tables

TABLE 1. A complete design space explorer should support representation, rules, memory and policy in both the system and HCI levels. These are examples of questions applied in the following reviews. 40

TABLE 2. LOOS supports exploration in representation, rule, memory and policy in the system level and only supports exploration in representation and rules in the HCI level. 41

TABLE 3. ABLOOS supports exploration in representation, rules, memory and policy in the system level. It supports partially in representation, rules and policy but memory in the HCI level. 45

TABLE 4. DiscoverForm supports exploration in representation, rules and policy in both the system and HCI levels. 49

TABLE 5. GENESIS mainly supports exploration in representation. There are rules in both the system and the HCI levels 53

TABLE 6. Tartan Worlds only supports exploration in representation and rules in the HCI level. 57

TABLE 7. Grammatica supports exploration partially in representation, rules and policy in the system level but only supports exploration in rules in the HCI level. 61

TABLE 8. SEED-Config supports exploration in representation, rules, memory and policy in both the system and the HCI levels. 65

TABLE 9. Gedit supports exploration in representation and rules in both the system and HCI levels. 70

TABLE 10. SG-CLIPS supports exploration in representation, rules and policy in both the system and HCI levels. 74

TABLE 11. A summary of key lessons learned from the reviews. (system level) 79

TABLE 12. A summary of key lessons learned from the reviews. (HCI level) 80

UNIVERSITY OF ADELAIDE LIBRARY

0.1 Abstract

This thesis explores an extension to the concept of carrying geometric information based on the theory of Typed Feature Structures. We introduce this extension, Geometric Typed Feature Structures, as the focus of this thesis. Geometric Typed Feature Structures covers an important aspect of design space explorers where their symbol level representation carries 3D geometric information. Order Types are the devices in Geometric Typed Feature Structures that carry the continuous infinite domain information—that is, geometry. In this extension, theories and algorithms are applied to two kinds of Order Type examples for carrying numerical values and geometric information.

The thesis uses the concept of *exploration* in *design spaces* by presenting a knowledge level and its corresponding symbol level representation. An informal equation of five concepts illustrates the design space exploration model and its falsification test—design space explorers. Realisations of the exploration model and of design space explorers are undertaken by creating coherent systems at both symbol and the human-computer interface (HCI) levels.

Using a knowledge level (the SEED knowledge level) and its symbol level realisation using Typed Feature Structures, we present a domain extension of Typed Feature Structures for carrying geometric information: Geometric Typed Feature Structures. Geometric Typed Feature Structures uses the types *(Order Types)* as the carriers for geometric information and other continuous domain information. We describe the requirements as well as the conditions in which an Order Type can be specified and synchronised with other domain knowledge.

We show an implementation for our Geometric Typed Feature Structures: Ordering Geometry System. This implementation demonstrates the significance and usefulness of representation in geometry by generating various floor design layouts for a typical Australian house—a single fronted cottage and the building enclosures. Finally, advanced geometric representation in the HCI level is left for future work and further research which lies within the computational design domain.

0.2 Statement

This work contains no material which has been accepted for the award of any other degree or diploma in any university or other tertiary institution and, to the best of my knowledge and belief, contains no material previously published or written by another person, except where due reference has been made in the text.

I give consent to this copy of my thesis, when deposited in the University Library, being available for loan and photocopying.

For my parents: Tsung-Fu Chang and Mei-Chi Li Chang

0.3 Acknowledgements

There are many people who have been generously supportive to this thesis throughout the process of research and writing. They made this thesis possible and, most important of all, made my years cheerful and rewarding. It is impossible to name them all, however, there are few people whom I would like to express my sincere appreciation to.

My most sincere appreciation goes to Associate Professor, Dr. Robert F. Woodbury, my supervisor, who always enriches my research knowledge and has given me strong support constantly throughout the research work during this thesis.

I would like to thank my research colleague Sambit Datta who has contributed to this thesis by engaging in useful discussions and spending time commenting on its various drafts. I also would like to thank Andrew Burrow who provided the Kryos software for supporting the implementation of the thesis and for his extra help with integrating his software with the implementation—Ordering Geometry System. I would like to thank Emma Pearce who generously spent time proof reading. I also would like to thank the group of seedlings on the other side of world (Carnegie-Mellon University): Zeyno Aygen, Sheng-Fen (Nik) Chien, Bongjin Choi, Magd Donia, Rana Sen, Jim Snyder and Wen-Jaw (Jonah) Tsai for their company during the years of SEED project.

In addition, I would like to thank to those who made my days in Adelaide memorable: Peter Scriver and Stan Fung for their endless noodles lunch company and intellectual conversation; Dean and Professor Antony Radford for his encouragement and inspiration; Dan Corbett for his generous friendship; Sonal, Gwenda and their dearest sons (Sagar, Ian and Cailean) for bringing me the joys of life. Special thanks give to Jessica Hsiao-Li Huang and her family who provide all the support I need in the last few months of writing. Without them, this thesis is not possible. I am deeply indebted to my family: my father (Tsung-Fu Chang), my mother (Mei-Chi Li Chang) and my three sisters (Teng-Ping, Pei-Ling and Pei-Cheng) who always have been the constant sources of support and faith on me through out these years, especially my father.

Finally, I would like to acknowledge School of Architecture, Landscape Architecture and Urban Design for supporting me during my Ph.D. study.

1

Introduction and Background

This thesis is the outcome of a personal realisation and exploration in the field of computational design research. It covers from general design points of view, specified domain knowledge and related research background to the proposed implementation and applicable examples. There are also several related research projects illustrated in order to explore their common interests in geometric information and to demonstrate the most needed domain knowledge in terms of the system level and the HCI level.

My original intent and motivation were to focus on 1) how geometric information can be represented in the symbol level and the knowledge level for a design space explorer; and 2) how geometric information and domain knowledge can be synchronised in the system level and the HCI level. They become clear and comprehensive from the analysis of tracing back to the history of computational research literature, emerging diversity of views and gaining experience of working design space explorers that assist human designers to explore design spaces in advanced levels to work on the representation of geometric information which is the core of the computational design research.

This thesis is organised in the manner of hierarchy research thinking and composed in four parts. It starts with related research background and design projects, theoretical and solid background to the implementation and examples which demonstrate the result of a personal exploration in computational design and its contribution to the field of computational design research.

Part I is divided into three chapters:

Chapter 1: "Introduction"

The very first chapter in Part I of this thesis provides a general introduction of computational design research literature and includes several related computational design research topics.

Chapter 2: "Background: From design to exploration"

Chapter two then describes the background from design to exploration. It commences with an account of the diversity of views on the computational paradigm of design research literature. By analysing several dominated views for this computational paradigm, this chapter provides a further understanding of the phenomenology of design through the use of models and metaphors.

Chapter 3: "Directly Supporting exploration—design space explorers"

Chapter three introduces a design space explorer which is the computational system for describing, generating and exploring large design spaces for designs in which its purpose is to gain experience and to learn from using design space explorers and, furthermore, to enhance a deeper understanding of the computational complexity involved. Several research projects are discussed and useful lessons are gained to evoke the realisation of knowledge level with the symbol level representation for domain information—geometric information and domain objects.

1
Introduction

Design encompasses a range of complex mental activities for human designers. This range covers from seeking design goals and creating forms to producing designs in the physical world. The complexity of design makes the design process very time consuming. It brings human designers under rapidly increasing pressure due to time constraints from the requisition of efficiency in professional practice. Over the past thirty years, researchers have attempted several different approaches to this problem. One of the most crucial challenges to be faced is enhancing the capability of design in any aspect for human designers. The approach of computational design research has achieved the further extension of enhancing design through models and implementations. Thus, the potential of computational design research is revealed and discussed.

Computational design research is a special field of design research literature which has been intensively exploring better ways of tackling and approaching designs. The initial vision of using computational systems was to assist human designers to visualise spaces and to draw them efficiently. By taking advantage of the recent rapid increases in the capabilities of computational systems, human designers can reach a more advanced level of design accomplishment in a short period of time. It is more vital than ever before to gain a deeper understanding of design concepts made possible by the interaction between human cognition and computer interpretation. In order to capture the essence of computational design research and the effectiveness of interaction between human and computer, the history of computational research background is now unfolded.

There have been over thirty years of research in the area of computational design, beginning with the development of SketchPad[Sutherland, 1963]. During this time, researchers and designers have endeavoured to characterise design using different perspectives. It is becoming clear that a design environment supported by a computational system is important because it can integrate the complementary nature of human and computer design capabilities. A human designer in conjunction with a computational system can perform in a dynamic, flexible and exploratory manner in design spaces[1]. By utilising the advantages of cooperating human and computer designs, the subsequent designs which may be generated unexpectedly could be beyond the capabilities of the human designer.

There are, however, limitations to the computational design systems currently available. The existing artificial or computational systems for design representation, automated reasoning and problem-solving can only match a very small and limited degree of human design behaviour. It is crucial and inevitable to extend the research domain of the existing computational systems to match a larger degree of human design behaviour. Finding a better approach to a representation of design problems is becoming central to the field of computational design research. A chosen approach is characterised by its terminology and its logic work.

It is not possible to formalise all of the mental activities involved during the design process. A terminology of "model" is introduced to represent an activity which models part of the design phenomena. A model allows us to study and analyse the characteristics of the abstract metaphor for the phenomenon of design. A useful and successful model in the field of computational design research is important as it allows us to express an abstract and partial understanding of intuitive design.

Within the domain of computational design, *exploration* is one of the most widely respected and useful models. It is commonly referred to as the exploration model. The exploration model is used as a prototype for exploring possible alternative designs in design spaces.

1. A design space is defined by the analogy to the problem space of Newell and Simon[Newell and Simon, 1972]. It consists of a representation scheme for designed artifacts, a set of constructive operators on the representation scheme, an initial state of a design problem, a design problem and a set of knowledge which may be applied to a given design problem.

One of the most fundamental concepts of computational design research was derived from the recognition of exploration for design spaces. To demonstrate the practicality and usefulness of exploration for design spaces, we refer to prototype design spaces systems, that is, design space explorers. Design Space Explorers are very important computational systems that fully support design exploration in the design process, however, there still is one issue that has not been successfully dealt with. The issue how design space explorers represent 3D design instances in the physical world deals with. *Geometric information* is the proposed solution to this question of representing 3D design instances in the physical world.

1.1 Overview of dissertation

The overview of this dissertation is presented in the clearly structured manner of research disciplines. Logical and hierarchical thinking are demonstrated through proposing a hypothesis, and by introducing the ultimate goals to be achieved with the proposed theoretical support in the field of computational design research. Potential for further extensions of research and other related research directions will be discussed later in this section.

1.1.1 Hypothesis

How to represent geometric information in design space explorers is still an unresolved research question. Utilising geometric information in a way of representing design instances in the physical world is essential in the field of computational design. According to this fact, the hypothesis of this thesis is that a uniformed treatment for geometric information and domain knowledge is beneficial for the computational design research, particularly for the realisation of design space explorers.

1.1.2 Goal

The hypothesis can only be tested through implementing a design space explorer which can carry geometric information uniformly with domain knowledge. Building a design space explorer allows us to gain experience in using design space explorers and to understand the computational complexity involved. For this purpose, a knowledge level[1] representation, a symbol level formalism and its functioning implementation are created. Within this symbol level formalism, geometric

1. The terms of "knowledge level" and "symbol level" are borrowed from [Newell, 1982] as describing the design process as an ideal knowledge process. A knowledge level has its corresponding symbol level representation.

information and domain knowledge is uniformly represented. However, the functioning implementation for geometry requires a disciplined and efficient way of representing geometric information.

1.1.3 Approach

In order to present geometry in a disciplined manner, a computational model is needed for large design spaces. The exploration model mentioned before needs a high-level representation which we refer to as its knowledge level representation. The realisation of the exploration model needs its symbol level representation. A set of domain concepts developed by SEED, called the *SEED* knowledge level, provides the underlying domain knowledge concepts. A symbol level representation using a formalism called Typed Feature Structures synchronises both domain knowledge and geometric information in a uniformly representation. We call such symbol level representation "Geometric Typed Feature Structures".

1.2 Organisation of dissertation

This dissertation consists of four parts:

Part I: Introduction and background

Chapters 1 and 2 provide an overview of this dissertation together with an introduction to the issues involved in research in the field of computational design. This is followed in the third chapter by a literature review of design space explorers such as Genesis, Tartan Worlds, LOOS, ABLOOS, DiscoverForm, Grammatica and SEED-Config.

Part II: Theories and algorithms

Part II outlines the theories and algorithms behind Geometric Typed Feature Structures. We describe the SEED knowledge level concepts as the domain objects for the computational system developed in this dissertation. Furthermore, a formal representation (Typed Feature Structures) is used and several definitions of its knowledge components and their algorithms are described in Chapter 5. The core theory of this dissertation—Geometric Typed Feature Structures—is described in Chapter 6.

PART III: Implementation and Examples

In the third part of this dissertation we document the system architecture of the Ordering Geometry System (OGS) with an instruction for using the OGS for specifying geometry. This is followed by two examples—a single fronted cottage (Chapter 8) and a building enclosure (Chapter 8.2).

PART IV: Conclusion and future work

The final part of this dissertation summarises the inroads made on current design problems using the working implementation and tools that we have developed and identifies remaining and emerging design problems which may be tackled in a further extension of our research.

2

Background: From design to exploration

Understanding the phenomenology of design has metaphorically been recognised as the basis for developing a characterised view on design. Since the domain of design is so large and the pursuit of individual interests is exclusive, progress in developing diversity of views is rapid. This chapter commences with an account of the diversity of views on the computational paradigm of design research literature. By analysing several dominating views for this computational paradigm, we gain an understanding of the phenomenology of design through the use of models and metaphors.

The potential view is often characterised as a metaphor or a model of design for design spaces. In general, a design space is defined to be the structured set of all solutions to a design problem, where the structure on the set allows one to traverse the space in an orderly fashion. However, the aim of this chapter is not to investigate relations between a particular view and its design spaces. The aim is attempting to understand the diversity of views, to question the importance of having these views, and to discuss the development of dominating views. Furthermore, we seek to introduce the chosen view—state space search.

This chapter gives an introduction to the background of research literature in relation to views of computational design in general. It then focuses in particular on the chosen view of concurrent computational paradigms—that is, *design as* search or *design as* exploration. The discussion includes concerns about particular approaches towards a unified, dynamic and flexible computational model. This model is the exploration model and its implementation is known as the design space explorer.

A design environment based on computational paradigms must take into account aspects of design space for the relationships between the roles of human designers and computers. Such a design environment must justify its design capability and emphasize the significance of its existence by exploring all possible alternative designs within the design spaces.

Observation and investigation of evolutionary design transformation reveal several difficult relationships between different views as well as a minimal common understanding. However, it is not important to seek either a higher degree of common understanding or improved relationships between the different views. The most important challenge is to take one potential view based on motivation of interests and current design goals for utilising geometric information to represent design instances in the physical world. Using this view we can explore the undiscovered implications of design.

2.1 Views on Design

Design research literature is seemingly fragmented into what might be called "views". Each view of design is distinct and has arisen with the development of its terminology and logic work within the domain of individual research. One has only to peruse a contemporary compendium of design research papers, *e.g.* the Proceedings of the *Artificial Intelligence* in Design 1998 (AID98) Conference [Gero and Sudweeks, 1998], to obtain a strong sense of the particular fragmentation in the computational design domain. These papers are mostly self-consistent within the same view. They are usually built upon a body of coherent literature that relates to design views similar to those they espouse, and yet seldom is a source outside that view referenced.

It is not simple to get an overview of a particular field just by reading a few relative papers, but there is still much to be gained from within these relative papers. The difference between the computational field and other design fields is that views do not cohere in the computational field. There seems less to be gained by comparing these relative papers with different views—each view has unique characterised terminology and logic and many different approaches are supported with convincing arguments. Each view clearly has its own logical and theoretical supports for the contribution derived from it. There is no clear basis for having different views in

the same field. On the other hand, the relationships between these views are not readily apparent.

Participating in a conference in computational design highlights the problems inherent in attempting to reconcile different views. Within the conference, much time seems to be spent trying to debate the relative merits of one view versus another and also in trying to understand the views espoused by other researchers. Such debates inevitably distract attention from actual discussion of current research contributions. It seems most desirable to focus solely on one pursuit within a particular view than to devote time to finding relationships between these views in computational design.

Despite the diversity of views, there is still some minimal common understanding of design research. It seems clear within the community that each individual is dealing with a complex enterprise and is constantly involving one or more agents and producing representations of artefacts. This minimal common understanding of design is, however, fundamentally built upon the notion of being goal-directed for a complex enterprise with the assistance of computational apparatus and systems. The notion of being goal-directed is embedded within each view since the ultimate goal of an individual view is to make purposeful and possible decisions in the process of design.

With such widely separated and divergent positions, one often discovers the dilemma of decision making in different views of design. Most progress within the field of computational design is made by presuming a particular position and then playing out the consequences of that presumption. In contrast, very little light is shed on design when we start to compare the views. A pursuit within a view becomes an essential first step for the ultimate goal one sets up to achieve and for the constructive contribution one is determined to make to the community.

2.1.1 Why do we have diversity of views?

The phenomenon of design and the minimal common understanding in design research literature have led to the development of distinctive individual terminology and corresponding logic works. It seems difficult and meaningless to attempt to seek a universal view cutting across the disciplines of different design research. Instead of debating the existence of possible relationships between views or a establishing hierarchy of views, we should be exploiting the range of related information

made possible by the diversity of views. It is essential to choose a particular view to work within. One might wonder why we have such a diversity of views on design and what the advantages of having such a range might be.

Some observations and outcomes of investigations have gone to answering these questions. The answers to both questions are focused particularly on the complexity of human enterprise and the common ground of understanding of views in computational research literature.

The rich history of design research literature can be traced back to Leonardo da Vinci in the fifteenth century and beyond. While what we now know as computational design has roots in this history, it can be viewed as having its own beginnings in several seminal works, for example, modelling programs such as SketchPad[Sutherland, 1963], the grammatical approach[Stiny, 1980], and the mathematics of designs[March and Earl, 1977]. Despite such relatively recent origins, computational designs have dominated design research literature for the last 35 years. Views for computational design are still emerging and displaying distinct tendencies to develop diversely. The notion of mixed-initiation for computational design embraces this diversity of views.

Each view on computational design is distinctive and its terminology is derived in order to characterise the difference from other views. Each view is also considered to originate from its logic work associated with a chosen framework in a particular field. The complexity of the phenomenon of design means that all aspects of computational design could not possibly be described fully by a single view only. Diversity of views is inevitable in attempting to enrich an understanding of design.

As with any complex enterprise created by humans, diversity of views can lead to greater design achievements. At the same time, it can challenge the limitations of human capability. One must, however, take into account the fact that views will change along with corresponding social structures, human needs and other constraints. For instance, a view is developed by focusing on one particular task of computational design. If the initial intention of this view fails to fulfil the latest changes and demands for social structures, human needs and other conditions, this view will be modified, or perhaps a new view will be even derived from it. The initial view of utilising the computer in design research was to use it as a drafting tool for assisting human designers. Recent dramatic technological advances in comput-

ing mean that this view is no longer sufficient or practically useful for updated computational design research performed in a professional role. The view must be modified or a new view must be derived from it in order to accomplish higher level requirements.

The theory of design, in general terms, gives an account of designs or designing that explains the structures of the design process. It also predicts possible outcomes or alternatives. Examples of views on theory are the concepts and knowledge supports for developing terminology and logic work. A conventional view concentrates on seeking solutions for problems in the domain of human intelligence. A comprehensive view for the design process should reflect the nature of design problems but not the particular ways in which a human solves a particular design problem.

Nevertheless, a concise explanation of design intelligence does not currently exist in the boundary of human capabilities. According to the initial design intelligence in computational design research, Newell and Simon's view[Newell and Simon, 1972] of "*human problem solving*" in the design process reflects the design limitations and the ill-structured form of human design performance and not the particular nature of design problems themselves.

It is important to recognise a potential view of utilising computers for assisting human designers in any possible aspect of design capabilities. It is also important to extend this view to envision a useful computational design tool for human designers. Given our interests in the computational domain, it is inevitable that we analyse the essence of views before committing to a pursuit within them. We seek a pursuit within views that provides an appropriate representation at the symbol level as well as a useful implementation.

The rapidity with which technological advances are being made means that current views are constantly being outdated. Researchers must either invent new views or modify those already existing in order to satisfy the demands of the real world. There are currently several dominating views in the field of computational design research. They are views on electronic media such as CAD and database, on constraints, on optimization and on state space search. A brief introduction of each view is given with an insightful analysis of the research framework supporting each view in order to demonstrate the distinctive characteristics of each. The ultimate intention of each view is also identified in the following sections.

2.1.2 What are some of these views?

Here we sample some views on computational design. We make no serious attempt to trace their historical origins, offering instead a snapshot of their current understanding and relevance.

Electronic representation and Database system

One of the dynamic views in computational design is an electronic media specialising in graphical representation. Its terminology is based upon the concept of Computer Aided Architecture Design[Eastman, 1976, Mitchell, 1977]. The contribution of this view is that the abstract design concepts are transformed into electronic representations in the form of database or geometrical computer graphics displayed on an electronic drawing board.

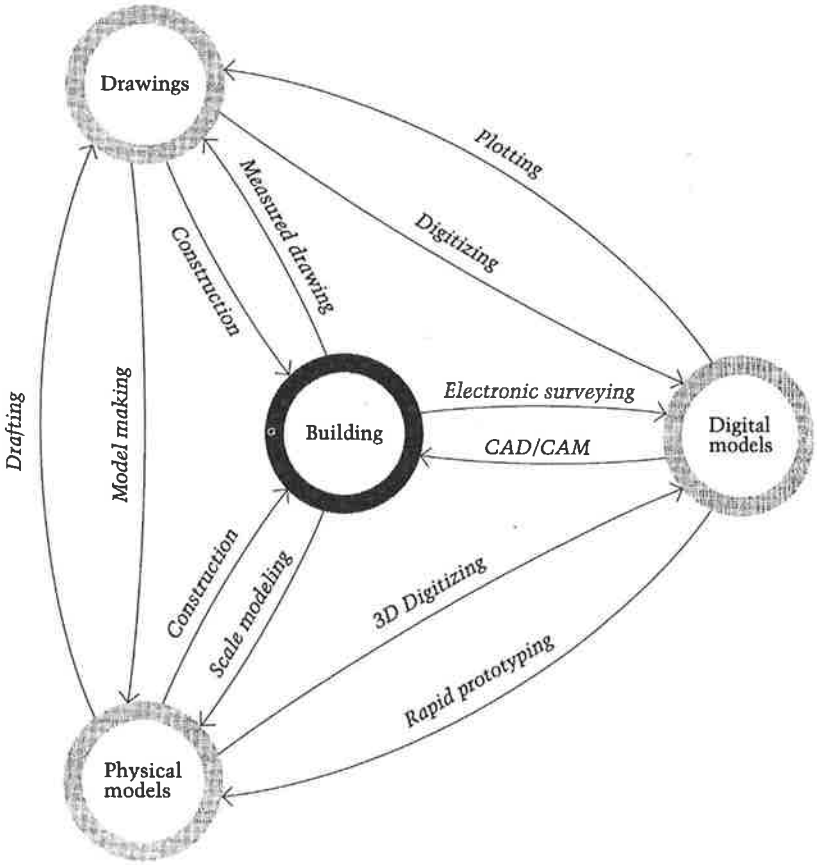


FIGURE 1. Interrelationships in the Design Process from *Digital Design Media*[Mitchell and McCullough, 1995b].

Design which utilises a computational approach for presenting shape information has transformed the process of designs from conventional hand drawing into an electronic drawing representation (a famous diagram done by W. Mitchell is shown in Figure 1). This electronic representation often involves computational operations such as transformation and the combination of different shapes in a two-dimensional drawing or in a three-dimensional geometric model. This view of using electronic media for representation takes an initial vocabulary of shapes integrated with a repertoire of shape transformation and combination operators [Mitchell, 1990b]. This initial vocabulary of shapes is established with shape algebra within the capability of computational design.

The earliest concept of utilising a computer drafting system is focused on assisting human designers to visualise shapes as spaces. Further approaches for operating shapes, such as rotation, translation, reflection and others advanced altered shape commands become formalisation and essential for computational implementation of a shape algebra later on. Such ideas are the fundamental basis for Ivan Sutherland's SketchPad system, released in 1963[Sutherland, 1963], which is regarded as the initial landmark for computational design research. Sutherland suggested that "the usefulness of computer drawings is precisely their structured nature". Such nature, he claimed, "is critically dependent upon the topological and geometric structure built up in the computer memory as a result of drawing operations"[Sutherland, 1975]. Traditional drawings, on the other hand, have no inherent structure and have unwanted black marks.

The graphical representation generated by electronic media also relies on the database system which stores the computation of shape information. This database system also plays the role of maintaining data over time, distributing the data information spatially and documenting sufficiently up-to-date information. The database system is mainly developed and organised to support the electronic representation of shapes at successive stages of the design process. Extended research of database systems has been undertaken and this research has been implemented in the integrated design environment[1] [Codd, 1970, Date, 1990, Katz, 1990, Mitchell

1. An integrated computer-aided design environment is one in which data files pertinent to a project, software tools and physical devices can be efficiently put to many different uses—by different design team members, at different places at different design stages in the design process.

and McCullough, 1995a] in which the advanced flow of data has been systematically organised to support any electronic representation by computational systems.

Constraints

The view on design with constraints is about demonstrating and exploring the computability of design. The concept of designing with constraints has been stated by Gross [Gross, 1986, Gross et al., 1987] to be the process of defining an initially ill-defined question or problem. It also proposes and examines possible solutions or answers. A description of the term of an ill-defined question or problem will be given later. The notion of designing with constraints is developed upon the process of seeking *a solution* or *possible solutions* to one particular problem. The view with constraints integrated with the notion of computational design is stated to be a metaphor for a "*constraint model*" which focuses on articulating given questions and on exploring possible alternative design solutions by utilising computers.

Its logic work is oriented towards explaining the design questions and towards specifying solutions to these questions of search within a state. Gross[Gross et al., 1987] has implemented a prototype Computer-Aided Design(CAD) system built around the concepts of constraint maintenance. He defines the view on design to be an "invention to a set of specifications" or as the theory of designing to account. The theory for such a view on design is ambiguous since "the result of designing is difference between the specific charged and the details delivered." Gross also suggested that this view on design represents the notion of an inner logic—both to the human designer and to the problem—where an ill-defined question is driven by a reason there can be as many designs as there are designers.

The view of designing with constraints was developed in the earliest years of computational design research. Important research which has been based upon this view includes Sutherland[Sutherland, 1963], Steele and Sussman[Steele and Sussman, 1980] and Borning[Borning, 1981]. All three use constraints to manage and analyse complex problems within computational design.

Optimization

A different approach of views for computational design is Optimization. This view takes advantages of the premier techniques in CAD by utilising the mathematical tools of optimization theory, in which calculations and comparisons of design solutions depend on different constraints[Cohon, 1978, Radford, 1981, Gero and

Radford, 1984]. The characterised terminology is determined by the notion that computers are used to prescribe a set of design decisions to accomplish a particular goal. The logic work has been developed by Radford and Gero[Radford and Gero, 1988] with the ultimate goal being to provide potential for better designs by subsuming wider selections and a diversity of opportunities.

The techniques provided by the development of Optimization give human designers insight into the nature and efficiency of good designs with a computational system. An abstract model of design is utilised to present the possible role of Optimization in relation to the general notion of design (the diagram is shown in Figure 2). The components of this abstract model of design are three characteristic phases—problem analysis, design synthesis and design evaluation. The key feature of this model is that the relationships between these three phases are not direct but cyclical[Radford and Gero, 1988, Snodgrass and Coyne, 1991][1]. However, the disadvantage of this model is that a comprehensive understanding cannot be provided for the reasoning involved in how a decision should be taken from an initial design. In brief, Optimization can only provide us with a useful model of design synthesis that can indicate how and on which basis design decisions are made.

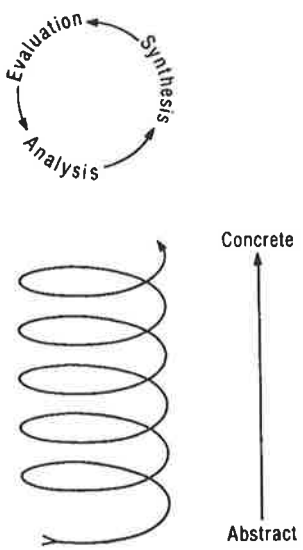


FIGURE 2. An Abstract Interpretation of three Phases in the Design Process[Radford and Gero, 1988].

1. Such models can be traced back further to, for example, Asimov's model[Asimov, 1962], which did not fully take the optimization viewpoint.

Optimization, as a model for creating information, forms a relationship between the performance in design goals and the result of decision making in the performance. This design model of Optimization takes the notion of design synthesis as the basis for the making of decisions within a language and within constraints. Given external information, this model (as illustrated in Figure 3) creates a link between possible decisions and their outcomes. A crucial link is between the desired decisions and the corresponding performance in specific design goals. The view of Optimization is focused on mapping design decisions and design performances along with modelling characteristics and any information for the required decision making.

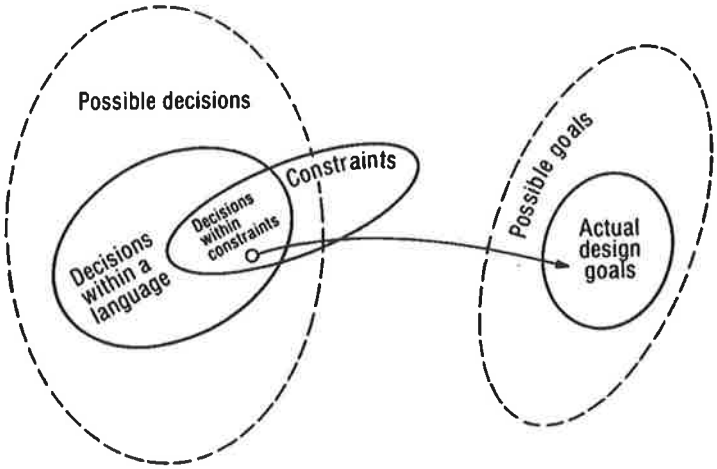


FIGURE 3. The Relationships Between Decisions with Constraints and Ideal Goals[Radford and Gero, 1988].

Optimization models in computational design research have demonstrated the effectiveness of this technical discipline for search*ing* a design field for possible alternative solutions. In addition, they simultaneously identify the best solutions suited to the human designer's ultimate goals. The direct approaches of Optimization provide a possible answer to the human designer's fundamental question of how to find the best solution for an individual design. However, the disadvantage of Optimization is that it imposes the inherently impossible task of formulating meaningful objectives in a discipline characterised by multiple and ill-defined objectives[Radford and Gero, 1988].

State space search

The terminology of the fourth view, that of state space search, was initially generated from the recognition that design spaces may possibly be infinite. In terms of formalised description, such a view has often been concerned with the rules and constraints involved in tracing out a design state (as shown in Figure 4) within a design space. The logic work of state space search utilises the rational and organised movement of search as a computational guided tool to reach the goal state in design space. The logic work also focuses on searching for relationships between generated design states and their edge[1] structures (the rule applications) of the design space. Modifications and refinements of the requirements within a state and within subsequent states are often changed by the rules. Often the modifications and refinements are determined and transformed into different design states by the rules and constraints applied. Further developments based on this view exert a significant influence on the view which we have chosen to work with later in this dissertation.

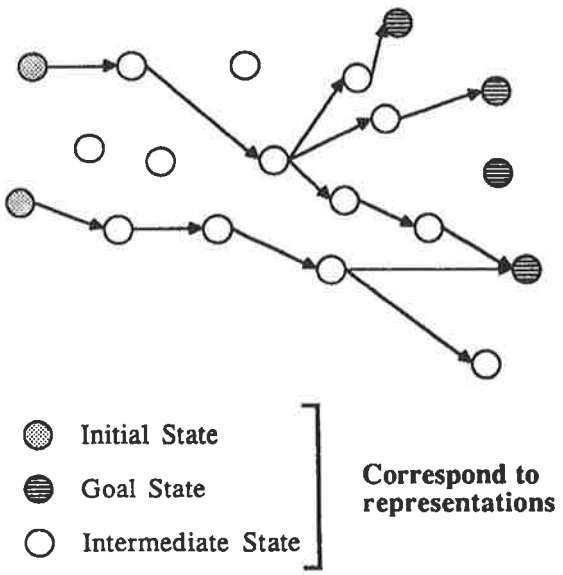


FIGURE 4. Possibilities of a Derivation Graph in the Design States[Woodbury, 1991a].

1. Design states can be described graphically—each design state is a node in the graph and the derivation association between any two states is defined to be the edge structure.

The earliest idea of state space search as a process of design was introduced by Herbert Simon in 1980 who defined this expression in *The Science of the Artificial*[Simon, 1980]. In this book, Simon suggested that the possession of effective heuristics for search*ing* was a significant part of design competence. An early precedent of such a notion was presented by Simon in a paper called "*Style in Design*"[Simon, 1971], in which he questioned the phenomenon of design and the behaviour of the thinking process for human designers, giving such conclusions as "*he solves his problem by moving through a large combinatorial space in which he adds one element after another to his design... The richness of the combinatorial space in which the problem solver moves... is the hallmark of design creativity.*"

The important aspects of state space search are both formalisible and computable. Recent computational design research literature covers this formal description of representations and search spaces, provides algorithms for rule applications and focuses in particular on rule-based generation of a corpora of designs. Evidence that a working computational implementation of a state space search is indeed achievable is provided (along with the corresponding code) in the later chapters of this dissertation.

A further development of the view of state space search is also seen in the work of Rob Woodbury[Woodbury, 1987]. He considers state space search to be a useful design model in which one or more agents could trace out or discover a space of designs by applying design-transforming operators. These design-transforming operators are stated to express currently existing designs as well as examining existing designs against the design criteria.

The views briefly summarised above are all well recognised within the domain of computational-based research. Although there is no common design discipline cutting across these different views, there is still a minimal common understanding which embraces them all within the field of computational design research. To that extent, there is no strong evidence to suggest that any one view is superior to any other.

2.1.3 Why and how to choose a view?

The crucial factor for pursuing a specific view must be concentrated on an assimilation of the common understanding behind all views. It also has to avoid the distraction of attempting to find relationships between different views as discussed in

the previous section. A clear vision of perspective for a specific view is determined by recognising the potential of one's interests and goals. A specific view, on the other hand, must necessarily be carefully chosen to assist and cohere with one's pursuit of specialised interests and goals.

Questioning the usefulness of contributions made by a chosen view is central to this dissertation. The view on which we shall focus is that of state space search in which the issue of exploration is centred. The reasoning behind this choice is given in Chapter 2.2.3 along with some basic engineering background for a design model called the exploration model. Recalling the ultimate goal and admitting one specified view on design we shall demonstrate a comprehension of design based on a derivation of state spaces in the design process.

2.2 *State space search* is a useful view

A pursuit within a view can only be chosen based on the initial intention and interests of this dissertation—understanding design with computational assistance and devising computer tools which support design. Following this logic, state space search clearly promises to be an appealing choice given its precise approach to design space*s* and given also the broad selection of related design research literature available.

There are many benefits in using the view of state space search. The actual process of design, for example, can be discussed in terms of a state space search; the design process can be an informal description of searching for a solution (design) in a state space[1]. Design researchers often use the term search as a metaphor for design, thus transforming the ambiguity of understanding design into characterised concepts and rules for solutions in a design state (referred to here simply as a *state*). Understanding design derivations and state transitions can lead to a deeper and more constructive understanding of the process of design itself (and thus of architecture) in a *state space*.

Furthermore, the body of literature on state spaces is extensive. The literature covers many facets of computational design—ranging from representation of one or more states to transitions between states and large scale state spaces. In addition, it

1. State space may be defined informally as a space which comprises states connected via a derivation relation.

also covers to a lesser extent the interactions between such spaces. To be able to draw on the expertise and accumulated knowledge of so many experienced researchers through their publications and computer packages can only benefit a researcher working in this field.

2.2.1 The origins of state space search

The origins of state space search in design go back to the 1960's when Simon [Simon, 1969, Simon, 1973] conducted empirical research which emphasized design problem solving. As pointed out by Schon[Schon, 1983], using scientific knowledge and technical skills to "*solve*" a broad range of "*problems*", and "*problem solving*", was the central agenda of this research. In the classic "*The Sciences of the Artificial*", Simon applied the model of "*problem solving*" to design problems and described the logic of design as "*finding alternatives*". This model of "*problem solving*" was to become the core of state space search. Later, Simon also discussed the features of ill-structured problems (design problems) in "*The structure of ill-structured problems*" [Simon, 1973], as well as in "*Human Problem Solving*" [Newell and Simon, 1972].

In brief, a theory of human problem solving that constitutes a basis for a conceptual framework is proposed by Newell and Simon in "*Human Problem Solving*" [Newell and Simon, 1972]. Such a theory of human problem solving is comprised of five major states. These are:

1. an "*information processing system*" (IPS);
2. a *problem space*;
3. using search as a metaphor for problem solving;
4. being influenced by a *task environment*; and
5. being supported by an external *memory*.

The human Information Process System (IPS) can be modelled as a group of numerous memories acted upon by the processors, which process information from sensory devices and perform physically in the real world through motor devices. This is the inherent structure of the IPS. Significant actions are features of the IPS as a whole which can affect the ability of the IPS to engage in problem solving. Such behaviour can be characterised as "search in a problem space" where this so-called problem space consists of the five constituent invariants proposed by Newell and Simon[Newell and Simon, 1972]. These are a set of symbol structures, a set of

operators, an initial state knowledge, a problem and a set of knowledge. These invariants are considered to be sufficient formalism to explain the complex and ill-structured domain of design[Simon, 1973].

An example of a human IPS constraining search with certain characteristics, such as the utility of depth first search for seeking information (such as data or documents) in a hierarchical directory system, was proposed by Akin Omer in 1983[Akin et al., 1983]. A task environment for problem solving is defined to be the physical world in which the performance of problem solving occurs. This task environment often provides information which may determine the type of search which should be used for problem solving. Searching with a particular task in mind can model the behaviour of a human problem solver who is often supplemented by an external memory.

Combining all these five invariants, the human IPS, the use of a problem space, the performance of search, the constraints of a particular problem solving task and the existence of an external memory device, provides a framework for comprehension of a given problem solving domain.

Such a framework provides a basic theory for modelling higher level design behaviours such as problem reconstructions and stages. Furthermore, based on the theory of human problem solving, researchers have performed empirical studies on this framework as well as exploring the corresponding design domain [Akin et al., 1983, Woodbury, 1984]. The framework and representation for the view of state space search have been transformed into a search-based model, sometimes also referred to as an exploration-based model[Archea, 1987, Woodbury, 1987, Smithers, 1992, Gero, 1994] which we shall describe in the following section.

2.2.2 The state space search model

This state space search modelling definition of the process of design has been through a significant transition since its origins as a view of design as search [Newell and Simon, 1963, Newell and Simon, 1972, Newell and Simon, 1976, Simon, 1980, Woodbury, 1984, Woodbury, 1985, Woodbury, 1987, Woodbury, 1991a] to the current extended view of search which is design as exploration[Smithers, 1992, Woodbury, 1991b].

Within the model, an acclaimed extension to the search metaphor is the concept of "*puzzle making*"[Archea, 1987]. Using the exploration metaphor, Archea argues that the design process is more like making a puzzle. By investigating the constraints and limitations of a unique design situation, designers can search for a possible solution for that situation without having knowledge of what the result will be. Smithers extends this concept by arguing that searching for solutions to design problems is not only a process of *puzzle-making* but also a process of *puzzle-solving*[Smithers, 1992]. Smithers also claims that an important characteristic of the design process for creating a problem is the revision of the initial requirement description as the process takes place. In other words, a fundamental and essential aspect of the design process is the transformation or interpretation of the initial description into a well-defined problem statement. This aspect is not explicitly dealt with in Woodbury's early work [Woodbury, 1991b], but is refined in his later work[Woodbury et al., 1998]. Solutions do, however, eventually have to fulfil the originally specified needs of the design problem. Smithers, on the other hand, extends the concept of "*puzzle-making*" to create a design model which focuses on "design as exploration" [Smithers, 1992], thus emphasizing the nature of the design process.

By using exploration as a metaphor, solving design problems becomes transformed into a process of defining a problem which satisfies the design requirement that motivated the problem in the first place. Smithers declares that the view of design as exploration is a discovery which concatenates the creation and solution of a problem—a feat which he claims can not be achieved by only using the search metaphor. However, Simon had written twenty years earlier in "Human Problem Solving", that "*using goals has real consequences for the structure of the IPS. If there are goals, then, the problem must contain processes for creating goals, test them, updating them, selecting methods for attempting them, evoking them, discarding them, and so on. Thus, a number of detailed specifications must be met by any IPS that represents human activity directed goals*" [Newell and Simon, 1972]. Changing goals during the search process is equivalent to conducting the exploration process. Thus, the exploration view is in fact addressed in the framework of the search-base model.

In brief, the so-called search metaphor, defined in Woodbury's early work, requires that the structure and relations of a problem space are known *a priori*. On the other hand, exploration can be characterised as a process to elaborate and refine the prob-

lem spaces, and thus, exploration can be treated as a meta-search. The creation or expansion of new states can be described as an exploration process. The subsequent finding of a solution in the defined state spaces can be characterised as a search process. In other words, the difference between these two metaphors is insignificant, and they seem to complement each other in a more unified view. We shall hereafter use the terms search and exploration interchangeably in this dissertation. On the other hand, the terminology for the exploration-based model called the "exploration model" used in this dissertation with one unified meaning of search and exploration, is espoused for the design spaces.

2.2.3 A description of the exploration model

For design researchers working within the view of state space search, the knowledge level theory[Newell, 1982] is of prime importance. However, knowledge level theory is significant not only to researchers working within a particular view but also to the general practice of designing and engineering knowledge-based systems. We shall now give a concise outline of the exploration model and in so doing will give an exposition of the use of the related knowledge level theory in the design process.

Several studies on design models give a formal treatment of the exploration model. No specialised study currently exists which can completely model the process of design. However, in the area of knowledge level theory at least, the exploration model is more thorough and performs better among the design models.

Woodbury develops a concise equation[Woodbury, 1996] which neatly encapsulates the approaches integral to the exploration model. The following equation accounts for exploration with four major structures and views exploration as the action of a set of *operators* on a *representation* all being guided by a *search strategy* and *memory*[Woodbury, 1992, Burrow and Woodbury, 1998, Woodbury et al., 1998].This characterised set of operators is made up of representation, rules, memory and policy. Each individual term recapitulates the compulsory work within the framework of design exploration.

We shall now give a detailed description of the characterised set of operators of the exploration model. We note that these operators are seen on the left-hand side of Woodbury's equation, illustrated in Figure 5

representation + rules + memory + policy = *search* ≡ *exploration*

FIGURE 5. Woodbury's Informal Equation for the Exploration Model[Woodbury, 1996].

Design spaces are regarded as being formal and systematic. However, their undiscovered structure can be dynamically altered at any time and external conditions may be applied by human designers who can change requirements and transformation as well as goal structure.

A *representation* is a formal symbolic language for the concepts used in some design domain. *Rules* are the transformations expressed with respect to this formal language and these comprise two parts: a precedent and a consequent, in which the precedent is a set of conditions to be matched and the consequent is a description of the changes that *rules* make to a *representation.* By performing on an initial design, *rules* often trace out a space of designs. This suggests that the rules are changeable and this recognition implies in turn that exploration systems in design spaces are open-ended. In other words, simply applying a rule to a design state in various ways can lead to a potentially infinite number of design spaces.

Memory holds the designs in a space that has been found by applying *rules*. The designs in a space are also defined by the *rules*. A search *policy* sets a guideline for all machinery (including humans). This is regarded as a critical intersection between human designers and machine control in the design spaces where the forms of information processing are inherently complementary.

In addition, the exploration model serves as a mechanism for organising, representing and expressing our understanding of design as a kind of exploration-based behaviour. It is also about exploring the problem definitions and then searching for alternatives which satisfy the problem definitions in some state space.

The exploration model of computational design was developed in order to understand the knowledge process and treats the design process as a knowledge-based

exploration task. Describing design knowledge is an exploration as much as is the actual design process. Human designers often build up their domain knowledge base progressively as the design exploration proceeds. Utilised in the exploration model is an existing cognition of a design prototype for exploring possible alternative solutions to the ill-structured problems in design spaces. The exploration model, however, needs to take a further extension in exploring design spaces. This is why the symbol level representation of geometric objects needs to be included during the process of exploration rather than after the exploration takes place.

The main intention of such an expressive equation is to describe the internal logic work of the search or exploration paradigm for a specified computational implementation described later.

2.2.4 Reification

Why reify?

To convert a theory into a concrete understanding of the phenomena this theory describes, a reification of this theory is needed. In order for reification of the theory behind a particular view of design to take place, a representation of the implementation for this model is needed, just as any other scientific research requires experimental justification for proposed models. The process of reifying a view of design is defined as being the building of a useful tool using a representation based on the model of that view. Given the nature of the exploration metaphor, such reification is even more important for the usability of such a design model (and hence for the corresponding views of design). The reason for this is straightforward. As stated before, views of design are diverse and hard to relate to each other. Without designers proving the usability of a model, such a view of design will contribute nothing to the design knowledge domain.

Additionally, a model needs its representation to justify itself, and a representation which supports exploration can prove nothing until it has been successfully implemented. The role of representation is vital for a model because it enables the abstract concepts of the model to be transformed into two or three-dimensional geometric objects. Without an implementation to utilise an intended representation, it is also hard to justify the capability of this representation and to demonstrate the capabilities of the model itself. We have, therefore, shown that reification is a needed

step for any computational design researcher who seeks to create a new model or to modify or even extend an existing one.

How to reify?

To find an answer to the question of how to go about reifying a design model, the role of reification in the real world needs to be considered. As we have described previously, the design process is a mixed initiative process which embraces both dynamic and complex behaviours. One would not be able to fully understand the impact of a view of design without actually using a tool based on that view. Thus one cannot only seek to create purely theoretical or abstract designs but must also seek to change or to modify existing designs as a result of what one learns from reification.

The motivation for reification is that should such a tool prove to be useful to the designers then this suggests that the design representation and the model which generated the tool may well also prove to be intrinsically valuable. We see that building a computational system which can perform state space search is the falsification test for the exploration model. Such a computational system is known as a "design space explorer". Details about design space explorers as well as a possible implementation of such an explorer are given in the next chapter.

3

Directly Supporting exploration—design space explorers

As a reification device for the exploration model, design space explorers are computational systems for describing, generating and exploring large design spaces for designs. The purpose of building a design space explorer is to gain experience in using design space explorers and to enhance a deeper understanding of the computational complexity involved. For this purpose, a knowledge level representation, a symbol level formalism and its functioning implementation are created to represent the design instances. Much is gained from actual experience with a design space explorer. This is why we emphasize implementation in the research process, as it is the way to gain meaningful design space explorer experience and, furthermore, to discover the needs for representing design instances in geometry.

We now review the design space explorers that have been reported as having actual implementations. We stress what we learn from the implementations and what we experience with them. Before these reviews, we sketch the components of design space explorers and describe their inter-relationships.

3.1 Components of design space explorers

Given the informal equation of Figure 5 shown previously, components of design space explorers include 1)a state representation (representation); 2) the mechanism to generate and refine the design representation(rules); 3) instances of design derivation(memory); and 4) rule control strategy and navigational assistance around the explored states(policy). Each component presents an important characteristic of design space explorers and is connected to each of the other components by a distinct relationship.

A computational system is generally discussed in two levels—the system level and the human-computer-interface (HCI) level. When designing a computational system, one usually separates the characteristics of the system and of its human-computer-interface for the simple purpose of modularisation. Implementing a meaningful and useful design space explorer should thus provide each component in both the *system level* and the *HCI level.*

The system level provides the formal mechanisms for representing, creating and manipulating design states and the space they occupy. The HCI level supports a user or users accessing, viewing and working with design instances along with the system level. The following paragraphs will describe briefly the four components of design space explorers and their issues in both the system level and the HCI level. These components are representation, rule, memory and policy.

Representation

At the system level, two common issues that need to be addressed in this component are 1) representation formalism and 2) geometric information. The first issue is simply a description of the formalism used. The second issue needs to be mentioned is because geometric information is one of the most common and required characteristics of design representation (representation) in the design domain. Most research projects have used 2D geometric information in order to reduce the complexity of both theory and implementation. An expected hope is that the result of using 2D geometry might be applicable to 3D geometric objects which can present the virtual simulation of design instances in the physical world. Such intricate transformation is not usually smooth and sometimes may be impossible.

Woodbury and Oppenheim suggested [Woodbury and Oppenheim, 1988a] that the process of how to "reason" with geometric information in conjunction with other information is the key for design representation. Historically, most formal knowledge representation work has been done for the language domain, in which atomic symbols and simple symbol structures suffice. Early design researchers used these knowledge representations as a whole. When they encountered a need for representing geometric information, they simply pointed to it. The purported "reasoning" processes supported by the knowledge representation stopped when the needed geometry was encountered. One identified key research problem, therefore, is to make geometry-equivalent information uniform with other design representation according to the representation used. In addition, Woodbury and Oppenheim

stressed a need for a capability for expressing abstractions of geometric objects as well as constraints over these objects.

At the HCI level, two issues are addressed: *visualisation* and *manipulation*. Visualisation of representation is a crucial issue in realising design space explorers. The simple reason is because one can not explore a design space without knowing what one is going to explore. Another issue in the visualisation of representation is to have strongly visual feedback that allows users to understand the exploring effects. Direct manipulation is also added as an important issue because certain design behaviours are easily represented by it. For example, it is easier to draw a line by dragging the mouse to point at two end-points on the screen than by writing a textual description of drawing behaviours. The visual impacts of direct manipulation over representation will benefit the exploration process greatly.

Rules

The mechanisms (rules) which generate and refine the design state from an existing state are the main constructers in design space explorers. Furthermore, rule application provides a mixed initiative framework for interaction in design space exploration. In exploration, the incremental process to resolve conflicts between design problems is more important to a designer than the set of solutions that can be resolved autonomously by some computational systems. Efficiency of recursive enumeration in finding alternatives is not an issue, since non-determinism in search can be resolved by human interaction. The other two reasons for wanting a mixed initiative environment are because 1) selection of alternatives depends on some intangible and/or incomplete criteria and 2) there exist design spaces so big that it is not possible to compute in them automatically.

It is hard to understand what rules are going to produce. A user is helped by 1) visualising rules; 2) visualising their matches; and 3) working forward possibly to many levels of results in the HCI level. For supporting rules in the HCI level, a visualisation of rules and their applications should be informative for comprehending the concepts of rules and should be supportive for providing rule controls (as policy component described later). A different approach is also proposed in which there are structured rules and a means is provided for decomposing these rules later

Memory

Designs are arranged in spaces. Understanding the logic of arrangement is the key to design space explorers. The third component (memory) in turn provides an electronic storage for recording design spaces. Most extant work has represented the design spaces as a derivation tree. For example, Carlson [CARLSON91] has implied this concept to higher aspirations.

Another rich structure allows indexing/retrieval of the design instances in an existing design process or in different design spaces. In this structure, a design state stored in memory can then be classified and applied to a different state later. In this aspect, Flemming [Flemming, 1994] describes an approach called "*case*"-base reasoning which is a method for designs that can be retrieved and refined later while still preserving the relations in a design space.

The issues in the HCI level are related to the visual understanding of design spaces. These include *visualisation* and *primary movement* of design spaces. Design spaces are incomprehensibly large and possibly infinite. Consequently, a visual assistant for a visual understanding of the design space plays an important role in state space search.

Policy

In the system level, three issues need to be addressed. These are 1) rule control mechanism; 2) the mixed initiative support and 3) the parallel search support. Rule control mechanisms in the system level provide the control mechanism to incrementally activate the rules and to be able to choose which rule can be applied to which state. In extending this control strategy to the design spaces, a support mixed initiative environment is also an important issue of policy. When multiple agents are search*ing* the same design space, the controlling strategy among multiple agents becomes another complex issue.

In the HCI level, interaction with the rule control mechanism is the key to exploration-assistant tools for the user. Such interaction will allow users to decide which set of rules should be applied, to know what these matched sets of rule application are, and to control how to apply selected rules. In a possibly infinitely large space such as a design space, an informative navigation assistant is very beneficial in helping users to move around an explored design space. A control assistant (either automatic or manual) is also an essential issue for policy.

Summary

A design space explorer is a device that presents goals, operators and design possibilities. It organises design states into spaces, provides means to access states and applies operators to extend the explored spaces. Design space explorers must present a mixed initiated environment based on the explanation of unfolding design spaces. In addition, the process of design is similar to the process of goal-oriented problem solving, except that the goals of design can change during the design process.

3.2 Several implementations of design space explorers

In this section, we review several research projects which have yielded implementations of design space explorers. Most projects only address a subset of the issues we described above, but all contribute important lessons to the enterprise of design space explorers.

The design representation and implementation that each project uses within the framework of design space explorers are analysed. At the end of each project we emphasize the substantial lessons gained from each project's particular contribution as a completed implementation of a design space explorer in order to propose emerging implementation for this dissertation. A completed version of the design space explorer is discussed in two levels in conjunction with four components. These are the system and HCI levels in conjunction with representation, rule, memory and policy. Critical criteria of each component in each level are presented through questions (as shown in Table 1). Given the constraints and focused interests of each project, some questions will not be tackled due to inapplicability. These unanswered questions will not be part of the structure of discussion framework presented in the following.

We include in this review only systems that provide significant insight on either memory or policy at the system or HCI level. This excludes such noteworthy work as the simple interpreter [Gips, 1975], shape grammar interpreter [Krishnamurti, 1982][Krishnamurti and Giraud, 1986], and shape grammar system [Chase, 1989].

A systematic and organised chart (as shown in Table 1) is presented to identify the specified questions corresponding to each component in both levels.

	representation	rule	memory	policy
system level	**1.** What is the representation? **2.** How is geometric information handled? **3.** How do we integrate geometric information and rule mechanism?	**1.** What are the descriptions of the rules? **2.** How do we generate new states using the rules? **3.** How do we control rule application?	**1.** What is the structure of a design space? **2.** How can we navigate in a design space?	**1.** What are the control strategies (of rules)? **2.** Is mixed initiative supported? **3.** Are multiple agents supported?
HCI level	**1.** How do we visualise the representation? **2.** How do we manipulate the representation?	**1.** How do we visualise the rules and their application? **2.** How do we author rules?	**1.** How does a user view the design space? **2.** How does a user move in the design space?	**1.** How do users interact with rule control strategies? **2.** What are the navigation assistants?

TABLE 1. A complete design space explorer should support representation, rules, memory and policy in both the system and HCI levels. These are examples of questions applied in the following reviews.

3.2.1 LOOS

	representation	rule	memory	policy
system level	**1.** 2D rectangles. **2.** Basic spatial relations between 2D rectangles.	**1.** A set of generation rules. **2.** Sequential alternative layout structures.	**1.** A complex derivation tree. **2.** goto-parent-state, goto-children-state and previous-alternative/next-alternative	**1.** A "tester" as an evaluation and control device. **2.** MI is supported. **3.** No Support
HCI level	**1.** 2D display. **2.** Generation of layouts.	**1.** Visual impacts on the current layout. **2.** Same as above. **3.** No Support	**1.** No Support **2.** No Support	**1.** "Controller" **2.** No Support

TABLE 2. LOOS supports exploration in representation, rule, memory and policy in the system level and only supports exploration in representation and rules in the HCI level.

Background

The approach of LOOS[Flemming et al., 1986] and its following project ABLOOS[Coyne and Flemming, 1990] was initially developed around a generative system based on a 2D rectangle representation. LOOS originates a mathematical equation which models schematic layout. An extension of LOOS, ABLOOS develops a framework which represents and manipulates decompositional abstraction levels and hierarchies. The system of LOOS/ABLOOS concentrates on problems of layout designs involving the many typical complex issues relating to the design of artifacts that need to satisfy some specific constraints. The layout design in the literature expects that computers can help in the generation and evaluation of acceptable spatial arrangements for displaying a complex pattern of trade-offs. Such trade-offs can actually accomplish some of the specified constraints imposed by the limitations of human capabilities on the performance of designers [Mitchell et al., 1976, Galle, 1986].

The layout design is an interesting domain for exploring a desirable structured method for creating multiple alternatives. The restriction to two dimensional rectangles provides chances for simplification without abandoning all practical relevance. On the other hand, the functional decomposition mechanism that LOOS/ABLOOS is using indicates a generic level of tasks that covers most disciplines and sub-levels of complexities.

system level

- Representation

  Representation formalism underlies LOOS presumes the formation of 2D rectangles into a directed graph. Each representation specifies discrete spatial relationships between 2D rectangles and suppresses continuous dimensional properties. A set of feasible layouts is divided into finite subsets with a common structure that can be enumerated via the generation rules of LOOS.

  Geometric information utilises basic spatial relations such as above and below, to the right of and to the left of, to define the structure of topological relations between pairs of 2D rectangles.

- Rules

  Rule formalism provided by LOOS is a set of generation *rules* that can be used for inserting rectangles into a structure one at a time. These *rules* specialise constructing sequentially alternative layout structures for a given number of rectangles.

  Rule application will generate a new design (state) automatically once the *rules* are applied. However, these generative *rules* are fixed and implemented with a presumption on the 2D rectangles representation.

  Intervention points are arranged in an evaluation stage after the layout has been generated. Given the generate-and-test paradigm, rule*s* are mainly the devices for generating layouts and testing by either a human or a machine tester.

  Geometry in rule*s* in LOOS presumes the 2D rectangle as the representation. The topology of a layout is defined, based on the basic fixed spatial relations "above", "below", "to the right of" and "to the left of".

- Memory

  Design space structure in LOOS is a complex derivation tree where LOOS follows the generate-and-test paradigm mentioned above. LOOS avoids any uncertain performance predictions by maintaining a current best set of alternatives and expanding in the minimal way of next round exploration. Even new states are generated during the exploration process and formed a hierarchical resolu-

tion result so that the generation process flows are revealed through LOOS supporting the multiple states.

Primitive movement in LOOS provides goto-parent-state, goto-children-state and previous-alternative/next-alternative.

- Policy

Control strategies are provided by LOOS using a "tester" as an evaluation and control device. The actions of a "tester" are, while the system is observed in operation, are stimulated by its mistakes to add to and to change or to withdraw the evaluation criteria. The role of users, as domain experts, often performs as a semantic filter (testers) which provides the evaluation criteria and discriminates between the solution alternatives.

Mixed initiative is supported by the generate-and-test paradigm. With the generate-and-test paradigm, a schematic layout generated by LOOS is either incomplete, which allows later elaboration, or complete that satisfies all the criteria. An incomplete layout defines a partially realised design space, and can be re-initialised by revisiting a different state.

Multiple agents are not supported in LOOS.

HCI level

- Representation

Representation visualisation in LOOS is through a simple 2D display which illustrates the generated layouts. Since LOOS representation is 2D rectangles, the visualisation of these rectangles is trivial.

Representation manipulation in LOOS is only via the generation process. The manipulation over the representation is about generation of layouts. In this version of LOOS[Flemming et al., 1986], there is no direct manipulation at the HCI level.

- Rules

Rule visualisation in LOOS is effected through its visual impacts on the current layout. By requesting another alternative layout, the 2D display reflects the result of generation immediately. The visualisation of rule*s* is equivalent to the rule

application visualisation, and the explored layouts are the targets for representation visualisation.

Rule application visualisation is supported in LOOS as stated above.

Rule authoring at the HCI level is not possible since the generation rule*s* in LOOS represent the domain specific knowledge, they are well defined in the system and need no further changes.

- memory

Design space visualisation is not completed in LOOS as seen in [Flemming et al., 1986].

Primitive movement in LOOS provides most of the functionality in the system level, but not in the HCI level as demonstrated in [Flemming et al., 1986].

- Policy

Invoking control is done by the "Controller" which is the system component of LOOS. By modifying the input object, that is, the problem domain, the layout will be controlled by the set of generation rule*s*.

An automated assistant for the memory is not implemented in the HCI level of this version of LOOS.

Lessons learned

Lessons learned from LOOS are:

1. A problem hierarchy is a powerful metaphor for design spaces, but it is also insufficient for expressing geometric information.
2. Function-form representation is effective for understanding the design problems and for decomposing more complex problems.
3. Strong integration of rule and representation is a powerful method for realising design space explorers.

3.2.2 ABLOOS

	representation	rule	memory	policy
system level	**1.** A hierarchical view of 2D rectangles. **2.** Basic spatial relations between 2D rectangles.	**1.** Specified GOB's (Goal Object's). **2.** Using the generate-and-test paradigm according to the levels. **3.** "Right", "left", "above" and "below".	**1.** A tree-like graph under one top-level GOB. **2.** goto-parent-state, goto-children-state and previous-alternative/next-alternative	**1.** "Tester" and decomposition of GOBs. **2.** Acceptance of different. level of abstraction. **3.** No Support
HCI level	**1.** A set of primitives viewed in a screen and a view assistant for zoom and others. **2.** No Support	**1.** Visual impacts on the current layout. **2.** No Support	**1.** No Support **2.** No Support	**1.** "Controller" and "GOB Hierarchy". **2.** No Support

TABLE 3. ABLOOS supports exploration in representation, rules, memory and policy in the system level. It supports partially in representation, rules and policy but memory in the HCI level.

Background

The approach of ABLOOS[Coyne and Flemming, 1990] was to develop a generative system based on a 2D rectangle representation. Based on LOOS, ABLOOS also conducts a framework which represents and manipulates decompositional abstraction levels and hierarchies. The system of ABLOOS, like LOOS, also concentrates on problems of layout design with further extension concerning constraint issues, specifically focusing on one aspect of abstraction of layout design in a hierarchical system.

The functional decomposition mechanism that ABLOOS also utilises indicates the generic levels of tasks, covers most disciplines and sub-levels of complexities in a hierarchical sense for a design layout of rectangles.

system level

- Representation

  Representation formalism that underlies ABLOOS provides a hierarchical view of 2D rectangles in a derivation graph, so that a 2D rectangle might contain another clusters of rectangles at different level of abstraction. Each node of a der-

ivation graph represents a rectangular layout which might be seen as a layout called "GOB" (Goal Object). A "GOB" is structured like a frame with slots and associated methods that determine its behaviour. A GOB implements one inheritance network thus allowing the inheritance of certain properties from the superclasses. The abstraction of ABLOOS is based on suppression of details. By choosing a level of abstraction, users would be able to search for more details in the same level.

Geometric information in ABLOOS is the same as in LOOS since they both use the 2D rectangle as symbolised representation.

- Rules

Rule formalism in ABLOOS, once again is very similar to LOOS. The only difference is that *rules* in ABLOOS are applied according to the methods specified in the GOB's slots rather than by dealing with a rectangle directly.

Rule application also uses the generate-and-test paradigm. The only difference is that ABLOOS generates new states according to their levels.

Intervention points of rules in ABLOOS are also similar to those in LOOS.

Geometry in rule*s* in ABLOOS presumes the 2D rectangle as representation. The topology of a layout is defined based on the basic fixed spatial relations "above", "below", "to the right of" and "to the left of".

- Memory

Design space structure supported by ABLOOS is a tree-like derivation graph under one top-level of GOBs. This derivation graph, like LOOS, also supports multiple states. However, there is no support for a storing or retrieving mechanism.

Primitive movement supported by ABLOOS in the system level is similar to the movement supported by LOOS (described in the previous review).

- Policy

Control strategies supported by ABLOOS are two fold-strategy; one is the tester from LOOS and the other one involves with the decomposition of GOBs. The single characteristic which separates ABLOOS from LOOS is the level of

abstraction able to be provided due to the introduction of the decentralised rule which is only applicable to the same level of GOBs. In the interaction between users and control mechanisms, the level of abstraction is always specified by users who can realise certain partial designs without dealing with others. Users also utilise the generate-and-test paradigm to visualise the control mechanism.

Mixed initiative in ABLOOS is indicated by the acceptance of a different level of abstraction in ABLOOS. Each GOB can also be realised in a later exploration of states and can also involve more than one state at a time.

Multiple agents are not supported by ABLOOS.

HCI level

- Representation

  Representation visualisation handles the same level of rectangular layouts as a set of primitives which can be displayed on the screen. The interface is similar to that implemented by LOOS. There is no direct manipulation for representation in the HCI level.

  Representation manipulation is not supported over 2D rectangles in the HCI level of ABLOOS.

- Rules

  Rules visualisation is similar to that in LOOS (described previously). There is no visualisation other than for the outcome derived from the generate-and-test.

  Rule application visualisation is also similar to that in LOOS (described previously).

  Rule authoring in ABLOOS provides more informative assistants than does LOOS. Each GOB can be specified via a textural description for the list of sub-GOBs and the methods to invoke them.

- Memory

  Design space visualisation has not yet been completed in ABLOOS[Coyne and Flemming, 1990].

Primitive movement in ABLOOS provides most of the functionality in the system level, but this has not yet been extended to the HCI level as demonstrated in [Coyne and Flemming, 1990].

- Policy

Invoking control in ABLOOS is effected in two places: one is the "Controller" component of LOOS and the other is the Problem Statement which describes the GOBs that forms the so-called "GOB Hierarchy". The "Controller" component allows users to decide whether to go further and to generate another alternative layout or to stop. The Problem Statement allows users to specify the domain specific knowledge in terms of a relationship between GOBs and their hierarchies. The decomposition of problems helps users to maintain control over the generation process. The control strategy is thus decentralised as the rule that is only applicable to the same level of GOBs.

An automated assistant is not yet completed in this version of ABLOOS.

Lessons learned

Lessons learned from ABLOOS are

1. Decomposition of representation and memory provides another dimension for the function form of representation.
2. Levels of abstraction provide a useful tool for a large-scale complex problem within the design spaces.

3.2.3 DiscoverForm

	representation	rule	memory	policy
system level	**1.** A set of symbols and structures. **2.** Instances of symbols with a 2D pointset.	**1.** Transformation of symbols. **2.** *Rules application.* **3.** Transformation.	**1.** The result of the exploration process. **2.** Moving upwards or downwards.	**1.** Generate-and-test paradigm. **2.** MI supported.
HCI level	**1.** A display window for motif in a given window. **2.** Independence of structure and motif.	**1.** Comparing outcome structure before and after generation. **2.** Assigning a symbol to a motif and changing the spatial relation of symbols.	**1.** No Support **2.** Moving up and down.	**1.** Combining with human tester and "what if" experiments.

TABLE 4. DiscoverForm supports exploration in representation, rules and policy in both the system and HCI levels.

Background

DiscoverForm was originally developed as a focused concept of Chris Carlson's masters thesis. The further development of DiscoverForm [Carlson, 1989, Carlson and Woodbury, 1994] demonstrates a recursive pattern of generation system with only one rule and a simple intuitive interface. With simplified representation—a set of symbols and their structures—DiscoverForm implements a version of design state navigation.

system level

- Representation

  Representation formalism in DiscoverForm is comprised of a set of symbols[1] and their structures in a spatial configuration. The position of a symbol in a structure is referred to as its orientation. Formally, a structure is comprised of a set of ordered pairs in which each pair contains one symbol and its orientation.

  Geometric information is handled as being the instances of symbols in which a 2D pointset is used. The structure maintains the spatial relation of each symbol only; in theory, such a relation could be expanded in an n-dimensional space.

1. Carlson refers to this symbol as a prototypical part from a set called an *alphabet*.

- Rules

  Rule formalism in DiscoverForm represents a transformation of symbols in the structure. It is also a formalisation for the intuitive notion of stepwise modification called the "rewrite rule". In brief, the DiscoverForm rules simply conduct the transformation and rewrite existing *rules* onto given structures.

  Rule application is supported in DiscoverForm. By transforming an existing representation, the rule*s* of DiscoverForm generate a new structure or even rewrite the existing one. These rule*s* are only focused on transformation, therefore they are parametric so that users can easily chose the motifs and transform them.

  Intervention points in DiscoverForm's rule*s* are invoked when one rule is applied. At any stage (DiscoverForm calls a stage of "level"), the rule can be rewritten or re-applied to a given structure.

  Geometry in rule*s* acts as a will-be-transformed symbol in given structures. The topological relation of two geometric objects is defined as the orientation of these two objects in the same structure. The generation of geometry does not take place in the conduct of rules.

- Memory

  Design space structure constructs the result of the exploration process. DiscoverForm supports only one derivation path, which means that only one path from an initial state to the current state is preserved.

  Primitive movements, such as moving up or moving down one level, are implemented in DiscoverForm. Moving up one level shows the motif before the transformation and moving down shows the motif after the transformation.

- Policy

  Control strategies in DiscoverForm are mainly generate-and-test paradigms plus *rules* rewriting. Users not only apply the rule applications but also "shape" the design spaces.

  Mixed initiative reaches its full support potential for the memory component of DiscoverForm, but it does not support multiple states. Within a single thread of

design exploration, users would be able to go back to levels above or to explore more alternatives.

Multiple agents are not supported in DiscoverForm.

HCI level

- Representation

  Representation visualisation in DiscoverForm means a display window for showing the motif in a given structure.

  Representation manipulation methods provided by DiscoverForm depend on the independence of structures and motif (representation) in the given structures. Users could change the assignment of motif independently or could change its spatial relations.

- *rules*

  Rule visualisation is supported in DiscoverForm.

  Rule application visualisation can be achieved by considering structures before and after the generation process. Implemented rule visualisation occur efficiently, the reaction of rule application is fast enough to give an instant feedback on the impact of rule application.

  Rule authoring in DiscoverForm is equivalent to 1) assigning a symbol to a motif and 2) changing the spatial relations of these symbols in a structure. These are all implemented in a simple graphic interface in DiscoverForm.

- Memory

  Design space visualisation is not supported in DiscoverForm.

  Primitive movement supported by DiscoverForm is a move-up or a move-down via a pull-down menu in the interface.

- Policy

  Invoking control in DiscoverForm is implemented by combining the given human tester and "what if" experiments. By testing a new state, users would be able to see different effects of rules.

  An automated assistant is not supported by DiscoverForm.

Lessons learned

Lessons learned from DiscoverForm include:

1. Decomposing a state representation into the "structure" of its own representation (symbol) is an effective method for carrying geometric information in a generation system in the system level and the HCI level.
2. Independence of structures and symbols is a satisfactory aspect of exploration with interpreters.
3. By combining interactive graphics with generate-and-test paradigm, we can evaluate the special effect brought by a change of rule on the form it generates.This lesson is learned again in Harada's direct manipulation model[Harada, 1997], which brings the grammar interpreter to the users.
4. A HCI level support of policy is helpful for users in controlling the behaviours of rules during the exploration.

3.2.4 GENESIS

	representation	rule	memory	policy
system level	**1.** A topology graph, a geometry, its labels, and a state. **2.** Coordinated geometry	**1.** A set of match conditions and a sequence of operations. **2.** Conditions applied and operations transformed in a boundary graph. **3.** Asserting or retracting a label onto a boundary graph.	Supporting neither structuring nor navigating in a design space.	**1.** Matching condition. **2.** No Support **3.** No Support
HCI level	**1.** A 3D geometry display window. **2.** CLP(R) interpreter	**1.** No Support **2.** A textual editor and CLP(R) interpreter.	Supporting neither rules *visualisation* nor rules *application*.	**1.** CLP(R) interpreter. **2.** No Support

TABLE 5. GENESIS mainly supports exploration in representation. There are rules in both the system and the HCI levels

Background

Genesis [Heisserman, 1991], developed by Jeff Heisserman, is a generative geometric design system. Genesis includes a grammar interpreter based on boundary solid grammar and a 3D geometric display. Grammar(rules) within Genesis is expressed in a boundary solid grammar formalism that admits logical reasoning about rigid 3D solids to the rule matching and application process. Genesis also improves the expressiveness of this grammar formalism in order to convey "the design's important aspects" ([Heisserman, 1991] pp37) in further depth.

system level

- Representation

  Representation formalism. Genesis expresses design instances as a set of boundary solids (in a so-called "topology graph" and "boundary graph"), their labels and a piece of information called the "state". The representation lies under boundary solids in two types of graphs (a topology graph and a boundary graph) and non-geometric information is associated with them via a set of labels.

Geometric information. Genesis handles its representation as boundary solids in a "topology graph" and a "boundary graph" which represent the coordinate geometric information of those solids. The characteristics of a topologic relation between two geometric entities[1] can be easily symbolised as an edge between two nodes in a topology graph. A boundary graph describes the boundary geometric information of each vertex.

- Rules

Rule formalism of Genesis is specified by a set of match conditions (LHS) and a sequence of operations (RHS). The match conditions decide when and which rule might be applied to a given boundary graph and when this might occur. The sequence of operations then, transform this given boundary graph into the outcome.

Rule application is not fully supported in Genesis since Genesis only supports one state. With only one state, the application of Genesis rules act as a refinement of this existing state. A new state will not be generated given this limitation of Genesis.

Intervention points are implemented in Genesis using "states" and matching conditions in general. The match conditions of a rule might satisfy some features of a given boundary graph that include the labels. Asserting or retracting a label onto a boundary graph will provide a simple control over the rules.

Geometry in rule*s* needs to express its conditions or features as clauses in first order logic. A rule in Genesis is applicable to a solid if it can match features of that solid. In this case, rules in Genesis can transform the representation (solids) into a new form. Geometric queries are treated in order to describe the explicit conditions of a given boundary graph and thus becomes a matter of satisfying goal clauses of a specific condition (of a boundary graph). The generation of geometry is then equivalent to specifying a set of partial conditions for a boundary graph as a result.

1. The geometric entities are the topological entities[Hoffmann, 1989] in a geometric representation including vertex, edge, loop, face, and so on.

- Memory

  Design space structure. There is no design space structure in the memory represented by Genesis due to the reason that only one design state is involved. All the changes and generations occur within a single design state. The "state" information is for rule control strategy, thus an alternative is not possible in Genesis.

  Primitive movement is not an issue as only one direct movement is involved within the design space structure in Genesis.

- policy

  Control strategies provided by Genesis are effected through the matching conditions of the Genesis rules stated previously. For instance, when a match function of a rule is matched, its paired operation function will prompt for the commitment. The control strategies can be automatic if the request is always granted or can find the next matched rule if the request is denied.

  Mixed initiative is not supported as Genesis only has one state.

  Multiple agents are not supported as Genesis does not even support multiple states.

HCI level

- Representation

  Representation visualisation is implemented by a 3D geometry display window. The main focus of representation in Genesis is the boundary solids, thus a 3D geometry display can surely visualise part of its representation—3D boundary solids. The geometric information shown in Genesis is a primitive 3D display, thus information such as colours and material are not considered.

  Representation manipulation in Genesis is implemented via a Prolog-like interpreter—the CLP(R) (Constraint Logic Programming)[Heintze et al., 1991] interpreter. As part of the representation, a set of low-level constructers are provided as a set of procedural functions that generate the boundary graph. The representation manipulation is then equivalent to describing the behaviour of a specific boundary graph. Such specification is via the CLP(R) interpreter.

- Rules

  Rules visualisation is not supported in Genesis.

  Rule application visualisation is not supported in Genesis.

  Rule authoring is through a textual editor. CLP(R) is embedded internally in, thus the rules are parsed as CLP(R) programs and are compiled at run-time. With this nature, Genesis rules need to be interpreted by the CLP(R) interpreter in order to understand what description of rule*s* has been written.

- Memory

  Design space visualisation is not fully supported by Genesis since Genesis only supports one state. With one state, the visualisation of a design state is trivially supported as the visualisation of that single state (a set of boundary graphs).

  Primitive movements is not supported by Genesis since Genesis only supports one state where there is only a direct movement.

- Policy

  Invoking control in Genesis is via a CLP(R) interpreter. Genesis does not implement a HCI level of control assistant but rather uses a plain CLP(R) interpreter. The controls that users have are either to apply a matched rule to a matched boundary graph or to ask for an alternative via a backtracking mechanism.

  An automated assistant is not supported in Genesis.

  Lessons learned

Lessons learned from Genesis are:

1. Reasoning over complex 3D geometric objects is possible and effective.
2. The effects and contents of "rules" need to be more informative.
3. The intra-object relation requires a more expressive representation.
4. Without a formal model of function, rules become harder to structure.
5. Labels are insufficient to carry all the non-geometric information.

3.2.5 Tartan Worlds

	representation	rule	memory	policy
system level	**1.** 2D configuration of symbols. **2.** No Support **3.** No Support	**1.** Object matcher and rule applier. **2.** Matching active rules with Tartan worlds. **3.** Active or inactive.	**1.** No Support **2.** Layers of tartan grids that contain configuration of 2D symbols.	**1.** *Rule* list and active/inactive selections. **2.** No Support **3.** No Support
HCI level	**1.** A layer of a "World window" **2.** symbols creation and grid manipulation.	**1.** LHS/RHS window pair with generate-and-test paradigm. **2.** Direct manipulation of composition over symbols.	**1.** No Support **2.** No Support	**1.** "*Rule* list" and "world list". **1.** No Support

TABLE 6. Tartan Worlds only supports exploration in representation and rules in the HCI level.

Background

Tartan Worlds is an interactive and productive 2D configuration of a symbol grammar system that generates designs on tartan grids[Woodbury et al., 1992]. The tartan grids are automatically generated and fixed within tartan worlds. Users create designs with invented rules inside the worlds simultaneously. One standard window represents an individual design world and any rules can be applied within the chosen world incrementally. The system of Tartan Worlds was developed and written in the C programming language and the multiple windows can be directly manipulated on a standard Macintosh user interface. The symbolised 3D geometric information is yet to be displayed and the 2D configuration of symbols is represented as point-sets which will be discussed in detail in later chapters.

One key feature of Tartan Worlds is the provision of an exploratory tool for investigating creative and generative cooperation between designers and interactive grammar systems. The main objective is to create an interesting and instinctive interface and a direct graphical feed-back which makes it possible to explore more design alternatives in large design spaces.

system level

- Representation

Representation formalism in Tartan Worlds utilises the 2D configuration of symbols on so-called "tartan grids" (in which the grid interval follows an unchanging a-b-a-b pattern).

Geometric information is not supported by Tartan Worlds. The created 2D symbols do not carry any geometric information or spatial relations other than a tartan grid.

- Rules

Rule formalism in Tartan Worlds also depends on tartan grids. Each rule contains two pair of grids: one is for the LHS (left hand side) and the other is for the RHS (right hand side). The list of rules is maintained by a "rule list" and only the rule*s* in the "rule list" are used in the individual world. A rule can be active or inactive. When a rule indicates an active conduct, the Object Matcher will match certain rules in certain worlds.

Rule application in Tartan Worlds is through two operations: Object Matcher and Rule Applier. The Object Matcher matches certain active rules with certain Tartan worlds. The rule Applier, then, applies the changes to the matched worlds.

Intervention points implemented in Tartan Worlds are active or inactive rule*s*. By making rules active or inactive during the matching process, the applicability of a rule is decidable at any stage.

Geometry in rule*s* is not supported in Tartan Worlds.

- Memory

Design space structure is not supported in Tartan Worlds. Tartan Worlds contains only one state, but multiple design instances are accomplished by a complete self-contained "world". Tartan Worlds is designed for multiple layers of tartan grids that contain configurations of 2D symbols in each layer.

Primitive movement depends on the complexity of users' design behaviour in Tartan Worlds. The arrangement of symbols is various and thus the movement between states is dynamic and active.

- Policy

Control strategies in Tartan Worlds are implemented via a "rule list" and active or inactive selections. The order in the rule list carries the sequence of rule matches. The selections of active or inactive combined with invisible or invisible provide a complete control over the rule of interest.

Mixed initiative is not supported in Tartan Worlds as it does not have the memory components of design space explorers.

Multiple agents are not supported in Tartan Worlds.

HCI level

- Representation

  Representation visualisation is implemented via a set of "World Windows". Each design (as a Tartan world) is shown in a "World Window" and acts as a layer.

  Representation manipulation in Tartan Worlds is two fold: the creation of symbols and the manipulation of grids. The symbols are bitmap description which can be created by any drawing or painting software, or even from scanned photographs. Once the symbols are created, they can be pasted in the symbol library for further use in exploration. Users also control changing spaces of grids in Tartan Worlds.

- Rules

  Rule visualisation is via the LHS and RHS window pair. LHS and RHS windows of rule*s* are also based on the tartan grids that make rule visualisation straightforward.

  Rule application visualisation is supported with the generate-and-test paradigm. By incremental rule applications, the results are shown in the "World Windows" of respecting worlds.

  Rule authoring is through the direct manipulation of compositions over the symbols in tartan grids. With supports like the delete/rename/create operations over the "Rule Window", users can author the rules interactively.

- Memory

  Design space visualisation is not supported in Tartan Worlds.

  Primitive movement is not supported in Tartan Worlds.

- Policy

  Invoking control in Tartan Worlds is supported in many ways. The "rule list" and "world list" windows are used to control the order of rule applications. The selections of active or inactive rule*s* provide the control to limit the range of rules that can be applied.

  An automated assistant is not supported in Tartan Worlds.

Lessons learned

Lessons learned from Tartan Worlds include:

1. A uniform representation of rule*s* and representation can further support the interaction of rules.
2. A set grammar is not sufficient for 3D geometric objects.
3. A rule with structures is reusable for further construction.
4. Visualisation of rule*s* provides informative support for policy.
5. The importance of strong integration between state representation and rule*s* has revealed its strength in this research project.

3.2.6 Grammatica

	representation	rule	memory	policy
system level	1. Grammatical programming-rules and grammars. 2. No Support 3. No Support	1. Operation of an algebra of nondeterministic functions. 2. No Support 3. No Support	1. No Support 2. No Support	1. Mechanism to decompose the rules and grammatical algebras. 2. No Support 3. No Support
HCI level	1. No Support 2. No Support	1. A graphical result of evaluation. 2. A textual editor.	1. No Support 2. No Support	1. No Support 2. No Support

TABLE 7. Grammatica supports exploration partially in representation, rules and policy in the system level but only supports exploration in rules in the HCI level.

Background

One of the aims of computational design is to provide extensive and flexible form-generation for designers. It is a dilemma that designers must be aware of at least broad outlines of what might be generated. Designers, on the other hand, can transform these broad outlines into specific instructions that computers can recognise and execute. Developing concepts for repositioning computers as form-generating partners is important. The interpretations and communication between human designers and computers are fundamental issues for developing a system—called "grammatical programming". This system is also referred to as Grammatica and was developed by Chris Carlson [Carlson, 1994]. Grammatica is a first attempt to build a system which supports grammatical programming. Such a system is grounded in formal semantics that can allow an algebra of control operators to construct modular specifications of design spaces.

Carlson gives a collection of definitions for recursive functions in a specified grammatical program[Carlson, 1994]. These functions can be reviewed as the bases of his research outcomes for a paradigm to describe design spaces. There are three types of function: (1) a type that defines a target algebra of designs and its subalgebras, (2) a type that defines primitive functions on design corresponding to the specified rules of traditional grammars, and (3) a type that defines components of recursive functions organised from the primitives corresponding to the sets of rules

of traditional grammars. Compared to traditional rules, grammatical programming accommodates rewriting rules even when they are not specified as right or left hand patterns.

The Grammatica prototype consists of two parts: the Grammatica kernel and a Mathematica [Wolfram, 1991] front-end. The interface and non-linear constraints are resolved within the Mathematica front-end. The textual description of a grammatical program is interpreted and compiled by the Grammatica kernel. It also exports to its Mathematica front-end for visualisation and evaluation. The kernel extends the capability of CLP(R)[Heintze et al., 1991], that is, of a Prolog[1]-like syntax. The linear constraints are resolved in the kernel, and the non-linear constraints are resolved in the Mathematica front-end.

system level

- Representation

  Representation formalism in Grammatica is grammatical programming. Unlike the traditional grammars, grammatical programs provide equal status to any created "rules", "grammars" and arbitrary functions. One of the components of grammatical programming is that a style of symbols is used as part of the representation.

  Geometric information is not handled in Grammatica as its main purpose is to demonstrate the usage of grammatical programming. However, as Carlson indicated, the theory behind Grammatica complements other approaches such as Genesis.

- Rules

  Rule formalism in Grammatica is actually grammatical programming. The components of grammatical programs consist of operations of an algebra of nondeterministic functions. These operations are composition, union, iteration and failure. These operations are also the algebraic operations and the style of symbols (as a representation)for them are chosen according to the demands of specific applications.

1. For an introduction to the syntax of Prolog, see [Sterling and Shapiro, 1986].

Rule application is not supported in Grammatica. In the system level, Grammatica only evaluates a set of grammatical expressions and provides output of the result.

Intervention points are not supported in Grammatica. The closest control is either an evaluation request or a solver request for evaluating the grammatical expression.

Geometry in rule*s* does not receive special formal treatments in Grammatica. In theory, grammatical programming treats geometric information in the same way as does any other design space explorer.

- Memory

Design space structure is not supported in Grammatica. There is no notion of design states in Grammatica.

Primitive movement. As the prototype did not touch the memory issue, Grammatica does not support primitive movement of memory either.

- Policy

Control strategies over the grammatical programming come from the mechanism for decomposing the rules and for determining how to interact with other modularised rules via grammatical algebras.

Mixed initiative is not supported in Grammatica.

Multiple agents are not supported in Grammatica.

HCI level

- Representation

Representation visualisation is not implemented in Grammatica. The main contributions of Grammatica, however, are the operations of composition, union and iteration over grammar*s*. The visualisation of representation is not completed in a sense like any other design space explorer.

Representation manipulation is also not implemented in Grammatica.

- Rules

Rules visualisation is through the evaluation tree generated by Grammatica as a result of evaluation. In the Mathematica front-end, this result is displayed graphically.

Rule application visualisation is not implemented in Grammatica.

Rule authoring in Grammatica is via a textual editor since the rule*s* themselves are textual descriptions of the grammatical programs. In theory, the modularisation of rule*s* shows a potential in visualising rule*s* and their construction, but this is not the case in the Grammatica prototype.

- Memory

  Design space visualisation is not implemented in Grammatica as Grammatica does not support memory in the system level.

  Primitive movement is also not implemented in Grammatica.

- Policy

  Invoking controls in the HCI level concepts are not implemented in Grammatica.

  An automated assistant is also not implemented in Grammatica.

Lessons learned

The lessons learned from Grammatica are:

1. Design spaces can themselves be considered as formal objects with which complex domain-proximate computations can be defined.
2. Using rewriting as an eloquent way to describe the transformation of designs, Grammatica demonstrates the possibility of combining part of a set of designs and rules for their configuration by embedding grammars in a large framework of programming with nondeterministic functions.
3. Constructing and controlling grammatical programs requires incremental matching and nonlinear constraints solving.
4. Parametric grammar enhances the modularity of design spaces.
5. Parametric designs require visualisation and direct manipulation.

3.2.7 SEED/SEED-Config

	representation	rule	memory	policy
system level	**1.** 3D geometric information. **2.** A version control. **3.** Integrated with 3D geometric information and a mixed initiated environment.	**1.** "Match and operation" mechanism and "technologies". **2.** By enforcing the "state" information. **3.** Controlled by rule authors.	**1.** A tree-like derivation graph supports multiple state and design space structure. **2.** Basic navigation around.	**1.** Technology devices that support a more user-controllable control strategy. **2.** "Version" control. **3.** No Support
HCI level	**1.** A 3D Design Window. **2.** "picking", "colouring" and "labelling".	**1.** Operation previous window and technology window. **2.** Textual editor.	**1.** A tree-like "design space window" in two-levels of visualisation. **2.** up/down/left/right, click-and-go and information window.	**1.** User-level organisation tools in "operation preview window". **2.** Not yet.

TABLE 8. SEED-Config supports exploration in representation, rules, memory and policy in both the system and the HCI levels.

Background

SEED[1], an abbreviation for "*S*oftware *E*nvironment for *E*arly phrase of building *D*esign"[Woodbury et al., 1994, Flemming et al., 1994, Flemming and Woodbury, 1995], started in 1992 as a joint project between The University of Adelaide and Carnegie-Mellon University. The SEED project targets the early phases of building design, where an exploratory design process can greatly shorten the process of generating internal design representations and of evaluating design alternatives.

The main concepts behind the development of SEED were based on a recognition of the lack of reliable mechanisms for supporting early design exploration [Flemming et al., 1993, Flemming and Woodbury, 1995]. SEED implements its system from a user-oriented perspectives which is different comparing with LOOS or Genesis which are the pilot projects for SEED. There are three modules in the SEED project and with specific interests of this dissertation, the three dimensional configuration module: SEED-Config is chosen and discussed in detail.

1. For a more detailed description of SEED see [Woodbury et al., 1994, Flemming et al., 1994, Flemming and Woodbury, 1995].

system level

- Representation

Representation formalism. SEED-Config inherits the boundary solids representation used in Genesis —3D geometric information (boundary solids and their associated labels).The extensions SEED-Config makes to the representation of Genesis include: a "version" control that allows multiple designs to be generated while maintaining the consistency of each topological element.

Geometric information is handled with the version technique described above. By constructing a complex graph with multiple version information, SEED-Config extends the boundary graph in a mixed initiative environment. Therefore, all the changes applied to a given set of solids within a state are preserved and then sealed when the state changes. This includes non-geometric information as well.

- Rules

The rule formalism in SEED-Config is also inherited from the match/operation mechanism of Genesis with several key extensions. One extension SEED-Config makes to the introduction of a rule*s* is "technologies" mechanism which allows a hierarchical abstraction over rule*s* and enhances the modularisation of rule*s*. In SEED-Config, rules are constructed via their own level of abstraction called "classes".

Rule application in SEED-Config is controlled by enforcing the 'state' information that generates a new state whenever the 'state' information changes. This is also implemented via the "version" technique described above. Informally, a state in SEED-Config is equivalent to a version. Logically speaking, unless the 'state' is changed, all rule*s* are applied to the current state.

Intervention points are controlled by the rule authors who change the 'state' when there is a need to preserve state information. However, once a 'state' is declared, the information of that declared 'state' will be frozen and changes are not allowed.

Geometry in rule*s.* As in Genesis, SEED-Config represents boundary solids as boundary graphs that can be transformed as an outcome of rule application. Col-

ours of boundary solids are used and represented as part of the geometric information.

- Memory

The design space structure supported in SEED-Config is like a tree-like derivation graph that supports multiple states.

Primitive movement is also provided as a basic navigation around explored design space such as 'goto-parent', 'goto-children', 'goto-previous-alternative' and 'goto-next-alternative'. As stated before, these 'states' are only viewed once they have been declared. Thus, applying a rule to a view-only 'state' will generate a new 'state'.

- Policy

Control strategies in SEED-Config are supported by the technology devices. Separate from the representation, technologies can have their own abstraction level that supports a more user-controllable control strategy. Instead of depending on the backtracking mechanism to find the rule*s*, technologies allow users to specify the range of rule*s* within individual design interests.

Mixed initiative lies under the 'version' control mechanism of SEED-Config. Multiple states are represented as different 'versions' in SEED-Config, thus, a set of navigation functions supported in the system level is able to provide switches between 'versions'.

Multiple agents are not supported in SEED-Config. SEED-Config is a single process in which parallel search over the same design space is not possible.

HCI level

- Representation

Representation visualisation is implemented using OpenGL (called a "design window") which enhances the visual effects as well as offering viewing assistance. In addition, the label information of a selectable boundary solid is also visualised via a "label window".

Representation manipulation methods allowed in SEED-Config are only "picking", "colouring" and "labelling" the boundary solids. Most other operations are provided in the system level but not yet to be implemented in the HCI level.

- Rules

  The rule visualisation that SEED-Config provides is through a window called the "operation previous window". Within this window (and associated windows) the information of a selected technology will be displayed and possible effects are also hinted at by a textural description.

  Rule application visualisation is provided by the "operation preview window" and the "technology window". A control over the selection of technologies is also available later in the current state.

  Rule authoring still needs to be edited by a textual editor.

- Memory

  Design space visualisation in SEED-Config is through a tree-like "design space window". Associated with the "design window", SEED-Config provides two levels of visualisation of the design space. One level is in the derivation graph (the "design space window") and the other displays multiple states ("design windows") simultaneously. Within the "design space window", a set of navigation assistant tools are given and the capability to connect the visualised designs ("design windows") with the derivation tree is demonstrated via a graphical interface.

  Primitive movement in SEED-Config is through three levels of mechanisms: up/down/left/right of the derivation tree, click-and-go on the "design space window" and an information display window which describes the information content of that design state.

- Policy

  Invoking controls are provided via user-level organisation tools (such as grouping) in the "operation preview window", which supports direct manipulation over the rule*s*. Within the "operation preview window", users can find the available technologies (rule*s*) for current state, preview a result of technologies or delete a result from an elaborated state. Associated with the "technology window", SEED-Config provides a more informative assistant for visualisation of the rule*s* as well as supports for policy.

Automated assistants provided by SEED-Config include the "operation preview window" and the automatic rule application. SEED-Config includes a set of rule application assistants such as: automatically generating all the possible first found solutions, or finding possible matched rules (technologies) in a current state.

Lessons learned

Lessons learned from SEED-Config are:

1. Envisaging the entire system from a user's perspective is essential for the implementation of a design space explorer.
2. A much deeper understanding of state space search is required.
3. Modularity in rules is desirable for expanding design space explorers.
4. Navigation of design spaces in the HCI level is crucial for exploration.
5. Function-form representation is useful for decomposition of design problems.
6. State spaces are multi-initiative.

3.2.8 Gedit

	representation	rule	memory	policy
system level	**1.** 2D maximal lines and 2D labelled points. **2.** 2D labelled shapes. **3.** geometry information is built-in inside of shape rules.	**1.** a pair of a pattern and its replacement. **2.** *Rules application.* **3.** Transformation.	**1.** No support in this implementation. **2.** No support in this implementation.	**1.** User control. **2.** Mixed initiative is not supported.
HCI level	**1.** A shape display window for the current shape. **2.** By rule application.	**1.** Using colour scheme for effects of rule application, and a rule display window. **2.** Specifying a pattern (shape) and its replacement in the rule window.	**1.** No Support **2.** No support	**1.** Combining with human tester and "what if" experiments.

TABLE 9. Gedit supports exploration in representation and rules in both the system and HCI levels.

Background

Gedit is a visual shape grammar implementation, developed by Mark Tapia[Tapia, 1999 and Tapia, 1996]. Although the shape representation can deal with three-dimensional objects, Gedit only supports shape computation in the algebras of 2D points and 2D lines. While Tapia claimed that the process of developing and using a shape grammar can be divided into three phrases[1], the implementation of Gedit only concentrates on compilation and directed execution of the shape rules.

system level

- Representation

  Representation formalism in Gedit is called *shapes* which are composed by the 2D maximal lines[Krishnamurti, 1992]. For preventing the emergence of extremely large or small shapes, Gedit representation provides a view of scope which restrict the way the rule can be applied, and further, transformations of the

1. Tapia divided the process into 1) creating and modifying the shape grammar; 2) compiling the grammar; 3) exploring the language of designs defined by the grammar.

representation must be compositions of integer translations multiples of 90 degree rotations, reflections along lines.

Geometric information in Gedit is represented by 2D labelled points and 2D non-zero length of lines of labelled shapes, which has not extended to 3D elements or non-linear elements such as circles yet. In Gedit, the shapes are drawn by designers in other tools, and then converted to the maximal line representation of Gedit.

- Rules

Rule formalism in Gedit follows the convention of shape grammars, which consists of a left hand side (the pattern) and a right hand side (the replacement). Each shape rule specifies a possible replacement operation. Such rule in Gedit will apply to a shape only if the correspondence of the shapes match the pattern of that rule, and the matched part of shape will be replaced/transformed according to the replacement.

Rule application in Gedit contains only the current shape. There is no mechanism in Gedit for state spaces or equivalent representation.

Intervention points of shape rules are purely user-control in Gedit. The rule is selected before the rule application. A set of rules are activated and fired in a sequential way when they are selected to apply to an initial shape.

Geometry in rules. As the left hand side and the right hand side of rules are described as shapes—the geometry of Gedit, the geometry (2D lines and labelled points) is built in the rule mechanism and application.

- Memory

Design space structure is not supported in Gedit.

Primitive movement is supported in Gedit for creating the shapes and the rules. In Gedit, the shapes are created via external drawing tools and are converted to maximal line representation. In addition, the created shapes can be manipulated directly within the program.

- Policy

Control strategies provided by Gedit are through the user controls. No other policy control strategies are implemented in the program.

Mixed initiative is not supported in Gedit.

Multiple agents are not supported in Gedit since the program has not implemented any states.

HCI level

- Representation

  Representation visualisation in Gedit is thought the display of shapes (the shapes window) which are done via 2D lines and points.

  Representation manipulation in Gedit is through the drawing program which allows the designers to use line tools or oval tools to create lines/points or labels. After the drawing tools, the shapes are converted into maximal line representation of Gedit for the use of rules.

- *rules*

  Rule visualisation supported in Gedit is through the rule window. Within the rule window, users specify the pattern as well as its replacement. Several techniques such as colours are used to identify the effects of that rule. Another window called "alternatives" will display the possible alternatives while applying the selected rule.

  Rule application visualisation is not supported in Gedit.

  Rule authoring in Gedit is via the rule window, where users can specify the left hand side of the rules and its replacement. While the pattern might be drawn from external tools and converted as internal data structure, the rule window specifies the replacement operation between these shapes.

- Memory

  Design space visualisation is not supported in Gedit.

  Primitive movement is not supported by Gedit.

- Policy

  Invoking control in Gedit is merely user-control. All the rules and target shapes are selected by the users. The program provides the information as well as how it will effect the current shape.

An automated assistant is not supported by Gedit.

Lessons learned

Lessons learned from Gedit include:

1. Maximal line representation provides a simple but powerful geometry information which provides a straight-forward visualisation of shape rules and rule authoring environment.
2. The actual computational algorithms for 3D maximal lines are still hard to be implemented.
3. User control strategy as similar to test-and-generate is good for restricting the scope of computational intensive problems such as lines emergency in shape grammar.

3.2.9 SG-CLIPS

	representation	rule	memory	policy
system level	**1.** 2D maximal lines and 2D labelled points. **2.** 2D labelled shapes. **3.** geometry information is built-in inside of shape rules.	**1.** a pair of a pattern and its replacement. **2.** *Rules application.* **3.** Transformation.	**1.** No support in this implementation. **2.** No support in this implementation.	**1.** User control. **2.** Mixed initiative is not supported.
HCI level	**1.** A shape display window for the current shape. **2.** By rule application.	**1.** Using colour scheme for effects of rule application, and a rule display window. **2.** Specifying a pattern (shape) and its replacement in the rule window.	**1.** No Support **2.** No support	**1.** Combining with human tester and "what if" experiments.

TABLE 10. SG-CLIPS supports exploration in representation, rules and policy in both the system and HCI levels.

Background

SG-CLIPS is a visual tool that supports the generation design from a set of grammar rules, developed by S.-F. Chien, M. Donia, J. Snyder, and W.-J. Tsai[Chien et al., 1998]. SG-CLIPS provides a graphical interface to allow the designers to explore the state space through firing the grammar rules. As conceived as a course project, SG-CLIPS is not a general grammatical system, but rather a generation system with grammatical flavour.

system level

- Representation

  Representation formalism in SG-CLIPS follows the shape grammar mechanism (points, lines, and line segments) and designed with object-oriented fashion. The classes include a Design State class and a set of shape primitives: Points, Line-Segments, Polygons, and Lines. Each shape primitive is based on the Shape class, which provides the basic operation and the relation of subParts.

Geometric information in SG-CLIPS is represented by the Shape primitives classes, which has not extended to 3D elements or non-linear elements such as circles yet. The relations between each shape primitives are defined within the class definition.

- Rules

Rule formalism in SG-CLIPS follows the convention of shape grammars, which consists of a left hand side (LHS, the shape) and a right hand side (RHS, the replacement). Each shape rule specifies a possible replacement operation, and the objects are asserted as "facts" for the inference engine of SG-CLIPS. Such rule in SG-CLIPS will apply to a shape only if the correspondence of the shapes match the pattern of that rule, and the matched part of shape will be replaced/ transformed according to the replacement.

Rule application is defined over a class called DesignState, which is created either by the initial rule or the RHS instruction of a matched LHS of a rule. Within any time in the application, there is only one active state.

Intervention points of shape rules are done all through the inference engine. A successful matched LHS of a rule will be put onto the execution agenda awaiting for being fired. The order of the rules in the agenda is controlled by the conflict resolution strategy.

Geometry in rules. As the left hand side and the right hand side of rules are described as shapes—the geometry of SG-CLIPS, the geometry (2D lines and labelled points) is built in the rule mechanism and application.

- Memory

Design space structure in SG-CLIPS is supported through the DesignState class in a tree-like structure, including explored state, selected state, un-explored state, and current activated state. In any case, there is only one active state.

Primitive movement supported in SG-CLIPS provides a basic exploring procedure for the design space, such as next, previous, goto states. With a known state identification, SG-CLIPS would be able to jump from one state to anther state, or to activate the selected state as the current active state.

- Policy

  Control strategies provided by SG-CLIPS are through two folds—the automatic generation and the user controls. The LHS of a rule matching process is controlled by the inference engine of SG-CLIPS—CLIPS. Within the matching process, the object (DesignState) is the key feature which the users can use as a control strategy.

  Mixed initiative is supported as SG-CLIPS allows the users to change the state during the exploration.

  Multiple agents are not supported in SG-CLIPS.

HCI level

- Representation

  Representation visualisation in Gedit is thought the display of 2D shapes in the current active state. As the geometric representation is defined in the class definition, the display of a shape object is to display all its sub-parts onto the screen.

  Representation manipulation in SG-CLIPS is fixed and throughout the rule application. SG-CLIPS does not implement manipulation of its shape/shape primitives via user interface.

- *rules*

  Rule visualisation is not supported in SG-CLIPS.

  Rule application visualisation is not supported in SG-CLIPS.

  Rule authoring in SG-CLIPS is by writing the rules in CLIPS. There is no graphic rule authoring assistant implemented in SG-CLIPS.

- Memory

  Design space visualisation. SG-CLIPS visualises the structure of explored DesignSpace objects in a tree window. The shapes of each DesignState object are displayed within a rectangle in the tree structure. Four different types of states are colour coded and linked with their derivation relations.

  Primitive movement of the design space structure supported by Gedit is via the manipulation of the tree window.

- Policy

  Invoking control. Most of control invoking in SG-CLIPS are done within the description of the rules. At the HCI level, SG-CLIPS provides the selection of DesignState which allows the users to control the rules and the initial shapes they would like to explore.

  An automated assistant is not supported by SG-CLIPS. All the rules are fired as facts and the users can only see the results by applying the rules.

Lessons learned

Lessons learned from SG-CLIPS include:

1. Maximal line representation provides a simple but powerful geometry information which provides a straight-forward visualisation of shape rules and rule authoring environment.
2. The data model of SG-CLIPS is designed by finding the common set of predicates used in the grammar system. This activity enlightens an interesting direction for dealing with the complexity of a general geometric information system.
3. By describing the behaviours and relations of the data structure which a grammatical system will invoke, SG-CLIPS simplifies the matching procedure of LHS of a rule.
4. User control strategy for selection of the active state is good for restricting the scope of computational intensive problems such as lines emergency in shape grammar.

3.3 Summary

As overall reviews of seven research projects have been demonstrated and discussed in details[1], a complete framework of a design space explorer is revealed. These lessons have identified amongst others, the vital issues of dealing with geometric information and the synchronisation of domain knowledge in the symbol level and the HCI level. These valuable lessons also emphasize the essential geometric information in terms of its representation in the symbol level as well as in the HCI level. One common interest as well as the primary focus of this thesis is that geometric information has been tackled across all research projects in 2D representation or 3D representation in the HCI level. Given the interests of this thesis, Genesis and SEED-Config are the only two projects that handle geometric information in terms of 3D representation which also synchronises with domain knowledge.

The fundamental concept of the proposed design space explorer introduced in the next chapter is based on a notion developed by Newell [Newell, 1982]—the "knowledge level". With the notion of the knowledge level and its symbol level representation, we unfold a design space explorer that represents geometry and knowledge in a seamless manner.

With the lessons learned from experiences with these research projects, a knowledge level is created, and its symbol level representation is discussed in the following chapters. An ideal design space explorer has by all means to fulfil all the critical criteria for the four components in both the system and the HCI levels, and the proposed design space explorer aims to achieve this goal. Given the limitations of adopted research projects on the HCI level, however, we hope with this proposed design space explorer to further study this aspect and to consider likely directions of future research in this area.

The following two tables (Table 11 and Table 12) illustrate the key lessons learned from the reviews above. These lessons are organised by the components of a complete design space explorer and the corresponding questionnaires listed in the Table 1 on page 40.

1. Two summaries of these key lessons are shown in Table 11 (the system level) and Table 12 (HCI level).

representation	*What is the representation?*	*How is geometric information handled?*	*How do we integrate geometric information and rule mechanism?*
	1. 2D information (LOOS/ABLOOS/ Gedit/SG-CLIPS) **2.** 3D geometrical information (Genesis/SEED-Config).	**1.** Instances of symbols with 2D pointsets (DiscoverForm). **2.** Coordinated geometry (Genesis/SEED-Config), and **3.** a version control system (SEED-Config).	**1.** Integrated with 3D geometric information and a mixed initiated environment (SEED-Config)
rule	*What are the descriptions of the rules?*	*How do we generate new states using the rules?*	*How do we control rule application?*
	1. Operation of an algebra of nondeterministic function (Grammatica). **2.** a set of match conditions and a sequence of operations (Genesis/ SEED-Config).	**1.** Conditions applied and operations transformed in a boundary graph (Genesis). **2.** by enforcing the "state" information (SEED-Config/SG-CLIPS).	**1.** User-control (Gedit/ SEED-Config/SG-CLIPS/ABLOOS). **2.** using labelling scheme and matching condition (Genesis/ SEED-Config).
memory	*What is the structure of a design space?*	*How can we navigate in a design space?*	
	1. A tree-like derivation graph (ABLOOS/ LOOS/SG-CLIPS). **2.** A derivation graph should support multiple states and possible directed cyclic graph (SEED-Config).	**1.** navigation through the graph (parent/children/previous alternative states) (LOOS/ ABLOOS/Discover-Form/SEED-Config/ SG-CLIPS).	
policy	*What are the control strategies (of rules)?*	*Is mixed initiative supported?*	*Are multiple agents supported?*
	1. Matching condition (Genesis/SEED-Config/ Gedit/SG-CLIPS). **2.** By user control (Gedit/SEED-Config/ SG-CLIPS). **3.** A "tester" as an evaluation and control device (LOOS/ ABLOOS).	**1.** ABLOOS supports different levels of abstraction in mixed initiative (ABLOOS). **2.** SEED-Config, DiscoverForm, and SG-CLIPS are all supporting mixed initiative with their state space representation. (SEED-Config/DiscoverForm/ SG-CLIPS)	**1.** no supports from all reviews.

TABLE 11. A summary of key lessons learned from the reviews. (system level)

representation	*How do we visualise the representation?*	*How do we manipulate the representation?*
	1. 2D lines/pointsets window with viewing assistant such as zoom/pan (Gedit/DiscoverForm/LOOS/ABLOOS/Tartan Worlds/SG-CLIPS). **2.** 3D geometry display (Genesis/SEED-Config).	**1.** "picking", "colouring", and "labelling" (SEED-Config). **2.** resize/scale of current shape (Gedit).
rule	*How do we visualise the rules and their application?*	*How do we author rules?*
	1. Visual side-effects on the current representation (Gedit/LOOS/ABLOOS/SG-CLIPS). **2.** Generate and test paradigm visual effect (Tartan Worlds/LOOS/ABLOOS). **3.** previewing the operation before and after application (SEED-Config). **4.** technologies/rules windows (Gedit/SEED-Config).	**1.** Textual editor with an interpreter (Genesis/SEED-ConfigSG-CLIPS). **2.** creating and modifying both the LHS/RHS visually (Gedit/Tartan Worlds).
memory	*How does a user view the design space?*	*How does a user move in the design space?*
	1. A tree-like design space window in two levels of visualisation (SEED-Config).	**1.** navigation through the derivation graph directly (SEED-Config/SG-CLIPS).
policy	*How do users interact with rule control strategies?*	*What are the navigation assistants?*
	1. Combining with human tester and "what if" experiment (DiscoverForm/ABLOOS/LOOS/SG-CLIPS). **2.** Users interact with specific windows (Tartan Worlds). **3.** Graph organisation assistant (SEED-Config). **4.** A grammatical interpreter (Gedit/Genesis/SEED-Config).	**1.** organisation assistant from operation preview window (SEED-Config).

TABLE 12. A summary of key lessons learned from the reviews. (HCI level).

2

Theories and Algorithms

Several lessons gained from studying other research projects relating to design space explorers are revealed in the previous chapter. Having provided this background analysis, one specific knowledge level and its symbol level representation are discussed in detail in Part II of this dissertation. An approach derived from the extension of such a knowledge level called "Geometric Typed Feature Structures" is then outlined in Chapter 6, which gives an important aspect of design space explorers—a state representation carrying 3D geometric information.

Part II is divided into three chapters:

Chapter 4: "Domain objects: the SEED knowledge level"

This chapter gives an introduction to the knowledge level concepts adopted from the SEED project, followed by a discussion of the realisation of the knowledge levels corresponding to the symbol level.

Chapter 5: "Representation of the knowledge level using Typed Feature Structures"

A formal system called "Typed Feature Structures" is adapted for implementing a symbol level representation for design space explorers. This chapter firstly describes the fundamental theories and algorithms of Typed Feature Structures related to the context of design space explorers. Then we discuss the incorporation of geometric information and the SEED knowledge level into Typed Feature Structures.

Chapter 6: "Geometric Typed Feature Structures"

Given the observation of domain objects, an extension of Typed Feature Structures called "Geometric Typed Feature Structures" is introduced. This chapter introduces Geometric Typed Feature Structures as a technique for representing continuous information in Typed Feature Structures in a manner compatible with the Typed Feature Structures mechanisms.

4

Domain objects: the SEED knowledge level

The approach taken in this research is to implement one design space explorer reifying the exploration model discussed in Chapter 2.2.3. In order to accomplish an implementation of a design space explorer, a representation of its domain objects is needed. Newell[Newell, 1982] characterises a layer of knowledge called the "knowledge level" to describe the domain objects of specified representation. Each knowledge level concept gives a definition for each domain object and describes relations between these objects. The knowledge level concepts adopted by this research come from the SEED project. These concepts are referred as the "SEED knowledge level". [1]

The SEED knowledge level concepts have been used to develop a specific knowledge representation for three SEED modules—1) the architectural programming module (SEED-Pro); 2) the schematic layout design module(SEED-Layout); and 3) the three-dimensional building configuration design module(SEED-Config). Two supporting modules—a standard and code checker, and a version object database are also included in SEED. The SEED knowledge level, nevertheless, is realised in the specified context of each module. However, the overall knowledge level remains largely identical across all modules.

1. We stress that we use one particular knowledge level in this dissertation. We do not here dive into the analogues or equivalents to the knowledge level in other research. Stiny's weight [Stiny, 1992], for example, are a substrate over which a knowledge level could be specified. There are also knowledge levels implicit in if not many most of the grammatical studies of design corpora, for example, Palladian villas [Stiny and Mitchell, 1978], Mughul gardens [Stiny and Mitchell, 1980], bungalows of Buffalo [Downing and Flemming, 1981], architecture of Frank Lloyd Wright [Koning and Eizenberg, 1981], Queen Anne [Flemming, 1987], architecture of Irving Gill [Gibbs, 1997], and others.

Because it focuses on the context of 3D configuration for generating spatial and physical building parts, SEED-Config is specially chosen for its 3D geometric support. We use "SEED" to refer to "SEED-Config" except when context requires it be otherwise.

4.1 The knowledge level concepts

The SEED knowledge level is an account of a specific design space explorer. It acts as a specification of a generic design space explorer. Some of the key features for the SEED knowledge level are:

- Function-form representation: Function-form separation allows designers to specify one function without worrying about its instance and to explore partially satisfied design spaces without fully instantiating the whole design.
- Use cases: A software engineering method called "use cases" is used in SEED. Use cases act as detailed system specifications from a designer's perspective. They can also be decomposed into more specific tasks performed by designers, such as to "refine a selected building wall with selected technology".
- Technologies: Technologies allow designers to describe the mechanism of instantiations when exploring through the design spaces.
- Functional decomposition: Functional decomposition allows one function to be decomposed further into smaller sub-components which might be shared by other functions. A reusability of partial design/function is ensured.

With such features, the SEED knowledge level provides a guideline for the realisation of a design space explorer in the design domain.

The SEED knowledge level contains five primary concepts that organise the basic conducts of SEED in generating alternative designs: (1) functional units (**FU**s); (2) design units (**DU**s); (3) technologies; (4) states; and (5) design spaces. States and design spaces are the high level structures of the SEED knowledge level, which organise the form of generation of alternative designs by utilising **FU**s, **DU**s and technologies. Each concept is defined and explained in the following sections (referred from SEED documentation). A short description with examples is also included.

4.1.1 Functional Unit

"A functional unit is an identifiable object intended to perform a specific function or combination of functions in a building (e.g., a living room, a load-bearing exterior wall). A **FU** *has associated constraints and criteria on its shape, size, placement, relations with other* **FU**s *and so on." [Woodbury et al., 1994]*

FUs and **DU**s (described in detail in the next sub-section) are the fundamental components which store the design data. **FU**s can be decomposed into hierarchies that provide the functional decomposition aspects of design. There is no notion of information inheritance in these **FU** hierarchies. However, a **FU** might occur in more than one **FU** hierarchy.

For example, a functional unit, specifying an insulation of a wall-enclosure, can be re-used in the roof-enclosure with some adjustments to the context. A typical example of **FU** hierarchy from SEED is shown in Figure 6: an enclosure system can be decomposed further into several layers. The layers of the wall are comprised of structural, insulation, air barrier and rainscreen components. The wall itself as described is a **FU** contained in an enclosure system. In a formal description, $\mathbf{FU}_{continuous_support}$ [1] can access all the information carried by **FU**s above it (in this case, these are $\mathbf{FU}_{building_enclosure}$ and $\mathbf{FU}_{wall_enclosure}$ in Figure 6). In addition, each **FU**, such as wall_enclosure, has its own unique corresponding **DU**. Figure 7a is a **DU** instance of $\mathbf{FU}_{wall_enclosure}$.

4.1.2 Design Unit

Definition

*"A design unit (***DU***) is a part of the spatial or physical structure of a building with an identifiable spatial boundary. In a complete design, each* **FU** *has a* **DU** *associated with it.* **DU**s *can, through their associated, induced* **FU** *hierarchy, implicitly contain other* **DU**s *(the latter units are said to elaborate the former) so that a hierarchical decomposition of* **DU**s *reflects a hierarchical decomposition of the associated* **FU**s *and vice-versa."*[Woodbury et al., 1994]

Design units (**DU**s) in SEED contain design information such as the geometric information of 3D objects. These 3D objects are bounded by different surfaces cor-

1. $\mathbf{FU}_{continous_support}$ stands for a continuous support functional unit as shown in Figure 8.

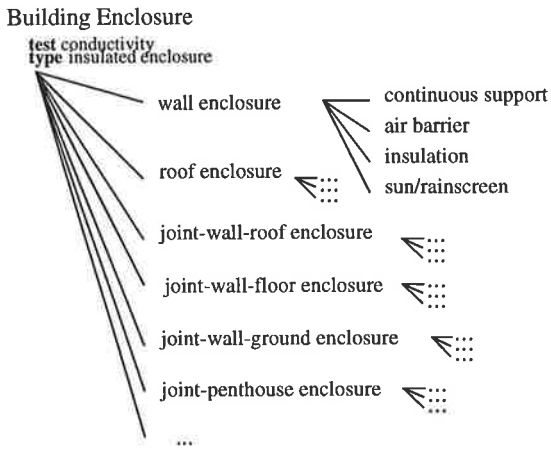


FIGURE 6. A Partial Functional Unit Hierarchy for an Insulated Enclosure.

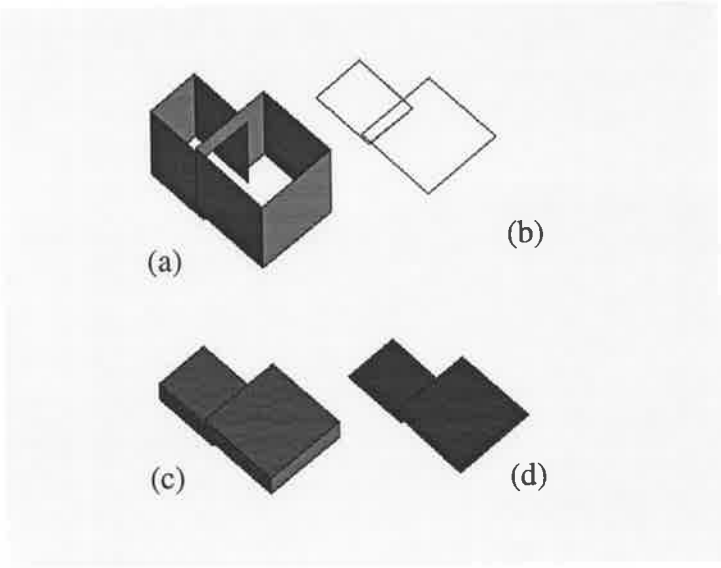
responding to the physical objects of everyday life, for instance, the constructed part of a building. Two characteristics of these 3D objects are non-manifold representation [Chang and Woodbury, 1997] and parametric objects. For non-manifold objects, the representations of design information are uniformly represented at levels of abstraction adequately for a design situation. For parametric objects, design decisions can be made before the actual sizes or form of objects are known.

In Figure 7, four abstract enclosures of a fire station from [Woodbury and Chang, 1995b] are shown. The joints of wall and roof enclosures (Figure 7b) are the lines shared by abstract roofs (2D surfaces) and abstract walls(2D surfaces surrounding the two spaces as shown in Figure 7a). Discarding dimensionality, these geometric objects are all carried by corresponding **DU**s, and are accessible via their corresponding **FU**s. Partiality is ensured by its higher structure and is implicitly represented in the **DU**s, thus, even $\mathbf{DU}_{we}$ (the geometry of wall enclosure) and the $\mathbf{DU}_{re}$ (the geometry of roof enclosure) are topological parts of $\mathbf{DU}_{enclosure}$, they are intentionally individual **DU**s.

4.1.3 Technology

Definition

"A technology *is a collection of computational mechanisms to create and instantiate design and functional units satisfying the requirements of a class of functional*



GURE 7. Abstract Enclosure Elements for (a) Wall Enclosure, (b) Enclosure for the Joint between the Wal and Roof, (c) Ground Enclosure and (d) Roof Enclosure.

units in a design context based on specific construction technology *or form generation principles"* [Woodbury et al., 1994].

The **FU**s capture the problems and the **DU**s reach solutions to solve these problems. Technologies, then, are used to describe the computational methods for problem solving in a design context. To describe the notion of a technology, several examples of technologies are illustrated in the following paragraphs.

An example of a technology is a set of grammar rules (grammars are a formalism to describe rule-based computations) that create a partition (**DU**) between two given building spaces (design context) in which the requirements of visual and acoustic privacy (**FU**s) are fulfilled by using the dry wall construction technology. Another example is an executable function in a procedural programming language that wraps an unelaborated wall (**DU**) around a set of spatial **DU**s (design context) in order to satisfy a requirement for weather enclosure (**FU**). This process may lead to the creation of additional **FU**s specifying requirements for the additional **DU**s needed as part of the instantiated **DU**s.

For demonstrating of a concrete example of technologies for a building enclosure see [Woodbury and Chang, 1995b, Woodbury and Chang, 1995a]. One of these technologies, a continuous support technology as shown in Figure 8 is used as an

example to demonstrate the behaviours of technologies and the relationships between technologies and **FUs/DUs**.

Continuous-support_wall-enclosure Technology

Pre-conditions

- A functional unit $\mathbf{FU}_{we}$ for a wall-enclosure and a design unit $\mathbf{DU}_{we}$ allocating it as an abstract surface.
- $\mathbf{FU}_{we}$ is elaborated as an insulated enclosure and thus contains a constituent for continuous support $\mathbf{FU}_{wcs}$.
- $\mathbf{FU}_{wcs}$ is unallocated.

Instantiation function

- Create a design unit allocating $\mathbf{FU}_{wcs}$. Its **geometry** is the same as $\mathbf{DU}_{we}$.
- Create a functional unit hierarchy under $\mathbf{FU}_{wcs}$ comprising constituents for a membrane $\mathbf{FU}_{wcsm}$ and an attachment $\mathbf{FU}_{wcsa}$ shown on the left.
- Create a design unit $\mathbf{DU}_{wcsm}$ respectively allocating $\mathbf{FU}_{wcsm}$.
- In $\mathbf{DU}_{wcsm}$ create a solid representing the wall on the inside of $\mathbf{DU}_{we}$. Constrain the **material** of $\mathbf{DU}_{wcsm}$ to be a member of **{concrete block, metal stud wall, metal panel}**. Choose a **thickness** appropriate to the material.
- $\mathbf{FU}_{wcsa}$ remains unallocated (it will be allocated in the structural technologies of SEED-Config).

A functional unit hierarchy

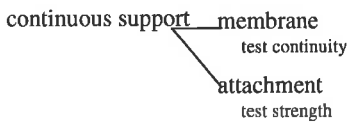


FIGURE 8. An Example of a Technology for the Continuous-Support of a Wall Enclosure is shown on the Top. The Partial **FU** Hierarchy is on the Bottom.

The technology shown in Figure 8 is for the continuous-support component of wall enclosures. The pre-condition describes the context in which this technology can be applied. In this case, the context ensures that there is a need to elaborate the $\mathbf{FU}_{wcs}$. The instantiation process is the action of generating $\mathbf{DU}_{wcsm}$, $\mathbf{FU}_{wcsm}$, and $\mathbf{FU}_{wcsa}$. Generation of $\mathbf{FU}_{wcsm}$ and $\mathbf{FU}_{wcsa}$ is the refinement of **FUs** specifying that at any time, the **FU** hierarchy is always partially completed. Allocating **FUs** into corresponding **DUs** allows us to elaborate the **DUs**. **FUs** and **DUs** can also be refined further into more detailed **FUs** and **DUs** by other technologies in different design contexts. The result of this technology is illustrated clearly in Figure 9.

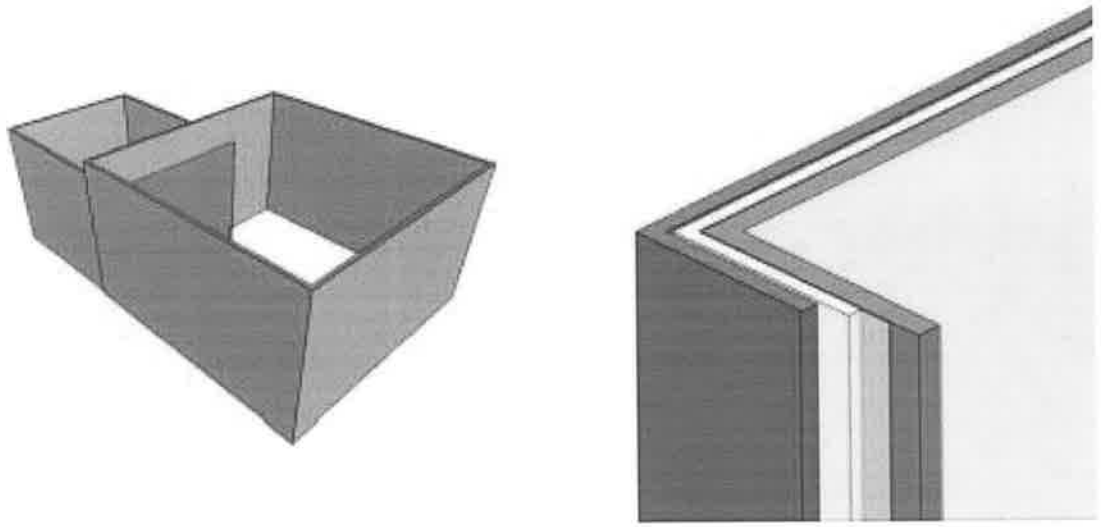


FIGURE 9. Left: the Initial Status of Spaces Enclosure. Right: Wall Continuous Support (Membrane) Design Unit (After Applying Technology).

4.1.4 State

Definition

*"A state in a design space comprises a set of **FUs** and any **DUs** that allocate them. A state is a starting state or the result of applying technologies to a starting state, where each state represents a partial or complete solution to the given problem. Every state defines a design problem, namely, the task of allocating the unallocated **FUs** in that state subject to the context established by the already allocated DUs. The technologies that allocate DUs may also create new **FUs**, so that states, and thus design problems, are elaborated dynamically."* [Woodbury et al., 1994]

States are the major components that form design spaces. They comprise problems as **FU**s and partial or completed solutions to those problems as **DU**s, where **DU**s and **FU**s exist in a relation of one-to-one correspondence. States also record the relation between a problem (**FU**), a solution (**DU**) and the means by which the problem is solved (technology). Design spaces, in general, compose states that connect with each other by the operations required to derive from one state to another.

Given the figure above, a design space is usually visualised using a graph notation such as a tree, or a direct acyclic graph. A state, thus, is a node of that graph. The derivation relation is the edge between two nodes. Such a node provides accessibil-

ity to all changes made to an existing context (**FU**s and **DU**s) and the methods (technologies) used within the time-frame of that particular node.

4.1.5 Design Space

Definition

"A design space is implicitly defined by the combination of a problem statement, a starting state, and a collection of technologies that derive states from states. The space is the collection of all states that can be derived by applying the operators to the starting state under the problem statement." [Woodbury et al., 1994]

Design spaces, regarded as processes of design thinking, record transformations from one state to another. Such transformations may take place based on changes in either the specified problems or solutions of a state (see Chapter 4.1.4) A design space or a partial of a design space is defined by a collection of obvious represented states. In each of these states both the design and the problem may depend on the situations.

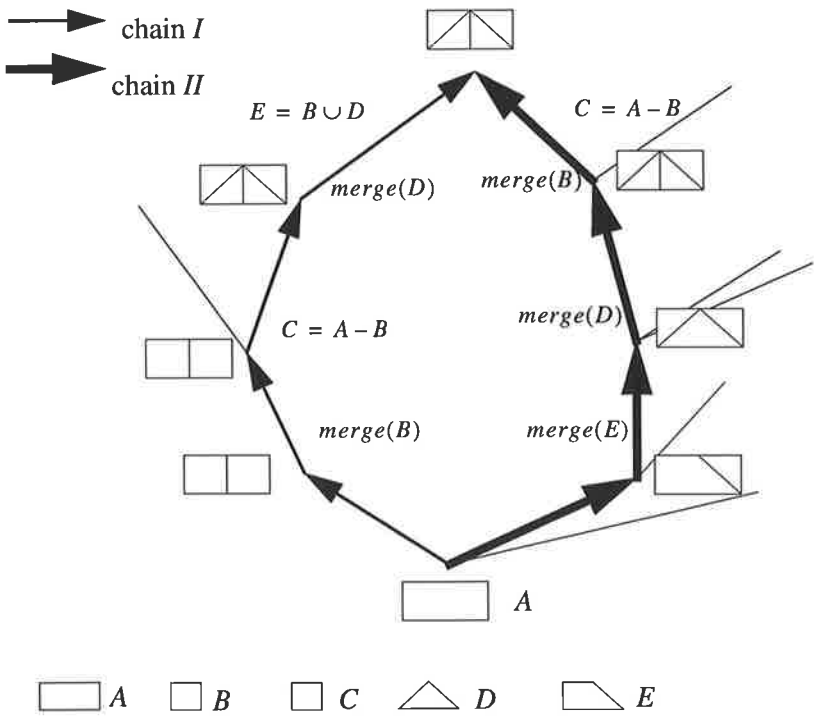


FIGURE 10. An Example of Design Spaces Demonstrates the Transformation of Geometry from one State to another.

Figure 10 is an example of a design space for geometry design. To each state of this design space at least one technology is applied which elaborates or refines **FU**s or **DU**s in its own way. This example shows the design spaces allowing a design state that can have multiple paths starting from the initial state. The existence of multiple paths from one design state to another suggests that there may be more than one possible sequence of design states in the achievement of a particular design state.

4.2 Realisation of the SEED knowledge level

Realisation of a knowledge level requires a computational mechanism that formalises key properties of the knowledge level. By describing a mapping between the knowledge level and an abstract knowledge system, we sketch the computational mechanisms needed for a realisation of the SEED knowledge level.

The symbol level representation might be envisaged as a generic abstract knowledge system that comprises three sets of objects: type, structure and description, and two mechanisms: *ordering over* structure*s* and *resolution methods*. We put this generic abstract knowledge system (as shown in Figure 11) into a representation scheme described in [Woodbury et al., 1998]. Types carry knowledge from the domain of interest, thus referred to as the symbol level representation of knowledge. Structure*s* model domain information which may, or may not be compatible with the knowledge at hand. Structure*s* are the symbol level of the designs (as shown in Figure 11). Description*s* stand for themselves as descriptions of the interaction with set structure*s* with knowledge in the symbol level. *Ordering over* structure*s* ensures consistent relations among structure*s* and resolution methods for generating structure*s*. Woodbury describes a generic version of such an abstract system [Woodbury et al., 1998].

knowledge level		**symbol level**
designs	←→	structures
knowledge	←→	types
descriptions	←→	descriptions
representation scheme		*abstract knowledge system*

FIGURE 11. Mapping between a Generic Abstract Knowledge System Concept and the Corresponding Representation Scheme[Woodbury et al., 1998].

With respect to the SEED knowledge level, a mapping between the SEED knowledge level and this abstract system is illustrated in Figure 12. **FU**s provide a partial decomposition of a design problem on the basis of functional requirements and **DU**s contain the design information. In the abstract system, they represent designs, thus structure*s* in the symbol level. Technologies are about describing how to instantiate a set of **FU**s into a set of **DU**s, then technologies represent knowledge (thus type*s* in the symbol level) from the design domain. State*s* provide accessibility to all **FU/DU** pairs and technologies explored within a certain timeframe, which is also the design information. By organising the design states in a structured fashion so that exploration can occur, design space*s* are the *ordering over* structure*s* in the symbol level. In addition, two relations are implicitly defined in the SEED knowledge level: interaction with users and policy for control strategy are respectively description and *resolution methods* in the symbol level.

In summary, a computational mechanism needed for realising the SEED knowledge level should contain structure, type, description (as illustrated in Figure 12) and two mechanisms (ordering over structures and resolution method). Such a knowledge system is common in the field of knowledge based systems and it occurs in many variations. Among these, a specific representation known as Typed Feature Structures (see Chapter 5), has been adapted [Woodbury et al., 1998] and is described in the next chapter.

knowledge level		**symbol level**
function units	⟷	structures
design units	⟷	structures
design states	⟷	structures
technologies	⟷	types
design spaces	⟷	ordering over structures
Interaction with users	⟷	description
policy	⟷	resolution methods
The SEED knowledge level		*abstract knowledge system*

FIGURE 12. Mapping Between the SEED Knowledge Level and its Realisation Concept.

5

Representation of the knowledge level using Typed Feature Structures

In order to grasp the essence of the SEED knowledge level described in the previous chapter, a computational mechanism is presented in this chapter. Its formalism and an approach for realising the SEED knowledge level using this formalism are demonstrated in this chapter.

In the development of computational linguistic theories, a record-like data structure known as feature structure*s* has been undertaken by many researchers[1], see [Kaplan and Bresnan, 1982, Ait-kaci, 1984, and Pollard and Sag, 1987]. One of the feature structure formalism called "Typed Feature Structures" developed by Bob Carpenter [Carpenter, 1992], is studied closely in this dissertation.

Typed Feature Structures comprises four elements: a set of type*s* organised in an inheritance hierarchy, a frame-like representation(feature structure*s*), a description language and the algorithms above those. The first three elements are referred to as the knowledge components of Typed Feature Structures in the next section. The algorithms are then explained in Chapter 5.2.

A brief description of the components of and Typed Feature Structures their algorithms relating to the SEED knowledge level is given in the following sections. The completed formalism of Typed Feature Structures characterised in "*The Logic of* Typed Feature Structures"[Carpenter, 1992] is recommended to readers who wish a detailed knowledge of the formalism.

1. For a more general introduction about this field see[Shieber, 1986].

Later in this chapter, issues involved in realising the SEED knowledge level using Typed Feature Structures formalism are discussed. This is followed by a discussion about carrying geometric information using this formalism.

5.1 Three Components

Three knowledge components of Typed Feature Structures are type*s*, feature structure*s*, and description*s*. Each component is portrayed in the next sections with discussion of the relationships between them. The formal definition of each term borrowed from [Carpenter, 1992] is also given in each section.

5.1.1 Types

Type*s* are defined as a means of efficiently encoding information. They are organised in a multiple inheritance hierarchy. In such an inheritance hierarchy, information associated with a type is extended in inheriting type*s*, that is, an informational ordering. We say that type α subsumes type τ, if and only if type τ contains more information than type α (the actual algorithm is described later in this chapter). A *bounded complete partial order* (BCPO)[1] condition is implied in this information ordering described above. This condition ensures that for every set of type*s*, there is a most general common subtype to join. Reasoning operations over types are available as the lattice operations such as $\wedge$ (a join symbol, or most general common specialisation) and $\vee$ (a meet symbol, or the most specific common generalisation). A universal type is described as Bottom (written as $\perp$), at which all the type*s* meet.

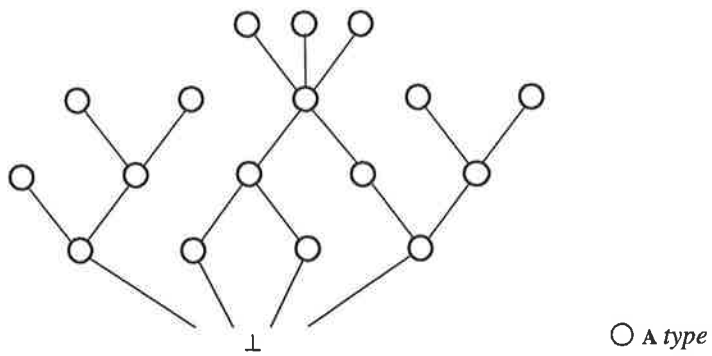


FIGURE 13. A Graph Notation of an Inheritance Hierarchy.

1. For the mathematic explanation of BCPO see [Davey and Priestley, 1994].

All types are organised in so-called "inheritance hierarchies" based on the information inheritance of any two types, that is a subtype carrying more information than its supertypes. With a graph-notation (as shown in Figure 13), inheritance hierarchies can be described as a directed acyclic graph[1]. Each vertex of this graph represents a type, and each edge between any two vertices represents the relationship of information inheritance between any two types. However, cycles are not allowed in inheritance hierarchies. Without cycles in inheritance hierarchies, a subtype of a type can therefore not become the super-type of that same type.

Given the condition of BCPO for ordering information over types, a primary construction of an inheritance hierarchy is known as the information inheritance of all types in which the information inheritance is also recognised as the appropriateness specification. The definition of appropriateness specification is shown in Figure 14). The partial order conditions such as reflexivity, anti-symmetry, and transitivity has to apply onto the appropriateness specification of these types. An inheritance hierarchy is not created unless unification over feature structures occurs. Therefore, types are not needed to be put into inheritance hierarchies before unification.

An *appropriateness specification* over the inheritance hierarchy $<Type, \sqsubseteq>$ and features *Feat* is a partial function $Approp: Feat \times Type \rightarrow Type$ that meets the following conditions:

- *Feature Introduction*
  for every feature $f \in Feat$, there is a most general type $Intro(f) \in Type$ such that $Approp(f, Intro(f))$ is defined

- *Upward Closure / Right Monotonicity*
  if $Approp(f, \sigma)$ is defined and $\sigma \sqsubseteq \tau$, then $Approp(f, \tau)$ is defined and $Approp(f, \sigma) \sqsubseteq Approp(f, \tau)$

FIGURE 14. Definition of Appropriateness Specification[Carpenter, 1992].

1. "A graph is a collection of vertices and edges. Vertices are simple objects that can have names and other properties; an edge is a connection between two vertices."[Sedgewick, 1990] A directed graph means the edges are 'one-way' from one vertex to another vertex. Thus a directed acyclic graph represents a directed graph without cycles—first and last vertex of a path are the same.

5.1.2 *Feature Structure*

A feature structure over Type *and* Feat *is a tuple* $F = \langle Q, \bar{q}, \theta, \delta \rangle$ *where*

- Q: *a finite set of nodes rooted at* $\bar{q}$
- $\bar{q} \in Q$: *the root node*
- $\theta : Q \rightarrow$ Type : *a total node typing function*
- δ: Feat $\times Q \rightarrow$ Q : *a partial feature value function*

FIGURE 15. Definition of a Feature Structure Over Types and Feats[Carpenter, 1992].

In brief, feature structures are frame-like representations based on a set of existing types. Following the definition given in Figure 15, a minimal feature structure simply contains a single node labelled by type information. An informal description of feature structures is that they are a representation for a labelled, rooted, directed, finite node and edge graph[1], which can usually be visualised in graph notation (as shown in Figure 16) or in AVM (attribute-value matrix) notation. Cycles are allowed in feature structures. With these characteristics, feature structures can easily organise domain information in an intricate structure.

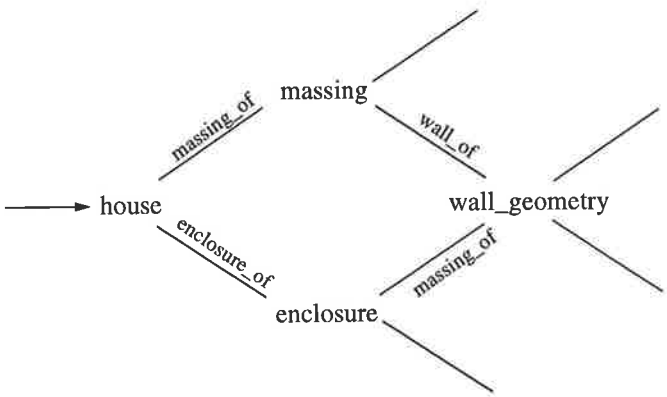


FIGURE 16. An Example Feature Structure in Graph Notion.

1. This graph departs from convention by representing the edges through a function rather than through a binary relation.

Naturally, traversing feature structures is important. Thus the transition function has to be able to be composed. The path is needed for retrieving the value at the end of a path. "*A* path *is a sequence of features and we let Path = Feat* be the collection of* paths" [Carpenter, 1992, page 37]. In terms of graph-notation, a path comprises all the edges from its root node to the final destined node. Using the graph notation in Figure 16, a partial path of *wall_geometry* from *house* is either massing_of → wall_of or enclosure_of → massing_of . The definition of path value is shown in Figure 17.

If $F = \langle Q, \bar{q}, \theta, \delta \rangle$ *and* $\delta(\pi, \bar{q})$ *is defined, then:*

$$F @ \pi = \langle Q', \bar{q}', \delta', \theta' \rangle$$

where:

$$\bar{q}' = \delta(\pi, \bar{q})$$

$$Q' = \{\delta(\pi', \bar{q}') \mid \pi' \in \text{Path}\} = \{\delta(\pi\pi', \bar{q}') \mid \pi' \in \text{Path}\}$$

$$\delta'(f, q) = \delta(f, q) \text{ if } q \in Q'$$

$$\theta'(q) = \theta(q) \text{ if } q \in Q'$$

FIGURE 17. Definition of Path Value[Carpenter, 1992].

5.1.3 Description

The set of descriptions over the collection Type *of types and* Feat *of features is the least set* Desc *such that:*

- $\sigma \in$ Desc *if* $\sigma \in$ Type
- $\top \in$ Desc
- $\pi : \phi \in$ Desc *if* $\pi \in$ Path, $\phi \in$ Desc
- $\pi_1 = \pi_2 \in$ Desc *if* $\pi_1, \pi_2 \in$ Path
- $\phi \wedge \psi, \phi \vee \psi \in$ Desc *if* $\phi, \psi \in$ Desc

FIGURE 18. Definition of Description[Carpenter, 1992].

A description is characterised as a notion for expressing a set of feature structures in a given inheritance hierarchy (as shown in Figure 18). A type, a path, path equation or in-equation and disjunction or conjunction are all descriptions. A description is

satisfied by a set of feature structure*s*. This set of feature structure*s*, however, might be empty if a description is unsatisfiable.

Every feature structure can be called out by some description*s* that includes no disjunction. In the counterpart, every description might have an empty feature structure or more than one satisfied feature structure. As expected, a recursive function will call out a set of pairwise incompatible feature structure*s*. When every feature structure among the selected set subsumes all other feature structure*s*, these feature structure*s* satisfy the description. The mathematical description of *satisfaction* is introduced and illustrated in Figure 19.

Type constraints are also included as a total function $Cons$: Type $\longrightarrow$ Desc in the description. Such a function maps from type*s* to description*s* as the constraint expressions of these type*s* in description*s*. Any feature structure, which satisfies a type τ, must also satisfy the recursive constraints given by $Cons*(\tau)$.

The satisfaction relation $\models$ *between feature structures and description is the least relation such that:*

- $F \models \sigma$ *if* $\sigma \in$ Type *and* $\sigma \sqsubseteq \theta(\bar{q})$
- $F \not\models \mathrm{T}$ *for any* $F \in \mathcal{F}$, *Let* $\mathcal{F}$ *denote the collection of feature structure*
- $F \models \pi : \phi$ *if* $F @ \pi \models \phi$
- $F \models \pi_1 = \pi_2$ *if* $\delta(\pi_1, \bar{q}) = \delta(\pi_2, \bar{q})$
- $F \models \phi \wedge \psi$ *if* $F \models \phi$ *and* $F \models \psi$
- $F \models \phi \wedge \psi$ *if* $F \models \phi$ *or* $F \models \psi$

FIGURE 19. Definition of Satisfaction[Carpenter, 1992].

5.2 Three Algorithms

With an understanding of the components of Typed Feature Structures—type*s*, feature structure*s* and description*s*, we now introduce a set of algorithms over these components. The algorithms are subsumption, unification and *π-resolution*, which are introduced in the following sub-sections.

5.2.1 Subsumption

A feature structure represents the partial information of domain objects at a certain stage. The ability to specify that one structure is more informative and specific than

the other is important. The informational ordering over types is extended to feature structures by the subsumption ordering. An intuitive explanation of subsumption over two feature structures is "*a* feature structure *F subsumes another* feature structure *F' if and only if every two paths which are shared in F are also shared in F', and every type assigned by F to a path subsumes the* type *assigned to the same path in F' in the* type *ordering*"[Carpenter, 1992, page 40]. The definition is shown in Figure 20.

$F = \langle Q, \bar{q}, \theta, \delta \rangle$ *subsumes* $F' = \langle Q', \bar{q}', \theta', \delta' \rangle$, $F \sqsubseteq F'$, *if and only if there is a total function* $h: Q \rightarrow Q'$, *called a morphism such that:*

- $h(\bar{q}) = \bar{q}'$
- $\theta(q) \sqsubseteq \theta'(h(q))$ *for every* $q \in Q$
- $h(\delta(f, q)) = \delta'(f, h(q))$ *for every* $q \in Q$ *and feature* f *such that* $\delta(f, q)$ *is defined*

FIGURE 20. Definition of Subsumption[Carpenter, 1992].

Since the same information may be represented by feature structures with distinct node sets, the informational equivalence must be decidable. It must be possible to distinguish between two distinct node sets with the same information and two distinct node sets with different information. Subsumption is therefore used to separate any two pairs of distinct node sets with the properties described above. These two distinct node sets carrying identical information are defined as having "*alphabetic variance*"[Carpenter, 1992], written as $A \sim B$.

5.2.2 Unification

With respect to subsumption ordering over two feature structures, a unification operation is defined as shown in Figure 21. A unification of two feature structures generates a feature structure carrying more or less information than them altogether. The information derived from a set of distinct sources can be combined with a unique node set to form modulo alphabetic variance. The only way in which unification can fail is when the unified feature structures are carrying inconsistent information. For example, the incomparable types in the same path are in these feature structures. Such inconsistency has been taken care of by subsumption which treats path values of two feature structures which are not subsumed to each other dis-

Suppose $F, F' \in \mathbf{F}$ are feature structures such that $F \sim \langle Q, \bar{q}, \theta, \delta \rangle$ and $F' \sim \langle Q', \bar{q}', \theta', \delta' \rangle$ are such that $Q \cap Q' = \emptyset$. We define an equivalence relation $\bowtie$ on $Q \cup Q'$ as the least equivalence relation such that:

- $\bar{q} \bowtie \bar{q}'$
- $\delta(f, q) \bowtie \delta(f, q')$ *if both are defined and* $q \bowtie q'$

The unification of F and F′ is then defined to be:

$$F \sqcup F' = \langle (Q \cup Q')/_{\bowtie}, [\bar{q}]_{\bowtie}, \theta^{\bowtie}, \delta^{\bowtie} \rangle$$

where:

$$\theta^{\bowtie}([q]_{\bowtie}) = \sqcup\{(\theta \cup \theta')(q') \mid q' \bowtie q\}$$

and

$$\delta^{\bowtie}(f, [q]_{\bowtie}) = \begin{cases} [(\delta \cup \delta')(f, q)]_{\bowtie} & \text{if } (\delta \cup \delta')(f, q) \text{ is defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

If all of the joins in the definition of $\theta^{\bowtie}$ exist. $F \sqcup F'$ is undefined otherwise.

FIGURE 21. Definition of Unification[Carpenter, 1992].

tinctly. The efficiency of unification, therefore, is dependent on the efficiency of the subsumption operation over two types in the same path.

5.2.3 π-resolution

With the definitions of type, feature structure, description now given and with the two algorithms in place, we can now turn our attention to the issue of constraints resolution. Constraint resolution is the attempt to satisfy the queries[1] of constraint equations. The resolution mechanism proceeds with the process of constraint resolution, substructure (of a given feature structure) by substructure, until all the substructures satisfy (or fail) the constraint on their types. During the constraint resolution stages, the mechanism adds as much information as is needed to the substructure in order to satisfy its constraints. This constraint resolution mechanism is called "*π-resolution*". It captures a relationship between descriptions and feature structures and performs as the main exploration mechanism in the design space explorer.

5.3 A representation using Typed Feature Structures

Issues relating the implementation of the SEED knowledge level using Typed Feature Structures are discussed in this section in the light of an understanding of the

1. "A query is simply an arbitrary description"[Carpenter, 1992].

UNIVERSITY OF ADELAIDE LIBRARY

basic mechanics of Typed Feature Structures. Chapter 4.2 shows how the SEED knowledge level corresponds to an abstract symbol level representation (illustrated in Figure 12). In this section, such representation is further considered in the theory of Typed Feature Structures (as shown in Figure 22).

Recall that the symbol level representation (as shown in Figure 12) in Chapter 4.2 contains three components (type, structure and description) and two mechanisms.(ordering over structures and resolution method)over these components. Implementing these concepts using Typed Feature Structures requires a mapping from these concepts to the Typed Feature Structures' components and algorithms.

symbol level		**symbol level**
structures	⟷	feature structures
types	⟷	TFS types
descriptions	⟷	TFS descriptions
ordering over structures	⟷	subsumption ordering
resolution methods	⟷	π-resolution
abstract knowledge system		*Typed Feature Structures*

FIGURE 22. Mapping Between the Abstract System Concepts and the Corresponding Typed Feature Structures(TFS) Concepts.

Among the abstract components of the symbol level, type*s* encompass the domain information, in terms of Typed Feature Structures, these correspond to TFS type*s*. Structures that express the models of this domain information represented by type*s*, are thus implemented by feature structure*s*. Description*s* in the knowledge system can be mapped to TFS description*s* to convey this domain information.

In terms of the two mechanisms: ordering over structures corresponds to is subsumption ordering over feature structure*s*, and the resolution methods used in Typed Feature Structures correspond to *π-resolution*. The relation between each pair of concepts are illustrated in Figure 22. Furthermore, a complete picture of mapping from the knowledge level to the symbol level representation is shown in Figure 23.

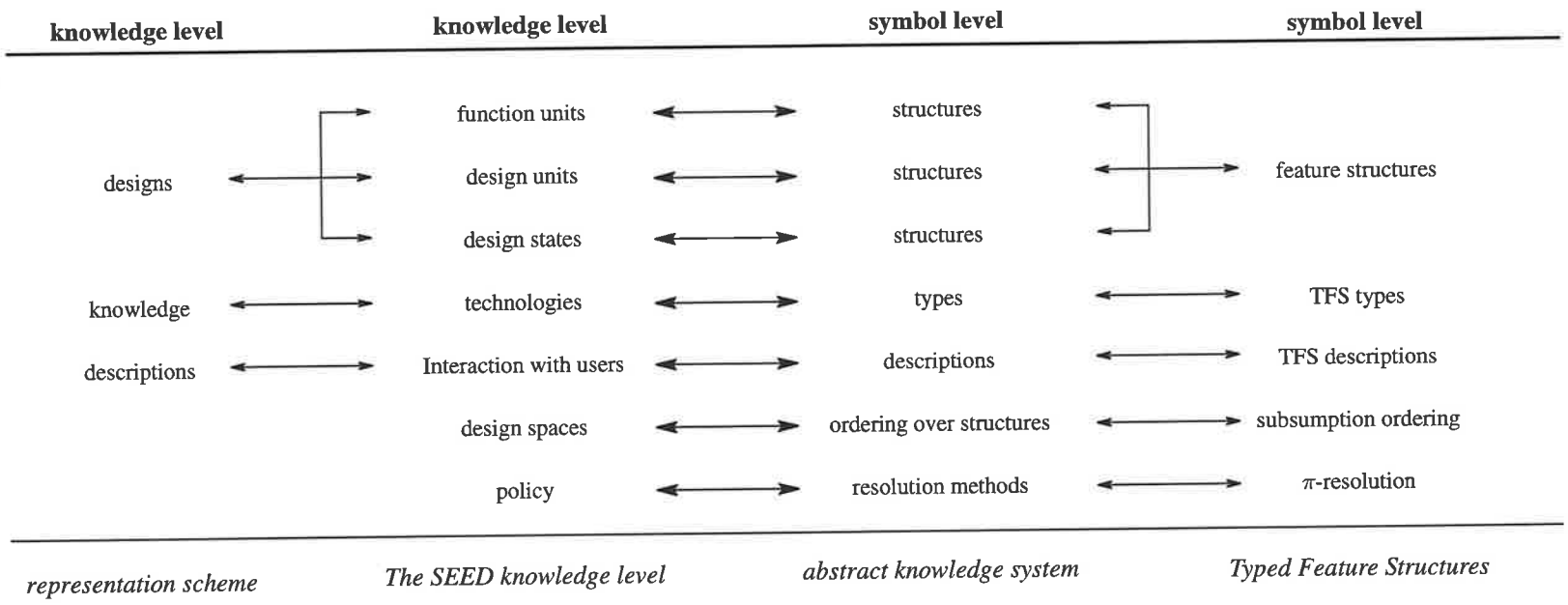


FIGURE 23. A Complete Mapping from the Knowledge Level to its Symbol Level Representation.

5.4 Toward representing geometric information

Having sketched the symbol level representation in the previous section, we now turn to the issue of representing geometric information using Typed Feature Structures. As stated in Part I, geometric information plays an essential role in design space explorers. Few design space explorers (as reviewed in Chapter 3.2) have achieved the goal of carrying geometric information in a clean and efficient formalism. Several issues need to be resolved to have symbol level representation carrying geometric information using Typed Feature Structures. These issues concern the—infinite and continuous domain; geometry that needs no sub-structure; minimal subsumption and minimal partiality.

Representing infinite, continuous domain information

A key characteristic of geometric information is its continuous and infinite domain. A work around solution for standard Typed Feature Structures representing an integer '4' is shown in Figure 24. This solution is not practical and is even impossible as far as geometric information is concerned. Firstly, such a special type can not cover all the integer values needed for the design domain, nor cover all the geometric information. Secondly, the structure of geometric information is intensional and imposed by users that will not need any sub-structure (as elaborated in the next subsection). Thus, a sound and effective solution is needed for representing this infinite continuous domain—geometric information.

Numbers inherit universal.

4 inherit Numbers.

3 inherit Numbers.

2 inherit Numbers.

1 inherit Numbers.

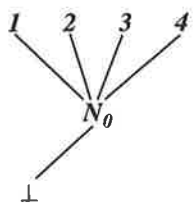


FIGURE 24. A Description of Atomic Values in a Formal System.

Geometric information needs no sub-structure

With Typed Feature Structures, we represent the structure of objects through feats, a finite sets of features. We also accept that there exist atomics that have no substruc-

ture with respect to a given type hierarchy. Therefore, we must represent geometric information in the same way—through Feats. Our geometric objects must be featureless. Now we hit shape grammars which show the importance of a uniform and general subpart relation on continuous information. Shapes contain infinite features. Thus a major question is how to resolve the apparent conflict between feature structures with their discrete finite parts and shape-like objects which, though finitely represented, contain infinite numbers of subparts?

Minimal subsumption, minimal partiality

Questions arise as to when is infinite information such as the geometric information encountered? How should infinite domain information be ordered? An appropriateness specification must apply to this infinite information and must confront the formal definition (as shown in Figure 14). In other words geometric information is less informative than other types of information and must therefore be expressed and computed efficiently. For this purpose, minimal subsumption and minimal partiality over the geometric information are required. This ensures that only sufficient geometric information is used in the system and provides incremental refinements over the geometric information.

Summary

Representing geometric information using Typed Feature Structures is the key to implementing a design space explorer. Three issues are unfolded in this section—the infinite and continuous domain; the fact that geometry needs no sub-structure; and minimal subsumption and minimal partiality. These issues require a special treatment from Typed Feature Structures. An approach called Geometric Typed Feature Structures is introduced and discussed in the next chapter. This approach tackles the issues described above and incorporates most of the functionality of Typed Feature Structures along with a geometry extension.

6
Geometric Typed Feature Structures

With an understanding of the knowledge level of a design space explorer and its symbol level realisation using Typed Feature Structures, we now turn toward representing geometric information within the theory of Typed Feature Structures. We seek a way of carrying geometric information within Typed Feature Structures that provides the computational representation needed for realising a design space explorer.

Firstly, if we presume Typed Feature Structures are the only carriers of information in our design space explorer, then we must regard feature structures and their types as the carriers for any and all information. We choose feature structures representing a piece of geometric information to be instances of atomic types; that is, such types carrying geometric information have no features. The type hierarchy itself thus does almost all the duty of defining our geometric representation.

The reasons for using types as the carriers for geometric information (as hinted in Chapter 5.4) as well as for other continuous domain objects are quite straightforward. First of all, if we use feature structures as the carriers, we still need to use types to carry partial domain information since types provide the atomic units of feature structures. Secondly, according to our observations, the domain objects can have no substructure (as mentioned in Chapter 5.4), which is similar to the concept of the atomic concept units of types. Thus, we must use types to carry the geometric information of the domain objects. It turns out that the infinite sub-features of shapes occur in a subpart lattice, which we will later show forms a valid inheritance hierarchy for Typed Feature Structures. So, infinitely featured shapes depend upon the same mechanism as completely unfeatured atomic geometric objects.

By using types as the carriers, we need to find a clear, formal and uniform means for carrying domain information between two types. As we have observed, the objects of interest to us can be arranged in relations that are also of interest to us. Some of these relations have the required properties of inheritance hierarchies; namely the BCPO properties [Carpenter, 1992, page 13]. For example, in the domain of real intervals, the join of $[-12.2, 17.74]$ and $[5.023, 20.47]$ is consistent as $[5.023, 17.74]$ under the pointset relation $\supseteq$, and a most general type (carrying an infinite interval $\perp$) is trivial to describe. An example of a partial order over these intervals is shown in Figure 25. An elaborated definition is shown in Chapter

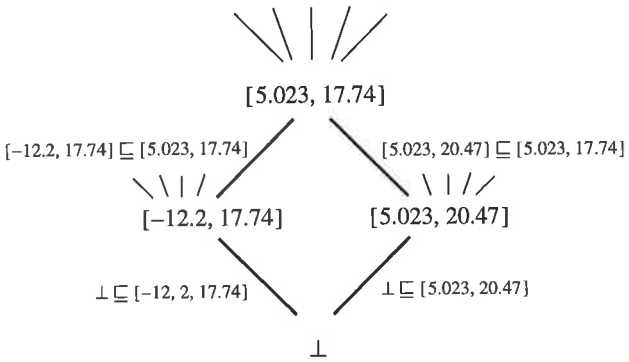


FIGURE 25. Example of a Complete Partial Order of Three Real Intervals and a Bottom.

6.3.10.

Following this example, a partial order $\langle [\Re, \Re], \supseteq \rangle$ (as shown in this interval example) under the relation $\supseteq$ can act as an inheritance hierarchy to a type system. Such inheritance hierarchies are not finite since the domain ($\Re$) is infinite. However in design space exploration we can trace out only a finite part of a space. Even if we could do it, enumerating an infinite type hierarchy such as that given by $\langle [\Re, \Re], \supseteq \rangle$ would be inefficient as it would contain an infinity of objects which we would never use. We therefore need a finitely describable way of representing such infinite hierarchies.

By reviewing the context of Typed Feature Structures, we found that there are two main reasons why Carpenter chose to define a finite inheritance hierarchy. One reason is that finite sets are sufficient for the application described in the book, and

another is that Carpenter uses construction method (for inheritance hierarchies) views which demand finite ISA networks yielding finite inheritance hierarchies. None of these reasons apply to our domain of interest so that having a finite inheritance hierarchy is not a necessary condition for Typed Feature Structures. We can relaxed Carpenter's definition to allow infinite types.

In summary, the problems of using types as carriers for domain relevant information are 1) continuous domains are infinite—thus we need a finitely describable way of representing infinite inheritance hierarchies and 2) the relations between domain objects must conform to an inheritance relation between the types carrying these objects.

Before describing of our solution to the problems described above, we need to address the terminology for the types we are using. The terms used here follow the literature[Carpenter, 1992, Burrow and Woodbury, 1998] very closely. First of all, the top of an inheritance hierarchy is called an absurd type, and the bottom is called a universal type. The standard types described in [Carpenter, 1992] are called Succession Types (as shown in Figure 26), as they succeed one another in an inheritance hierarchy specification. We describe a different kind of types called Order Types in which the ordering of types can be computed by direct comparison (as shown in Figure 27). The name for "Order Type" is given because of its *ordering* characteristic.

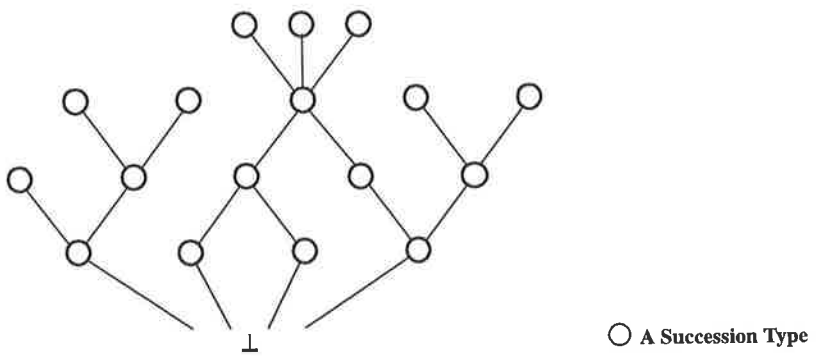


FIGURE 26. Example of a Succession *InheritanceHierarchy*.

The elements of a set of types comprising both Succession Types and Order Types are called *regular* types, to distinguish them from absurd types. Consequently, regular types and the absurd type are from the *universe of* types. A detailed definition of the set of Order Types is given in the following section. The theory following that of Typed Feature Structures with the necessary extensions to include regular types is called Geometric Typed Feature Structures.

Geometric Typed Feature Structures is a domain extension of Typed Feature Structures. The theory and algorithms described in the previous chapter are still valid within Geometric Typed Feature Structures. Therefore, the remainder of this chapter only describe the extension—Order Types—and its behaviour.

In the following sections we describe Order Types in terms of

1. their characteristics;
2. what kind of objects can serve as Order Types (models);
3. discuss the representation of Order Types carrying two kinds of domain information: numerical values and geometric information.

6.1 The characteristics of Order Types

In order to solve these two problems—representing the continuous domain and defining the inheritance relation of domain relevant objects—several differences between Order Types and Succession Types are discovered. In this section, we describe the four characteristics of Order Types which allow such types to carry continuous domain information. Then, in the following section, we will describe the model space which Order Types will represent.

Five characteristics of Order Types are 1) Order Types are un-evaluated, 2) Order Types have no feats, 3)descriptions of Order Types have no paths, 4)Order Types carry no constraints and 5)some Order Types are intentional. We devote a section describing each of these characteristics in the following text.

6.1.1 Order Types are un-evaluated

Regular types, that is, both Succession Types and Order Types, are defined by inheritance. In Succession Type finite inheritance hierarchies are explicitly constructed, either directly or through closure operations over ISA networks

[Carpenter, 1992]. In the Order Type we do not explicitly store a type hierarchy. Instead we determine information consistency by direct examination of type tokens.

Therefore, if the value has not been evaluated, the Order Type carrying this value will not exist in the type system. This reflects the first characteristic of Order Types—Order Type hierarchies are un-evaluated until users need to describe certain domain information that is implicit in them.

We treat Order Types simply as non-decomposable values, that is, as far as feature structures are concerned they only carry information in terms of their subsumption relation. This gets rid of the need to construct infinite Order Types and to incorporate them into the type system, and makes it clear that Order Types are simply acting like values.

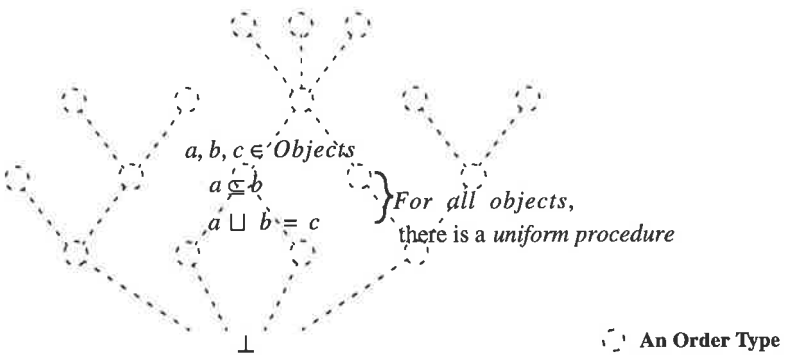


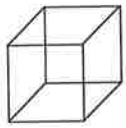
FIGURE 27. Example of an Order Type Hierarchy.

Therefore, Order Types need not be evaluated until a description specifies a particular value, *i.e.*, Order Type hierarchies are presumed to exist but remain un-evaluated until needed in the universe of types.

6.1.2 Order Types have no feats

The main motivation for employing Order Types is to represent continuous domain objects, such as point-sets, numbers, and intervals. These objects have no sub-structures (as hinted in chapter 5). If we want to represent the structure of domain information, we need to utilise higher level structures. For example, in Figure 28, the topological relation between a solid and its faces can be organised in a feature structure-fashion, such that each feat of this feature structure carries a face as its value.

Thus, there is no need to provide a sub-structure for Order Types and consequently, Order Types need no feats.



A cube

$$\begin{bmatrix} \textit{Cube} & \\ \text{FACE_1:} & [\text{face}] \\ \text{FACE_2:} & [\text{face}] \\ \text{FACE_3:} & [\text{face}] \\ \text{FACE_4:} & [\text{face}] \\ \text{FACE_5:} & [\text{face}] \\ \text{FACE_6:} & [\text{face}] \\ \text{GEOM:} & [\text{solid}] \end{bmatrix}$$

face and solid are order types

A *feature structure* representing the cube

FIGURE 28. An Example Represents the Properties of a Cube with a Higher Level Structure (in AVM Notation).

Furthermore, all properties of an Order Type instance can be computed from the Order Type instance alone. Once again, for an integer '4', there is a property that it is divisible by two giving an integer result. We can compute this property by computing over the value itself (4), without using feats at all. We argue again that Order Types have no feats.

6.1.3 Descriptions of Order Types have no paths

Descriptions contain four key components: types, values at the end of paths, path equations and and/or expressions of descriptions. Without feats of their own, Order Types will have no path values and no path equations. Thus a description of Order Types comprises only a set of Order Types.

Let us recall that the definition of a path value $F@\pi$ is the value of the feature structure F at a path π (which must be defined). If F is typed by an Order Type in Geometric Typed Feature Structures, there will be no path for F as F will have no feats at all. Thus, there will be no path values for F. Consequently, there are no path equations and path description for Order Types either, since Order Types have no path values. This means that the descriptions over Order Types are merely

descriptions of those types, along with the disjunctions and conjunctions over them.

Typed Feature Structures' Description	*Geometric Typed Feature Structures' Description*
$\sigma \in$ Desc *if* $\sigma \in$ Type	$\sigma \in$ Desc *if* $\sigma \in$ Type
$\top \in$ Desc	$\top \in$ Desc
$\pi : \phi \in$ Desc *if* $\pi \in$ Path, $\phi \in$ Desc	
$\pi_1 = \pi_2 \in$ Desc *if* $\pi_1, \pi_2 \in$ Path	
$\phi \wedge \psi, \phi \vee \psi \in$ Desc *if* $\phi, \psi \in$ Desc	$\phi \wedge \psi, \phi \vee \psi \in$ Desc *if* $\phi, \psi \in$ Desc

FIGURE 29. The Descriptions of Order Types have no Path Values.

In Figure 29, we compare Carpenter's[1] definition of description[Carpenter, 1992, page 52] with Geometric Typed Feature Structures[2]. Without path values, the definition of description becomes only the type description and the conjunction/disjunction of types, which are also types (or sets of types). A definition of a Geometric feature structure is given in Figure 30.

A Geometric feature structure over Order Types *is a tuple* $F = \langle Q, \bar{q}, \theta, \delta \rangle$ *where*

- Q: *a singleton, rooted at* $\bar{q}$
- $\bar{q} \in Q$: *the root node, and the sole member of* Q
- $\theta : Q \rightarrow$ Type : *a total node typing function*

FIGURE 30. Definition of a Geometric Feature Structure Over Order Types.

The satisfaction of Order Types is also simplified by such a characteristic. As shown[3] in Figure 31, the definition of satisfaction of Geometric Typed Feature

1. The symbols are borrowed from Carpenter[Carpenter, 1992], and the definitions are given in the previous chapter.
2. In the right side of Figure 29, this amounts to saying that descriptions are combinations of types under conjunction and disjunction. Remember that the base cases for descriptions are $\sigma \in$ Desc *if* $\sigma \in$ Type , so all combinations of *Desc* are combinations of types.
3. Same as above.

Structures covers the absurd type, type satisfaction and conjunction/disjunction of satisfaction of two types.

Typed Feature Structures' Satisfaction	*Geometric Typed Feature Structures' Satisfaction*
$F \models \sigma$ *if* $\sigma \in$ Type *and* $\sigma \sqsubseteq \theta(\bar{q})$	$F \models \sigma$ *if* $\sigma \in$ Type *and* $\sigma \sqsubseteq \theta(\bar{q})$
$F \not\models \top$ *for any* $F \in \mathcal{F}$	$F \not\models \top$ *for any* $F \in \mathcal{F}$
$F \models \pi : \phi$ *if* $F @ \pi \models \phi$	
$F \models \pi_1 = \pi_2$ *if* $\delta(\pi_1, \bar{q}) = \delta(\pi_2, \bar{q})$	
$F \models \phi \wedge \psi$ *if* $F \models \phi$ *and* $F \models \psi$	$F \models \phi \wedge \psi$ *if* $F \models \phi$ *and* $F \models \psi$
$F \models \phi \wedge \psi$ *if* $F \models \phi$ *or* $F \models \psi$	$F \models \phi \wedge \psi$ *if* $F \models \phi$ *or* $F \models \psi$

FIGURE 31. Comparison of Satisfaction of Typed Feature Structures with Satisfaction of Geometric Typed Feature Structures.

6.1.4 Order Types carry no constraints

As the domain described comprises a set of objects under a natural relation such as *point-set containing*, there is no need for further structure. What needs to be represented is nothing more or less than a chosen relation over a chosen set of domain objects. The only restriction is that the relation has to have the BCPO properties to form a well-behaved inheritance hierarchy, so that it can stand in an isomorphism to that hierarchy. Introducing $cons(t) \neq \phi$ will change that desired state. For example, in Figure 32, three solids are under a point-set superset relation. If the constraints over these Order Types are allowed then there will be no way of representing the cuboid marked *c* in Figure 32.

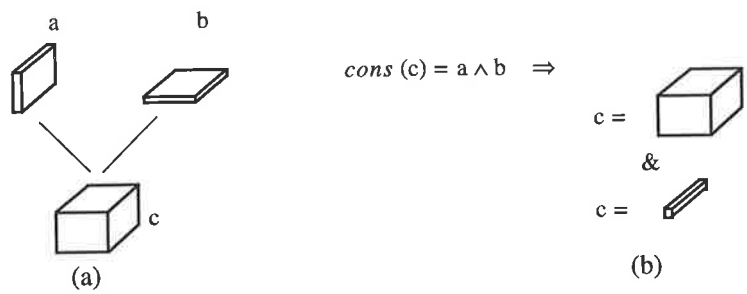


FIGURE 32. Order Types Need no Constraints.

Besides, we do not explicitly represent Order Types prior to their creation. There can be no type tokens with which constraints could be associated.

6.1.5 Some Order Types are intentional

We shall now discuss some other properties of Order Types. Although it seems natural to think of an integer interval as an extensional type, it can be a nuisance since it generates structure sharing which prevents us from specialising a value by unification.

For example, using Burrow and Woodbury's feature structure example[1][Burrow and Woodbury, 1998]:

```
human
FOODSTUFF: animal
        LEG_COUNT: interval_0_4
PET: animal
    LEG_COUNT: interval_0_4
```

If intervals were extensional the above is the alphabetic variant of the following:

```
human
FOODSTUFF: animal
        LEG_COUNT: [1] interval_0_4
PET: animal
    LEG_COUNT: [1]
```

Then if a constraint identifies that we only eat four legged animals, we get

```
human
FOODSTUFF: animal
        LEG_COUNT: [1] interval_4_4
PET: animal
    LEG_COUNT: [1]
```

preventing us from owning snakes! If instead, we treat these types as being intensional, we avoid this problem.

6.2 Order Type Models

With an understanding of the characteristics of Order Types, we are going to reveal what kind of objects can serve as Order Types—the Model space of the Order Types. There are three criteria: domain relevance, the chosen relation has to be a BCPO, and efficiency of the computations corresponding to subsumption and unification. Each requirement is described in detail in the following sub-sections:

1. The syntax of this example follows that used in the description of Kryos: an implementation of Typed Feature Structures.

6.2.1 Relevance to domain for objects and relations

The values carried by the instances of Order Types must be domain relevant as the relation over these values that must be the inheritance hierarchy. One example of such objects regarding of geometry is two-dimensional rectangles. This 2D rectangle representation is used for several domain representation. Efficient constraint resolution algorithms computed over those rectangles are well developed. Another example is the 3D point-set, which is also of great interest in this dissertation.

Relevant relations over those objects are also essential for Order Types. The example of the 3D point-sets, point-set containing is one relevant relation, that $P(A) \supseteq P(B)$ devoting all the points of B are also points of A.

6.2.2 The Bounded Complete Partial Order (BCPO) condition

A BCPO contains these conditions: partial order condition (reflexivity, anti-symmetry, and transitivity), type consistency and a unique bottom. As shown in Figure 33, partial order conditions form the basic property of a BCPO: partial order.

In addition, the existence of consistent joins is a defining characteristic of inheritance hierarchies for Typed Feature Structures. Informally, if a set of types is consistent, there is a single most general satisfied type that is subsumed by them all. This type subsumes all other types subsumed by all members of the set. More formally, defining $S \in \wp(\text{OrderTypes})$ [1] and the upper bound t, $S \otimes t, s \sqsubseteq t, s \in S$. Let $\mathrm{T} = \{t | S \otimes t, s \sqsubseteq t, s \in S\}$ be a set of upper bounds of S. Let $\bigsqcup S = t, when\ \exists t \in \mathrm{T}$, such that $S \otimes t \wedge t \sqsubseteq v, v \in \mathrm{T}$. A unique bottom is ensured by enforcing a universal type as described before. These two conditions associated with the partial order condition are the key of describing Order Types. In other words, Order Types and their associated relations have to confront the BCPO conditions for being included

1. As a power set $\wp(OrderTypes)$ consists of all subset of Order Types.

in inheritance hierarchies. Given the indication in [Carpenter, 1992], the simplest partial order that confronts the BCPO conditions is a tree.

Let P be a partial order set under relation $\leq$ $for \forall a, b, c \in P$

(i) reflexivity	$a \leq a$
(ii) anti-symmetry	$a \leq b$ and $b \leq a$, $imply$ $a = b$
(iii) transitivity	$a \leq b$ and $b \leq c$, $imply$ $a \leq c$

FIGURE 33. Requirements of a Partial Order Set[Davey and Priestley, 1994].

6.2.3 Efficient algorithms for subsumption/unification

Considering a mapping from a continuous domain, such as real intervals, an efficient domain representation is required. When one specifies an Order Type to carry a certain domain information, such information has to be efficiently represented for a straightforward reason: computability.

Order Types are specified under domain relations with given mechanisms, so that the subsumption/unification relations are defined in each Order Type. Consequently, each Order Type is responsible to the algorithms over that specified type. Thus, when one defines a useful Order Type, extra care has to be put on the efficiency of subsumption/unification algorithms defined over that Order Type simultaneously. In addition, without path values, unification of two Order Types is simply based on the subsumption relations which are defined within the domain of interest.

One useful example is the interval subset. Computing the interval subset of two intervals is trivial if two intervals are specified. The intervals under subset relation can be used as the information carried by Order Types. The other example is point-set inclusion over 3D point-sets. The algorithms are more complex than interval subset. But if given an efficient geometric representation such as non-manifold geometry (described in more detail in the later this chapter), the Boolean algorithms are linear.

In contrast, the sub-graph of a graph is not an ideal candidate for the relevant domain relation even if it might be domain relevant in some cases.

6.3 Representing Order Types

With the knowledge of Order Types, we are going to present a set of Order Types which are domain approximated and to demonstrate the representation of Order Types carrying geometric information in Section 6.4. Before that, the numerical values are focused as they are needed for geometric information as well as their atomic values which are not representable in Succession Types. As indicated, two sets of values identified are numerical values and pointset geometric information. Each information paired with one domain relevant relation, such as *less than* relation $\leq$, define one Order Type.

Given three requirements described in the previous section: domain relevancy, the BCPO condition and efficient algorithms for subsumption and unification, four crucial conditions are raised. Firstly, the need for numeric data and geometric information is trivial, as well as the relations including lifted, $\leq$, $\geq$, and interval. The BCPO condition (including partial order, a unique bottom and a least upper bound) and the efficient algorithms are three key conditions for expressing Order Types. In addition, a way to express the members of the set is the fourth condition. For clarification purpose, the BCPO condition is carried on as two steps; partial order (*reflexivity, anti-symmetry* and *transitivity)* and the BCPO (a unique bottom and a *least upper bound*) condition as mentioned above.

As the remaining of this chapter, we discuss a set of examples for carrying numerical values. 12 Order Types demonstrated in the following sections carry a domain information which pair one of four relevant relations (antichain lifted, chain ascending, chain descending and interval inclusion) with one of three given domains (real numbers, integers and whole numbers) respectively. These identified relation/domain pairs are:

1. The lifted reals
2. The lifted integers
3. The lifted whole numbers
4. The ascending reals
5. The descending reals
6. The ascending integers
7. The descending integers
8. The ascending whole numbers

9. The descending whole numbers

10. The real intersection intervals

11. The integer intersection intervals

12. The whole number intersection intervals

These 12 examples applied with four Order Type conditions described above are illustrated in the following sections.

6.3.1 The lifted reals

Given the ordered set $\overline{\Re} = \langle \Re, \bullet \rangle$ where $a \bullet b$ iff $a = b$ we form $\overline{\Re_\perp}$, which we call the *'lifted' reals* by adding an element $\perp$, $\Re_\perp := \Re \cup \{\perp\}$. We define an order relation '$\bullet$' ($\overline{\Re}$ is an antichain[1]) on $\Re_\perp$ as $\overline{\Re_\perp} = \langle \Re_\perp, \bullet \rangle$. The definition is

$$a \bullet b \text{ if } a \bullet b \text{ in } \overline{\Re} \qquad \textbf{(EQ 1)}$$

$$a \bullet b \text{ if } a = \perp \text{ and } b \in \Re_\perp \qquad \textbf{(EQ 2)}$$

Usually we refer to $\overline{\Re_\perp}$ simply as $\Re_\perp$. A partial inheritance hierarchy formed by $\langle \Re_\perp, \bullet \rangle$ is shown in Figure 34. With this definition, the four conditions described above apply as follows:

1. $\langle \Re_\perp, \bullet \rangle$ is a partial order.

Let $a, b, c \in \Re_\perp$,

reflexivity: $a \bullet a$ is true by definition (see equation: 1)

anti-symmetry: $a \bullet b$ and $b \bullet a$ imply $a = b$

by exhaustive enumeration,

if $a \bullet b$, either $a = b$ or $a = \perp, b \neq a$,

in the former,

$$a \bullet b \text{ and } b \bullet a \Rightarrow$$

$$a \bullet a \text{ and } a \bullet a \Rightarrow$$

$$a \bullet a \Rightarrow a = b$$

in the latter,

$$a \bullet b \text{ but } \neg b \bullet a$$

---

1. "*The ordered set P is an antichain if* $x \leq y$ *in P only if* $x = y$." [Davey and Priestley, 1994].

$\therefore (a \bullet b)$ and $(b \bullet a)$ and $b \neq a$ are false.

$\therefore (a \bullet b$ and $b \bullet a$ imply $a = b)$ is true. □

transitivity: $a \bullet b$ and $b \bullet c$ imply $a \bullet c$

by exhaustive enumeration.

if $a \bullet b$, either $a = b$ or $a = \perp, a \neq b$ **(EQ 3)**

in the former,

$a \bullet b$ and $b \bullet c \Rightarrow$

$a \bullet a$ and $a \bullet c \Rightarrow a \bullet c$

in the latter,

$a \bullet b$ and $b \bullet c \Rightarrow$

$\perp \bullet b$ and $b \bullet c \Rightarrow$

Note that $b = c$, as $b \neq a$ from [equation: 3].

$\perp \bullet b$ and $b \bullet c \Rightarrow$

$\perp \bullet c$ and $c \bullet c \Rightarrow \perp \bullet c$,

but $a = \perp$, so $a \bullet c$.

$\therefore ((a \bullet b)$ and $(b \bullet c)$ imply $(a \bullet c))$ is true.□

2. $\langle \Re_{\perp}, \bullet \rangle$ is a BCPO.

As mentioned before, a tree is a BCPO. A 'lifted' anti-chain is a tree, thus, $\langle \Re_{\perp}, \bullet \rangle$ is a BCPO. □

3. There must be an efficient means of computing subsumption and unification. We take $\langle \Re_{\perp}, \bullet \rangle$ as an inheritance hierarchy such that $a \sqsubseteq b = a \bullet b$.

Subsumption

$\perp \sqsubseteq a, a \in \Re$

$a \sqsubseteq a, a \in \Re_{\perp}$

The subsumption algorithm is trivial and computed based on its simple symbol comparison.

Unification

$\perp \sqcup a = a, a \in \Re_{\perp}$

$a \sqcup a = a, a \in \Re$

The unification algorithm is trivial and computed based on its simple symbol comparison.

4. There must be a way to specify members of the set. Elements of $\Re_{\perp}$ stand as names for themselves. Explicit expressions over $\Re$ with the usual operators of from $\{ +, -, \times, / \}$, can also stand as names for elements of $\Re_{\perp}$.

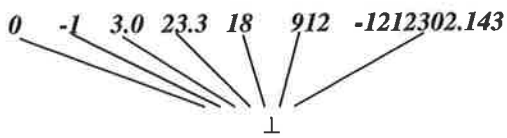


FIGURE 34. A Part of 'Lifted' Reals as an *InheritanceHierarchy*.

6.3.2 The lifted integers

Given the ordered set $\bar{Z} = \langle Z, \bullet \rangle$ where $a \bullet b$ iff $a = b$ we form $\overline{Z_\perp}$, which we call the *'lifted' integers* by adding an element $\perp$, $Z_\perp := Z \cup \{\perp\}$. We define an order relation '$\bullet$' ($\bar{Z}$ is an antichain[1]) on $Z_\perp$ as $\overline{Z_\perp} = \langle Z_\perp, \bullet \rangle$. The definition is

$$a \bullet b \text{ if } a \bullet b \text{ in } \bar{Z} \qquad \textbf{(EQ 4)}$$

$$a \bullet b \text{ if } a = \perp \text{ and } (b \in Z_\perp) \qquad \textbf{(EQ 5)}$$

Usually we refer to $\overline{Z_\perp}$ simply as $Z_\perp$. A partial inheritance hierarchy formed by $\langle Z_\perp, \bullet \rangle$ is shown in Figure 35. With this definition, the four conditions described above apply as follows:

1. $\langle Z_\perp, \bullet \rangle$ is a partial order.

   Let $a, b, c \in Z_\perp$,

   reflexivity: $a \bullet a$ is true by definition (see equation: 4)

   anti-symmetry: $a \bullet b$ and $b \bullet a$ imply $a = b$

   by exhaustive enumeration,

   if $a \bullet b$, either $a = b$ or $a = \perp, b \neq a$,

   in the former,

   $$a \bullet b \text{ and } b \bullet a \Rightarrow$$

   $$a \bullet a \text{ and } a \bullet a \Rightarrow$$

   $$a \bullet a \Rightarrow a = b$$

   in the latter,

   $$a \bullet b \text{ but } \neg b \bullet a$$

---

1. *"The ordered set P is an antichain if $x \leq y$ in P only if $x = y$."* [Davey and Priestley, 1994].

$\therefore (a \bullet b)$ and $(b \bullet a)$ and $b \neq a$ are false.

$\therefore (a \bullet b$ and $b \bullet a$ imply $a = b)$ is true. □

transitivity: $a \bullet b$ and $b \bullet c$ imply $a \bullet c$

by exhaustive enumeration.

if $a \bullet b$, either $a = b$ or $a = \perp, a \neq b$ **(EQ 6)**

in the former,

$a \bullet b$ and $b \bullet c \Rightarrow$

$a \bullet a$ and $a \bullet c \Rightarrow a \bullet c$

in the latter,

$a \bullet b$ and $b \bullet c \Rightarrow$

$\perp \bullet b$ and $b \bullet c \Rightarrow$

Note that $b = c$, as $b \neq a$ from [equation: 6].

$\perp \bullet b$ and $b \bullet c \Rightarrow$

$\perp \bullet c$ and $c \bullet c \Rightarrow \perp \bullet c$,

but $a = \perp$, so $a \bullet c$

$\therefore ((a \bullet b)$ and $(b \bullet c)$ imply $(a \bullet c))$ is true. □

2. $\langle Z_\perp, \bullet \rangle$ is a BCPO.

As mentioned before, a tree is a BCPO. A 'lifted' anti-chain is a tree, thus, $\langle Z_\perp, \bullet \rangle$ is a BCPO. □

3. There must be an efficient means of computing subsumption and unification. We take $\langle Z_\perp, \bullet \rangle$ as an inheritance hierarchy such that $a \sqsubseteq b = a \bullet b$.

Subsumption

$\perp \sqsubseteq a, a \in Z$

$a \sqsubseteq a, a \in Z_{\perp}$

The subsumption algorithm is trivial and computed based on its simple symbol comparison.

Unification

$\perp \sqcup a = a, a \in Z_{\perp}$

$a \sqcup a = a, a \in Z$

The unification algorithm is trivial and computed based on its simple symbol comparison.

4. There must be a way to specify members of the set. Elements of $Z_{\perp}$ stand as names for themselves. Explicit expressions over Z with the usual operators of from $\{+, \times, -\}$, can also stand as names for elements of $Z_{\perp}$ ($/$ is not closed in Z so are not included here).

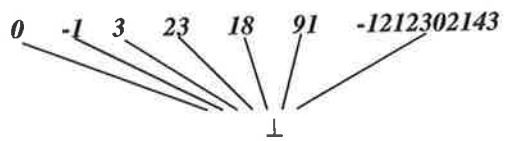


FIGURE 35. A Part of 'Lifted' Integers in an *InheritanceHierarchy*.

6.3.3 The lifted whole numbers

Given the ordered set $\overline{N_0} = \langle N_0, \bullet \rangle$ where $a \bullet b$ iff $a = b$ we form $\overline{N_{0_\perp}}$, which we call the *'lifted' whole numbers* by adding an element $\perp$, $N_{0_\perp} := N_0 \cup \{\perp\}$. We define an order relation '$\bullet$' ($\overline{N_0}$ is an antichain[1]) on $N_{0_\perp}$ as $\overline{N_{0_\perp}} = \langle N_{0_\perp}, \bullet \rangle$. The definition is

$$a \bullet b \text{ if } a \bullet b \text{ in } \overline{N_0} \qquad \textbf{(EQ 7)}$$

$$a \bullet b \text{ if } a = \perp \text{ and } (b \in N_{0_\perp}) \qquad \textbf{(EQ 8)}$$

Usually we refer to $\overline{N_{0_\perp}}$ simply as $N_{0_\perp}$. A partial inheritance hierarchy formed by $\langle N_{0_\perp}, \bullet \rangle$ is shown in Figure 36. With this definition, the four conditions described above apply as follows:

1. $\langle N_{0_\perp}, \bullet \rangle$ is a partial order.

Let $a, b, c \in N_{0_\perp}$,

reflexivity: $a \bullet a$ is true by definition (see equation: 7)

anti-symmetry: $a \bullet b$ and $b \bullet a$ imply $a = b$

by exhaustive enumeration,

if $a \bullet b$, either $a = b$ or $a = \perp, b \neq a$,

in the former,

$$a \bullet b \text{ and } b \bullet a \Rightarrow$$

$$a \bullet a \text{ and } a \bullet a \Rightarrow$$

$$a \bullet a \Rightarrow a = b$$

in the latter,

---

1. *"The ordered set P is an antichain if $x \leq y$ in P only if $x = y$."* [Davey and Priestley, 1994].

$a \bullet b$ but $\neg b \bullet a$

$\therefore (a \bullet b)$ and $(b \bullet a)$ and $b \neq a$ are false.

$\therefore (a \bullet b$ and $b \bullet a$ imply $a = b)$ is true. □

transitivity: $a \bullet b$ and $b \bullet c$ imply $a \bullet c$

by exhaustive enumeration.

if $a \bullet b$, either $a = b$ or $a = \perp, a \neq b$ **(EQ 9)**

in the former,

$a \bullet b$ and $b \bullet c \Rightarrow$

$a \bullet a$ and $a \bullet c \Rightarrow a \bullet c$

in the latter,

$a \bullet b$ and $b \bullet c \Rightarrow$

$\perp \bullet b$ and $b \bullet c \Rightarrow$

Note that $b = c$, as $b \neq a$ from [equation: 9].

$\perp \bullet b$ and $b \bullet c \Rightarrow$

$\perp \bullet c$ and $c \bullet c \Rightarrow \perp \bullet c$,

but $a = \perp$, so $a \bullet c$

$\therefore ((a \bullet b)$ and $(b \bullet c)$ imply $(a \bullet c))$ is true.□

2. $\langle N_{0_\perp}, \bullet \rangle$ is a BCPO.

As mentioned before, a tree is a BCPO. A 'lifted' anti-chain is a tree, thus, $\langle N_{0_\perp}, \bullet \rangle$ is a BCPO. □

3. There must be an efficient means of computing subsumption and unification. We take $\langle N_{0_\perp}, \bullet \rangle$ as an inheritance hierarchy such that $a \sqsubseteq b = a \bullet b$.

Subsumption

$\perp \sqsubseteq a, a \in N_0$

$a \sqsubseteq a, a \in N_{0_\perp}$

The subsumption algorithm is trivial and computed based on its simple symbol comparison.

Unification

$\perp \sqcup a = a, a \in N_{0_\perp}$

$a \sqcup a = a, a \in N_0$

The unification algorithm is trivial and computed based on its simple symbol comparison.

4. There must be a way to specify members of the set. Elements of $N_{0_\perp}$ stand as names for themselves. Explicit expressions over N_0 with the usual operators of from $\{+, \times\}$, can also stand as names for elements of $N_{0_\perp}$ ($-$ and $/$ are not closed in N_0 so are not included here).

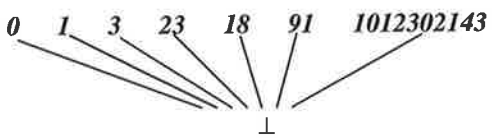


FIGURE 36. A Part of 'Lifted' Whole Numbers as an *InheritanceHierarchy*.

6.3.4 The ascending reals

Given the ordered set $\Re_{\leqslant} = \langle \Re, \leqslant \rangle$ where $\leqslant$ denotes the usual "*less than or equals*" relation, we form $\Re_{\perp_{\leqslant}}$ which we call the *ascending reals* by adding an element $\perp$ to $\Re$, $\Re_{\perp} := \Re \cup \{\perp\}$. We define an ordering relation '$\leqslant$' ($\Re_{\leqslant}$ is a chain[1]) on $\Re_{\perp}$ as $\Re_{\perp_{\leqslant}} = \langle \Re_{\perp}, \leqslant \rangle$. The definition is

$$a \leqslant b, \text{ if } a \leq b \in \Re_{\leqslant} \qquad \textbf{(EQ 10)}$$

$$a \leqslant b \text{ if } a = \perp \text{ and } b \in \Re_{\perp} \qquad \textbf{(EQ 11)}$$

We usually refer to $\Re_{\perp_{\leqslant}}$ as $\Re_{\uparrow}$; the arrow gives the sense of the ascending reals. A partial inheritance hierarchy formed by $\Re_{\uparrow}$ is shown in Figure 37. With this definition, the four conditions described above apply as follows:

1. $\Re_{\uparrow}$ is a partial order.

Let $a, b, c \in \Re_{\perp}$,

reflexivity: $a \leqslant a$ is true by definition. (see equation: 10)

anti-symmetry: $a \leqslant b$ and $b \leqslant a$ imply $a = b$

As a chain, if $a \neq b$, $b \leqslant a$ is false, so $a = b$ □

transitivity: $a \leqslant b$ and $b \leqslant c$ imply $a \leqslant c$

$(a \leqslant b)$ and $(b \leqslant c)$ form a sequence of $a \leqslant b \leqslant c$. (see equation: 10).

If $c \leqslant a$, thus the sequence is $a \leqslant b \leqslant c \leqslant a$ which is true

only when $a = b = c \Rightarrow a = c$

else $a \leqslant c$

$\therefore a \leqslant b$ and $b \leqslant c$ imply $a \leqslant c$ is true. □

1. "*P is a chain if, for all $x, y \in P$, either $x \leq y$ or $y \leq x$ (that is, if any two elements of P are comparable)*"[Davey and Priestley, 1994].

2. $\Re_{\uparrow}$ is a BCPO.

A chain is a tree. As mentioned in [Carpenter, 1992], a tree is a simple partial order that meets the BCPO condition. Thus, $\Re_{\uparrow}$ is a BCPO.□

3. There must be an efficient means of computing subsumption and unification. We take $\langle \Re_{\perp}, \leqslant \rangle$ as an inheritance hierarchy such that $a \sqsubseteq b = a \leqslant b$.

Subsumption

$\perp \sqsubseteq a, a \in \Re$

$a \sqsubseteq a, a \in \Re_{\perp}$

The subsumption algorithm is trivial and computed based on its symbol comparison.

Unification

$\perp \sqcup a = a, a \in \Re_{\perp}$

$a \sqcup a = a, a \in \Re$

The subsumption algorithm is trivial and computed based on its symbol comparison.

4. There must be a way to specify members of the set. Elements of $\Re_{\perp}$ stand as names for themselves. Explicit expressions over $\Re_{\perp}$ and the usual operators from $\{ +, -, \times, / \}$, can also stand as names for elements of $\Re_{\perp}$.

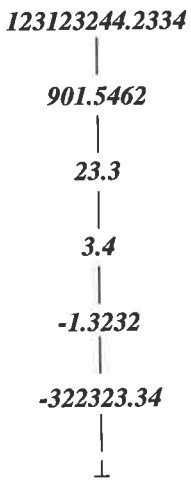


FIGURE 37. A Part of Ascending Reals as an *InheritanceHierarchy*.

6.3.5 The descending reals

Given the ordered set $\Re_{\geqslant} = \langle \Re, \geqslant \rangle$ where $\geqslant$ denotes the usual "*greater than or equals*" relation, we form $\Re_{\perp_{\geqslant}}$ which we call the *descending reals* by adding an element $\perp$ to $\Re$, $\Re_{\perp} := \Re \cup \{\perp\}$. We define an ordering relation '$\geqslant$' ($\Re_{\geqslant}$ is a chain[1]) on $\Re_{\perp}$ as $\Re_{\perp_{\geqslant}} = \langle \Re_{\perp}, \geqslant \rangle$. The definition is

$$a \geqslant b, \text{ if } a \geq b \in \Re_{\geqslant} \qquad \textbf{(EQ 12)}$$

$$a \geqslant b \text{ if } a = \perp \text{ and } (b \in \Re_{\perp}) \qquad \textbf{(EQ 13)}$$

We usually refer to $\Re_{\perp_{\geqslant}}$ as $\Re_{\downarrow}$; the arrow gives the sense of the descending reals. A partial inheritance hierarchy formed by $\Re_{\downarrow}$ is shown in Figure 38. With this definition, the four conditions described above apply as follows:

1. $\Re_{\downarrow}$ is a partial order.

Let $a, b, c \in \Re_{\perp}$,

reflexivity: $a \geqslant a$ is true by definition. (see equation: 12)

anti-symmetry: $a \geqslant b$ and $b \geqslant a$ imply $a = b$

As a chain, if $a \neq b, b \geqslant a$ is false, so $a = b$ □

transitivity: $a \geqslant b$ and $b \geqslant c$ imply $a \geqslant c$

$a \geqslant b$ and $b \geqslant c$ form a sequence of $a \geqslant b \geqslant c$. (see equation: 12).

If $c \geqslant a$, thus the sequence is $a \geqslant b \geqslant c \geqslant a$ which is true

only when $a = b = c \Rightarrow a = c$

else $a \geqslant c$

$\therefore (a \geqslant b$ and $b \geqslant c$ imply $a \geqslant c)$ is true. □

1. "*P is a chain if, for all $x, y \in P$, either $x \leq y$ or $y \leq x$ (that is, if any two elements of P are comparable)*"[Davey and Priestley, 1994].

2. $\mathfrak{R}_{\downarrow}$ is a BCPO.

A chain is a tree. As mentioned in [Carpenter, 1992], a tree is a simple partial order that meets the BCPO condition. Thus, $\mathfrak{R}_{\downarrow}$ is a BCPO.□

3. There must be an efficient means of computing subsumption and unification. We take $\langle \mathfrak{R}_{\perp}, \geqslant \rangle$ as an inheritance hierarchy such that $a \sqsubseteq b = a \geqslant b$.

Subsumption

$\perp \sqsubseteq a, a \in \mathfrak{R}$

$a \sqsubseteq a, a \in \mathfrak{R}_{\perp}$

The subsumption algorithm is trivial and computed based on its symbol comparison.

Unification

$\perp \sqcup a = a, a \in \mathfrak{R}_{\perp}$

$a \sqcup a = a, a \in \mathfrak{R}$

The subsumption algorithm is trivial and computed based on its symbol comparison.

4. There must be a way to specify members of the set. Elements of $\Re_{\perp}$ stand as names for themselves. Explicit expressions over $\Re_{\perp}$ and the usual operators from $\{ +, -, \times, / \}$, can also stand as names for elements of $\Re_{\perp}$.

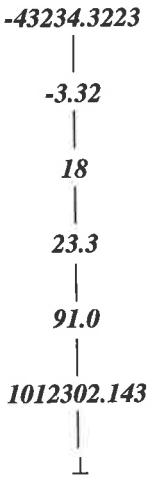


FIGURE 38. A Part of Descending Reals as an *InheritanceHierarchy*.

6.3.6 The ascending integers

Given the ordered set $Z_{\leqslant} = \langle Z, \leqslant \rangle$ where $\leqslant$ denotes the usual "*less than or equals*" relation, we form $Z_{\perp_{\leqslant}}$ which we call the *ascending integers* by adding an element $\perp$ to Z, $Z_{\perp} := Z \cup \{\perp\}$. We define an ordering relation '$\leqslant$' ($Z_{\leqslant}$ is a chain[1]) on $Z_{\perp}$ as $Z_{\perp_{\leqslant}} = \langle Z_{\perp}, \leqslant \rangle$. The definition is

$$a \leqslant b, \text{ if } a \leq b \in Z_{\leqslant} \qquad \textbf{(EQ 14)}$$

$$a \leqslant b \text{ if } a = \perp \text{ and } (b \in Z_{\perp}) \qquad \textbf{(EQ 15)}$$

We usually refer to $Z_{\perp_{\leqslant}}$ as $Z_{\uparrow}$; the arrow gives the sense of the ascending integers. A partial inheritance hierarchy formed by $Z_{\uparrow}$ is shown in Figure 39. With this definition, the four conditions described above apply as follows:

1. $Z_{\uparrow}$ is a partial order.

Let $a, b, c \in Z_{\perp}$,

reflexivity: $a \leqslant a$ is true by definition. (see equation: 14)

anti-symmetry: $a \leqslant b$ and $b \leqslant a$ imply $a = b$

As a chain, if $a \neq b, b \leqslant a$ is false, so $a = b$ □

transitivity: $a \leqslant b$ and $b \leqslant c$ imply $a \leqslant c$

$(a \leqslant b)$ and $(b \leqslant c)$ form a sequence of $a \leqslant b \leqslant c$. (see equation: 14).

If $c \leqslant a$, thus the sequence is $a \leqslant b \leqslant c \leqslant a$ which is true

only when $a = b = c \Rightarrow a = c$

else $a \leqslant c$

$\therefore a \leqslant b$ and $b \leqslant c$ imply $a \leqslant c$ is true. □

1. "*P is a chain if, for all $x, y \in P$, either $x \leq y$ or $y \leq x$ (that is, if any two elements of P are comparable)*"[Davey and Priestley, 1994].

2. $Z_{\uparrow}$ is a BCPO.

A chain is a tree. As mentioned in [Carpenter, 1992], a tree is a simple partial order that meets the BCPO condition. Thus, $Z_{\uparrow}$ is a BCPO.□

3. There must be an efficient means of computing subsumption and unification. We take $\langle Z_{\perp}, \leqslant \rangle$ as an inheritance hierarchy such that $a \sqsubseteq b = a \leqslant b$.

Subsumption

$\perp \sqsubseteq a, a \in Z$

$a \sqsubseteq a, a \in Z_{\perp}$

The subsumption algorithm is trivial and computed based on its symbol comparison.

Unification

$\perp \sqcup a = a, a \in Z_{\perp}$

$a \sqcup a = a, a \in Z$

The subsumption algorithm is trivial and computed based on its symbol comparison.

4. There must be a way to specify members of the set. Elements of $Z_{\perp}$ stand as names for themselves. Explicit expressions over $Z_{\perp}$ and the usual operators

from $\{+, \times, -\}$, can also stand as names for elements of $Z_\perp$ ($/$ is not closed in $Z_\perp$ so are not included here).

FIGURE 39. A Part of Ascending Integers as an *InheritanceHierarchy*.

6.3.7 The descending integers

Given the ordered set $Z_{\geqslant} = \langle Z, \geqslant \rangle$ where $\geqslant$ denotes the usual "*greater than or equals*" relation, we form $Z_{\perp_{\geqslant}}$ which we call the *descending integers* by adding an element $\perp$ to Z, $Z_{\perp} := Z \cup \{\perp\}$. We define an ordering relation '$\geqslant$' ($Z_{\geqslant}$ is a chain[1]) on $Z_{\perp}$ as $Z_{\perp_{\geqslant}} = \langle Z_{\perp}, \geqslant \rangle$. The definition is

$$a \geqslant b, \text{ if } a \geq b \in Z_{\geqslant} \qquad \textbf{(EQ 16)}$$

$$a \geqslant b \text{ if } a = \perp \text{ and } (b \in Z_{\perp}) \qquad \textbf{(EQ 17)}$$

We usually refer to $Z_{\perp_{\geqslant}}$ as $Z_{\downarrow}$; the arrow gives the sense of the descending integers. A partial inheritance hierarchy formed by $Z_{\downarrow}$ is shown in Figure 40. With this definition, the four conditions described above apply as follows:

1. $Z_{\downarrow}$ is a partial order.

Let $a, b, c \in Z_{\perp}$,

reflexivity: $a \geqslant a$ is true by definition. (see equation: 16)

anti-symmetry: $a \geqslant b$ and $b \geqslant a$ imply $a = b$

As a chain, if $a \neq b$, $b \geqslant a$ is false, so $a = b$ □

transitivity: $a \geqslant b$ and $b \geqslant c$ imply $a \geqslant c$

$(a \geqslant b$ and $b \geqslant c)$ form a sequence of $a \geqslant b \geqslant c$. (see equation: 16).

If $c \geqslant a$, thus the sequence is $a \geqslant b \geqslant c \geqslant a$ which is true

only when $a = b = c \Rightarrow a = c$

else $a \geqslant c$

$\therefore (a \geqslant b$ and $b \geqslant c$ imply $a \geqslant c)$ is true. □

1. "*P is a chain if, for all $x, y \in P$, either $x \leq y$ or $y \leq x$ (that is, if any two elements of P are comparable)*"[Davey and Priestley, 1994].

2. $Z_{\downarrow}$ is a BCPO.

A chain is a tree. As mentioned in [Carpenter, 1992], a tree is a simple partial order that meets the BCPO condition. Thus, $Z_{\downarrow}$ is a BCPO.□

3. There must be an efficient means of computing subsumption and unification. We take $\langle Z_{\perp}, \geqslant \rangle$ as an inheritance hierarchy such that $a \sqsubseteq b = a \geqslant b$.

Subsumption

$\perp \sqsubseteq a, a \in Z$

$a \sqsubseteq a, a \in Z_{\perp}$

The subsumption algorithm is trivial and computed based on its symbol comparison.

Unification

$\perp \sqcup a = a, a \in Z_{\perp}$

$a \sqcup a = a, a \in Z$

The subsumption algorithm is trivial and computed based on its symbol comparison.

4. There must be a way to specify members of the set. Elements of $Z_{\perp}$ stand as names for themselves. Explicit expressions over $Z_{\perp}$ and the usual operators from

{+, × , – }, can also stand as names for elements of $Z_\perp$ (/ is not closed in $Z_\perp$ so are not included here).

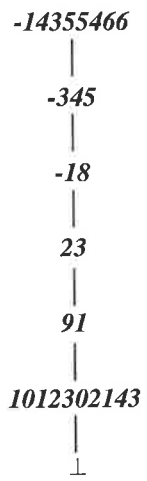


FIGURE 40. A Part of Descending Integers as an *InheritanceHierarchy*.

6.3.8 The ascending whole numbers

Given the ordered set $N_{0_{\leqslant}} = \langle N_0, \leqslant \rangle$ where $\leqslant$ denotes the usual "*less than or equals*" relation, we form $N_{0_{\leqslant}}$ which we call the *ascending whole numbers*. We define an ordering relation '$\leqslant$' ($N_{0_{\leqslant}}$ is a chain[1]) on N_0 as $N_{0_{\leqslant}} = \langle N_0, \leqslant \rangle$. The definition is

$$a \leqslant b, \text{ if } a \leq b \in N_{0_{\leqslant}} \qquad \textbf{(EQ 18)}$$

We usually refer to $N_{0_{\leqslant}}$ as $N_{0_{\uparrow}}$; the arrow gives the sense of the ascending whole numbers. A partial inheritance hierarchy formed by $N_{0_{\uparrow}}$ is shown in Figure 41. With this definition, the four conditions described above apply as follows:

1. $N_{0_{\uparrow}}$ is a partial order.

Let $a, b, c \in N_0$,

reflexivity: $a \leqslant a$ is true by definition. (see equation: 18)

anti-symmetry: $a \leqslant b$ and $b \leqslant a$ imply $a = b$

As a chain, if $a \neq b, b \leqslant a$ is false, so $a = b$ □

transitivity: $a \leqslant b$ and $b \leqslant c$ imply $a \leqslant c$

$(a \leqslant b)$ and $(b \leqslant c)$ form a sequence of $a \leqslant b \leqslant c$. (see equation: 18).

If $c \leqslant a$, thus the sequence is $a \leqslant b \leqslant c \leqslant a$ which is true

only when $a = b = c \Rightarrow a = c$

else $a \leqslant c$

$\therefore a \leqslant b$ and $b \leqslant c$ imply $a \leqslant c$. □

2. $N_{0_{\uparrow}}$ is a BCPO.

---

1. "*P is a chain if, for all $x, y \in P$, either $x \leq y$ or $y \leq x$ (that is, if any two elements of P are comparable)*"[Davey and Priestley, 1994].

A chain is a tree. As mentioned in [Carpenter, 1992], a tree is a simple partial order that meets the BCPO condition. In addition, $N_{0_\uparrow}$ already has an element (0) which confronts the definition of a bottom, therefore, $N_{0_\uparrow}$ does not need to be lifted. Thus, $N_{0_\uparrow}$ is a BCPO with a bottom (0). ▯

3. There must be an efficient means of computing subsumption and unification. We take $\langle N_0, \leqslant \rangle$ as an inheritance hierarchy such that $a \sqsubseteq b = a \leqslant b$.

Subsumption

$0 \sqsubseteq a, a \in N_0$

The subsumption algorithm is trivial and computed based on its symbol comparison.

Unification

$0 \sqcup a = a, a \in N_0$

$a \sqcup a = a, a \in N_0$

The subsumption algorithm is trivial and computed based on its symbol comparison.

4. There must be a way to specify members of the set. Elements of N_0 stand as names for themselves. Explicit expressions over N_0 and the usual operators

from $\{+, \times\}$, can also stand as names for elements of N_0 ($-$ and $/$ are not closed in N_0 so are not included here).

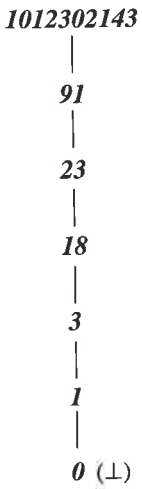


FIGURE 41. A Part of Ascending Whole Numbers as an *InheritanceHierarchy*.

6.3.9 The descending whole numbers

Given the ordered set $N_{0_{\leqslant}} = \langle N_0, \geqslant \rangle$ where $\geqslant$ denotes the usual "*greater than or equals*" relation, we form $N_{0_{\perp\geqslant}}$ which we call the *descending whole numbers* by adding an element $\perp$ to N_0, $N_{0_\perp} := N_0 \cup \{\perp\}$. We define an ordering relation '$\geqslant$' ($N_{0_\geqslant}$ is a chain[1]) on $N_{0_\perp}$ as $N_{0_{\perp\geqslant}} = \langle N_{0_\perp}, \geqslant \rangle$. The definition is

$$a \geqslant b, \text{ if } a \geq b \in N_{0_\geqslant} \qquad \textbf{(EQ 19)}$$

$$a \geqslant b \text{ if } a = \perp \text{ and } (b \in N_{0_\perp}) \qquad \textbf{(EQ 20)}$$

We usually refer to $N_{0_{\perp\geqslant}}$ as $N_{0_\downarrow}$; the arrow gives the sense of the descending whole numbers. A partial inheritance hierarchy formed by $N_{0_\downarrow}$ is shown in Figure 42. With this definition, the four conditions described above apply as follows:

1. $N_{0_\downarrow}$ is a partial order.

Let $a, b, c \in N_{0_\perp}$,

reflexivity:

$a \geqslant a$ is true by definition. (see equation: 19)□

anti-symmetry: $a \geqslant b$ and $b \geqslant a$ imply $a = b$

As a chain, if $a \neq b$, $b \geqslant a$ is false, so $a = b$.□

transitivity: $a \geqslant b$ and $b \geqslant c$ imply $a \geqslant c$

$a \geqslant b$ and $b \geqslant c$ form a sequence of $a \geqslant b \geqslant c$. (see equation: 19).

If $c \geqslant a$, thus the sequence is $a \geqslant b \geqslant c \geqslant a$ which is true

only when $a = b = c \Rightarrow a = c$

else $a \geqslant c$

---

1. "*P is a chain if, for all $x, y \in P$, either $x \leq y$ or $y \leq x$ (that is, if any two elements of P are comparable)*"[Davey and Priestley, 1994].

$\therefore (a \geqslant b$ and $b \geqslant c$ imply $a \geqslant c)$ is true. □

2. $N_{0_\downarrow}$ is a BCPO.

   A chain is a tree. As mentioned in [Carpenter, 1992], a tree is a simple partial order that meets the BCPO condition. Thus, $N_{0_\downarrow}$ is a BCPO.□

3. There must be an efficient means of computing subsumption and unification. We take $\langle N_{0_\perp}, \geqslant \rangle$ as an inheritance hierarchy such that $a \sqsubseteq b = a \geqslant b$.

   Subsumption

   $\perp \sqsubseteq a, a \in N_0$

   $a \sqsubseteq a, a \in N_{0_\perp}$

   The subsumption algorithm is trivial and computed based on its symbol comparison.

   Unification

   $\perp \sqcup a = a, a \in N_{0_\perp}$

   $a \sqcup a = a, a \in N_0$

   The subsumption algorithm is trivial and computed based on its symbol comparison.

4. There must be a way to specify members of the set. Elements of $N_{0_\perp}$ stand as names for themselves. Explicit expressions over $N_{0_\perp}$ and the usual operators

from $\{+, \times\}$, can also stand as names for elements of $N_{0_\perp}$ ($-$ and $/$ are not closed in $N_{0_\perp}$ so are not included here).

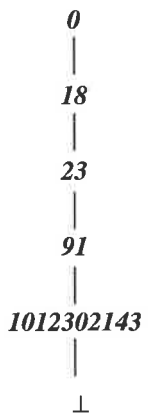


FIGURE 42. A Part of Descending Whole Numbers as an *InheritanceHierarchy*.

6.3.10 The real intersection intervals

The ordered set $\Re_{\uparrow\downarrow} = \langle [\Re_{\uparrow}, \Re_{\downarrow}], \supseteq \rangle$, where $\supseteq$ denotes the "interval inclusion" relation we call the "*real intersection interval*". $a \supseteq b$ is read as "a includes b". First we define $val(n)$ as a polymorphic function giving the real corresponding to an element in $\Re_{\uparrow}$ or $\Re_{\downarrow}$.[1]

In $\Re_{\uparrow}$

$$val(\perp) < val(a), a \in \Re_{\uparrow} - \{\perp\}$$

In $\Re_{\downarrow}$

$$val(\perp) > val(a), a \in \Re_{\downarrow} - \{\perp\}, \text{ and}$$

$$val(\perp_{\Re_{\uparrow}}) < val(\perp_{\Re_{\downarrow}}).$$

We write an interval as $a = [a_l, a_u]$ and use a_l and a_u to denote the lower and upper bound of the interval respectively, such that $val(a_l)$ is less than or equal to $val(a_u)$. A partial inheritance hierarchy formed by $\Re_{\uparrow\downarrow}$ is shown in Figure 43. The definition of $\Re_{\uparrow\downarrow}$ is

Let $a \in \Re_{\uparrow}, b \in \Re_{\downarrow}, [a, b] \in \Re_{\uparrow\downarrow}$, if $val(a) \leq val(b)$. **(EQ 21)**

$a \supseteq b$ if $a_l \leqslant b_l$ and $a_u \geqslant b_u$ **(EQ 22)**

1. $\Re_{\uparrow\downarrow}$ is a partial order.

Let $a, b, c \in \Re_{\uparrow\downarrow}$,

reflexivity: $a \supseteq a$

As a special case shown in equation: 22, $a \supseteq a$ is true by definition. □

anti-symmetry: $a \supseteq b$ and $b \supseteq a$ imply $a = b$

1. $\Re_{\uparrow}$ and $\Re_{\downarrow}$ are described in Chapter 6.3.4 and Chapter 6.3.5 respectively.

The definition of $a \sqsupseteq b$ relies directly on the relations "*less than*" and "*greater than*" over the sets $\Re_{\uparrow}$ and $\Re_{\downarrow}$ respectively. Both $\Re_{\uparrow}$ and $\Re_{\downarrow}$ are partial orders and anti-symmetric. Therefore, $\Re_{\uparrow\downarrow}$ is anti-symmetric as its definition (see equation: 22) is simply the conjunction of the above two conditions. □

transitivity: $a \sqsupseteq b$ and $b \sqsupseteq c$ imply $a \sqsupseteq c$

By the argument used for anti-symmetry, $\Re_{\uparrow\downarrow}$ also satisfies the transitivity condition. □

2. $\Re_{\uparrow\downarrow}$ is a BCPO.

Bottom.

An bottom of $\Re_{\uparrow\downarrow}$ exists as $[\bot_{\Re_{\uparrow}}, \bot_{\Re_{\downarrow}}]$, $[\bot_{\Re_{\uparrow}}, \bot_{\Re_{\downarrow}}] \sqsupseteq a, a \in \Re_{\uparrow\downarrow}$.

A least upper bound.

Following the definition of upper bound[1] from [Davey and Priestley, 1994],

Let $Q \in \wp(\Re_{\uparrow\downarrow})$, the powerset of $\Re_{\uparrow\downarrow}$,

$Q^u = \{x \in \Re_{\uparrow\downarrow} | \forall(q \in Q), q \sqsupseteq x\}$, such that $q_l \leqslant x_l$ and $q_u \geqslant x_u$.

Consider a least upper bound $r \in Q^u$, and $r \sqsupseteq y, \forall(y \in Q^u)$. $\Rightarrow$

$r_l \leqslant y_l$, $r_u \geqslant y_u$ for all y in Q^u, and with upper bound condition $\Rightarrow$

$x_l \leqslant r_l \leqslant y_l$, $x_u \geqslant r_u \geqslant y_u$ for all y in Q^u and all x in Q. $\Rightarrow$

r_l and r_l are the least upper bounds of the sets containing all lower and upper bounds of intervals in Q^u respectively. Both $\Re_{\uparrow}$ and $\Re_{\downarrow}$ are BCPOs. Therefore, $\Re_{\uparrow\downarrow}$ has a least upper bound r as its definition is simply the conjunction of the above two least upper bounds.

1. "Let P be an ordered set and let $S \subseteq P$. An element $x \in P$ is an *upper bound* of S if $s \leqslant x$ for all $s \in S$. ... The set of all upper bounds of S is denoted by S^u"[Davey and Priestley, 1994]

Together with the interval restriction (see equation: 21), such interval r is the least upper bound of Q. The interval r either exists or is undefined if $val(r_l) > val(r_u)$.

Therefore, $\Re_{\uparrow\downarrow}$ is a BCPO. $\square$

3. There must be an efficient means of computing subsumption and unification. We take $\Re_{\uparrow\downarrow}$ as an inheritance hierarchy such that $a \sqsubseteq b = a \supseteq b$.

Subsumption

The subsumption algorithm is trivial and computed based on its interval inclusion.

Unification

The subsumption algorithm is trivial and computed based on its interval comparison.

4. There must be a way to specify members of the set $\Re_{\uparrow\downarrow}$. Elements of $\Re_{\uparrow\downarrow}$ can be expressed in terms of their lower bounds and upper bounds that can stand as names for themselves. Explicit expressions over $\Re_{\uparrow\downarrow}$ and the set operators from $\{ \cup, \cap \}$ can be used to describe the members of the set.

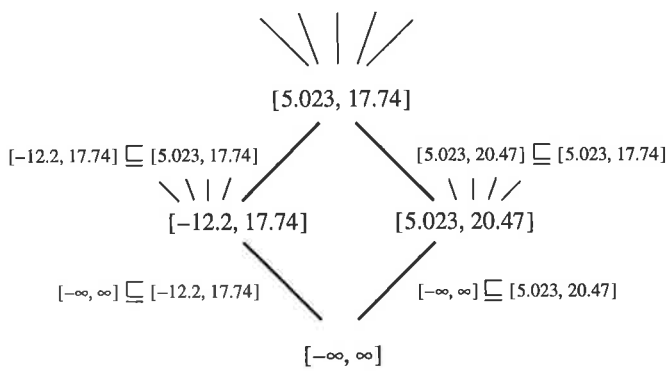


FIGURE 43. A Part of Real Intersection Intervals as an *InheritanceHierarchy*.

6.3.11 The integer intersection intervals

The ordered set $Z_{\uparrow\downarrow} = \langle [Z_{\uparrow}, Z_{\downarrow}], \supseteq \rangle$, where $\supseteq$ denotes the "interval inclusion" relation we call the "*integer intersection interval*". First we define $val(n)$ as a polymorphic function giving the integer corresponding to an element in $Z_{\uparrow}$ or $Z_{\downarrow}$.[1]

In $Z_{\uparrow}$

$$val(\perp) < val(a), a \in Z_{\uparrow} - \{\perp\}$$

In $Z_{\downarrow}$

$$val(\perp) > val(a), a \in Z_{\downarrow} - \{\perp\}, \text{ and}$$

$$val(\perp_{Z_{\uparrow}}) < val(\perp_{Z_{\downarrow}}) .$$

We write an interval as $a = [a_l, a_u]$ and use a_l and a_u to denote the lower and upper bound of the interval respectively, such that $val(a_l)$ is less than or equal to $val(a_u)$. A partial inheritance hierarchy formed by $Z_{\uparrow\downarrow}$ is shown in Figure 44. The definition of $Z_{\uparrow\downarrow}$ is

Let $a \in Z_{\uparrow}, b \in Z_{\downarrow}, [a, b] \in Z_{\uparrow\downarrow}$, if $val(a) \leq val(b)$. **(EQ 23)**

$a \supseteq b$ if $a_l \leqslant b_l$ and $a_u \geqslant b_u$ **(EQ 24)**

1. $Z_{\uparrow\downarrow}$ is a partial order.

   Let $a, b, c \in Z_{\uparrow\downarrow}$,

   reflexivity: $a \supseteq a$

   As a special case shown in equation: 24, $a \supseteq a$ is true by definition. □

   anti-symmetry: $a \supseteq b$ and $b \supseteq a$ imply $a = b$

---

1. $Z_{\uparrow}$ and $Z_{\downarrow}$ are described in Chapter 6.3.6 and Chapter 6.3.7 respectively.

The definition of $a \sqsupseteq b$ relies directly on the relations "*less than*" and "*greater than*" over the sets $Z_{\uparrow}$ and $Z_{\downarrow}$ respectively. Both $Z_{\uparrow}$ and $Z_{\downarrow}$ are partial orders and anti-symmetric. Therefore, $Z_{\uparrow\downarrow}$ is anti-symmetric as its definition (see equation: 24) is simply the conjunction of the above two conditions. □

transitivity: $a \sqsupseteq b$ and $b \sqsupseteq c$ imply $a \sqsupseteq c$

By the argument used for anti-symmetry, $Z_{\uparrow\downarrow}$ also satisfies the transitivity condition. □

2. $Z_{\uparrow\downarrow}$ is a BCPO.

Bottom.

A bottom of $Z_{\uparrow\downarrow}$ exists as $[\bot_{Z_{\uparrow}}, \bot_{Z_{\downarrow}}]$ and $[\bot_{Z_{\uparrow}}, \bot_{Z_{\downarrow}}] \sqsupseteq a, a \in Z_{\uparrow\downarrow}$.

A least upper bound.

Following the definition of upper bound[1] from [Davey and Priestley, 1994],

Let $Q \in \wp(Z_{\uparrow\downarrow})$,

$Q^u = \{x \in Z_{\uparrow\downarrow} | \forall(q \in Q), q \sqsupseteq x\}$, such that $q_l \leqslant x_l$ and $q_u \geqslant x_u$.

Consider a least upper bound $r \in Q^u$, and $r \sqsupseteq y, \forall(y \in Q^u)$. $\Rightarrow$

$r_l \leqslant y_l$, $r_u \geqslant y_u$ for all y in Q^u , and with upper bound condition $\Rightarrow$

$x_l \leqslant r_l \leqslant y_l$, $x_u \geqslant r_u \geqslant y_u$ for all y in Q^u and all x in Q . $\Rightarrow$

r_l and r_l are the least upper bounds of the sets containing all lower and upper bounds of intervals in Q^u respectively. Both $Z_{\uparrow}$ and $Z_{\downarrow}$ are BCPOs. Therefore, $Z_{\uparrow\downarrow}$ has a least upper bound r as its definition is simply the conjunction of the above two least upper bounds.

1. "Let P be an ordered set and let $S \subseteq P$. An element $x \in P$ is an *upper bound* of S if $s \leqslant x$ for all $s \in S$ . ... The set of all upper bounds of S is denoted by S^u"[Davey and Priestley, 1994]

Together with the interval restriction (see equation: 23), such interval r is the least upper bound of Q. The interval r either exists or undefined if $val(r_l) > val(r_u)$.

Therefore, $Z_{\uparrow\downarrow}$ is a BCPO. $\square$

3. There must be an efficient means of computing subsumption and unification. We take $Z_{\uparrow\downarrow}$ as an inheritance hierarchy such that $a \sqsubseteq b = a \supseteq b$.

Subsumption

The subsumption algorithm is trivial and computed based on its interval inclusion.

Unification

The subsumption algorithm is trivial and computed based on its interval comparison.

4. There must be a way to specify members of the set $Z_{\uparrow\downarrow}$. Elements of $Z_{\uparrow\downarrow}$ can be expressed in terms of their lower bounds and upper bounds that can stand as names for themselves. Explicit expressions over $Z_{\uparrow\downarrow}$ and the set operators from $\{ \cup, \cap \}$ can be used to describe the members of the set.

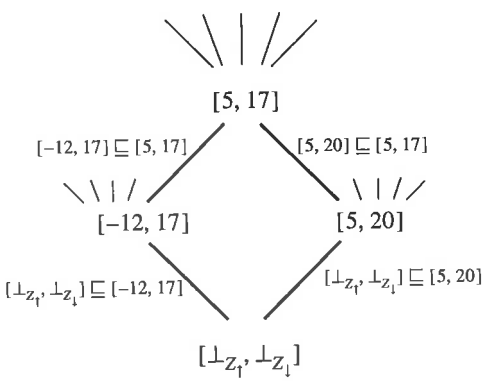


FIGURE 44. A Part of Integer Intersection Intervals as an *InheritanceHierarchy*.

6.3.12 The whole number intersection intervals

The ordered set $N_{0_{\uparrow\downarrow}} = \langle [N_{0_\uparrow}, N_{0_\downarrow}], \supseteq \rangle$, where $\supseteq$ denotes the "interval inclusion" relation we call the "*whole number intersection interval*". First we define $val(n)$ as a polymorphic function giving the whole number corresponding to an element in $N_{0_\uparrow}$ or $N_{0_\downarrow}$.[1]

In $N_{0_\uparrow}$

$$val(0) < val(a), a \in N_{0_\uparrow} - \{0\}$$

In $N_{0_\downarrow}$

$$val(\perp) > val(a), a \in N_{0_\downarrow} - \{\perp\}, \text{ and}$$

$$val(\perp_{N_{0_\uparrow}}) < val(\perp_{N_{0_\downarrow}}).$$

We write an interval as $a = [a_l, a_u]$ and use a_l and a_u to denote the lower and upper bound of the interval respectively, such that $val(a_l)$ is less than or equal to $val(a_u)$. A partial inheritance hierarchy formed by $N_{0_{\uparrow\downarrow}}$ is shown in Figure 45. The definition of $N_{0_{\uparrow\downarrow}}$ is

Let $a \in N_{0_\uparrow}, b \in N_{0_\downarrow}, [a, b] \in N_{0_{\uparrow\downarrow}}$, if $val(a) \leq val(b)$. **(EQ 25)**

$a \supseteq b$ if $a_l \leqslant b_l$ and $a_u \geqslant b_u$ **(EQ 26)**

1. $N_{0_{\uparrow\downarrow}}$ is a partial order.

Let $a, b, c \in N_{0_{\uparrow\downarrow}}$,

reflexivity: $a \supseteq a$

As a special case shown in equation: 26, $a \supseteq a$ is true by definition. □

1. $N_{0_\uparrow}$ and $N_{0_\downarrow}$ are described in Chapter 6.3.8 and Chapter 6.3.9 respectively.

anti-symmetry: $a \sqsupseteq b$ and $b \sqsupseteq a$ imply $a = b$

The definition of $a \sqsupseteq b$ relies directly on the relations "*less than*" and "*greater than*" over the sets $N_{0_\uparrow}$ and $N_{0_\downarrow}$ respectively. Both $N_{0_\uparrow}$ and $N_{0_\downarrow}$ are partial orders and anti-symmetric. Therefore, $N_{0_{\uparrow\downarrow}}$ is anti-symmetric as its definition (see equation: 26) is simply the conjunction of the above two conditions. □

transitivity: $a \sqsupseteq b$ and $b \sqsupseteq c$ imply $a \sqsupseteq c$

By the argument used for anti-symmetry, $N_{0_{\uparrow\downarrow}}$ also satisfies the transitivity condition. □

2. $N_{0_{\uparrow\downarrow}}$ is a BCPO.

Bottom.

A bottom of $N_{0_{\uparrow\downarrow}}$ exists as $[0, \perp_{N_{0_\downarrow}}]$, $[0, \perp_{N_{0_\downarrow}}] \sqsupseteq a, a \in N_{0_{\uparrow\downarrow}}$.

A least upper bound.

Following the definition of upper bound[1] from [Davey and Priestley, 1994],

Let $Q \in \wp(N_{0_{\uparrow\downarrow}})$,

$Q^u = \{x \in N_{0_{\uparrow\downarrow}} | \forall(q \in Q), q \sqsupseteq x\}$, such that $q_l \leqslant x_l$ and $q_u \geqslant x_u$.

Consider a least upper bound $r \in Q^u$, and $r \sqsupseteq y, \forall(y \in Q^u)$. $\Rightarrow$

$r_l \leqslant y_l$, $r_u \geqslant y_u$ for all y in Q^u, and with upper bound condition $\Rightarrow$

$x_l \leqslant r_l \leqslant y_l$, $x_u \geqslant r_u \geqslant y_u$ for all y in Q^u and all x in Q. $\Rightarrow$

r_l and r_l are the least upper bounds of the sets containing all lower and upper bounds of intervals in Q^u respectively. Both $N_{0_\uparrow}$ and $N_{0_\downarrow}$ are

1. "Let P be an ordered set and let $S \subseteq P$. An element $x \in P$ is an *upper bound* of S if $s \leqslant x$ for all $s \in S$. ... The set of all upper bounds of S is denoted by S^u"[Davey and Priestley, 1994]

BCPOs. Therefore, $N_{0_{\uparrow\downarrow}}$ has a least upper bound r as its definition is simply the conjunction of the above two least upper bounds.

Together with the interval restriction (see equation: 25), such interval r is the least upper bound of Q. The interval r either exists or undefined if $val(r_l) > val(r_u)$.

Therefore, $N_{0_{\uparrow\downarrow}}$ is a BCPO. □

3. There must be an efficient means of computing subsumption and unification. We take $N_{0_{\uparrow\downarrow}}$ as an inheritance hierarchy such that $a \sqsubseteq b = a \sqsupseteq b$.

Subsumption

The subsumption algorithm is trivial and computed based on its interval inclusion.

Unification

The subsumption algorithm is trivial and computed based on its interval comparison.

4. There must be a way to specify members of the set $N_{0_{\uparrow\downarrow}}$. Elements of $N_{0_{\uparrow\downarrow}}$ can be expressed in terms of their lower bounds and upper bounds that can stand as

names for themselves. Explicit expressions over $N_{0_{\uparrow\downarrow}}$ and the set operators from $\{ \cup, \cap \}$ can also be used to describe the members of the set.

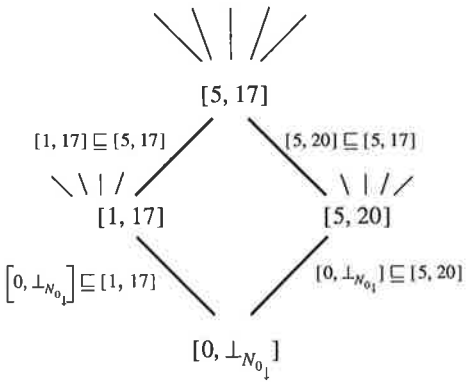


FIGURE 45. A Part of Whole Number Intersection Intervals as an *InheritanceHierarchy*.

6.4 Order Types carrying geometric information

By gaining experience from previous section, we now move toward a set of Order Types carrying geometric information. We called such Order Types "geometry types". The goal of geometry types is to represent the physical design information (geometric objects) within Geometric Typed Feature Structures. As stated before, geometry types require an efficient geometric representation. An approach called "*non-manifold cell complex*" is adopted. A representation based on non-manifold cell complex is described in Chapter 6.4.1. A set of relations over this geometric representation is discussed in Chapter 6.4.2, followed by four Order Type examples.

6.4.1 Representation

An efficient and powerful spatial representation is critical for representing the geometric information carried by design space explorers. Furthermore, geometric objects should remain invariant within a single session in order to provide the information consistency needed by the representation of its symbol level.

In general, boundary representation (B-rep), in which geometric objects are represented via structures representing their boundaries, is a common representation schemata used in solid modelling. With simplifying assumption that geometric

objects have only two-manifold boundaries[Mantyla, 1988, Hoffmann, 1989], early B-rep scheme defines a topological space in which every point of the boundaries (of geometric objects) has a neighbourhood topologically equivalent to an open disk. As stated in [Chang and Woodbury, 1996], several drawbacks of two-manifold scheme indicate that an extended scheme is required for the design domain—a non-two manifold scheme (usually referred to as *non-manifold*).

In the non-manifold approach the modelling domain comprises objects of mixed dimensionality occurring in the same geometric model as well as objects whose boundaries are non-manifold. Various representation schemes over this domain have been developed, notably Weiler's[Weiler, 1986] edge-based data structure and Gursoz et al.'s[Gursoz et al., 1990] vertex-based data structure.

cell-complex non-manifold

An approach called *cell-complex non-manifold geometric modelling* has been used in several implementations, though it does not seem to have received formal treatment [Gursoz, 1991, Masuda, 1993, Fenves et al., 1994, XOX, 1995]. In essence, cell-complex modelling maintains a collectively exhaustive and mutually exclusive subdivision of space in a geometric model. Models contain distinguished dimension-independent cells, which are represented as collections of parts of the subdivision. Models are combined with a single operation *merge* which subdivides two models each by the other yielding a single model which contains all the distinguished cells from both models, represented as collections of parts from the subdivision of the new model. The cell-complex approach is typically implemented over a non-manifold geometric domain and comprises the main application programming interface to the modelling environment. Based on the representation scheme described in [Chang and Woodbury, 1996], we proceed to an elaborated version of representation using the cell complex approach in the next subsections.

A cell complex

We use the characterisation of cell complex with focus on cell-decomposition over merge operator. A model (a cell complex) will remain invariant in terms of point-set

topology even though that model might contain more cells after the cell-decomposition process of merge. An example of a cell complex is shown in Figure 46.

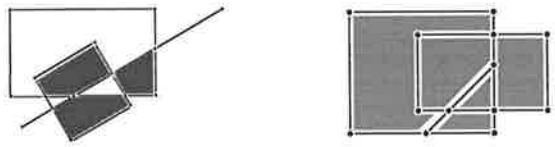


FIGURE 46. Examples of Cell Complex: a Cell Complex Comprises Multi-Dimensional Cells.

Regions and Cells

A labelling scheme is used on top of a non-manifold cell-complex to represent individual non-manifold object occurring in the model space described in previous section. In essence, a representation of a non-manifold object called region comprises a set of cells taken from the carrying complex. A region comprises a mutually exclusive and collectively exhaustive subdivision of space into zero-dimension (0D), one-dimension (1D), two-dimension (2D) and three-dimension (3D) topological elements called cells (see Figure 47).

We treat the notion of a cell informally here, as does the rest of the literature of which we are aware[1] of. Informally, a cell is of homogeneous dimensionality (all points have a neighbourhood topologically equivalent to an open ball in the dimen-

1. For example, in (Masuda, 1993), an *n* cell is homeomorphic to an *n*D open sphere. However, a 3D cell with an internal hole is not homeomorphic to a 3D open sphere.

sionality of the cell), has a finitely describable boundary in all relevant aspects, and contains only its interior points with respect to its dimensionality.

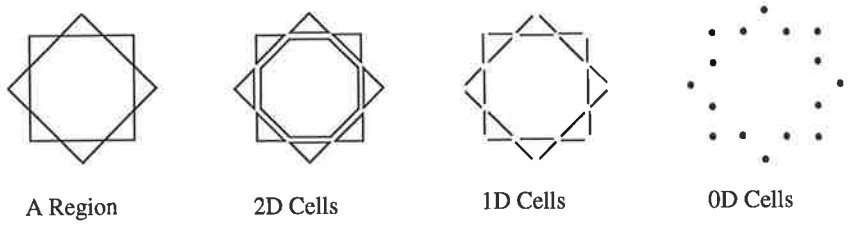


FIGURE 47. The Boundary of a Region is Comprised by Cells from Different Dimensions, *e.g.* the Vertices of the Region Above are 0D Cells.

The boundary of an nD cell comprises the (n-k)D cells $1 \leq k \leq n$ that bound it. For example, in Figure 48, the boundary of a 2D cell consists of 0D cells (as vertices) and 1D cells (as edges).

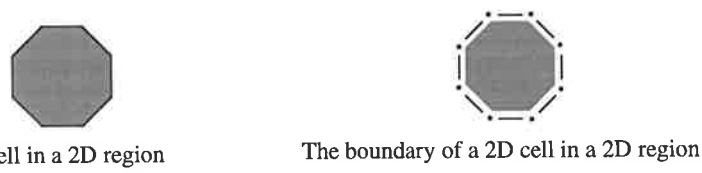


FIGURE 48. The Boundary of an nD Cell Comprises the (n-k)D Cells $1 \leq k \leq n$ that Bound it.

A region contains no structure other than a set of cells (see Figure 49). As indicated in Chapter 5.4, geometry information needs no further structure. Consequently, a region used to describe the geometric information of domain interest needs no further structure other than the cells. Furthermore, *regions* denote point-sets within the

carrier point-set of the default cell complex. The function $pset(R)$ gives the point-set denoted by region R.

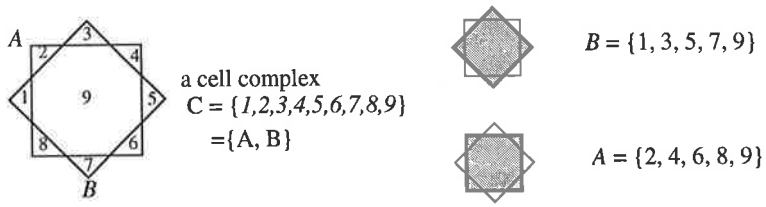


FIGURE 49. Two Regions: A and B are Part of a Cell Complex C. Each Region Comprises a Set of Cells and two Regions might Share a Set of Cells.

The boundary of a region is represented as a region containing cells of lower dimensionality than the given region. A *closed region* implicitly contains the boundaries of its cells. For example, in Figure 50, the boundary of a 2D *closed* region consists of 2D cells (the faces), 1D cells (the edges), and 0D cells (vertices).

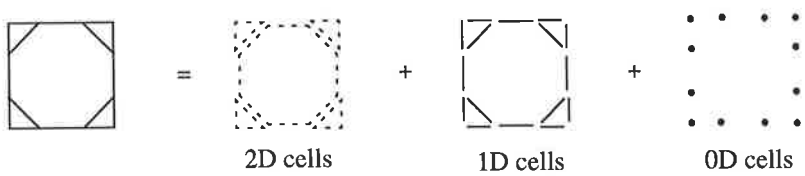


FIGURE 50. The Boundary of a Region is Represented as a Region Containing Cells of Lower Dimensionality Than the Given Region.

merge operator

Without loss of generality, in what follows we presume that regions are closed, that is, their region sets implicitly include the cell boundaries of constituent regions.

Regions are combined with the single operation *merge*. Merge takes one region and merge that region into a cell complex which containing subdivided cells from that merged region, such that each cell of either region has been subdivided by its intersection with the cells of the other region and common cells eliminated. For example, Figure 51 demonstrates a merge operation which merges two regions: A and B

into an existing cell complex that contains only region B originally. The cells of both regions have been changed, but the regions remain the same.

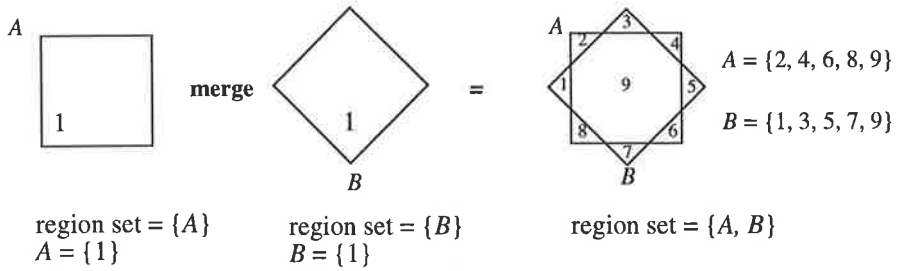


FIGURE 51. The Cells of a Region do not Maintain their Identities Through the Merge Operation.

The point-set of a region (given by $pset(R)$) is invariant across a merge. However, the cells composing a region may change as Figure 52 below. When a region C which comprises partial cells of region A merges another region D which comprises partial cells of region B, the cells of both regions change but the point-set of both regions remain invariant.

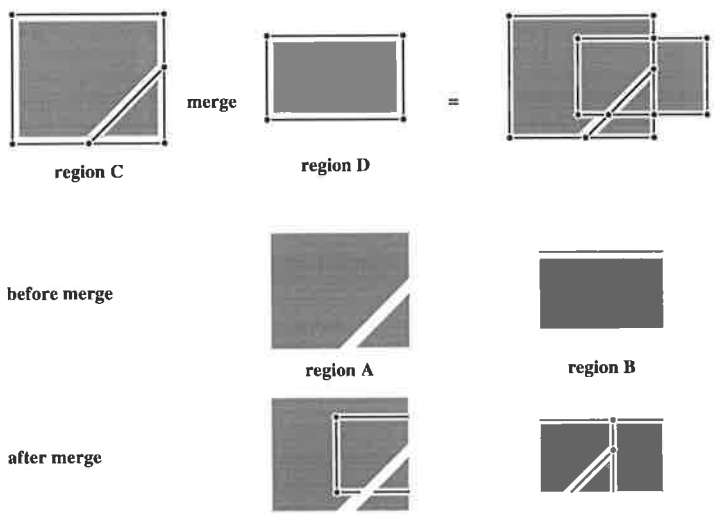
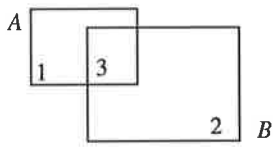


FIGURE 52. The Point-Set of a Region Remain Invariant Across a Merge Operation, but the Cells it Comprises might Change, *e.g.* the Cells that Region A Comprises Change After Merge Operation, but the Boundary of Region A still Remains Invariant.

Boolean operators over Regions

Boolean operations are defined over regions and amount to set operations over sets. Algorithms for determining the regularised union, intersection or set difference of two regions can be viewed conceptually as set operations over the cells they comprise. For example, a union operation of two regions is a set union operation of two set of cells contained by these two regions. The regularised Set Boolean operations are shown in Figure 53.



region $A = \{1, 3\}$

region $B = \{2, 3\}$

Boolean operations:

$C = A \overset{*}{\cup} B$	region $C = \{1, 2, 3\}$
$D = A \overset{*}{\cap} B$	region $D = \{3\}$
$E = A \overset{*}{-} B$	region $E = \{1\}$
$F = B \overset{*}{-} A$	region $F = \{2\}$
$G = (A \overset{*}{\cup} B) \overset{*}{-} (A \overset{*}{\cap} B) = C \overset{*}{-} D$	region $G = \{1, 2\}$

FIGURE 53. Boolean Ops over Regions are the Set Boolean Ops (After Merging).

Regions need not model closed sets. Since regions are merely subsets of the cells of a cell complex, the usual mathematics of sets apply including closure of Boolean operations within the powerset of a set. For example, in Figure 54, both region A and region B are not closed. The results of Boolean operation of these two regions are valid but not closed.

FIGURE 54. Regions Need not Model Closed Sets. Use two Regions: A and B from Figure 52, the Results of Boolean Ops Over these two Regions are not Closed.

Summary

In brief, cell complex non-manifold representation described above provides 1)dimensional independence and 2)non-homogeneous representation, 3)one monotonic operator (merge) and 4) linear time algorithm for boolean operation. A basic object called region*s*, denotes a point-set within the carrier point-set in a default cell complex. Each region models invariant point-set in space, and remain monotonic across Boolean operation. With this cell complex representation, we describe four examples domain relevant relations in the next section.

6.4.2 Representation with Relations

With the efficient Boolean algorithms over cell complex representation, we illustrate four domain relevant relations in this section. The domain relevant relations identified for the cell-complex representation are 1)point-set 'lifted', 2)point-set subset, 3)point-set inclusion and 4)point-set interval inclusion. The relevancy of the first two relations are simple. For example, point-set subset relation can be used to describe an exterior wall of a room as *subset* of an abstract room space that specifies the boundary of that room. The third relation indicates that an abstract room includes its exterior wall. The fourth relation is similar to an interval that can be useful to describe an interval constraint over geometry.

Each *geometry* type *is* defined across a particular domain-relevant relation according to its proposition, such as point-set inclusion/subset (as seen in Figure 55). Four domain relevant relations described above over the cell complex geometric objects (regions) are used to form the *geometry* type*s* in the following sections.

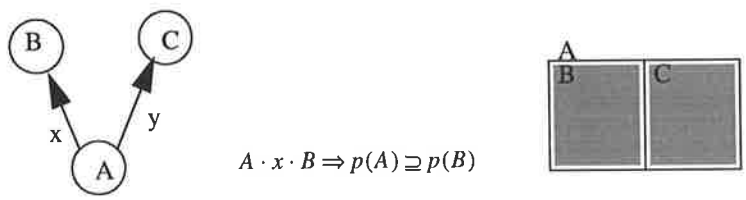


FIGURE 55. Example of Point-Set Containing Relation Among Geometry Types.

As described before, the four properties of these relations are

1. the partial order condition (reflexivity, anti-symmetry, and transitivity),
2. the BCPO condition (a unique bottom and the least upper bound),
3. an efficiency algorithms for subsumption/unification and
4. a way to express the members of the set.

Four these relations over non-manifold point-set domain are proposed as followed:

- Pset: points sets in space compatible with cell-complex-based non-manifold geometric modelling as an antichain and lifted to include a bottom—the space.
- IPSet: points sets in space under point-set inclusion relation.
- OPSet: points sets in space under point-set subset relation.
- IOPSet: points sets in space under subset/inclusion relation.
- OIPSet: points sets in space under inclusion/subset relation.

For representing geometric information, the linear algorithm of non-manifold solids described in the previous section provides an efficient mechanism for subsumption/unification over non-manifold domain, which is the requirement number 3 above. On the other hand, since a CSG[1]-like expression which describes the generation of geometry. With enhanced access through efficient adjacency computations, the properties of non-manifold also satisfies the criteria 4 above. Therefore, with these four conditions and combined with four different relations over them, we describe a set of *geometry* types in detail in the following sections.

1. CSG stands for "constructive solid geometry", in which a "CSG object is 'built' from the standard primitive using regularised Boolean operations and rigid motion".[Hoffmann, 1989] A CSG expression described here is an expression comprising regularised Boolean operations over regions.

6.4.3 The 'lifted' point-sets

Given the ordered set $\overline{PSet} = \langle PSet, \bullet \rangle$ where $PSet$ is the set of all connected point sets and $a \bullet b$ iff $a \subseteq b$ and $a \supseteq b$ [1], we form $\overline{PSet_\perp}$, which we call the *'lifted'* point-sets by adding an element $\perp$, $PSet_\perp := PSet \cup \{\perp\}$. We define an order relation '$\bullet$' ($PSet$ is an antichain[2]) on $PSet_\perp$ as $\overline{PSet_\perp} = \langle PSet_\perp, \bullet \rangle$. The definition is

$$a \bullet b \text{ if } a = \perp \text{ and } b \in PSet \qquad \textbf{(EQ 27)}$$

$$a \bullet a \text{ if } a \in PSet_\perp \qquad \textbf{(EQ 28)}$$

Usually we refer to $\overline{PSet_\perp}$ simply as $PSet_\perp$. A partial inheritance hierarchy formed by $\langle PSet_\perp, \bullet \rangle$ is shown in Figure 56. With this definition, the four conditions described above apply as follows:

1. $\langle PSet_\perp, \bullet \rangle$ is a partial order. Let $a, b, c \in PSet_\perp$

   reflexivity: $a \bullet a$ is true by definition (see equation: 28)□

   anti-symmetry: $a \bullet b$ and $b \bullet a \Rightarrow a = b$

   by exhaustive enumeration,

   if $a \bullet b$, either $a = b$ or $a = \perp, b \neq a$,

   in the former,

   $a \bullet b$ and $b \bullet a \Rightarrow$

   $a \bullet a$ and $a \bullet a \Rightarrow$

   $a \bullet a \Rightarrow$

   $a = b$

   in the latter,

---

1. We define two point-sets (a and b) are equal if these point-sets occupy the same space.
2. *"The ordered set P is an antichain if $x \leq y$ in P only if $x = y$."*[Davey and Priestley, 1994].

$a \bullet b$ but $\neg b \bullet a$

$\therefore a \bullet b$ and $b \bullet a$ and $b \neq a$ are false. □

transitivity: $a \bullet b$ and $b \bullet c \Rightarrow a \bullet c$

by exhaustive enumeration.

if $a \bullet b$, either $a = b$ or $a = \perp, a \neq b$ **(EQ 29)**

in the former,

$a \bullet b$ and $b \bullet c \Rightarrow$

$a \bullet a$ and $a \bullet c \Rightarrow \quad a \bullet c$

in the latter,

$a \bullet b$ and $b \bullet c \Rightarrow$

$\perp \bullet b$ and $b \bullet c \Rightarrow$

Note that $b = c$, as $b \neq a$ from [equation: 29].

$\perp \bullet b$ and $b \bullet c \Rightarrow$

$\perp \bullet c$ and $c \bullet c \Rightarrow \quad \perp \bullet c$,

but $a = \perp$, so $a \bullet b$ □

2. $\langle PSet_{\perp}, \bullet \rangle$ is a BCPO.

As mentioned before, a tree is a BCPO. A 'lifted' anti-chain is a tree, thus, $\langle PSet_{\perp}, \bullet \rangle$ is a BCPO. □

3. There must be an efficient means of computing subsumption and unification. We take $\langle PSet_{\perp}, \bullet \rangle$ as an inheritance hierarchy such that $a \sqsubseteq b = a \bullet b$.

Subsumption

$\perp \sqsubseteq a, a \in PSet$

$a \sqsubseteq a, a \in PSet_{\perp}$

The subsumption algorithm is trivial and computed based on its simple symbol comparison.

Unification

$\perp \sqcup a = a, a \in PSet_{\perp}$

$a \sqcup a = a, a \in PSet$

The subsumption algorithm is trivial and computed based on its simple symbol comparison.

4. There must be a way to specify members of the set. Elements of $PSet$ stand as names for themselves. Explicit expressions over $PSet$ with Boolean operators[1] can also stand as names for elements of $PSet_{\perp}$.

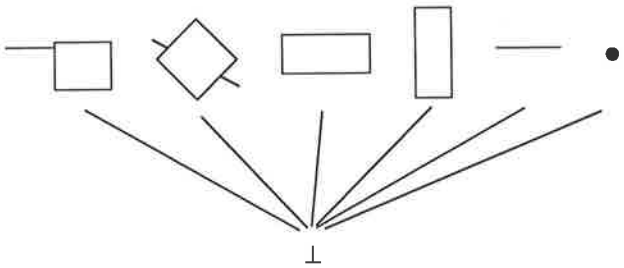


FIGURE 56. A Part of 'Lifted' Point-Set as an *InheritanceHierarchy*.

1. Boolean operators are *union*, *difference* and *intersection*.

6.4.4 The growing point-sets

Given the ordered set $PSet_{\subseteq} = \langle PSet, \subseteq \rangle$ where $PSet$ is the set of all connected point sets[1] and $\subseteq$ denotes the point-set subset (contained by) relation, which we call the *growing* point-sets. We denote the point-set *subset* relation as

> Let A and B be point-sets. Then A *is a subset of* B, denoted $A \subseteq B$, if all the points of A are within B. **(EQ 30)**

$PSet_{\subseteq}$ has a bottom, the null set as ϕ.

We usually refer to $PSet_{\subseteq}$ as $IPSet$; the I indicates the sense of that a given set is informally an "innermost" set of points all of which must be contained in any specialisation. A partial inheritance hierarchy formed by $IPSet$ is shown in Figure 57. With this definition, the four conditions of Order Types apply as follows:

1. $IPSet$ is a partial order.

   > $\subseteq$ is a subset relation where the sets in question are the sets of points that comprise the objects in $PSet$. Like all such subset relations over sets it is a partial order. □

2. $IPSet$ is a BCPO. We define a subsumption ordering of $PSet$ over relation $\subseteq$, such that $a \sqsubseteq b = a \subseteq b$[2].

   Bottom. $OPSet$ has a bottom as ϕ.

   A least upper bound.

   Let Q be a set of point-sets from $PSet$

   > Each upper bound of Q must contain all the points in every member of Q. $\bigcup Q$ is precisely the set of such points. So every upper bound of Q will contain $\bigcup Q$. Thus, $\bigcup Q$ is a least upper bound for Q. □

---

1. Any representation scheme will choose a subset of $PSet$ as its domain. We require simply that the chosen subset of $PSet$ has a bottom under $\subseteq$. This restriction can, of course, be lifted by the simple expedient of lifting the set.
2. We could also define $a \sqsubseteq b = a \subseteq c(i(b))$ where c denotes *closure* and i denotes *interior*. This would make unification equivalent to regularised union. We do not show here that the resulting type hierarchy is a BCPO.

3. There must be an efficient means of computing subsumption and unification. We take $\langle PSet, \subseteq \rangle$ as an inheritance hierarchy such that $a \sqsubseteq b = a \subseteq b$.

Subsumption

In non-manifold cell complex-based geometric modelling, the subsumption algorithm is linear (after merging). Its algorithm is the point-set inclusion algorithm in a non-manifold cell complex.

Unification

In non-manifold cell complex-based geometric modelling, the unification algorithm is linear (after merging). Its algorithm is the union algorithm in a non-manifold cell complex.

4. There must be a way to specify members of the set $IPSet$. Primitive parametric objects, transformations of these and combination of both under the unregularised or regularised Boolean operation can stand as names of elements of $IPSet$. Let the set of such expressions be called Exp . Other algebras, for example, sweeps are also possible. An advantage of using the Boolean ops is that it is pos-

sible to write expressions that name subsumed elements of $a \in IPSet$. These comprise any expression that can be reduced to $a \cup b, b \in \text{Exp}$.

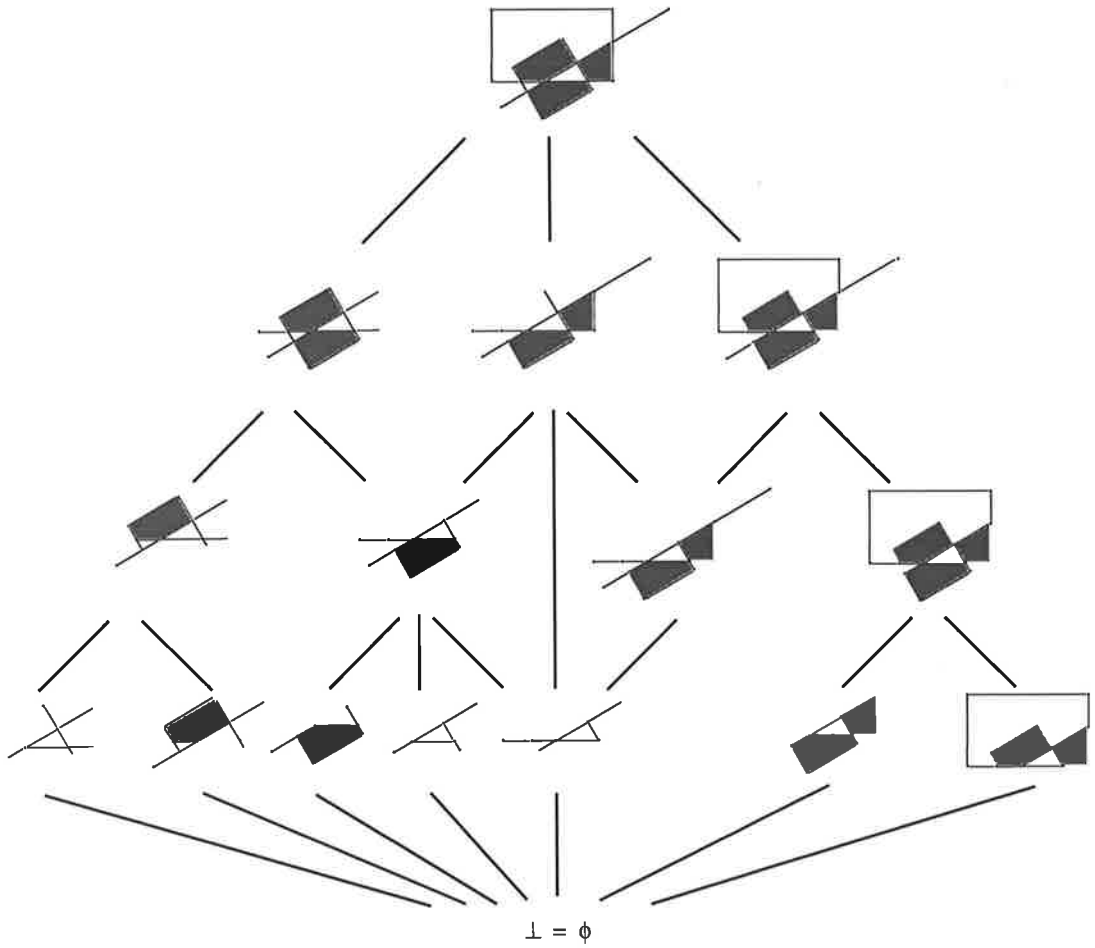


FIGURE 57. A Part of *IPSet* as an *InheritanceHierarchy*.

6.4.5 The shrinking point-sets

Given the ordered set $PSet_{\supseteq} = \langle PSet, \supseteq \rangle$ where $PSet$ is the set of all connected point sets except for the empty set and $\supseteq$ denotes the point-set superset relation, which we call the *shrinking* point-sets. We denote the point-set superset (containing) relation as

> Let A and B be point-sets. Then A *is a superset of* B, denoted $A \supseteq B$, if all the points of B are within A. **(EQ 31)**

$PSet_{\supseteq}$ has a bottom, the set of all points and denoted as $\infty \in PSet$.

We refer to $PSet_{\supseteq}$ as $OPSet$; the O indicates the sense of that a given set is informally an "outermost" set of points all of which must be contained in any specialisation. A partial inheritance hierarchy formed by $OPSet$ is shown in Figure 58. With this definition, the four conditions of Order Types apply as follows:

1. $OPSet$ is a partial order.

   > $\supseteq$ is a superset relation where the sets in question are the sets of points that comprise the objects in $PSet$. Like all such superset relations over sets it is a partial order. □

2. $OPSet$ is a BCPO. We define a subsumption ordering of $PSet$ over relation $\supseteq$, such that $a \sqsubseteq b = a \supseteq b$ [1].

   Bottom. $OPSet$ has a bottom that comprises all the points in $PSet$.

   A least upper bound.

   Let Q be a set of point-sets from $PSet$

   > Each upper bound of Q must contain all the common points in every member of Q. $\bigcap Q$ is precisely the set of such points. So every upper bound of Q will contain $\bigcap Q$. Thus, $\bigcap Q$ is a least upper bound for Q. □

---

1. We could also define $a \sqsubseteq b = a \supseteq c(i(b))$ where c denotes *closure* and i denotes *interior*. This would make unification equivalent to regularised union. We do not show here that the resulting type hierarchy is a BCPO.

3. There must be an efficient means of computing subsumption and unification. We take $\langle PSet, \supseteq \rangle$ as an inheritance hierarchy such that $a \sqsubseteq b = a \supseteq b$.

Subsumption

In non-manifold cell complex-based geometric modelling, the subsumption algorithm is linear (after merging). Its algorithm is the point-set intersection algorithm in a non-manifold cell complex.□

Unification

In non-manifold cell complex-based geometric modelling, the subsumption algorithm is linear (after merging). Its algorithm is the point-set intersection algorithm in a non-manifold cell complex.□

4. There must be a way to specify members of the set $OPSet$. Primitive parametric objects, transformations of these and combination of both under the unregularised or regularised Boolean operation can stand as names of elements of $OPSet$. Let the set of such expressions be called Exp . Other algebras, for example, sweeps are also possible. An advantage of using the Boolean ops is that it is pos-

sible to write expressions that name subsumed elements of $a \in OPSet$. These comprise any expression that can be reduced to $a \cap b, b \in \text{Exp}$.

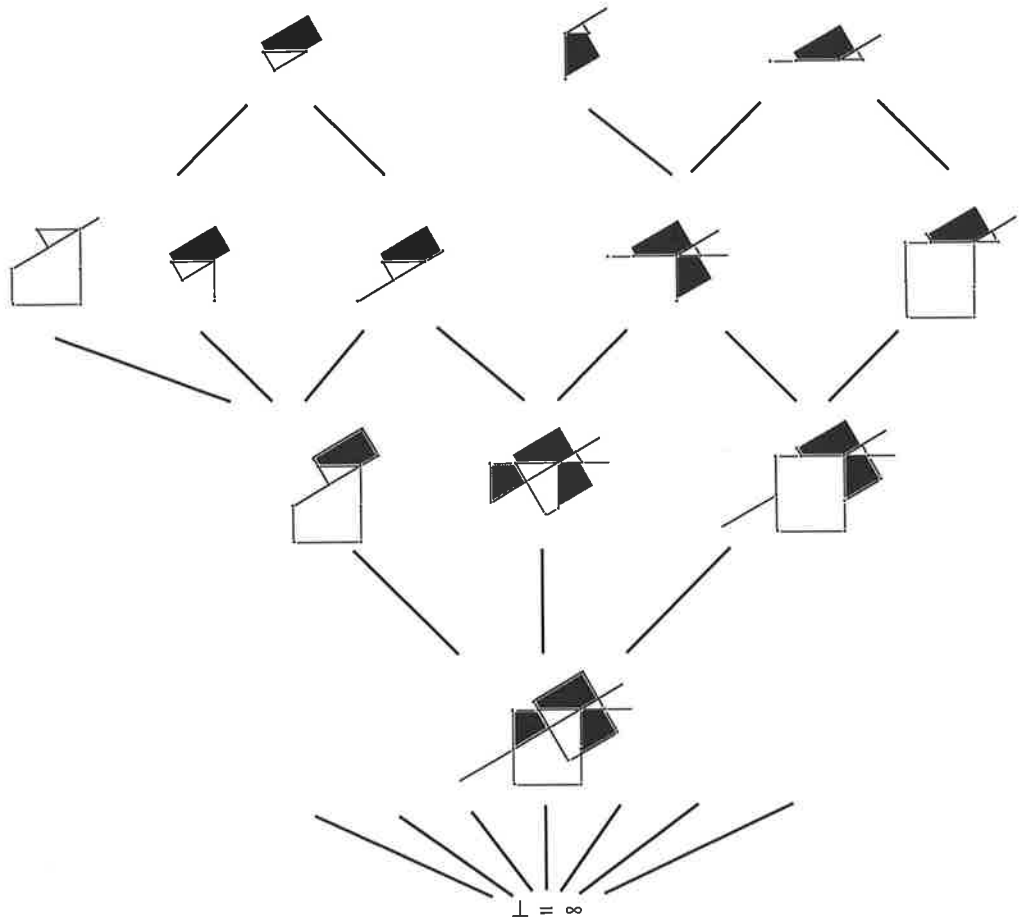


FIGURE 58. A Part of *OPSet* as an *InheritanceHierarchy*.

6.4.6 The IGOSet intervals

The ordered set $PSet_{\circledcirc} = \langle [PSet_{\subseteq}, PSet_{\supseteq}], \circledcirc \rangle$, where $PSet$ is the set of all point-set and $\circledcirc$ denotes inner growing outer shrinking relation. We call the elements of $PSet_{\circledcirc}$ the *IGOSET intervals*. First we define $pset(n)$ as a polymorphic function giving the point-set corresponding to an element in $PSet_{\subseteq}$ or $PSet_{\supseteq}$. $pset(a)$ and $pset(b)$ are comparable by $\subseteq$ and $\supseteq$ irrespective of the set membership of a and b in $PSet_{\subseteq}$ and $PSet_{\supseteq}$. We distinguish the set $\infty \in PSet$ as the set of all points and the set $\phi \in PSet$ as the null set.

We refer to $PSet_{\circledcirc}$ as $IOPSet$ (inner-outer point-sets). We write an IGOSet interval a as $a = [a_i, a_o]$ and use a_i and a_o to denote the inner and outer bound of the interval respectively, such that $pset(a_i) \subseteq pset(a_o)$. A partial inheritance hierarchy formed by $PSet_{\circledcirc}$ is shown in Figure 60. The definition of $PSet_{\circledcirc}$ is

Let $a \in PSet_{\subseteq}, b \in PSet_{\supseteq}, [a, b] \in PSet_{\circledcirc}$, if $pset(a) \subseteq pset(b)$. **(EQ 32)**

$a \circledcirc b$ if $a_i \subseteq b_i$ and $a_o \supseteq b_o$ **(EQ 33)**

1. $PSet_{\circledcirc}$ is a partial order.

Let $a, b, c \in PSet_{\circledcirc}$

reflexivity: $a \circledcirc a$

As a special case shown in equation: 33, $a \circledcirc a$ is true by definition.

anti-symmetry: $a \circledcirc b$ and $b \circledcirc a$ imply $a = b$

The definition of $a \circledcirc b$ relies directly on the relations $\subseteq$ and $\supseteq$ over the sets $PSet_{\subseteq}$ and $PSet_{\supseteq}$ respectively. Both $PSet_{\subseteq}$ and $PSet_{\supseteq}$ are partial orders and anti-symmetric. Therefore, $PSet_{\circledcirc}$ is anti-symmetric as its definition (see equation: 33) is simply the conjunction of the above two conditions. □

transitivity: $a \circledcirc b$ and $b \circledcirc c$ imply $a \circledcirc c$

By the argument used for anti-symmetry, $PSet_{\circledcirc}$ also satisfies the transitivity condition. $\square$

2. $PSet_{\circledcirc}$ is a BCPO.

Bottom. $[\perp_{PSet_{\subseteq}}, \perp_{PSet_{\supseteq}}] \circledcirc a, a \in PSet_{\circledcirc}$.

A least upper bound.

Following the definition of upper bound[1] from [Davey and Priestley, 1994],

Let $S \in \wp(PSet_{\circledcirc})$, and let S be consistent, that is, there is some $x \in PSet_{\circledcirc}$ such that $s \circledcirc x, s \in S$.

Informally, two point-set intervals are consistent if the union of their inner sets is inside the intersection of their outer sets. The least upper bound of S is

$$[\bigcup s_i, \bigcap s_o], s \in S, \text{ if } (\bigcup s_i \subseteq \bigcap s_o), \textit{undefined} \text{ otherwise.} \square$$

3. There must be an efficient means of computing subsumption and unification. We take $PSet_{\circledcirc}$ as an inheritance hierarchy such that $a \sqsubseteq b = a \circledcirc b$. An example is shown in Figure 59.

Subsumption

The subsumption algorithm is computed based on its lower bound ($PSet_{\subseteq}$) and upper bound ($PSet_{\supseteq}$) subsumption algorithms, which is linear in the non-manifold representation.

1. "Let P be an ordered set and let $S \subseteq P$. An element $x \in P$ is an *upper bound* of S if $s \leqslant x$ for all $s \in S$. ... The set of all upper bounds of S is denoted by S^u"[Davey and Priestley, 1994]

Unification

The unification algorithm is computed based on its lower bound ($PSet_{\subseteq}$) and upper bound ($PSet_{\supseteq}$) unification algorithms, which is linear in the non-manifold representation.

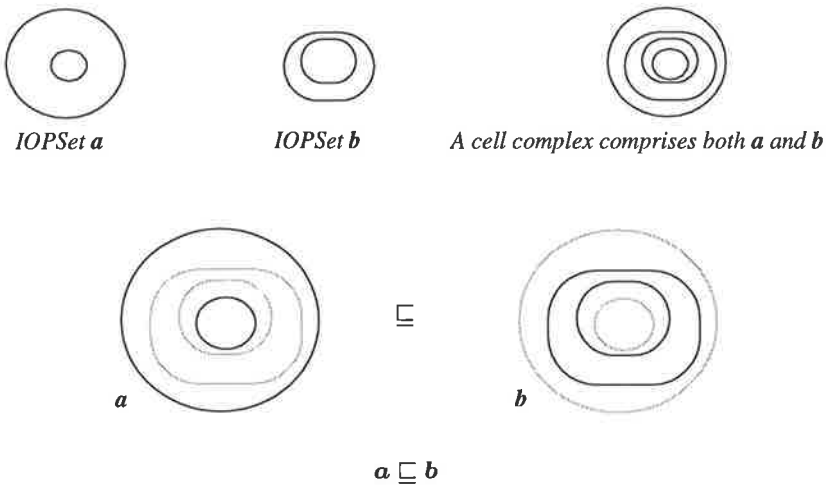


FIGURE 59. Example of Two IOPSets a and b, Such that $a \sqsubseteq b$.

4. There must be a way to specify members of the set $PSet_{\circledcirc}$. Primitive parametric objects, transformations of these and combination of both under the unregularised or regularised Boolean operation can stand as names of inner/outer bounds of elements of $PSet_{\circledcirc}$.

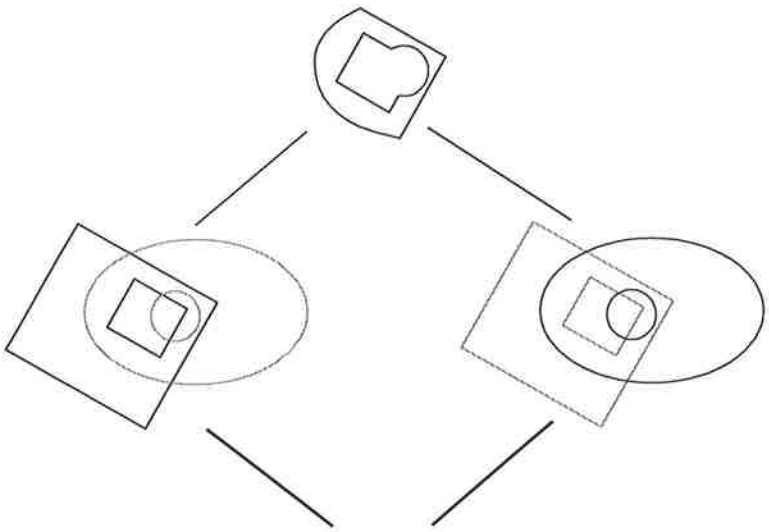
Let's define two operations: $\circledcirc$ and $\bullet$ as:

$$a \circledcirc b = [(a_i \cup b_i), (a_o \cap b_o)], a, b \in PSet_{\circledcirc}, (a_i \cup b_i) \subseteq (a_o \cap b_o)$$

$$a \bullet b = [(a_i \cap b_i), (a_o \cup b_o)], a, b \in PSet_{\circledcirc}, (a_i \cap b_i) \subseteq (a_o \cup b_o)$$

Thus, the set of such expressions using these two operations be called Exp. Exp is not close. Other algebras, for example, sweeps are also possible. An advantage of using these two operations is that it is possible to write expressions that

name subsumed elements of $a \in IPPSet$. These comprise any expression that can be reduced to $a \circledcirc b, b \in \text{Exp}$.



$I = \phi, O = \infty$

FIGURE 60. A Part of *IOPSet* as an *InheritanceHierarchy*.

6.4.7 The OGISet intervals

The ordered set $PSet_{\bullet} = \langle [PSet_{\supseteq}, PSet_{\subseteq}], \bullet \rangle$, where $PSet$ is the set of all point-sets and $\bullet$ denotes outer growing and inner shrinking relation. We call the elements of $PSet_{\bullet}$ the *OGISET intervals*. By adding an element $\perp$, we form $PSet_{\bullet_{\perp}} := PSet_{\bullet} \cup \{\perp\}$. We refer $PSet_{\bullet_{\perp}}$ simply as $PSet_{\bullet}$. First we define $pset(n)$ as a polymorphic function giving the point-set corresponding to an element in $PSet_{\supseteq}$ or $PSet_{\subseteq}$. $pset(a)$ and $pset(b)$ are comparable by $\supseteq$ and $\subseteq$ irrespective of the set membership of a and b in $PSet_{\supseteq}$ and $PSet_{\subseteq}$. We distinguish the set $\infty \in PSet$ as the set of all points and the set $\phi \in PSet$ as the null set.

We refer to $PSet_{\bullet}$ as $OIPSet$ (outer-inner point-set). We write an OGISet interval a as $a = [a_i, a_o]$ and use a_i and a_o to denote the inner and outer bound of the interval respectively, such that $pset(a_i) \subseteq pset(a_o)$. A partial inheritance hierarchy formed by $PSet_{\bullet}$ is shown in Figure 62. The definition of $PSet_{\bullet}$ is

Let $a \in PSet_{\supseteq}, b \in PSet_{\subseteq}, [a, b] \in PSet_{\bullet}$, if $pset(a) \subseteq pset(b)$. **(EQ 34)**

$a \bullet b$ if $a_i \supseteq b_i$ and $a_o \subseteq b_o$ **(EQ 35)**

1. $PSet_{\bullet}$ is a partial order.

Let $a, b, c \in PSet_{\bullet}$

reflexivity: $a \bullet a$

As a special case shown in equation: 35, $a \bullet a$ is true by definition.

anti-symmetry: $a \bullet b$ and $b \bullet a$ imply $a = b$

The definition of $a \bullet b$ relies directly on the relations $\supseteq$ and $\subseteq$ over the sets $PSet_{\supseteq}$ and $PSet_{\subseteq}$ respectively. Both $PSet_{\supseteq}$ and $PSet_{\subseteq}$ are partial orders and anti-symmetric. Therefore, $PSet_{\bullet}$ is anti-symmetric as its definition (see equation: 35) is simply the conjunction of the above two conditions. □

transitivity: $a \bullet b$ and $b \bullet c$ imply $a \bullet c$

By the argument used for anti-symmetry, $PSet_{\bullet}$ also satisfies the transitivity condition. □

2. $PSet_{\bullet}$ is a BCPO.

Bottom. We have a bottom $\perp$ and $\perp \bullet a, a \in PSet_{\bullet}$.

A least upper bound.

Following the definition of upper bound[1] from [Davey and Priestley, 1994],

Let $S \in \wp(PSet_{\bullet})$, and let S be consistent, that is, there is some $x \in PSet_{\bullet}$ such that $s \bullet x, s \in S$.

Informally, two OGISet intervals are consistent if the intersection of their inner sets is inside the union of their outer sets. The least upper bound of S is

$$[\bigcap s_i, \bigcup s_o], s \in S, \text{ if } (\bigcap s_i \subseteq \bigcup s_o), \textit{undefined} \text{ otherwise. } \square$$

3. There must be an efficient means of computing subsumption and unification. We take $PSet_{\bullet}$ as an inheritance hierarchy such that $a \sqsubseteq b = a \bullet b$. An example is shown in Figure 61.

Subsumption

The subsumption algorithm is computed based on its lower bound ($PSet_{\supseteq}$) and upper bound ($PSet_{\subseteq}$) subsumption algorithms, which is linear (after merging) in the non-manifold representation.

1. "Let P be an ordered set and let $S \subseteq P$. An element $x \in P$ is an *upper bound* of S if $s \leqslant x$ for all $s \in S$. ... The set of all upper bounds of S is denoted by S^u"[Davey and Priestley, 1994]

Unification

The unification algorithm is computed based on its lower bound ($PSet_{\supseteq}$) and upper bound ($PSet_{\subseteq}$) unification algorithms, which is linear (after merging) in the non-manifold representation.

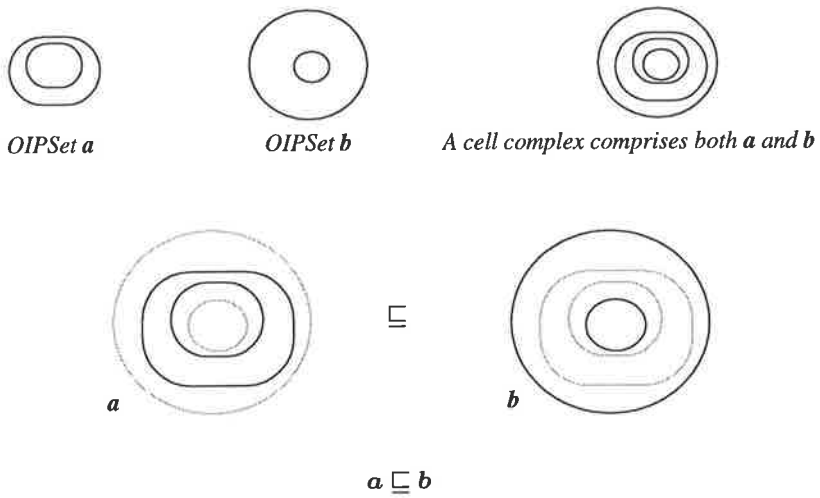


FIGURE 61. Example of Two OIPSets a and b, Such That $a \sqsubseteq b$.

4. There must be a way to specify members of the set $OIPSet$. Primitive parametric objects, transformations of these and combination of both under the unregularised or regularised Boolean operation can stand as names of the inner/outer bounds of elements of $OIPSet$.

Let's define two operations: $\circledcirc$ and $\bullet$ as:

$$a \circledcirc b = [(a_i \cup b_i), (a_o \cap b_o)], a, b \in PSet_{\bullet}, (a_i \cup b_i) \subseteq (a_o \cap b_o)$$

$$a \bullet b = [(a_i \cap b_i), (a_o \cup b_o)], a, b \in PSet_{\bullet}, (a_i \cap b_i) \subseteq (a_o \cup b_o)$$

Thus, let the set of such expressions comprise these two operations be called Exp. Exp is not close. Other algebras, for example, sweeps are also possible. An advantage of using these operations is that it is possible to write expressions

that name subsumed elements of $a \in OIPSet$. These comprise any expression that can be reduced to $a \bullet b, b \in \text{Exp}$.

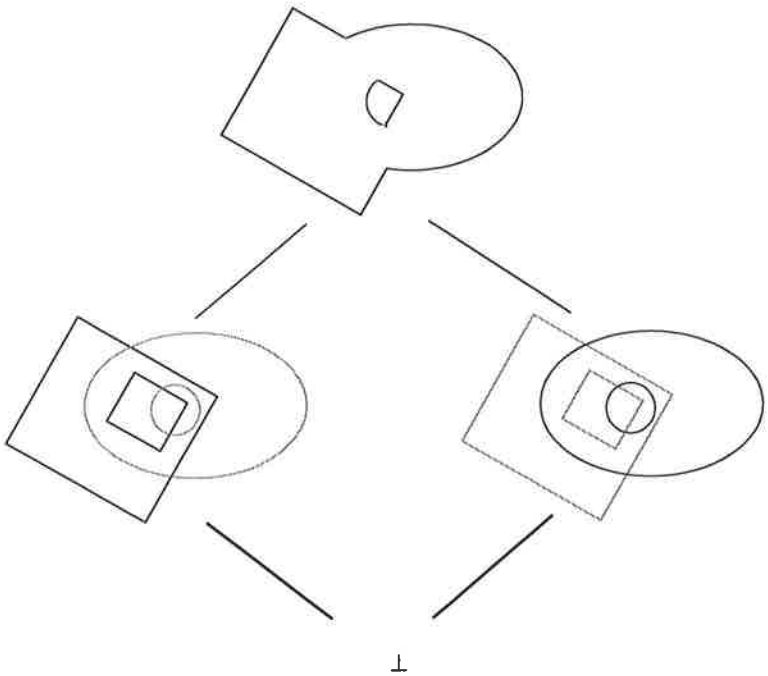


FIGURE 62. A Part of *OIPSet* as an *InheritanceHierarchy*.

6.5 Summary

The knowledge domain of Typed Feature Structures has been extended through the realisation of knowledge level corresponding with its symbol level representation—Geometric Typed Feature Structures. The future framework of design space explorers is also demonstrated through the capability of Geometric Typed Feature Structures which specialises in carrying 3D domain information as well as in dealing with synchronisation of domain knowledge in the system level and the HCI level.

Two major questions are immediately raised. Firstly, the way of finitely representing the continuous domain of domain information which uniforms geometric information and other domain knowledge. Secondly, the way of finding a clear, uniform and formal means for carrying domain information which covers not only geometric information but also other domain knowledge. The first question has been achieved by utilising Order Types to define the domain information which can finitely represent the continuous domain (infinite type hierarchy). The second question has been resolved by the algorithms which provide Order Types as the carriers for carrying domain information within the framework of Geometric Typed Feature Structures.

The representation of Order Types also has been characterised through three requirements and satisfied for four main conditions in order to identify the given knowledge objects in the domain information.

Mapping between the abstract knowledge representation and Typed Feature Structures is to find the connection of Order Types and geometric information in which geometry types and geometric information are synchronised with domain knowledge. Five significant geometry types, 'lifted' point-set, IPSet, OPSet, IOPSet and OIPSet, are given and described according to its distinct features in order to demonstrate four Order Type condition for geometry types.

By using the symbol level representation (non-manifold solids) of carrying 3D domain information in Geometric Typed Feature Structures, we can then sketch a software architecture—implementation of Ordering Geometry System in the next part. Implementation with explanation and methods of using is segmented stated to demonstrate applicability of this formalism statement by applying within two examples—a single fronted cottage and a building enclosure.

3

Implementation and Examples

Given the potential framework of design space explorers through the realisation of the knowledge level with its symbol level representation—Geometric Typed Feature Structures, a practicable implementation is essential. An implementation called "Ordering Geometry System" (OGS) not only is required for a prospect design space explorer but also demonstrate the main concept of geometric information which can be represented in the core of Geometric Typed Feature Structures.

The description of Ordering Geometry System utilises the syntax of Kryos description and Order Type expression discussed in the examples of the previous chapter. Two examples—a single fronted cottage and a building enclosure are also used to illustrate the ordering of geometric objects and to define the expression of these geometric objects. Chapter 8 then demonstrates the process of integration of Ordering Geometry System with practical building design instances.

Part 3 is divided into three chapters:

Chapter 7: "Implementation—Ordering Geometry System"

This chapter addresses the implementation issues of the design space explorer as well as gives a sketch of an implementation conducted by this dissertation: Ordering Geometry System (OGS). Geometric representation and geometry generation are the two vital concepts of Ordering Geometry System, which are depicted in two examples in the next chapter.

Chapter 8: "Examples"

A typical Australian house plan (of a single fronted cottage) and a building enclosure design are used as examples to demonstrate the ordering of geometric objects (as the implementation of Order Type described in the previous chapter) and to demonstrate how to specify the expression in which is needed to instantiate the given geometric objects.

7

Implementation—Ordering Geometry System

Ordering Geometry System (OGS) forms the kernel of the design space explorer developed by this dissertation. OGS allows users to interact with this kernel via a layer of an application programming interface, thus providing an opportunity for directly manipulating well-defined geometric objects in a reasoning framework. As the kernel, the main programming task of OGS is to provide a unique framework which allows Order Types (as their instances in the inheritance hierarchies) as well as Succession Types to communicate with each other by the values they represent. One of those values is geometric information, which is implemented via a non-manifold modeller. An initial system concept is shown in Figure 63.

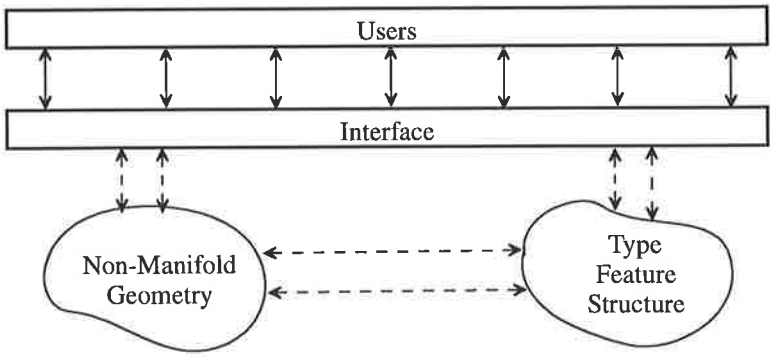


FIGURE 63. System for a Design Space Explorer Needs to Communicate with Both Typed Feature Structures and Non-Manifold Geometry.

In order to accomplish this task, two system design issues arise—system integration, which describes the structure of OGS prototype; and Order Type implementation, which is an implementation design according to this prototype. Each of the

system design issues and Order Type implementation issues are recorded in a separate section (Chapter 7.1 and Chapter 7.2), followed by two examples written in C++ of Order Types carrying real numbers and geometric information.

7.1 System Integration

The core of Geometric Typed Feature Structures (described in Chapter 6) is Typed Feature Structures which implements the symbol level of design space explorers. By extending Typed Feature Structures, Geometric Typed Feature Structures provide a set of specific Order Types which represent the domain-relevant objects/relation, particularly the geometric objects. This prototype uses two software components to achieve this goal—Kryos, an implementation of Typed Feature Structures; and SHAPES[XOX, 1995], a non-manifold geometric kernel. A sketch of interaction between these two systems with a viewer (a light-weight OpenInventor clone on the top of SHAPES' geometric objects), and a general interface library which connects the kernel to users is shown in Figure 64.

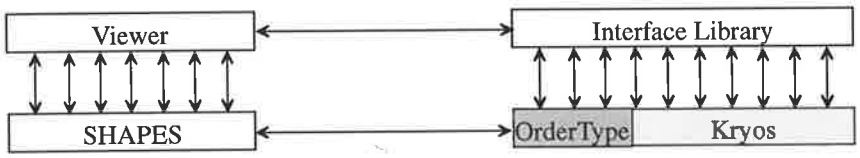


FIGURE 64. Implementation of Geometric Typed Feature Structures Comprises Both Kryos and SHAPES (and its Display Library: Viewer).

In Figure 65, design representation and associated objects represent the instance of design processes in a design space. The interface, as the "search" metaphor of design space explorers, provides users with the capability of directly manipulating those instances. Users elaborate/refine the design instances via the interface, which

then communicates with the design representation to retrieve the results of constraint resolution, or to visualise the design exploration.

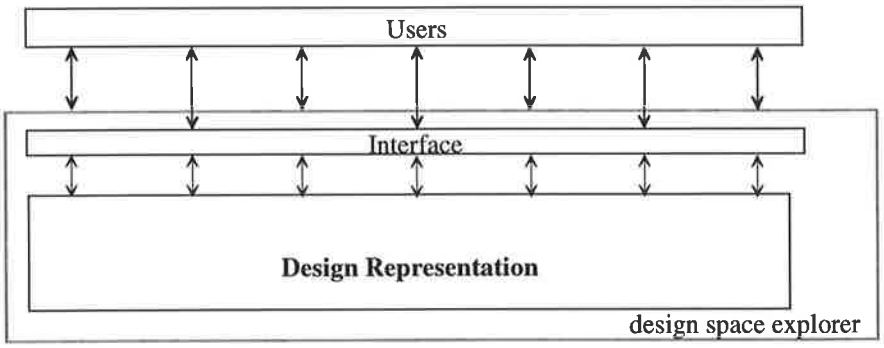


FIGURE 65. A Generic System Design for a Design Space Explorer Requires a Design Representation.

For the focus of this dissertation, we will only address the geometry related issues of describing those knowledge representation, especially for the geometric objects, as we have done throughout this thesis.

7.1.1 A System for Geometric Typed Feature Structures

With Geometric Typed Feature Structures, the main implementation challenge is to combine Typed Feature Structures objects and non-manifold solids. Thus, the strategy of such a system architecture is to realise Order Types as a group of objects which then connect to the internal Typed Feature Structures representation to perform the unification/subsumption mechanisms. These objects thus must contain the geometric information internally. An illustration of this concept is shown in Figure 66

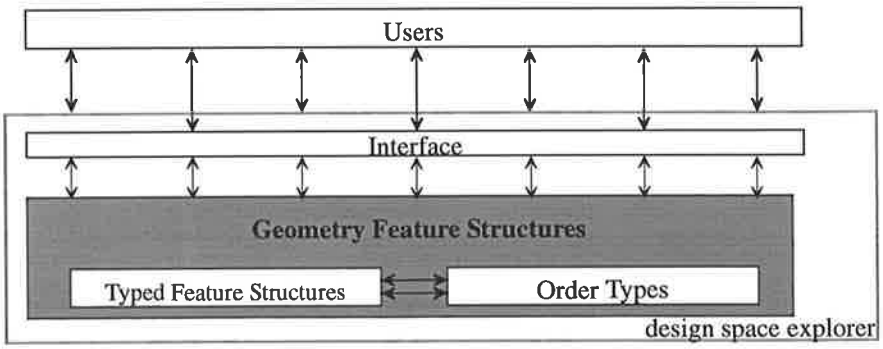


FIGURE 66. An Implementation of Representation: Geometric Typed Feature Structures which Contains Typed Feature Structures and Order Types.

The Typed Feature Structures implementation that we have used is called "Kryos", an objects-oriented reasoning framework developed by Andrew L. Burrow at The University of Adelaide. Programming in C++, Kryos provides the fundamental class library for an ordered object system such as Inheritance Hierarchy(for instance of regular types), *StructNode* (for the instances of feature structure) and *DescNode* (for the description) classes. Based on Kryos, we implement the concepts of Order Types.

Another implementation which uses the logic of Typed Feature Structures is ALE[Carpenter and Penn, 1997]. Unlike Kryos, ALE is built on top of a Prolog environment which comes with all the basic built-in functions. The main drawbacks using ALE as a platform to implement design space explorers are its lack of incremental resolution control and its scalability which is needed for developing a large scale environment such as a design space explorer.[1] On the other hand, Kryos, as a software framework, provides the necessary components in the form of linkable objects, which makes the communication with the geometric modeller possible. In addition, Kryos provides an incremental exploration algorithm called *lazy* π *-resolution algorithm* [Burrow and Woodbury, 1998] which allow us to control the incremental resolution process in the design process. This, in brief, makes support for rules control (*policy*) through the design space possible.

The other external system module is the non-manifold geometric modeller—the SHAPES geometric kernel[XOX, 1995]. As a non-manifold geometric kernel, the basic non-manifold cellular representation, as well as the linear algorithms over these representations described in Chapter 7, are provided in SHAPES. Bearing this non-manifold geometric representation in mind, several research systems and commercial products, including ACIS kernel[STI, 1995], have been evaluated. Among these systems, SHAPES is one of the few modellers which provides the full non-manifold support needed for this prototype. Furthermore, SHAPES is distributed as a C linkable library, which makes the linking with C++ straightforward. A refined system design using these two external systems is shown in Figure 67. Using both systems, a uniformed representation (Order Type) for geometric objects

1. However, as the constraint resolution of Kryos had not been finished at the time this dissertation was being written and we also use ALE as a generation tool for the examples.

is implemented with aid from Kryos which calls on geometric functions via the API of the SHAPES kernel.

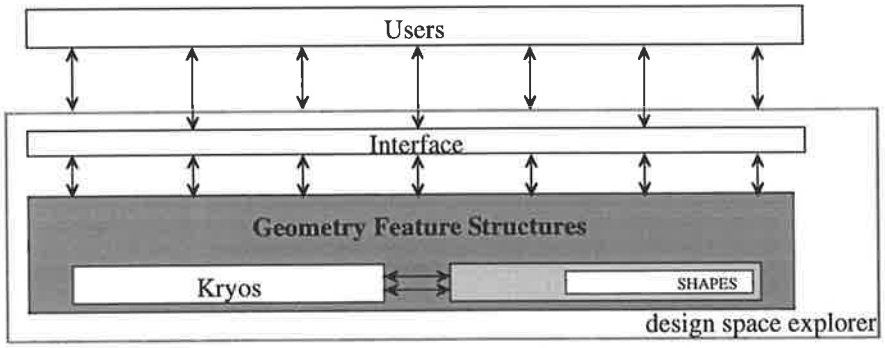


FIGURE 67. A Proposed System Comprises two External Components: Kryos and SHAPES.

7.2 Order Type implementation

There are several issues involved in the implementation of Order Types within OGS. The life time of Order Types in the system needs to be defined in terms of their timing of the construction/destruction of their instances. The resources management of Order Types needs to be considered when implementing Order Type identifier persistence. Furthermore, the construction/destruction of each Order Type needs careful consideration to enforce the pre-orderliness of Order Types. Lastly, the expression of each Order Type is different according to the values they represent. For example, the expression of type *IntInterval* can be specified as $12 + 23 \times int(\sqrt[3]{a}) - A$, and the expression for the point set types will be something different such as $transform(box(12, 765, 304), [0, 0, 1, 0, 1, 0, 1, 0, 0])$. Each of these issue is discussed in detail in the following section.

7.2.1 Order Type lifetime

Order Types represent the types that are not explicitly placed in the inheritance hierarchies. There might be an infinite number of these types. For reasons of efficiency, we must be able to represent only the types that are actually used. The system provides an interface to trigger the constructor of Order Types and records created object in a central database. Each object can "require" several Order Type instances. The destruction of the instances of Order Types should happen only when they are no longer required by other objects.

7.2.2 Order Type resources management

Resources of Order Types are maintained in two singleton[Erich Gamma and Vlissides, 1995] classes; one is the Label of an Order Type which is maintained by the *LabelManager* class; the other is *GeomNameManager* which is employed if the domain object is a geometric object. *LabelManager* maintains a central resource manager of all Typed Feature Structures objects using a simple labelling mechanism and pointer to the object (Index) as object identification in OGS. *GeomNameManager* maintains the bijection relation between the geometric objects these Order Types represent and their labels(*GeomName*)—a generated object identification. The reason for implementing the labels in this way rather than simply using type labels (maintained by *LabelManager*) is because by separating these two labels, the geometric object's identity can be retrievable by other Order Types. For example, an instance of *GeomAscend* might carry identical geometric information to that for an instance of *GeomDescend*. Thus, a *GeomName* is acting as an identification of a geometric object.

The purpose of *LabelManager* is to preserve the communication with type systems by using a simple textual label for each type. This is trivial for representing numbers, but needs an extra mechanism to work with external objects such as geometric objects. Such a mechanism is implemented by *GeomNameManager*. In addition, geometric objects are constructed within the parser, therefore back pointers to these objects have to be stored in the memory during the lifetime of those Order Types representing them.

7.2.3 Order Type Identifier Persistence

As mentioned above, persistence of type identifiers is maintained via textual strings as well as via the pointers of their instances (in *LabelManager*). Thus, communication between the instances of Order Types and the rest of the system is reliant on the textual names of those Order Types. The naming scheme of those geometric objects needs a policy to preserve their consistency and logic in the geometric representation. These names are also used to generate generic type labels that act as their type identification in the system. OGS provides a singleton class—*GeomNameManager* mentioned above to maintain the persistency of such a name scheme. Its associated type label depends on each Order Type carrying geometric information.

7.2.4 Construction of Order Types and domain objects

Construction of Order Types occurs in two conditions—one follows from a description of a feature structure and the other from the unification of two feature structures. Either one is filtered first by a specific preprocessor of OGS to generate the unique label and is then registered in *LabelManager*. Therefore, the system will call the constructor of the corresponding Order Type to construct the instance of an Order Type. The precondition is that the value of a representing Order Type has to be computed before the construction take place. The main purposes of this are to preserve the pre-orderliness of those instances and to enforce a unique label for each type. This is trivial for numbers, since the value of numbers can be represented via textual strings, but needs an extra mechanism to work with an existing data structure such as geometric objects. Such a mechanism is also implemented by the extra resource management described above.

7.2.5 Dereference of Order Types

Dereference of Order Types in OGS is determined by the references of their owner. That is, an instance of an Order Type is deleted when no objects in the system own this instance. This will enforce the efficiency of memory usages. On the other hand, efficient construction of instances of Order Types is required.

7.2.6 Type expression and preprocessor

As described in Chapter 6.1.3, an Order Type description is equivalent to the type expression of those types. Furthermore, those expressions are represented as a vehicle to compute those values. For example, an expression of an instance of IntegerAscend ($<Z, \geq>$) may contain an equation (such as $(12 + 4^2) \cdot |97 - 123|$) to compute an integer represented by this type.

Since in theory, every Order Type has its own way of expressing the value, and the value must be computed before constructing the instance of this Order Type, a preprocessing mechanism is proposed in order to parse and compute these values before the type system picks up the instance of those types. Thus, the syntax of the Expression of each Order Type is defined within the preprocessor classes.

7.2.7 Subsumption relation

The subsumption/unification relations of Order Types are determined by their domain relevant relations. Therefore, these relations exist in the representation of

those values themselves. An implementation of subsumption of Order Types has to be built into the class definition of those types, according to their ordering function.

7.3 Two examples of Order Type implementation using C++

Each Order Type implementation in OGS requires a data module and a parser. The data module specifies the data information and the subsumption algorithms which operate among them. The parser will parse the expression given in the description and will then call out the data in the system. An Order Type instance communicates with the kernel via a set of public methods which are explained in the two examples described below. Thus, subsumption and unification can be computed by each individual Order Type instance via a defined set of public methods.

A data module of an Order Type comprises three C++ classes—a domain information object, a bridge class for communicating with the parser of this Order Type, and a type system extension class. The first class is for the domain object which implements the domain information and the domain relevant relation, which can then be called on for computing the subsumption/unification algorithm. The second class is the preprocessor class which performs the preprocessing filter to deal with the interface with the yacc/lex grammar. The third class is for specifying a type containing these domain objects, and this acts as a generic interface for work with both Succession Type and Order Type. For demonstration purposes, two examples—*RealInterval* and *GeomAscend* are described in the following section.

7.3.1 RealInterval

A domain class[1]*: RealInterval*

This class (RealInterval) is a representation of intervals over the reals(R). Closed intervals are specified as a pair '[lb, ub]' of lower and upper bounds. Open and half-open intervals are specified via bounds at infinity. For example,

- [0, 2.3] is the set {x in R | 0 <= x <= 2.3},
- [-i, 8.7865] is the set {x in R | x <= 8.7865},
- [10, +i] is the set {x in R | 10 <= x}, and
- [-i, +i] is the set R.

1. Details of this class are documented in Appendix B.1 on page 245.

"RealInterval" values are picked out by a collection of strings corresponding to the type labels for these objects. An informational ordering is supplied through the "<=" operator that interprets the intervals as constraints ordered by the domain-relevant relation. An interval 'r' subsumes another 's' iff 's' is a subset of 'r'. "RealInterval" values are also manipulated by an additional constructor which computes the join or meet according to the given informational ordering.

For identification of this Order Type, a unique name scheme has to be specified. For a real interval, the name of this Order Type is "RealInterval" and the format of this object is specified as "real_interval_L_U", where L is the place for specifying the lower bound of the real interval, and U is for specifying the upper bound of the real interval. For example, a valid type name for a real interval type can be: real_interval_2.3_4.8 or real_interval_-i_100.534 (-i is for representing the negative infinite real number).

The constructors of this class are specified as

```
RealInterval();
    // Construct the open interval containing all reals.

RealInterval(RealInterval const& r);
    // Construct a copy of the interval 'r'.

RealInterval(
    Op op, RealInterval const& r, RealInterval const& s
);
    // If 'op == int_interval_join', then construct
    // the join of 'r' and 's' in the informational order "<=".
    // Namely, 'r intersection s'.
    // If 'op == int_interval_meet', then construct the meet
    // in the informational order "<=". Namely, the smallest
    // interval containing 'r union s'.

RealInterval(char const* l);
    // Construct the interval described by the string 'l',
    // where 'l' is of the form described by
    // "RealInterval::is_type_label()".

RealInterval(
        bool has_lb, double lb,
        bool has_ub, double ub
);
    // Construct the interval described by the string 'l',
    // where 'l' is of the form described by
    // "RealInterval::is_type_label()".
```

Once an object is created, this object will register itself with the main resource manager: *LabelManager class* with its label (the name of this type) as a key. Therefore, another construction method is by specifying a type name (in this case, could be real_interval_2_3 for a real interval between 2.0 and 3.0), then, by parsing this type name, the system will create a RealInterval object representing this real interval.

```
static bool is_type_label(char const* l);
    // Return true if 'l' is a non-NULL string, and matches
    // "RealInterval::format". Namely, characters 'L' and 'U'
    // replaced by
    // either:
    // + the string '-i', representing the negative infinite;
    // + a string matching regular expression '(0|[1-9][0-9']*)',
    //     which is a decimal representation of a finite bound; or
    // + the string '+i', representing the absence of an upper bound.

static bool parse_type_label(
    char const* l,
    bool& has_lb, double& lb, bool& has_ub, double& ub
);
    // Parse the type label 'l'. Record the parsed interval bounds in
    // 'has_lb' 'lb', 'has_ub', and 'ub'. Return true if 'l' matches
    // "RealInterval::format" as for "RealInterval::is_type_label()"
```

The domain relevant relation is specified over a set of C++ operators: <=, ==, and !=, which are defined as below:

```
bool operator <= (RealInterval const& r) const;
    // Return true if 'this' is less specific than 'r', i.e., 'r' is a
    // subset of 'this'.

bool operator == (RealInterval const& r) const;
    // Return true if 'this' and 'r' describe the same interval.

bool operator != (RealInterval const& r) const;
    // Return true if 'this' and 'r' describe different intervals.
```

A preprocessing class for RealInterval

The *LabelManager class* implements a preprocessor for "RealInterval" values by extending the "PreprocessorBase". Two classes are defined here for preprocessing the expressions of one Order Types called "RealIntervalSymValues" (for semantic values of preprocessing parser) and "RealIntervalPreprocessor" (for the parser itself).

"RealIntervalSymValues" defines a set of possible semantic values for symbols used in the "RealInterval" preprocessing parser. Semantic values point to read only objects, thus updates of the semantic values are restricted to the "RealIntervalPre-processor" class.

```
union RealIntervalSymValues
{
   struct {
   }  empty;
      // No resources.

   struct {
      char const* label;
   }          identifier;
      // Resources:
      // label a label resource registered with the "LabelManager"

   struct {
      double value;
   }          real;

   struct {
      bool closed;
      double value;
   }          endpoint;
      // Resources:
      // closed a boolean value, 'true' if 'value' represents a finite
      // endpoint and 'false' if 'value' represents the sign of an
      // endpoint at infinity; and a double value.

}; // union RealIntervalSymValues
```

The RealIntervalPreprocessor implements a concrete preprocessor class for "Real-Interval" values by extending the "PreprocessorBase". The main purpose of this class is to implement resource managements on semantic values specified in "Real-IntervalSymValues". The reason of doing this is that yacc parsers do not deconstruct discarded and reduced tokens. Instead, inactive tokens are simply over-written. This causes no problems for reduced tokens since their resources may be reclaimed in the rule's action. However, during error recovery, tokens are discarded and their resources are lost. Besides this resource management issues, this class acts as an integration between the parsers (specified in Yacc/Bison grammar), such as accept_interval() and accept_type_label() which are specified in this class and then redirected to its domain objects.

```
bool accept_interval(
      char const* ident,
      bool has_lb, double lb, bool has_ub, double ub
```

```
);
    // Assign the interval to 'ident'. Consume the arguments. Return
    // true if the arguments are well formed, otherwise return
    // false and collect an error report.

bool accept_type_label(char const* ident, char const* label);
    // Assign the "RealInterval" type label to 'ident'. Return
    // true if the arguments are well formed, otherwise return
    // false and collect an error report.
```

The last part is to specify a lexer for recognising the tokens for the parser which is specified in the yacc/bison grammar and also is needed to be able to access internal data.

```
friend int
real_interval_lex(RealIntervalSymValues* lvalp, void* context);
    // Retrieve the next token recognised in the preprocessor directive
    // syntax.
```

A preprocessing parser for RealInterval

The parser is specified in Bison/Yacc grammar for the preprocessor handling of real interval values. For each resource assigned to the parser (such as a string recognised in the lexical analyser), an appropriate method must be called in "TypesParse" in order to create a managed resource. For each production, consuming a resource carrying tokens, another method must be called in "TypesParse" in order to recover the resource. This is carried out according to the action taken from the resource: if the resource is transferred to the reduced token, it is not relinquished. However, if the resource is not transferred, it is relinquished. An example of grammar is described below[1]:

```
range_def:
    IDENTIFIER '=' '[' low_expr ',' high_expr ']'
      {
        RealIntervalPreprocessor* preprocessor
          = (RealIntervalPreprocessor*) context;

        if (
          preprocessor->accept_interval(
            $1.label,
            $4.closed, $4.value,
            $6.closed, $6.value
          )
        ) {
```

1. The full version of this yacc/bison grammar is documented in Appendix B.1 on page 245.

```
        // Accept the parse.
        YYACCEPT;
      }
      else {
        // Abort the parse.
        YYABORT;
      }
    }

  | IDENTIFIER '=' IDENTIFIER
    {
      RealIntervalPreprocessor* preprocessor
        = (RealIntervalPreprocessor*) context;

      // Record the type label in the symbol table.
      if (
        preprocessor->accept_type_label(
          $1.label, $3.label
        )
      ) {
        // Accept the parse.
        YYACCEPT;
      }
      else {
        // Abort the parse.
        YYABORT;
      }
    }

  | error
    {
      RealIntervalPreprocessor* preprocessor
      = (RealIntervalPreprocessor*) context;

      // Reject the parse.
      preprocessor->reject_parse();

      // Abort the parse.
      YYABORT;
    }
  ;
```

OrderTypes/types_with_ real_intervals.hh

This class implements a concrete class providing a proxy (or smart reference) to an inheritance hierarchy of feature structure types. This proxy augments the subject inheritance hierarchy with the informational order of real intervals. The "TypeHierarchy" methods "identifier()", "join()", and "meet()" act to pick out real interval types. It is accompanied by a type resource management scheme that holds reference counts on real interval types. In addition, the method "is_label()" syntactically identifies labels of real interval types so that the "identifier()" precondition is satisfied by all syntactically correct labels. The collection of real interval types, as

described in Chapter 6.3.10, forms a BCPO. Since meet is defined as union for all but the empty set of intervals. This structure is inserted above the universal type and is topped by the absurd type. Therefore, the empty interval is represented by the absurd type.

```
class TypesWithRealIntervals
   : public ManualTypes
{
public:

   TypesWithRealIntervals(ManualTypes* real_types);
      // Construct a proxy whose real subject is the type hierarchy
      // 'real_types', and which augments the inheritance hierarchy in
      // 'real_types' by the real intervals. Take sole ownership of
      // the real subject.

...
   virtual ~TypesWithRealIntervals();
   bool pre_D_TypesWithRealIntervals() const
      {
         return InheritanceHierarchy::pre_D_InheritanceHierarchy();
      }
      // Free this proxy and the real subject. Free the
      // representations of all interval types.
```

"TypesWithRealIntervals" passes these methods to the domain subject (RealInterval). It intervenes to test for real interval type labels and identifiers, as described in the "TypeHierarchy" application programming interface.

```
virtual bool is_type(Index t) const;
virtual bool are_types(IndexSet const& T) const;
virtual bool is_regular_type(Index t) const;
virtual bool are_regular_types(IndexSet const& T) const;
virtual bool is_universal_type(Index t) const;
virtual bool is_absurd_type(Index t) const;
virtual bool is_label(char const* l) const;
virtual Index universal() const;
virtual Index absurd() const;
virtual Index identifier(char const* l) const;
virtual char const* label(Index t) const;
virtual bool subsumes(Index s, Index t) const;
virtual bool comparable(Index s, Index t) const;
virtual Index join(Index s, Index t) const;
virtual Index meet(Index s, Index t) const;
virtual Index accept_type(Index t) const;
virtual void relinquish_type(Index& t) const;
```

Thus the "InheritanceHierarchy" Interface for this Order Type is defined as below, which also delegates to the real subject:

```
virtual bool is_succession_type(Index t) const;
virtual bool are_succession_types(IndexSet const& T) const;
virtual bool is_leaf_succession_type(Index t) const;
virtual bool are_sound_parents(IndexSet const& T) const;
virtual IndexSet* succession_types() const;
virtual IndexSet* maximal_succession_types() const;
virtual IndexSet* children(Index t) const;
virtual IndexSet* parents(Index t) const;
virtual Index insert(char const* l, IndexSet const& S);
virtual void remove(Index t);
```

7.3.2 *GeomAscend*

The second example is chosen for the purpose of demonstrating how an external data structure such as a geometric modeller can be integrated in OGS. In addition, this class demonstrates an implementation of a geometric object within the framework as described in Chapter 6.4.1.

The domain objects[1]

Domain objects in Order Type GeomAscend include two classes: the OGS object (class *GeomAscend*) and the geometric object (class *Region*). GeomAscend is similar to RealInterval described in the previous section. On the other hand, Region implements non-manifold geometric objects as described in Chapter 6.4.1 on top of the SHAPES geometric kernel.

As an interface to the OGS kernel, GeomAscend represents subset relation on the point-set geometry (*P*). The empty pointset (bottom) is specified via '*-p*'. Other geometric objects are specified via their unique *geom_names*, for example

- abox is a point set
- -p is the empty set.

GeomAscend values are picked out by mapping a collection of strings corresponding to the type labels for these objects to a resource manager called *GeomNameManager*. An informational ordering is supplied through the "<=" operator that interprets the subset relation as constraints over the non-manifold solids. An ascend 'r' subsumes another 's' iff the pointset 's' contains pointset 'r'. "GeomAscend" values are also manipulated by an additional constructor which computes the join or

1. The full detail is documented in Appendix B.2 on page 259.

meet according to the given informational ordering. The constructors are defined as:

```
GeomAscend();
    // Construct the empty pointset.

  GeomAscend(GeomAscend const& r);
    // Construct a copy of the geom_ascend 'r'.

GeomAscend(
    Op op, GeomAscend const& r, GeomAscend const& s
  );
    // If 'op == geom_ascend_join', then construct the join of 'r'
    // and 's' in the informational order "<=". Namely, the union
    // of pointset 'r' and 's'.

    // If 'op == geom_ascend_meet', then construct the meet in
    // the informational order "<=". Namely, the intersection of
    // pointset of 'r' and 's'.

GeomAscend(char const* l);
    // Construct the interval described by the string 'l', where
    // 'l' is of the form described by
    // "GeomAscend::is_type_label()".
    // If l contains the geomname registered with GeomNameManager,
    // then, l is a type_label,
    // otherwise, No.

GeomAscend( bool is_empty_set, Region const* g);
    // Construct the ascend described by the bool is_empty_set
    // and a char* g (for geom_name)
```

The interface methods (is_type_label, parse_type_label, and their associated methods) with the central resource management: *LabelManager* are defined below:

```
static bool is_type_label(char const* l);
    // Return true if 'l' is a non-NULL string, and matches
    // "GeomAscend::format". Namely, characters 'G' replaced by
    // either:
    //  + the string '-p', representing the empty geom set
    //  + a string matching regular expression
    //    '([a-z][0-9a-z_+-]*)', which is a machine generated
    //    geometry name of a finite bound


static bool is_ascend(
    bool is_empty_set, Region const* geom
  );
    // Return true if 'is_empty_set' and 'gname' describe a
    // well-formed ascend. Namely one of:
    //  + 'is_empty_set' is 'false' and 'gname' is registered with
    //    GeomNameManager;
    //  + 'is_empty_set' is 'true', so that the ascend is an empty
    //    pointset.
```

```
static bool parse_type_label(
    char const* l,
    bool& is_empty_set, char *& g
);
    // Parse the type label 'l'. Record the parsed geom number in
    // 'is_empty_set' 'r'. Return true if 'l' matches
    // "GeomAscend::format" as for "GeomAscend::is_type_label()".
```

The domain relevant relation is specified over a set of C++ operators: <=, == and !=, which are defined below:

```
bool operator <= (GeomAscend const& r) const;
    // Return true if 'this' is less specific than 'r', i.e., the
    // boundary of 'this' is within the boundary of 'r'.

bool operator == (GeomAscend const& r) const;
    // Return true if 'this' and 'r' describe the same ascend geom.

bool operator != (GeomAscend const& r) const;
    // Return true if 'this' and 'r' describe different ascend geoms.
```

Besides GeomAscend we have the geometric representation of OGS: *Region*. Region represents a non-manifold geometric object which provides persistent geometric boundary information. A Region is either an empty_pset/universal_pset or an non-empty_pset. All "Region" objects are constructed on the heap, thus constructors have to be protected so that only the methods of this class can use them. For this purpose, static methods, such as "region_union()" and "region_difference()", are provided to construct objects on the heap.

```
...
protected:
    Region():_xg(NULL){ }
    // create a Region class representing an empty pointset.

    Region::Region(xCELL cell);
    // a protected constructor: ensure the input is a xCELL--the end
    // node of a group graph.

    Region::Region(xGROUP grp);
    // a protected constructor: ensure the input is a xGROUP.
    // the xGROUP is might be three cases:
    // 1. a feature group which contains the xCELLs.
    // 2. a feature group which has no region_attrib
    // 3. a plain group which contains a set of feature groups.
    // 4. a null group, which is an error.

public:
    static Region* geom_region(xGEOM geom);
    // this function takes a geometry object and return a Region
    // object contains that geometry object.
```

```
static Region* union_region(Region const* a, Region const* b);
// Union:
// this function takes two Regions (scra, scrb) and return a
// Region object contains the unioned geometric object, if success.
// otherwise, the result is a NULL pointer.

static Region* intersection_region(Region const*, Region const*);
// Intersection:
// this function takes two Regions (scra, scrb) and return a
// Region object contains the intersected geometric object,
// if success. otherwise, the result is a NULL pointer.

static Region* difference_region(Region const*, Region const*);
// difference:
// this function takes two Regions (scra, scrb) and return a
// Region object contains the geometric object= a - b,
// if success. otherwise, the result is a NULL pointer.
```

The transformation of geometric objects is implemented as public methods to allow the internal location to be changed when the transformation occurs. There is the need for persistency for geometric information, since once one region is merged into another region, transformation over this region will affect the region it is merged to.

```
bool translate(double x, double y, double z);
// this function translate this object(_xg) via a vector(x, y, z)
//   return true if success, otherwise false.

bool scale(double factor, double px, double py, double pz);
// this function scale this object(_xg) with a factor, and
// at a base point at (px, py, pz)
//   return true if success, otherwise false.

bool rotateX(double radians, double px, double py, double pz);
// this function rotate this object(_xg) around X-axis with
// a radians, and a basis point at (px, py, pz)
//   return true if success, otherwise false.

bool rotateY(double radians, double px, double py, double pz);
// this function rotate this object(_xg) around Y-axis with
// a radians, and a basis point at (px, py, pz)
//   return true if success, otherwise false.

bool rotateZ(double radians, double px, double py, double pz);
// this function rotate this object(_xg) around Z-axis with
// a radians, and a basis point at (px, py, pz)
//   return true if success, otherwise false.
```

A preprocessing class for GeomAscend[1]

This concrete preprocessing class implements a preprocessor for "GeomAscend" values by extending "PreprocessorBase". Two classes are defined here for preprocessing the expression of this Order Type: "GeomAscendSymValues" (for semantic values of preprocessing parser) and "GeomAscendPreprocessor" (for the parser itself).

GeomAscendSymValues defines the set of possible semantic values for symbols in the "GeomAscend" preprocessor. Semantic values point to read only objects. Semantic value updates are restricted to the "GeomAscendPreprocessor" class.

```
union GeomAscendSymValues
{
   struct {
   }empty;
      // No resources.

   struct {
      char const* label;
   }identifier;
      // Resources:
      //  labela label resource registered with the "LabelManager"
      //        singleton.

   struct {
      double value;
   } real;
      // Resources:
      //  valuea non-NULL pointer to a Region *

   struct {
     bool empty_pset;
     Region const* geom;
   } endpoint;
      // Resources:
      //  closed a boolean value, 'true' if 'value' represents a finite
      //        endpoint and 'false' if 'value' represents the sign of
      //        an endpoint at infinity; and
      //  valuea non-NULL pointer to a GNU variable length double.


}; // union GeomAscendSymValues
```

Another class (GeomAscendPreprocessor) is for the parser described in the next section. GeomAscendPreprocessor is a concrete class implementing a preprocessor for "GeomAscend" values by extending "PreprocessorBase".

1. The full detail is documented in Appendix B.2 on page 259.

```
class GeomAscendPreprocessor
   : public PreprocessorBase
{
public:

   GeomAscendPreprocessor();
      // Construct an unattached preprocessor.

   virtual ~GeomAscendPreprocessor();
   bool pre_D_GeomAscendPreprocessor() const
      {
         return LineSource::pre_D_LineSource();
      }
      // Deconstruct this preprocessor.

...
protected:

      // "PreprocessorBase" Protected Interface.
      //
      // The protected interface provides the basis for a concrete
      // derivation.
   virtual bool parse();
```

Yacc parsers do not deconstruct discarded and reduced tokens. Instead, inactive tokens are simply over-written. This causes no problems for reduced tokens since their resources may be reclaimed in the rule's action. However, during error recovery, tokens are discarded and their resources are lost. Each "GeomAscendPreprocessor" instance holds pointers to resources held by unconsumed tokens. Each resource is registered when it is constructed and deregistered when it is consumed by a rule's action. A resource is 'outstanding' while it remains registered. Resources outstanding at the destruction of the "GeomAscendPreprocessor" instance are deallocated.

A preprocessing parser for GeomAscend

The parser is specified in Bison/Yacc grammar for the preprocessor handling of real interval values. For each resource assigned to the parser (such as a string recognised in the lexical analyser), an appropriate method must be called in "TypesParse" to create a managed resource. For each production consuming a resource carrying token, another method must be called in "TypesParse" to recover the resource. This is carried out according to the action taken on the resource: if the resource is transferred to the reduced token, it is not relinquished, but if the resource is not transferred, it is relinquished. The details on managing parser resources are documented in "*geom_ascend_preprocessor.hh*" (Appendix B.2 on page 259). The tokens are defined as

```
%token    <identifier>   IDENTIFIER
%token    <identifier>   GEOMNAME
%token    <real>         REAL
%token    <empty>        BOX
%token    <empty>        EMPTY_SET
%token    <empty>        UNION
%token    <empty>        DIFF
%token    <empty>        INTERSECT
%type     <empty>        ascend_geom_def
%type     <endpoint>     geom_expr
%type     <endpoint>     geom_paren_expr
%type     <endpoint>     prim_geom_expr
%type     <real>         sum_expr
%type     <real>         prod_expr
%type     <real>         paren_expr
```

An example of partial grammar is described below[1]:

```
geom_expr:
    geom_paren_expr
      {
        $$.geom = $1.geom;
        $$.empty_pset = $1.empty_set;
      }
    | geom_paren_expr UNION geom_expr
      {
        if($1.empty_set) {
        $$.geom = $3.geom;
        $$.empty_pset = $3.empty_set;
        }
        else if($3.empty_set) {
        $$.geom = $1.geom;
        $$.empty_pset = $1.empty_set;
        }
        else {
        $$.geom = $1.geom->_union($3.geom);
        $$.empty_pset = false;
        }
      }
    | geom_paren_expr DIFF geom_expr
      {
        GeomAscendPreprocessor* preprocessor
        = (GeomAscendPreprocessor*) context;

        if($3.empty_set) {
        $$.geom = $1.geom;
        $$.empty_pset = $1.empty_set;
        }
        else if (! $1.empty_set) {
        $$.geom = $1.geom->_diff($3.geom);
        $$.empty_pset = false;
        }
        else {
        // error
```

1. The full version of this yacc/bison grammar is documented in Chapter B.2.

```
        preprocessor->collector()->append_count(
            GeomAscendPreprocessor::err_eset_diff_non_eset,
            strlen(GeomAscendPreprocessor::err_eset_diff_non_eset)
            );

        // Abort the parse.
        YYABORT;

        }
      }
    | geom_paren_expr INTERSECT geom_expr
      {
        if($1.empty_set) {
        $$.geom = $1.geom;
        $$.empty_pset = $1.empty_set;
        }
        else if($3.empty_set) {
        $$.geom = $3.geom;
        $$.empty_pset = $3.empty_set;
        }
        else {
        $$.geom = $1.geom->_intersect($3.geom);
        $$.empty_pset = false;
        }
      }
    ;
```

OrderTypes/types_with geom_ascends.hh

Another concrete class *TypeWithGeomAscends* provides a proxy (or smart reference) to an inheritance hierarchy of feature structure types. This proxy augments the subject inheritance hierarchy with the informational order of real ascends. The "TypeHierarchy" methods, "identifier()", "join()" and "meet()" act to pick out real ascend types. This role is discussed in "FeatureStructures/type_hierarchy.hh". It is accompanied by a type resource management scheme that holds reference counts on real ascend types. In addition, the method *is_label()* syntactically identifies labels of real ascend types, so that the *identifier()* precondition is satisfied by all syntactically correct labels. The collection of real ascend types forms a BCPO, since meet is defined as being smaller for all but the empty set of ascends. This structure is inserted above the universal type and is also topped by the absurd type. Therefore, the infinite ascend is represented by the absurd type.

```
class TypesWithGeomAscends
  : public ManualTypes
{
public:

  TypesWithGeomAscends(ManualTypes* real_types);
    // Construct a proxy whose real subject is the type hierarchy
```

```
    // 'real_types', and which augments the inheritance hierarchy in
    // 'real_types' by the real ascends. Take sole ownership of the
    // real subject.

virtual ~TypesWithGeomAscends();
    // Free this proxy and the real subject. Free the
    // representations of all ascend types.
```

"TypesWithGeomAscends" passes these methods to the domain subject (GeomAscend). It intervenes to test for geom ascend type labels and identifiers, as described in the "TypeHierarchy" application programming interface. Thus, the methods are the same as TypesWithRealIntervals. This is also applied to the interface to "InheritanceHierarchy".

7.4 How to specify geometric types in OGS

There are two ways to specify a geometric type in OGS—1) by geometric expressions which then are parsed by the preprocessors OGS provides; and 2) by unification of existing type*s* finding their meet or join in an Order Type hierarchy.

Geometric expression examples

The syntax of expression of each Geometric Type depends on its own specific expression (as expressed in a yacc/bison grammar). With the examples of GeomAscend type, OGS provides a set of primitive operations over geometry such as geometry creation functions, the boolean operators and transformation. An example of a geometric expression is given as follows:

```
# GeomAscend Box = union(box (2.0, 3.0, 4.0), translate(box(3.0, 4.1, 5.0), 1, 3, 4))
```

With this description, OGS will generate a non-manifold cell-complex which is the result of the union of these two boxes. "Box" is an identifier for using within the preprocessor.The *GeomName* is generated from the system. An example of the type for naming this *GeomAscend* might be *geom_ascend_geom23.*

The primary geometric functions provided are box, point[1], line[2] and sphere. The Boolean operations are *diff*, *union*, *intersect*, for difference, union, and intersection operation. Three transformation operators are *translate, rotate* and *scale.*

1. By default, all the points are embedded as three dimensional objects.
2. By default, all the lines are embedded as three dimensional objects.

Unification of two instances of geometric Order Types

As requested by the description, OGS will redirect the join or meet of each pair of Order Types instances to the domain objects they are carrying. As in the example of implementation described before, the behaviour of subsumption ordering is embedded in the objects themselves. As a result, the unification of two geometric Order Types instances will generate a new geometric object as the join of these two types if that is subsumed by both Order Types.

Having described these two mechanisms, we now demonstrate them in the following chapters in two examples—a single fronted cottage and a building enclosure.

8
Examples

Two examples are described in this chapter: single fronted cottages and building enclosures. Through these examples, we demonstrate 1) representation for geometric objects; 2) creation of new geometric objects; 3) incremental refinement of geometry objects; 4) multiple models related via a cell-complex scheme.

8.1 Single Fronted Cottages

Background

The very first example used is a typical house type in Australia, known as a single fronted cottage. Single fronted cottages are simple houses which are composed of a single-loaded hallway, rooms off this hallway and a cluster of different spaces at the end of the hallway. Single fronted cottages and newly styles which have also been derived from single fronted cottages are recognised by some features: 1) the hallway and its adjacent rooms are under one single roof; 2)the cluster of spaces at the end of hallway is arranged as a skillion or as separate structures such as a kitchen and a toilet; and 3) there is a front porch which gives the picture of a single fronted cottage. The example of single fronted cottages is chosen for analysing the layout problem of spaces, especially when some spaces such as bedrooms, dining spaces, studies and lounges are also generic in function.

An inheritance hierarchy

By separating the function from the single fronted cottage space, we get two hierarchies: *sfc* and *house* (a complete inheritance hierarchy is shown in the appendix). A type called "*sfc_house*" inherits both *sfc* and *house*. Thus, a most general satisfier of *sfc_house* will generate all the possible layouts of the single fronted cottage in an

abstract geometric representation. Each geometry represents a space in function. Some geometry may be identical, such as for bedrooms, yet each geometry remains distinct. For example, in the following example, a feature with *massing_d* type contains 4 feats and each feat is typed to a geometrical type representing a distinct geometry[1]. The type definition is shown below.

```
brief
   inherit universal;

house
   inherit brief;

building
   inherit universal;

massing
   inherit universal;

massing_a
   inherit massing;

massing_t
   inherit massing ;

massing_b
   inherit massing_a;

massing_c
   inherit massing_b;

massing_d
   inherit massing_c;

sfc
   inherit massing_d & building;


sfc_house
   inherit sfc & house;
```

Spatial constraints of the types specified above are

```
sfc
   cons PORCH == MASS_EL1
      & HALL == MASS_EL2
      & ROOM_ROW == MASS_EL3
      & SKILLION == MASS_EL4;

house
      cons SLEEPING : X
```

1. The main reason for using the name "massing_d" rather than "massing_4" is dued to the limitation of current implementation.

```
& LOUNGE : (=/= X)
& KITCHEN : (=/= X)
& DINING : (=/= X)
& BATHROOM : (=/= X);
```

The *π-resolution* algorithm will incrementally generate alternative layouts. The abstract alternative layouts with the specification of each low-left coordinations and up-right coordinations then form a geometric expression to generate the geometric types in OGS.

8.1.1 Representation for geometric objects

The constraint resolution of Kryos is not available at the time of writing this thesis. Thus, constraints are specified and resolved in another Typed Feature Structures implementation (ALE), and then, are parsed as a valid OGS expression to generate the required geometry. This sfc description generates 32 alternative layouts along with the variation of geometry. The generation mechanism that Geometric Typed Feature Structures provides is not a major issue in this example. Instead, geometric representation and geometry generation are the main lessons to be learned from this simple layout example.

The variation of geometry is achieved by applying adjacency constraints such as the ALE macros: "inline" and "in" onto a fixed range of width and height. As a simple example, each geometry is assigned a low left and a up right coordinate that is bounded by a range of x-axis and y-axis values. For example, the width of a geometry representing the *sfc_house* is within the interval [1, 2], and the length is within [3, 7]. Using structure sharing, the constraints are applied to parametric geometry objects. Examples of these constraints are shown below:

```
inline1(Zone) macro
    @ share([Zone, geometry, ll], [Zone, mass_el1, geometry, ll]),
    @ share([Zone, geometry, ur, x], [Zone,mass_el1,geometry,ur, x]),
    @ share([Zone, geometry, ur, y], [Zone,mass_el1,geometry,ur, y]).

inline2(Zone) macro
    @ share([Zone, geometry, ll], [Zone, mass_el1, geometry, ll]),
    @ share([Zone, geometry, ur, x], [Zone,mass_el1,geometry,ur, x]),
    @ share([Zone, mass_el2, geometry, ll, y],
        [Zone, mass_el1, geometry, ur, y]),
    @ share([Zone, mass_el2, geometry, ll, x],
        [Zone, mass_el1, geometry, ll, x]),
    @ share([Zone, mass_el2, geometry, ur, x],
        [Zone, mass_el1, geometry, ur, x]),
    @ share([Zone, geometry, ur, y],
```

```
[Zone, mass_el2, geometry, ur, y]).
```

The constraints over sfc are a description of constraints over its feats.

```
sfc cons
    (
    @ share([porch],[mass_el1]),
    @ share([hall],[mass_el2]),
    @ share([room_row],[mass_el3]),
    @ share([skillion],[mass_el4]),

    @ share([skillion, geometry, ll, x], [geometry, ll, x]),
    @ share([skillion, geometry, ll, y], [geometry, ll, y]),
    @ share([skillion, geometry, ur, x], [geometry, ur, x]),
    @ share([hall, geometry, ll, x], [skillion, geometry, ll, x]),
    @ share([hall, geometry, ll, y], [skillion, geometry, ur, y]),
    @ share([room_row, geometry, ll, x], [hall, geometry, ur, x]),
    @ share([room_row, geometry, ll, y], [skillion,geometry,ur, y]),
    @ share([porch, geometry, ll, x], [hall, geometry, ll, x]),
    @ share([porch, geometry, ll, y], [hall, geometry, ur, y]),
    @ share([porch, geometry, ll, y], [room_row, geometry, ur, y]),
    @ share([porch, geometry, ur, x], [room_row, geometry, ur, x]),
    @ share([porch, geometry, ur, x], [geometry, ur, x]),
    @ share([porch, geometry, ur, y], [geometry, ur, y]),
    geometry:ll:x:a0,   geometry:ll:y:a0,
    @ bounds( geometry ),

    skillion:( @ bounds( geometry ) ),
    hall:( @ bounds( geometry )),
    room_row:( @ bounds( geometry )),
    porch: (@ bounds( geometry ))

    ).
```

8.1.2 Creation of new geometric objects

One of the geometric feature structure that satisfied the *sfc_house* description (generated from ALE) is shown in Appendix C.1 on page 283. The geometric part of the result is interpreted in Ordering Geometry System with the geometric Order Types expression as below:

```
# GeomAscend Geom1 = translate(box(1, 1, 3), 1, 3, 0)
# GeomAscend Geom2 = translate(box(1, 1, 3), 1, 2, 0)
# GeomAscend Geom3 = translate(box(1, 1, 3), 1, 1, 0)
# GeomAscend Geom4 = translate(box(2, 1, 3), 0, 3, 0)
# GeomAscend Geom5 = Geom1 & Geom2 & Geom3
# GeomAscend Geom6 = translate(box(1, 4, 3), 0, 1, 0)
# GeomAscend Geom7 = box( 2, 1, 3)
# GeomAscend Geom8 = Geom4 & Geom5 & Geom6 & Geom7
```

Thus, a geometric feature structure which satisfies this description is shown below:

```
sfc_house
BATHROOM [14] massing
      GEOMETRY geom_ascend_geom4
DINING [10] massing
     GEOMETRY geom_ascend_geom2
GEOMETRY geom_geom1
HALL [6] massing
   GEOMETRY geom_ascend_geom6
KITCHEN [13] massing
      GEOMETRY geom_ascend_geom4
LOUNGE [11] massing
     GEOMETRY geom_ascend_geom1
MASS_EL1 [15] massing
      GEOMETRY geom_ascend_geom7
MASS_EL2 [6]
MASS_EL3 [16] massing3
      GEOMETRY geom_ascend_geom5
      MASS_EL1 [18] massing
            GEOMETRY geom_ascend_geom3
      MASS_EL2 [10]
      MASS_EL3 [11]
MASS_EL4 [17] massing2
      GEOMETRY geom_ascend_geom4
      MASS_EL1 [13]
      MASS_EL2 [14]
PORCH [15]
ROOM_ROW [16]
SKILLION [17]
SLEEPING [18]
```

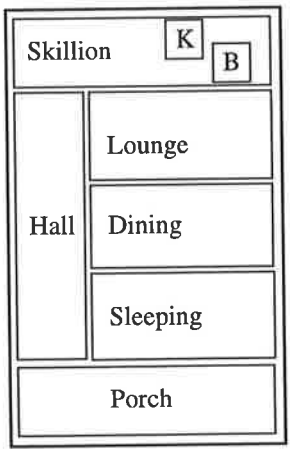


FIGURE 68. A 2D Display of the Schematic Layout Generated by OGS.

As indicated in the graph displayed above, the geometry for bathroom and kitchen is not instantiated as the numeric values are not yet specified. With the pointset inclusion relation, bathroom and kitchen are all within the boundary of skillion:

geom_ascend_geom4. An abstract version of this particular layout is shown in Figure 68. The 3D appearance of the alternative layout is shown in Figure 69.

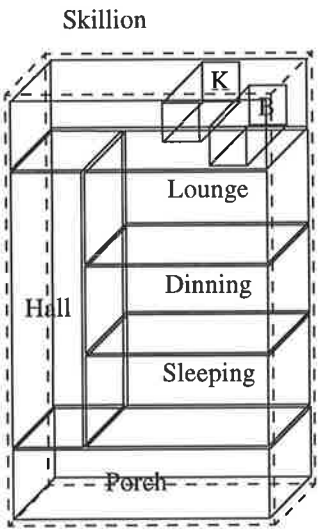


FIGURE 69. A 3D View of the Abstract Layout: Each Room is Represented as a 3D Region.

8.1.3 Multiple models related via a cell-complex scheme

The relation between two geometric objects is specified via two methods: Geometric type expression or by call-out by the unification mechanism. As for the geometric feature structure shown in the last section, the relations between geometric objects are expressed in a higher level. The actual cell complex of geom_ascend_geom8 looks like the abstract figure shown in Figure 70. All of the spatial elements of this *sfc_house* instance are instantiated as sub-cells of this Cell*sfc_house*, and thus cells are sharing their boundaries with adjacent cells. For example, the $Cell_{room_row}$ shares two edges and three points with $Cell_{lhall}$.

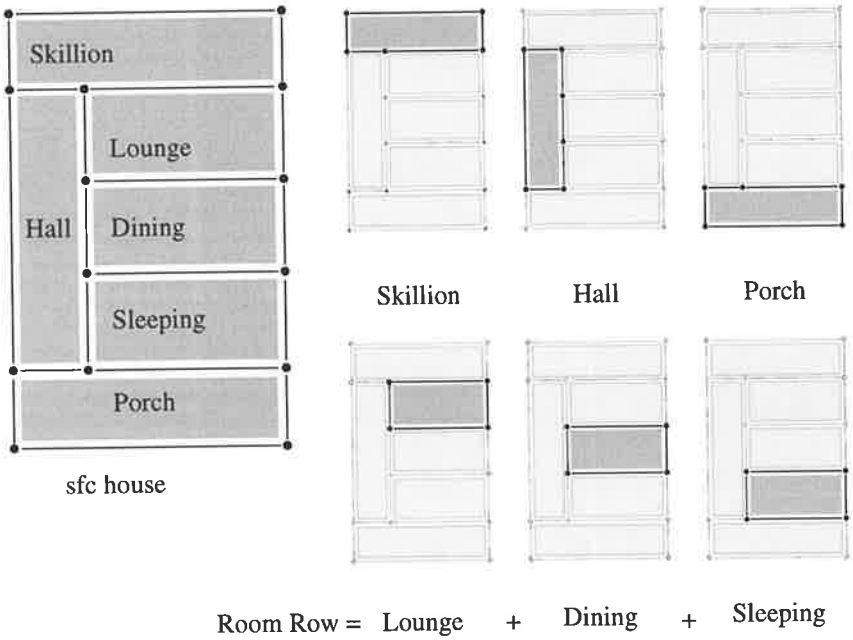


FIGURE 70. Each Room is a Sub-Cell of a Cell-Complex: $Cell_{sfc_house}$.

8.2 Examples—Building Enclosures

Background

The initial notion of building enclosures is derived from the project of SEED-Config in which an enclosure is considered as a physical system that separates two spaces. An example of a sound wall of an interior and exterior space separated by an insulated rainscreen. For all the building systems in SEED-Config, enclosures are created by applying technologies to a design state. Furthermore, these technologies allocate design units(DU) to functional units(FU), modify design units in place and, to a extend, may create functional units.

One special chosen type of building enclosure, with the developing technologies and research carried out primarily in countries like Canada, is insulated enclosures. Insulated enclosures, according to our understanding, relate their functions directly to the elements that comprise them and provide conditions so that these elements must intersect individually as well as a whole. Such research has been developed

and centred at the Division of Building Research(DBR) at the National Research Council of Canada. A series of publications [Latta, 1973, Brand, 1990] has made concrete and comprehensive contributions to the research work on enclosure systems.

The main purpose of choosing building enclosures is to demonstrate possible alternatives for the development of a functional unit(**FU**) hierarchy in Geometric Typed Feature Structures. In other words, to understand the **FU** of decomposition and to elaborate each element of the decomposition. In general, there are two alternatives, one is to place the hierarchy in the problem as early as possible in the design process; the other is to place the hierarchy via the application of technologies as the design process proceeds. Such information and knowledge should be developed individually. One technology called "insulated enclosure" with a high-level **FU** hierarchy is proposed. It contains a single-level of constituents and each constituent is identified as one region of an enclosure. Another proposed technology called "insulated_X enclosure"(where "X" is a wall, roof, ground or joint) seeks for allocated abstract enclosures and indicates them within the **FU** hierarchy appropriate to their types.

An inheritance hierarchy

For demonstration purposes, only wall enclosure technologies are discussed in this example. The full detailed description of building enclosure technologies in [Woodbury and Chang, 1995a] will not be elaborated upon in this section. In brief, a wall insulation contains four layers—continuous support, air-barrier, insulation and sun/rainscreen. Each layer is built on an existing layer in order: continuous support, air-barrier, insulation, and sun/rainscreen. The technology for building wall enclosure is simply to incrementally elaborate these four layers one by one.

A type definition for function units is described in OGS and is shown below:

```
function_unit
   inherit universal;

design_unit
   inherit universal;

torched
   inherit universal;

material
   inherit universal;
```

```
insulated_encl
   inherit universal;

real
   inherit universal ;

region
   inherit universal ;

fu_building_encl
   inherit function_unit ;

fu_layers
   inherit function_unit ;

fu_cap
   inherit function_unit ;

fu_air_space
   inherit function_unit ;

fu_drainage_holes
   inherit function_unit ;

fu_membrane
   inherit function_unit ;

fu_attachment
   inherit function_unit ;

fu_wall_encl
   inherit fu_roof_encl ;

fu_continuous_support_air_barrier_roof_gap
   inherit fu_layers ;

fu_sun_rainscreen_wall
   inherit fu_layers ;
```

As described before, a **FU** hierarchy is not equivalent to an inheritance hierarchy. The inheritance hierarchy instantiated from the description above is not related to its location in the **FU** hierarchy. The relations between **FU**s and their constituent units are via the paths. A satisfied feature structure is shown below:

```
MOST GENERAL SATISFIER OF: fu_wall_encl

fu_wall_encl
AIR_BARRIER fu_air_barrier
        ATTACHMENT fu_attachment
                DU du_attachment
                  FACE_OF region
                  GEOM region
                  THICKNESS real
                  TORCHED ballast
        DU du_material
```

```
        FACE_OF [0] region
        GEOM [0]
        MATERIAL [1] rubberised_asphalt
        THICKNESS [2] 1
      MEMBRANE fu_membrane
          DU du_membrane
            FACE_OF [3] region
            GEOM region
            MATERIAL [1]
            THICKNESS [2]
CONTINUOUS_SUPPORT fu_continuous_support
          ATTACHMENT fu_attachment
                DU du_attachment
                  FACE_OF region
                  GEOM region
                  THICKNESS real
                  TORCHED ballast
          DU design_unit
            FACE_OF [0]
            GEOM [0]
            THICKNESS real
          MEMBRANE fu_membrane
                DU du_membrane
                  FACE_OF [0]
                  GEOM region
                  MATERIAL metal_stud_wall
                  THICKNESS 2
DU du_wall_encl
  AREA real
  BOTTOM_CHORD [6] region
  FACE_OF [0]
  GEOM [0]
  THICKNESS real
  TOP_CHORD [5] region
INSULATION fu_layers
      ATTACHMENT fu_attachment
            DU du_attachment
              FACE_OF region
              GEOM region
              THICKNESS real
              TORCHED ballast
      DU design_unit
        FACE_OF region
        GEOM [3]
        THICKNESS real
      MEMBRANE fu_membrane
          DU du_membrane
            FACE_OF [4] region
            GEOM region
            MATERIAL fibreglass
            THICKNESS 2
SUN_RAINSCREEN fu_sun_rainscreen_wall
        AIR_SPACE fu_air_space
              DU design_unit
                FACE_OF region
                GEOM region
                THICKNESS real
        ATTACHMENT fu_attachment
              DU du_attachment
```

```
            FACE_OF region
            GEOM region
            THICKNESS real
            TORCHED ballast
DRAINAGE_HOLES fu_drainage_holes
            DU design_unit
              FACE_OF region
              GEOM region
              THICKNESS real
DU design_unit
  FACE_OF region
  GEOM [4]
  THICKNESS real
MEMBRANE fu_membrane
      DU du_membrane
        FACE_OF region
        GEOM region
        MATERIAL material
        THICKNESS real
[5] == [6]
```

As shown above, accessing $DU_{membrane}$ of $DU_{sun_rainscreen}$ is via its path: sun_rainscreen:membrane:du. Once again, ALE is used for its constraint resolution.

8.2.1 Representation for geometric objects

Half of the geometric objects represented in this technology are surfaces. For example, the geometry of $\mathbf{DU}_{we}$ is the abstract surface as shown in Figure 71, which shared by $\mathbf{DU}_{wcs}$(wall continuous support layer) and $\mathbf{DU}_{wab}$ (wall air-barrier layer). Another issue is that such a surface is often the outer surface of another geometric objects as shown in Figure 71. Thus, a "face_of" operator is needed for retrieving the sub-cells of an existing cell complex.

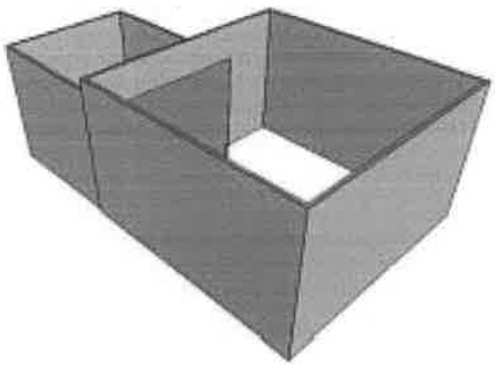


FIGURE 71. The Wall Continuous Support (Membrane Layer). Outfaces of this Wall from the Geometry of $\mathbf{DU}_{we}$.

8.2.2 Creation of new geometric objects

Creation of new geometric objects in this technology is via extrusion of an abstract surfaces. For example, the satisfied $\mathbf{FU}_{we}$ with a given geometry (wall enclosure) can be parsed in OGS as

```
# GeomAscend Geom1 = face_of( WallEncl )
# GeomAscend Geom2 = extrude( Geom1, -2 )
# GeomAscend Geom3 = extrude( Geom1, 1 )
# GeomAscend Geom4 = face_of( Geom3 )
# GeomAscend Geom5 = extrude (Geom4, 2 )
# GeomAscend Geom6 = face_of( Geom5 )
# GeomAscend Geom7 = extrude( Geom6, 2 )
# GeomAscend Geom8 = extrude( Geom7, 2 )

fu_wall_encl
AIR_BARRIER fu_air_barrier
        ATTACHMENT fu_attachment
                DU du_attachment
                  GEOM region
                  TORCHED ballast
        DU du_material
          GEOM [0]geom_ascend_geom1
          MATERIAL [1] rubberised_asphalt
        MEMBRANE fu_membrane
              DU du_membrane
                GEOM geom_ascend_geom3
                MATERIAL [1]
CONTINUOUS_SUPPORT fu_continuous_support
            ATTACHMENT fu_attachment
                    DU du_attachment
                      GEOM region
                      TORCHED ballast
            DU design_unit
              GEOM [0]
            MEMBRANE fu_membrane
                  DU du_membrane
                    GEOM geom_ascend_geom2
                    MATERIAL metal_stud_wall
DU du_wall_encl
  AREA real
  BOTTOM_CHORD [6] region
  GEOM [0]
  TOP_CHORD [5] region
INSULATION fu_layers
        ATTACHMENT fu_attachment
                DU du_attachment
                  GEOM region
                  TORCHED ballast
        DU design_unit
          GEOM geom_ascend_geom4
        MEMBRANE fu_membrane
              DU du_membrane
                GEOM geom_ascend_geom5
                MATERIAL fibreglass
SUN_RAINSCREEN fu_sun_rainscreen_wall
          AIR_SPACE fu_air_space
```

```
        DU design_unit
          GEOM geom_ascend_geom7
ATTACHMENT fu_attachment
        DU du_attachment
          GEOM region
          TORCHED ballast
DRAINAGE_HOLES fu_drainage_holes
          DU design_unit
            GEOM region
            THICKNESS real
DU design_unit
  GEOM [4] geom_ascend_geom6
  THICKNESS real
MEMBRANE fu_membrane
        DU du_membrane
          GEOM geom_ascend_geom8
          MATERIAL material
[5] == [6]
```

The 3D appearance of this instance is shown in Figure 72.

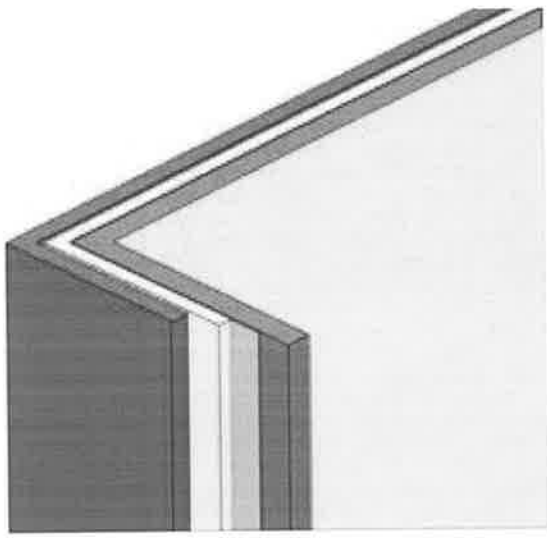


FIGURE 72. A 3D Display of Four Layers of Wall Enclosures: Sun/Rainscreen, Insulation, Air-Barrier, and Wall Continuous Support (from Left to Right).

8.2.3 multiple models related via cell-complex scheme

The relation between two geometric objects is specified via two methods: Geometric type expression or by the unification mechanism. As for the geometric feature structures shown in the last section, the relations between geometric objects are expressed in a higher level. The actual cell complex of geom_ascend_geom8 looks

like the abstract figure shown in Figure 70. All of the spatial elements of this *sfc_house* instance are instantiated as sub-cells of this Cells*sfc_house*, and thus cells are sharing their boundaries with adjacent cells. For example, the $Cell_{room_row}$ shares two edges and three points with $Cell_{lhall}$.

8.3 Summary

Understanding the knowledge domain of Typed Feature Structures, the crucial realisation of its domain extension known as Geometric Typed Feature Structures has been discussed and presented in terms of the knowledge level and its symbol level representation. The essential implementation of Geometric Typed Feature Structures—Ordering Geometry System (OGS) not only provides a unique framework for a design space explorer but also testifies the practicability of concepts of domain information (geometric information and other domain knowledge) carried by Order Types in Geometric Typed Feature Structures.

Geometric representation—Region which represents a non-manifold geometric object plays an important role in Ordering Geometry System because it provides persistent geometric boundary and efficient Boolean algorithms. The main lessons we have gained from geometric representation and geometry generation are from the two examples of single fronted cottages and of building enclosures.

By investigating these two examples, the evidence of feasibility for this implementation is revealed through the representation of geometric objects, creation of new geometric objects, incremental refinement of geometric objects and multiple models related via a cell-complex scheme. The layout problem of spaces in a single fronted cottage has been represented in the form of 3D geometric objects via the implementation we have proposed. The external data structure such as a non-manifold geometric modeller is also integrated sucessfully in the implementation as well.

However, there is still the limitation of this implementation can achieve for representing the geometric information due to the partial potential it holds. That leaves to the further development of this implementation for the future research work within the framework of domain information of the computational design research.

4
Conclusion

9
Conclusion

This dissertation presents a particular design of a design space explorer and the knowledge level concepts as well as its symbol level representation—Geometric Typed Feature Structures. The approach of this dissertation is that we synchronise the geometric information with other domain knowledge rather than separate them from non-geometric information. In this case, the theory of Typed Feature Structures is brought in as the basis of our symbol level representation. The key observation is that geometric information with certain domain relevant relations can be organised in inheritance hierarchies. Geometric Typed Feature Structures and its implementation—Ordering Geometry System shows the possibility and demonstrates how this particular set of types called Order Types can carry the continuous domain information such as geometric information.

One of the strengths of this approach is certain properties of knowledge behaviours such as partiality, intentionality and monotonicity can be brought in geometric representation schemes. In addition, certain domain relations over geometric information can be coherent with other domain knowledge. The second strength is the unification and subsumption provided by Typed Feature Structures which can provide meaningful behaviours for geometry. Another benefit from Geometric Typed Feature Structures is such mechanisms provide an efficient way of describing geometric information in domain.

In the realisation of search spaces, as a habit of a student, my research derivation path went from design space explorer knowledge levels to the hypothesis: a particular formal approach outside the computational domain might work for us. Due to the understanding of that approach, finally my research path looked into represent-

ing geometric information within this approach. In the end, in another derivation path lies "the Algebra of Shapes"[Stouffs, 1994], it remains to be seen and my current understanding these two to be unified.

I conclude this dissertation with a concise overview of contributions to and the prospect of future research work within the field of computational design.

10
Contributions and future work

10.1 Contributions

This dissertation offers a complete and detailed description of the Geometric Typed Feature Structures for geometric information and its implementation of Ordering Geometry System. The main contributions are:

Both geometric information and other domain knowledge can be uniformly represented in the symbol level representation of a particular design of design space explorers.

With such functionality, geometric information and domain knowledge can be treated uniformly within the framework, thus, the reasoning algorithm over geometric information is simplified and possible.

Geometric information with certain domain relations can be organised in inheritance hierarchies

With this contribution, the unification and subsumption algorithms of Typed Feature Structures can not only apply onto domain knowledge but also onto this specialised group of geometric representation schemes. This provides the fundamental structure of unification-based reasoning over geometric information.

Continuous geometric information and real numbers can be implemented using Geometric Typed Feature Structures

Continuous geometric information and real numbers are two main difficult problems for implementing a design space explorer. The implementation of Geometric Typed Feature Structures is a useful technique for representing such continuous domain since ordered relation over these continuous domain can be specified within the theory of Geometric Typed Feature Structures.

Point-sets paired with combination of subset/inclusion relations are analogue to the real intervals.

Following the previous contribution, subset/inclusion relations over sets is a logical solution for describing the relation of two point-sets. In addition, such relations paired with point-sets are similar to the real intervals. However, in the last part of geometric Order Types such as $IOPSet$, such Order Types provide an interesting direction of geometric representation schemes.

A working mechanism of geometric representation in the system level builds the foundations for other mechanisms of geometric representation in the HCI level.

An working exploration-based mechanism of geometric representation such as Geometric Typed Feature Structures provides an important role in implementing the HCI level of design space explorers. As described in the reviews of design space explorers, in order to provide a full-scaled design space explorer, such working mechanism is essential for being built-in the system levels, thus, this contribution is a big step forward for implementing a design space explorer described in the first part of this dissertation.

10.2 Prospect of future research work

The prospect of future research work lies upon the limitations and discoveries of the research work undertaken for this dissertation. I have defined a symbol level of representation—Geometric Typed Feature Structures as the domain extension of Typed Feature Structures. In order to develop this representation to its full potential for displaying geometric information, a geometric representation in the HCI level inevitably needs to be constructed. Several other further research directions such as geometric queries and the BCPO domain relations are also revealed in this section.

The great possibilities of extending Geometric Typed Feature Structures and creating an advanced Ordering Geometry System would require lots of research efforts. Further research would be necessary to step forwards and find possible solutions to the dilemmas posed by the future direction unfolded in this chapter in which all geometric information can be parsed within a complete design environment.

10.2.1 Domain relevant Geometric relations

The most crucial Order Type condition is the BCPO condition. In this dissertation, we discuss several geometric relations paired with continuous domain information

that provide the BCPO condition. The geometric relations such as adjacency and offset are used often in the design domain, but is hard to adopt in Geometric Typed Feature Structures. Finding a way to represent these relations in the framework of Geometric Typed Feature Structures is an interesting future research topic. Also, understanding the limitation of the BCPO condition to a more wide range of usual geometric relations is another research direction.

10.2.2 Geometric queries and expression

As stated in Part II, our geometric objects have no sub-structures. The sub-cells of our geometric objects are called out by a higher structure as shown in Figure 28. This provides a basic form of geometric queries such as *face_of* or *outer_faces_of* expression in Geometric Typed Feature Structures. Another example is the *pset()* function described in Chapter 6.4.6, using such polymorphic function, the information carried by a particular Order Type can be shared by a different Order Type. As hinted in the examples of Chapter 6, other geometric operations other than Boolean operations are possible and feasible within the framework of Geometric Typed Feature Structures. However, more elaborated mechanisms for geometric queries and expressions for each Order Type remain another unexplored research direction.

10.2.3 Incremental constraint resolution over Order Types

As argued earlier in this dissertation, incremental constraint resolution is an important step for realising an infinite *design spaces* partially and intentionally. However, such mechanism is not available at the time of writing this dissertation, Ordering Geometry System did not implement incremental constraint resolution over Order Types. The behaviours of this mechanism remains undiscovered.

10.2.4 Representation visualisation in HCI level

Geometric Typed Feature Structures demonstrates the representation and rules for a design space explorer. It provides a potential for implementing memory and policy at both the system level and the HCI level. In terms of representation visualisation, Geometric Typed Feature Structures carry the necessary geometric information but only give supports in the system level and only display 3D geometric objects using a simple viewing program. Other visualisation described in the reviews of Chapter 3 provides an interesting research topic for understanding the behaviours of these Order Types. Exploring the extension work of Genesis and SEED-config for inte-

grating with Geometric Typed Feature Structures offers a potential subject for future research.

Bibliography

Ait-kaci, H. (1984). *A lattice-theoretic approach to computation based on a calculus of partially ordered types*. PhD thesis, University of Pennsylvania.

Akin, O., Baykan, C., and Rao, R. (1983). Searching in the unix directory space. Submitted to Human-Computer Interaction, 1983.

Archea, J. (1987). Puzzle-making: What architects do when no one is looking. In Kalay, Y., editor, *Computability of Design*, Principles of Computer-Aided Design, chapter 2, pages 37–52. John-Wiley and Sons, New York, N.Y.

Asimov, M. (1962). *Introduction to Design*. Prentice-Hall, Englewood Cliffs, NJ.

Borning, A. (1981). Thinglab - an object-oriented system for building simulations using constraints. In *Fifth International Joint Conference on Artificial Intelligence*.

Brand, R. (1990). *Architectural Details for Insulated Buildings*. Van Nostrand Reinhold, New York.

Burrow, A. and Woodbury, R. (1999). Pi-resolution in design space exploration. submitted to CAADFutures 1999. http://www.arch.adelaide.edu.au/~rw/publications/in-progress.html.

Carlson, C. (1989). *The Explorer's Guide to discoverForm*. DiscoverWare Corporation, Pittsburgh, P.a., 1st edition.

Carlson, C. (1994). Design space description formalisms. In Gero, J. and Tyugu, E., editors, *In Proceedings of Formal Design Methods for CAD*, number 18 in B, pages 121–131, North-Holland. IFIP WG5.2, Elsevier Science Publishers B.V.

Carlson, C. and Woodbury, R. (1994). Hands-on exploration of recursive patterns. *Languages of Design*, 2:121–142.

Carlson, C. N. and Woodbury, R. F. (1991). Structure grammars and their application to design. In Brown, D., Waldron, M., and Yoshikawa, H., editors, *IntCAD'91 Preprints*, pages 103–126, Columbus, OH. An earlier version appeared in First International Workshop on Formal Methods in Engineering Design, Manufacturing, and Assembly, Colorado Springs, Co., January 1990.

Carpenter, B. (1992). *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.

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.

Chang, T.-W. and Woodbury, R. F. (1996). Sufficiency of the SEED knowledge-level representation for grammatical design. In Narasimhan, V. and Jain, L., editors, *Proceedings of 1996 Australian New Zealand Conference on Intelligent Information Systems*, pages 129–132, Adelaide, South Australia. Institute of Electrical and Electronics Engineers.

Chang, T.-W. and Woodbury, R. F. (1997). Efficient design spaces of non-manifold solids. In Yu-Tung Liu, J.-Y. Tsou, J.-H. H., editor, *CAADRIA 97*, volume 2, pages 335–344, Hsinchu, Taiwan. Computer Aided Architectural Design Research in Asia, National Chia Tung University.

Chase, S. C. (1989). Shapes and shape grammars: from mathematical model to computer implementation. *Environment and Planning B: Planning and Design*, 16(2):215–241.

Chien, S.-F., Donia, M., Snyder J. D., and Tsai, W.-J. (1998). SG-CLIPS: A System to Support the Automatic Generation of Designs from Grammars. *CAADRIA 98*, volume 3, pages 445–454, Osaka, Japan. Computer Aided Architectural Design Research in Asia.

Codd, E. F. (1970). A relational model of data for large shared data banks. *Communications of the association for Computing Machinery*, 13(6).

Cohon, J. (1978). *Multiobjective Programming and Planning*. Academic Press, New York.

Coyne, R. and Flemming, U. (1990). Planning in design synthesis: Abstraction-based loos. In Gero, J., editor, *Artificial Intelligence in Engineering V*, volume 1: Design, pages 91–111, New York. Springer.

Date, C. (1990). *An Introduction to Database Systems, Version 2*. Addison Wesley, Reading, Mass.

Davey, B. and Priestley, H. (1994). *Introduction to Lattices and Order*. Cambridge Mathematical Textbooks. Cambridge University Press, Cambridge, U.K.

Downing, F. and Flemming, U. (1981). The bungalows of Buffalo. *Environment and Planning B: Planning and Design,* 8:269–293

Durand, J.-N.-L. (1821). *Partie Graphique des cours d'architecture.* Ecole Polytechnique, Paris.

Durer, A. ([1528]1970). *Vier Bucher von menschlicher Proportion(Four Books of Human Proportion).* G.M. Wagner, London.

Eastman, C. (1976). General purpose building description systems. *Computer Aided Design*, 8(1):17–26.

Eastman, C. M. (1991). Use of data modeling in the conceptual structuring of design problems. In Schmitt, G. N., editor, *CAAD futures'91*, pages 207–224, Zurich, Switzerland. CAAD futures, Department of Architecture, Swiss Federal Institute of Technology.

Erich Gamma, Richard Helm, R. J. and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Publishing.

Fenves, S., Flemming, U., Hendrickson, C., Maher, M., Quadrel, R., Terk, M., and Woodbury, R. (1994). *Concurrent Computer-Integrated Building Design.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 07632.

Flemming, U. (1994). Case-based design in the SEED system. *Automation in Construction*, 3:123–133.

U. Flemming, U. (1987). More than the sum of parts: the grammar of Queen Anne houses. *Environment and Planning B: Planning and Design*, 14:323–350.

Flemming, U., Coyne, R., and Woodbury, R. (1993). SEED: A Software Environment to support the Early phases in building Design. In *IV International Symposium on Computer Aided Design in Architecture and Civil Engineering*, Barcelona. Institut de Technologia de la Construccio de Catalunya.

Flemming, U., Coyne, R., Woodbury, R., Bhavnani, S., Chien, S.-F., Chiou, S.-C., Choi, B., Kiliccote, H., Stouffs, R., Chang, T.-W., Han, S.-J., Jo, C., Shaw, J., and Suwa, K. (1994). SEED-Layout requirements analysis. http://seed.edrc.cmu.edu/SL/SL-start.book.html.

Flemming, U., Rychener, M. D., Coyne, R. F., and Glavin, T. J. (1986). A generative expert system for the design of building layouts: Version 1. Technical report, Center for Arts and Technology, Carnegie-Mellon University.

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.

Galle, P. (1986). Abstraction as a tool of automated floor-plan design. *Envi-*

ronment and Planning B: Planning and Design, 13(1):21–46.

Gero, J. (1994). Towards a model of exploration in computer-aided design. In Gero, J. and Tyugu, E., editors, *Formal Design Methods for CAD*, pages 315–336. Elsevier.

Gero, J. and Radford, A. (1984). The place of multicriteria optimization in design. In Langdon, R. and Purcell, P., editors, *Design Theory and Practice*, pages 81–85. Design Council, London.

Gero, J. S. and Sudweeks, F., editors (1998). *Artificial Intelligence in Design '98*, Dordrecht, the Netherlands. Kluwer Academic Publishers.

Gross, M., Ervin, S., Anderson, J., and Fleisher, A. (1987). Designing with constraints. In Kalay, Y. E., editor, *Principles of Computer-Aided Design: Computability of Design*, chapter 3, pages 53–88. John Wiley & Sons Publishing Company.

Gibbs, J. (1997). *Irving J Gill's Architectural Grammar.* Ph.D. Dissertation, Department of Architecture (Design and Computation), University of California, Los Angeles.

Gips, J. (1975). *Shape Grammar and their Uses: Artificial Perception, Shape Generation and Computer Aesthetics.* Birkhauser Verlag, Basel, Switzerland.

Gross, M. D. (1986). *Design as Exploring Constraints*. PhD thesis, Department of Architecture, Massachusetts Institute of Technology.

Gursoz, E., Choi, Y., and Prinz, F. (1990). Vertex-based representation of non-manifold boundaries. In Wozny, M. and (eds.), J. T., editors, *Geometric Modeling for Product Engineering*, New York. IFIP WG 5.2 / NSF, North-Holland.

Gursoz, E. L. (1991). *Reference Manuals for the libraries of NOODLES*. Engineering Design Research Center, Carnegie-Mellon University, Pittsburgh, PA, USA.

Harada, M. (1997). *Discrete/Continuous Design Exploration by Direct Manipulation*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA.

Harada, M., Witkin, A., and Baraff., D. (1995). Interactive physically-based manipulation of discrete/continuous models. In *SIGGRAPH '95 Conference Proceedings*, volume 29, pages 199–208. ACM Siggraph, ACM.

Heintze, N., Jaffar, J., Michaylov, S., Stuckey, P., and Yap, R. (1991). *The CLP(R) Programmer's manual, Version 1.1*. IBM.

Heisserman, J. A. (1991). *Generative geometric design and boundary solid grammars*. PhD thesis, Department of Architecture, Carnegie Mellon University.

Hoffmann, C. M. (1989). *Geometric & Solid Modeling: An Introduction*. Series in Computer Graphics And Geometric Modeling. Morgan Kaufmann Publishers, Inc., San Mateo, California 94403.

Kaplan, R. and Bresnan, J. (1982). Lexical-functional grammar: A formal system for grammatical representation. *The Mental Representation of Grammatical Relations*, pages 173–281.

Katz, R. H. (1990). Towards a unified framework for version modeling in engineering databases. *ACM Computing Surveys*, 22(4):375–408.

Koning, H. and Eizenberg, J. (1981). The language of the prairie: Frank Lloyd Wright's prairie houses. *Environment and Planning B: Planning and Design*, 8:295–323.

Krishnamurti, R. (1982). SGI: a Shape Grammar Interpreter. Research Report, Centre for Configurational Studies, The Open University, Milton Keynes, England.

Krishnamurti, R. (1992). The arithmetic of maximal planes. *Environment and Planning B: Planning and Design*, 19:431–288.

Krishnamurti, R. and Giraud, C. (1986). Towards a shape editor: the implementation of a shape generation system. *Environment and Planning B: Planning and Design*, 13(4):391–403.

Latta, J. (1973). *Walls, Windows and Roofs for the Canadian Climate*. Number 1 in Special Technical Publications. Division of Building Research, National Research Council of Canada, Ottawa, Ontario, Canada.

Mantyla, M. (1988). *An Introduction to Solid Modeling*. Computer Science Press, Rockville, Md.

March, L. and Earl, C. F. (1993). On counting architectural plans. *Environment and Planning B: Planning and Design*, 4(1):57–80.

Masuda, H. (1993). Topological operators and boolean operations for complex-based nonmanifold geometric models. *Computer Aided Design*, 25(2):119–129.

Mitchell, W., Steadman, J., and Liggett, R. (1976). Synthesis and optimization of small rectangular floor plans. *Environment and Planning B: Planning and Design*, 3:37–70.

Mitchell, W. J. (1977). *Computer-Aided Architectural Design*. Petrocelli/Charter, New-York.

Mitchell, W. J. (1990a). *The Logic of Architecture*. MIT Press, Cambridge, MA., USA.

Mitchell, W. J. (1990b). A new agenda for computer-aided deisgn. In McCullough, M., Mitchell, W., and Purcell, P., editors, *The Electronic Design Studio*, chapter Introduction, pages 1–17. MIT Press, Cambridge,

MA.

Mitchell, W. J. and McCullough, M. (1995a). Design database management. In McCullough, M. and Mitchell, W., editors, *Digital Design Media*, chapter 16, pages 353–382. Van Nostrand Reinhold, New York.

Mitchell, W. J. and McCullough, M. (1995b). *Digital Design Media*. Van Nostrand Reinhold, New York, 2nd edition.

Newell, A. (1982). The knowledge level. *Artificial Intelligence*, 18(1):87–127.

Newell, A. and Simon, H. (1963). A program that simulates human thought. In *Computers and Thought*, pages 279–296. McGraw-Hill, New York.

Newell, A. and Simon, H. (1972). *Human Problem Solving*. Prentice-Hall Inc., Englewood Cliffs, N.J.

Newell, A. and Simon, H. (1976). Computer science as empirical inquiry: symbol and search. *Association of Computing Machinery*, 19(3).

Pollard, C. and Sag, I. (1987). Information-based syntax and semantics: Volume i, fundamental. In *CSLI Lecture Notes*, volume 13. Center for the Study of Language and Information.

Radford, A. D. (1981). Optimization, simulation and multiple criteria in window design. *Computer-Aided Design*, 13(6):345–350.

Radford, A. D. and Gero, J. S. (1988). *Design by Optimization in Architecture, Building, and Construction*. Van Nostrand Reinhold, New York, N.Y.

Schon, D. (1983). *The Reflective Practitioner: How Professionals Think in Action*. Basic Books, New York.

Sedgewick, R. (1990). *Algorithms in C*. Addison-Wesley Publishing, New York.

Shieber, S. (1986). An introduction to unification-based approaches to grammar. In *CSLI Lecture Notes, Volume 4*. Center for the Study of Language and Information.

Simon, H. (1969). *The Sciences of the Artificial*. MIT Press, Cambridge, MA, USA.

Simon, H. (1971). Style in design. In *Proceedings of the 2nd Annual Design Research Association Conference*.

Simon, H. (1973). The structure of ill-structured problems. *Artificial Intelligence*, 4:181–201.

Simon, H. A. (1980). *The Sciences of the Artificial - Second Edition*. MIT Press, Cambridge, MA.

Smithers, T. (1992). Design as exploration: Puzzle-making and puzzle-solving. In *AID92, Workshop Notes, Exploration-based models of design and search-based models of design*, pages 1–21. Carnegie-Mellon University, Pittsburgh, PA.

Snodgrass, A. and Coyne, R. (1991). Models, metaphors and the hermeneutics of designing. Technical report, The University of Syndey, Sydney.

Steele, G. and Sussman, G. J. (1980). Constraints - a language for expressing almost-hierarchical descriptions. *Artificial Intelligence*, 14:1–39.

Sterling, L. and Shapiro, E. (1986). *The Art of Prolog*. MIT Press, Cambridge, Masschusetts.

STI (1995). *ACIS Documentation*. Spatial Technology, Inc., Boulder, CO, USA.

Stiny, G. (1980). Introduction to shape and shape grammars. *Environment and Planning B: Planning and Design*, 7(3):343–352.

Stiny, G. (1992). Weights. *Environment and Planning B: Planning and Design*, 19:413–430.

Stiny, G. and Mitchell, W. J. (1978). The Palladian Grammar. *Environment and Planning B: Planning and Design*, 5:5–18.

Stiny, G. and Mitchell, W. J. (1980). The grammar of paradise: on the generation of Mughul gardens. *Environment and Planning B: Planning and Design*, 7:209–226.

Stouffs, R. (1994). *The Algebra of Shapes*. PhD thesis, Carnegie-Mellon University.

Sutherland, I. (1963). Sketchpad: A man-machine graphical communication system. Technical Report 296, MIT Lincoln Lab.

Sutherland, I. (1975). Structure in drawings and the hidden-surface problem. *Reflections on Computer Aids to Design and Architecture*.

Tapia, M. (1996). *From Shape to Style, Shape Grammars: Issues in representation and computation*. PhD thesis, Department of Computer Science, University of Toronto, Toronto, Canada.

Tapia, M. (1999). A Visual implementation of a shape grammar system. *Environment and Planning B: Planning and Design*, 26:59–73.

Weiler, K. J. (1986). *Topological Structures for Geometric Modeling*. PhD thesis, Computer and Systems Engineering, Rensselaer Polytechnic Institute.

Wolfram, S. (1991). *Mathematica: A System for Doing Mathmatics by Computer*. Addison Wesley, Reading, Mass., 2nd edition.

Woodbury, R. (1984). Issues and approaches. *ACADIA Newsletter*, 3(1):1.

Woodbury, R. (1991a). 'Realities' of design. In Woodbury, G., editor, *The Technology of Design*, pages 155–168, G.P.O. Box 498, Adelaide, SA 5001, Australia. ANZAScA, University of Adelaide. A later version of this paper appeared in ACADIA91.

Woodbury, R. (1991b). Searching for designs: Paradigm and practice. *Building and Environment*, 26(1):61–73. An earlier version of this paper was published as EDRC 48-16-90.

Woodbury, R. (1992). What's in a part? In Dana Vanier, C. D. and Thomas, R., editors, *Joint CIB Workshops on Computers and Information in Construction*, pages 502–509, Montreal, PQ, Canada. CIB.

Woodbury, R. (1996). Exploring large spaces of building designs. ARC (Australia Research Concil) large grant proposal.

Woodbury, R. and Chang, T.-W. (1995a). Enclosures in SEED-Config. http://www-seed.arch.adelaide.edu.au/docs/enclosures/Enclosure-rep.html.

Woodbury, R., Fenves, S., Flemming, U., Coyne, R., Chang, T.-W., Chiou, S.-C., and Gomez, N. (1994). SEED-Config requirements analysis. http://seed.edrc.cmu.edu/SC/requirements-new/SC-req-new.book.html.

Woodbury, R. F. (1985). Computer techniques for designers: Strategies. In *4th Canadian Building Congress*, pages 108–113, Ottawa, Ontario. National Research Council.

Woodbury, R. F. (1987). Strategies for interactive design systems. In Kalay, Y., editor, *The Computability of Design*, volume 2 of *Principles of Computer-Aided Design*, pages 11–36. John Wiley and Sons. An earlier version of this paper was published as EDRC 48-02-87.

Woodbury, R. F. and Burrow, A. L. (1998). Computer-based support for conceptual design. In Srivastava, N., editor, *Structural Engineering Worldwide 1998*, page 332. Elsevier Science Ltd. Paper T159-3 on proceedings CD-ROM.

Woodbury, R. F., Burrow, A. L., Datta, S., and Chang, T.-W. (1999). Typed feature structures in design space exploration. accepted in AIEDAM Special Issue on Generative Systems in Design, manuscript available at http://www.arch.adelaide.edu.au/ rw/publications/progress.html.

Woodbury, R. F. and Chang, T.-W. (1995b). Building enclosures using SEED-Config. In Tan, M. and Teh, R., editors, *The Global Design Studio: Proceedings of the Sixth International Conference on Computer-Aided Architectural Design Futures*, pages 49–58, Singapore. CAADFutures 95 Conference.

Woodbury, R. F. and Oppenheim, I. J. (1988a). Geometric reasoning: Concepts and demonstration. In Gero, J., editor, *Artificial Intelligence in*

Engineering: Robotics and Processes, pages 61–76. Third International Conference on Applications of Artificial Intelligence to Engineering, Computational Mechanics Inc.

Woodbury, R. F. and Oppenheim, I. J. (1988b). Geometric reasoning: Motivation and demonstration. Technical Report EDRC 48-07-88, Engineering Design Research Center, Carnegie-Mellon University. A later version of this paper appeared in AIENG 88.

Woodbury, R. F., Radford, A. D., Taplin, P. N., and Coppins, S. A. (1992). Tartan worlds: A generative symbol grammar system. In Noble, D. and Kensek, K., editors, *ACADIA 92*, pages 211–220, Charleston, SC.

XOX (1995). *Reference Manual for SHAPES kernel*. XOX corporation, St. Paul, MN, USA.

Appendix

A
The syntax of Kryos descriptions

A.1 syntax of *Feature Structure*

```
//          terminated_types -> types EOF
//
//          types -> /* empty */
//
//          types -> types terminated_inheritance_def
//
//          terminated_inheritance_def -> inheritance_def ';'
//
//          terminated_inheritance_def -> ';'
//
//          inheritance_def -> TYPE 'inherit' inheritance_desc
//
//          inheritance_desc -> type_conjunction
//
//          type_conjunction -> TYPE
//
//          type_conjunction -> type_conjunction '&' TYPE
//
//    - Lexical Rules:
//
//          TYPE -> '[a-z][a-z_]*'
//
//          FEATURE -> '[A-Z][A-Z_]*'
//
//    - Source Tracking Directives:
//
//          SYNC_LINE -> '^# +[0-9]+ +"[^"]*".*'
//
```

A.2 The syntax for Specification

```
//          terminated_feats -> feats EOF
//
//          feats -> /* empty */
//
//          feats -> feats terminated_approp_def
//
//          terminated_approp_def -> approp_def ';'
//
//          terminated_approp_def -> ';'
//
//          approp_def -> TYPE 'intro' approp_desc
//
//          approp_desc -> FEATURE ':' TYPE
//
//          approp_desc -> approp_desc AND FEATURE ':' TYPE
//
//    - Lexical Rules:
//
//          TYPE -> [a-z][a-z0-9_+-]*
//
```

```
//          FEATURE -> [A-Z][A-Z0-9_+-]*
//
//    - Source Tracking Directives:
//
//          SYNC_LINE -> '^# [0-9]+ "[^"]*"( [1234])+$'
//
```

B

Examples of Order Type Implementation

B.1 RealInterval

B.1.1 RealInterval.hh

```
// ---------------------------------------------------------------------------
//   Order Types                                                Teng-Wen Chang
// ---------------------------------------------------------------------------
//
// OrderTypes/real_interval.hh
//   - Representation of intervals on the real number line (R).
//


#ifndef OG_ORDERTYPES_REAL_INTERVAL_HH
#define OG_ORDERTYPES_REAL_INTERVAL_HH


// Included Header Files.


#include "TFSShells/preconditions.hh"


// Declaration of classes to be defined.

class RealInterval;// Intervals on the real number line.


// ---------------------------------------------------------------------------
//   class RealInterval
//
//   Representation of intervals on the real number line (R).  Closed
//   intervals are specified as a pair '[lb, ub]' of lower and upper bounds.
//   Open and half-open intervals are specified via bounds at infinity.  For
//   example,
//
//       [0, 2.0]is the set { x in R | 0 <= x <= 2.0 },
//       [-i, 8]is the set { x in R | x <= 8 },
//       [ 10, +i]is the set { x in R | 10 <= x }, and
//       [ -i, +i]is the set R.
//
//   "RealInterval" values are picked out by a collection of strings
//   corresponding to the type labels for these objects.  An informational
//   ordering is supplied through the "<=" operator that interprets the
//   intervals as constraints ordered by specificity.  An interval 'r'
//   subsumes another 's' iff 's' is a subset of 'r'.  "RealInterval" values
//   are also manipulated by an additional constructor which computes the join
//   or meet according to the given informational ordering.
//
//   Note that the empty interval is not represented.  This role is already
```

```
//   catered for by the absurd type.
// ---------------------------------------------------------------------------

class RealInterval
{
public:

    // The following nested typedef distinguishes between constructors.

    typedef enum {
     real_interval_join, real_interval_meet
    } Op;


    // "RealInterval" Public Interface.

    RealInterval();
    bool pre_RealInterval() const
     {
         return true;
     }
     // Construct the open interval containing all reals.

    RealInterval(RealInterval const& r);
    bool pre_RealInterval(RealInterval const& r) const
     {
         return true;
     }
     // Construct a copy of the interval 'r'.

    RealInterval(
     Op op, RealInterval const& r, RealInterval const& s
    );
    bool pre_RealInterval(
     Op op, RealInterval const& r, RealInterval const& s
    ) const
     {
         return op == real_interval_join && r.overlaps_with(s)
        || op == real_interval_meet;
     }
     // If 'op == int_interval_join', then construct the join of 'r' and
     // 's' in the informational order "<=".  Namely, 'r intersection s'.
     // If 'op == int_interval_meet', then construct the meet in the
     // informational order "<=".  Namely, the smallest interval containing
     // 'r union s'.

    RealInterval(char const* l);
    bool pre_RealInterval(char const* l) const
     {
         return is_type_label(l);
     }
     // Construct the interval described by the string 'l', where 'l' is of
     // the form described by "RealInterval::is_type_label()".

    RealInterval(
         bool has_lb, double lb,
         bool has_ub, double ub
    );
    bool pre_RealInterval(
     bool has_lb, double lb,
     bool has_ub, double ub
    ) const
     {
         return is_interval(has_lb, lb, has_ub, ub);
     }
     // Construct the interval described by the string 'l', where 'l' is of
     // the form described by "RealInterval::is_type_label()".

    static bool is_type_label(char const* l);
    static bool pre_is_type_label(char const* l)
     {
         return true;
     }
     // Return true if 'l' is a non-NULL string, and matches
     // "RealInterval::format".  Namely, characters 'L' and 'U' replaced by
     // either:
     //   + the string '-i', representing the absence of a lower bound;
     //   + a string matching regular expression '(0|[1-9][0-9']*)',
     //     which is a decimal representation of a finite bound; or
     //   + the string '+i', representing the absence of an upper bound.
```

```
static bool is_interval(
 bool has_lb, double lb, bool has_ub, double ub
);
static bool pre_is_interval(
 bool has_lb, double lb, bool has_ub, double ub
)
 {
     return true;
 }
 // Return true if 'has_lb', 'lb', 'has_ub', and 'ub' describe a
 // well-formed interval.  Namely one of:
 //   + 'has_lb' and 'has_ub' are both 'false' so that the interval is
 //     open;
 //   + 'has_lb' is 'false and 'has_ub' is 'true' and 'ub' is the upper
 //     bound, so that the interval is half open with no lower bound;
 //   + 'has_lb' is 'true' and 'has_ub' is 'false' and 'lb' is the
 //     lower bound, so that the interval is half open with no upper
 //     bound; or
 //   + 'has_lb' and 'has_ub' are both 'true' and 'lb <= ub', so that
 //     the interval is closed.

static bool parse_type_label(
 char const* l,
 bool& has_lb, double& lb, bool& has_ub, double& ub
);
static bool pre_parse_type_label(
 char const* l,
 bool has_lb, double const& lb, bool has_ub, double const& ub
)
 {
     return l != NULL;
 }
 // Parse the type label 'l'.  Record the parsed interval bounds in
 // 'has_lb' 'lb', 'has_ub', and 'ub'.  Return true if 'l' matches
 // "RealInterval::format" as for "RealInterval::is_type_label()".

bool overlaps_with(RealInterval const& r) const;
bool pre_overlaps_with(RealInterval const& r) const
 {
     return true;
 }
 // Return true if 'this' and 'r' have a non-empty intersection.

bool has_lower_bound() const;
bool pre_has_lower_bound() const
 {
     return true;
 }
 // Return true if the interval has a finite lower bound.

double lower_bound() const;
bool pre_lower_bound() const
 {
     return has_lower_bound();
 }
 // Return the finite lower bound of this interval.

bool has_upper_bound() const;
bool pre_has_upper_bound() const
 {
     return true;
 }
 // Return true if the interval has a finite upper bound.

double upper_bound() const;
bool pre_upper_bound() const
 {
     return has_upper_bound();
 }
 // Return the finite upper bound of this interval.

char* type_label() const;
bool pre_type_label() const
 {
     return true;
 }
 // Return a string matching "RealInterval::format" as for
 // "RealInterval::is_type_label()", and representing this interval.
 // The client must dispose of the string.

bool operator <= (RealInterval const& r) const;
```

```
    // Return true if 'this' is less specific than 'r', i.e., 'r' is a
    // subset of 'this'.

    bool operator == (RealInterval const& r) const;
    // Return true if 'this' and 'r' describe the same interval.

    bool operator != (RealInterval const& r) const;
    // Return true if 'this' and 'r' describe different intervals.


    // The following static data member provides the format for type labels.
    // 'L' indicates position of lower bound, and 'U' indicates position of
    // upper bound.  Therefore, 'L' and 'U' should not be used elsewhere in
    // the format string.

    static char const* format;
    // "real_interval_L_U".


private:

    KR_DECLARE_PRECONDITION_SUPPORT;
    // Precondition checking state.

    bool _has_lower;
    // True if lower bound exists.

    double _lower;
    // Value of lower bound if it exists.

    bool _has_upper;
    // True if upper bound exists.

    double _upper;
    // Value of upper bound if it exists.


}; // class RealInterval


#endif
```

B.1.2 RealInterval_preprocessor.hh

```
// ---------------------------------------------------------------------------
//   Order Types                                                Teng-Wen Chang
// ---------------------------------------------------------------------------
//
// OrderTypes/real_interval_preprocessor.hh
//   - Concrete class implementing a preprocessor for "RealInterval" values
//     by extending "PreprocessorBase".
//   - Grammar Rules:
//
//      range_def -> IDENTIFIER '=' '[' low_expr ',' high_expr ']'
//
//      low_expr -> sum_expr
//
//      low_expr -> '-' INFINITY
//
//      high_expr -> sum_expr
//
//      high_expr -> '+' INFINITY
//
//      sum_expr -> prod_expr
//
//      sum_expr -> sum_expr '+' prod_expr
//
//      sum_expr -> sum_expr '-' prod_expr
//
//      prod_expr -> paren_expr
//
//      prod_expr -> prod_expr '*' paren_expr
//
//      prod_expr -> prod_expr '/' paren_expr
//
//      paren_expr -> REAL
//
//      paren_expr -> '-' REAL
//
```

```
//      paren_expr -> '+' REAL
//
//      paren_expr -> '(' sum_expr ')'
//
//   - Lexical Rules:
//
//      IDENTIFIER -> '[A-Z][0-9A-Za-z_+-]*'
//
//      NATURAL -> '(0|[1-9][0-9]*)'
//      REAL -> '(0|[1-9][0-9]*d?[0-9]*)'
//
//      INFINITY -> 'i'
//
//   - Source Tracking Directives:
//
//         SYNC_LINE -> '^# +[0-9].*'
//
//         INT_INTERVAL_LINE -> '^# +RealInterval .*'
//
//         OTHER_LINE -> '^# +[A-Z][0-9A-Za-z_+-]* (.*)'
//


#ifndef OG_ORDERTYPES_REAL_INTERVAL_PREPROCESSOR_HH
#define OG_ORDERTYPES_REAL_INTERVAL_PREPROCESSOR_HH


// Included Header Files.

#ifndef KR_CON_INDEX_SET_HH
#include "Containers/index_set.hh"
#endif

#ifndef KR_MGSATSH_PREPROCESSOR_BASE_HH
#include "TFSShells/preprocessor_base.hh"
#endif

#include "TFSShells/preconditions.hh"


// Declaration of classes to be defined.

union RealIntervalSymValues;// Semantic values for preprocessor.
class RealIntervalPreprocessor;// "RealInterval" preprocessor.


// ---------------------------------------------------------------------------
//   union RealIntervalSymValues
//
//   Union type defining the set of possible semantic values for symbols in
//   the "RealInterval" preprocessor.  Semantic values point to read only
//   objects.
//
//   Semantic values updates are restricted to the "RealIntervalPreprocessor"
//   class.  This underpins a resource management policy where
//   "RealIntervalPreprocessor" owns all semantic values, and deallocates
//   resources in semantic values discarded during error recovery.
// ---------------------------------------------------------------------------

union RealIntervalSymValues
{

    struct {
    }       empty;
     // No resources.

    struct {
     char const* label;
    }       identifier;
     // Resources:
     //   labela label resource registered with the "LabelManager"
     //     singleton.

    struct {
     double value;
    }       real;
     // Resources:
```

```
    //   valuea non-NULL pointer to a GNU variable length double.

    struct {
     bool closed;
     double value;
    }       endpoint;
     // Resources:
     //   closed a boolean value, 'true' if 'value' represents a finite
     //     endpoint and 'false' if 'value' represents the sign of
     //     an endpoint at infinity; and
     //   valuea non-NULL pointer to a GNU variable length double.


}; // union RealIntervalSymValues


// ---------------------------------------------------------------------------
//   class RealIntervalPreprocessor
//
//   Concrete class implementing a preprocessor for "RealInterval" values
//   by extending "PreprocessorBase".
// ---------------------------------------------------------------------------

class RealIntervalPreprocessor
    : public PreprocessorBase
{
public:

    RealIntervalPreprocessor();
    bool pre_RealIntervalPreprocessor() const
     {
         return true;
     }
     // Construct an unattached preprocessor.

    virtual ~RealIntervalPreprocessor();
    bool pre_D_RealIntervalPreprocessor() const
     {
         return LineSource::pre_D_LineSource();
     }
     // Deconstruct this preprocessor.


     // "LineSource" Public Interface.
     //
     // Fully implemented by "PreprocessorBase".


     // "LineSourceFilter" Public Interface.
     //
     // Fully implemented by "PreprocessorBase".


     // "PreprocessorBase" Public Interface.
     //
     // Define the preprocessor tag as "RealInterval".

     virtual char const* preprocessor_name() const;


     // The following static data members provide format lines for error
     // reports.

     static char const* err_divide_by_zero;
      // Divide by zero.

     static char const* err_existing;
      // Identifier already assigned a value.

     static char const* err_interval;
      // Ill formed interval.

     static char const* err_label;
      // Right hand side is not an interval type label.

     static char const* err_syntax;
      // Syntax error in preprocessor directive.


 protected:
```

```
// "PreprocessorBase" Protected Interface.
//
// The protected interface provides the basis for a concrete derivation.

virtual bool parse();


// The following methods implement resource management on semantic values
// in the Yacc parser.
//
// Yacc parsers do not deconstruct discarded and reduced tokens.  Instead,
// inactive tokens are simply overwritten.  This causes no problems for
// reduced tokens since their resources may be reclaimed in the rule's
// action.  However, during error recovery tokens are discarded, and their
// resources lost.
//
// Each "RealIntervalPreprocessor" instance holds pointers to resources
// held by unconsumed tokens.  Each resource is registered when
// constructed and deregistered when consumed by a rule's action.  A
// resource is 'outstanding' while it remains registered.  Resources
// outstanding at the deconstruction of the "RealIntervalPreprocessor"
// instance are deallocated.
//
// Several subtleties must be observed.
//   - An identifier may be referenced many times across the collection of
//     unconsumed tokens, therefore a single reference is held for the
//     life of the preprocessor.  This is handled in "PreprocessorBase".
//   - "double" values are not shared objects, therefore a simple pointer
//     to each object is sufficient.

double collect_real();
bool pre_collect_real() const
 {
   char c=get_char();
   return isdigit(c) || (c == 'd');
 }
 // Collect the next unsigned real on the input buffer.  Construct
 // an "double" instance holding the value, register it as an
 // outstanding resource, and return a "const" pointer to act as a
 // handle.

bool accept_interval(
 char const* ident,
 bool has_lb, double lb, bool has_ub, double ub
);
bool pre_accept_interval(
 char const* ident,
 bool has_lb, double lb, bool has_ub, double ub
) const
 {
     return ident != NULL && is_identifier(ident);
 }
 // Assign the interval to 'ident'.  Consume the arguments.  Return
 // true if the arguments are well formed, otherwise return false and
 // collect an error report.

bool accept_type_label(char const* ident, char const* label);
bool pre_accept_type_label(char const* ident, char const* label) const
 {
     return ident != NULL && is_identifier(ident)
    && label != NULL && is_identifier(label);
 }
 // Assign the "IntRealerval" type label to 'ident'.  Return true if the
 // arguments are well formed, otherwise return false and collect an
 // error report.

void reject_parse();
bool pre_reject_parse() const
 {
     return true;
 }
 // Collect an error report indicating the parse failed.


// Protected methods and private data members of this class are visible to
// the parser implementation.

friend int real_interval_lex(RealIntervalSymValues* lvalp, void* context);
 // Retrieve the next token recognised in the preprocessor directive
 // syntax.
```

```
    friend int real_interval_parse(void* context);
     // Parse the preprocessor directive.


private:

    KR_DECLARE_PRECONDITION_SUPPORT;
     // Precondition checking state.

    char const* _preprocessor_name;
     // Name of preprocessor type.


}; // class RealIntervalPreprocessor



#endif
```

B.1.3 real_interval_preprocessor.y

```
/*
 * ---------------------------------------------------------------------------
 *   Order Types                                                Teng-Wen Chang
 * ---------------------------------------------------------------------------
 *
 * OrderTypes/real_interval_preprocessor.y
 *   - Bison/Yacc grammar to specify preprocessor handling of real interval
 *     values.
 *   - Management of parser resources.
 *       + For each resource assigned to the parse (e.g., a string recognised
 *         in the lexical analyser), an appropriate method must be called in
 *         "TypesParse" to create a managed resource.
 *       + For each production consuming a resource carrying token, another
 *         method must be called in "TypesParse" to recover the resource.
 *         This is carried out according to the action taken on the resource:
 *           + if the resource is transfered to the reduced token, it is not
 *             relinquished, but
 *           + if the resource is not transferred, it is relinquished.
 *   - See: "MGSatShell/real_interval_preprocessor.hh:RealIntervalPreprocessor"
 *     for further details on managing parse resources through the enclosing
 *     "RealIntervalPreprocessor" instance.
 */


%{

// Included header files.

#ifndef OG_ORDERTYPES_REAL_INTERVAL_PREPROCESSOR_HH
#include "OrderTypes/real_interval_preprocessor.hh"
#endif



// ---------------------------------------------------------------------------
//   #define YYPARSE_PARAM context
//   #define YYLEX_PARAM context
//
//   Void pointer to the encapsulating preprocessor.
// ---------------------------------------------------------------------------

#define YYPARSE_PARAM context
#define YYLEX_PARAM context



// ---------------------------------------------------------------------------
//   #define YYSTYPE RealIntervalSymValues
//
//   Union type defining the set of possible semantic values returned by the
//   scanner and the productions in the preprocessor.
// ---------------------------------------------------------------------------

#define YYSTYPE RealIntervalSymValues



// ---------------------------------------------------------------------------
```

```
//   int real_interval_error(char* s)
//
//   Accept the parse error.
//
//   Error reporting via the error stream is suppressed.  Instead errors are
//   stored in the preprocessor's '_err_line' data member.  This behaviour is
//   explicitly implemented in the parser.
// ---------------------------------------------------------------------------

int real_interval_error(char* s);

%}


/* ---------------------------------------------------------------------------
 * Bison Declarations
 */


/*
 *  Construct a reentrant parser.
 */

%pure_parser


/*
 *  Expect 0 shift/reduce conflicts.
 */

%expect 0


/*
 *  Standard tokens in the preprocessor directive syntax.
 */

%token<identifier>IDENTIFIER
%token<real>  REAL
%token<empty>INFINITY

%type<empty>range_def
%type<endpoint>low_expr
%type<endpoint>high_expr
%type<real>   sum_expr
%type<real>   prod_expr
%type<real>  paren_expr


/*
 *  Topmost production.
 */

%startrange_def


/* ---------------------------------------------------------------------------
 * Grammar Rules
 */

%%

range_def:
     IDENTIFIER '=' '[' low_expr ',' high_expr ']'
         {
         RealIntervalPreprocessor* preprocessor
             = (RealIntervalPreprocessor*) context;

         if (
             preprocessor->accept_interval(
             $1.label,
             $4.closed, $4.value,
             $6.closed, $6.value
             )
         ) {
             // Accept the parse.
             YYACCEPT;
         }
         else {
```

```
            // Abort the parse.
            YYABORT;
        }
        }

    | IDENTIFIER '=' IDENTIFIER
        {
        RealIntervalPreprocessor* preprocessor
            = (RealIntervalPreprocessor*) context;

        // Record the type label in the symbol table.
        if (
            preprocessor->accept_type_label(
            $1.label, $3.label
            )
        ) {
            // Accept the parse.
            YYACCEPT;
        }
        else {
            // Abort the parse.
            YYABORT;
        }
        }

    | error
        {
          RealIntervalPreprocessor* preprocessor
        = (RealIntervalPreprocessor*) context;

          // Reject the parse.
          preprocessor->reject_parse();

          // Abort the parse.
          YYABORT;
        }
    ;


low_expr:
    sum_expr
        {
        // Collect value from subexpression.
        $$.closed = true;
        $$.value  = $1.value;
        }

    | '-' INFINITY
        {
        RealIntervalPreprocessor* preprocessor
            = (RealIntervalPreprocessor*) context;

        // Open lower bound.
        $$.closed = false;
        $$.value = -1.0;
        }
    ;


high_expr:
    sum_expr
        {
        // Collect value from subexpression.
        $$.closed = true;
        $$.value  = $1.value;
        }

    | '+' INFINITY
        {
        RealIntervalPreprocessor* preprocessor
            = (RealIntervalPreprocessor*) context;

        // Open upper bound.
        $$.closed = false;
        $$.value  = 1.0;
        }
    ;


sum_expr:
    prod_expr
```

```
        {
        // Collect value from subexpression.
        $$.value = $1.value;
        }

    | sum_expr '+' prod_expr
        {
        RealIntervalPreprocessor* preprocessor
            = (RealIntervalPreprocessor*) context;

        // Compute sum of operands.
        $$.value = $1.value + $3.value ;
        }

    | sum_expr '-' prod_expr
        {
        RealIntervalPreprocessor* preprocessor
            = (RealIntervalPreprocessor*) context;

        // Compute difference of operands.
        $$.value = $1.value - $3.value;
        }
    ;


prod_expr:
    paren_expr
        {
        // Collect value from subexpression.
        $$.value = $1.value;
        }

    | prod_expr '*' paren_expr
        {
        RealIntervalPreprocessor* preprocessor
            = (RealIntervalPreprocessor*) context;

        // Compute product of operands.
        $$.value = $1.value * $3.value;
        }

    | prod_expr '/' paren_expr
        {
        RealIntervalPreprocessor* preprocessor
            = (RealIntervalPreprocessor*) context;

        // Compute ratio of operands.
        if ($3.value == 0 ) {
          // Divide by zero error occurred.
          preprocessor->collector()->append_count(
            RealIntervalPreprocessor::err_divide_by_zero,
            strlen(RealIntervalPreprocessor::err_divide_by_zero)
            );
          // Abort the parse.
          YYABORT;
        }
        else {
          $$.value = $1.value / $3.value;
        }
        }
    ;


paren_expr:
    REAL
        {
        // Collect value from scanner token.
        $$.value = $1.value;
        }

    | '-' REAL
        {
        RealIntervalPreprocessor* preprocessor
            = (RealIntervalPreprocessor*) context;

        // Negate value  collected from scanner token.
        $$.value *= -1.0;
        }

    | '+' REAL
        {
```

```
        // Collect value from scanner token.
        $$.value = $2.value;
         }

    | '(' sum_expr ')'
         {
        // Collect value from subexpression.
        $$.value = $2.value;
         }
    ;

%%


/* ---------------------------------------------------------------------------
 * C++ Code
 */


// ****************************************************************************
// *** int real_interval_lex(RealIntervalSymValues* lvalp, void* context)
// ****************************************************************************

int real_interval_lex(RealIntervalSymValues* lvalp, void* context)
{
    RealIntervalPreprocessor* preprocessor
     = (RealIntervalPreprocessor*) context;
    char c  // Next charcter in input buffer.
     = preprocessor->get_char();

    // Consume white space.
    while ( c == ' ' || c == '\t' )
     c = preprocessor->next_char();

    // Is this an identifier?
    if ( isalpha(c) || c == '_' ) {
     // Collect the token's label.
     lvalp->identifier.label = preprocessor->collect_identifier();

     if ( strcmp(lvalp->identifier.label, "i") == 0 )
         return INFINITY;
     else
         // Return as identifier.
         return IDENTIFIER;
    }

    // OR, is this an real?
    else if ( isdigit(c) || c == 'd' ) {
     // Collect real.
     lvalp->real.value = preprocessor->collect_real();

     // Return as natural number.
     return REAL;
    }

    // OR, is this end of input.
    else if ( c == '\n' || c == '\0' ) {
     return 0;
    }

    // Otherwise, return this as a character?
    else {
     int d = c;
     c = preprocessor->next_char();
     return d;
    }
} // int real_interval_lex(RealIntervalSymValues*, void*)


// ****************************************************************************
// *** int real_interval_error(char* s)
// ****************************************************************************

int real_interval_error(char* s)
{
    // Error reporting via the error stream is suppressed.  Instead errors are
    // stored in the preprocessor's '_err_line' data member.  This behaviour
    // is explicitly implemented in the parser.
```

```
    // Mandatory return statement.
    return 0;
} // int real_interval_error(char*)
```

B.1.4 types_with_real_intervals.hh

```
// ---------------------------------------------------------------------------
//   Order Types                                                Teng-Wen Chang
// ---------------------------------------------------------------------------
//
// OrderTypes/types_with_real_intervals.hh
//   - Representation of intervals on the real number line (R), and proxy
//     class to extend an inheritance hierarchy with the informational order
//     of real intervals.
//


#ifndef OG_ORDERTYPES_TYPES_WITH_REAL_INTERVALS_HH
#define OG_ORDERTYPES_TYPES_WITH_REAL_INTERVALS_HH


// Included Header Files.

#ifndef KR_CON_INDEX_SET_HH
#include "Containers/index_set.hh"
#endif

#ifndef KR_CON_INDEX_INJECTION_HH
#include "Containers/index_injection.hh"
#endif

#ifndef KR_TFS_MANUAL_TYPES_HH
#include "FeatureStructures/manual_types.hh"
#endif

#ifndef KR_TFSSHELLS_DEFINITIONS_HH
#include "TFSShells/definitions.hh"
#endif

#include "TFSShells/preconditions.hh"


// Declaration of classes to be defined.

class TypesWithRealIntervals;// Inheritance hierarchy plus real intervals.


// ---------------------------------------------------------------------------
//   class TypesWithRealIntervals
//
//   Concrete class providing a proxy (or smart reference) to an inheritance
//   hierarchy of feature structure types.  This proxy augments the subject
//   inheritance hierarchy with the informational order of real intervals.
//
//   The "TypeHierarchy" methods "identifier()", "join()", and "meet()" act to
//   pick out real interval types.  This role is discussed in
//   "FeatureStructures/type_hierarchy.hh".  It is accompanied by a type
//   resource management scheme that holds reference counts on real
//   interval types.  In addition, the method "is_label()" syntactically
//   identifies labels of real interval types.  So that the "identifier()"
//   precondition is satisfied by all syntactically correct labels.
//
//   The collection of real interval types form a BCPO in their own right,
//   since meet is defined as union, for all but the empty set of intervals.
//   This structure is inserted above the universal type, and topped by the
//   absurd type.  Therefore, the empty interval is represented by the absurd
//   type.
// ---------------------------------------------------------------------------

class TypesWithRealIntervals
    : public ManualTypes
{
public:

    TypesWithRealIntervals(ManualTypes* real_types);
    bool pre_TypesWithRealIntervals(ManualTypes const* real_types) const
```

```
        {
            return real_types != NULL;
        }
        // Construct a proxy whose real subject is the type hierarchy
        // 'real_types', and which augments the inheritance hierarchy in
        // 'real_types' by the real intervals.  Take sole ownership of the
        // real subject.

    virtual ~TypesWithRealIntervals();
    bool pre_D_TypesWithRealIntervals() const
        {
            return InheritanceHierarchy::pre_D_InheritanceHierarchy();
        }
     // Free this proxy and the real subject.  Free the representations of
     // all interval types.


     // "TypeHierarchy" Interface.
     //
     // "TypesWithRealIntervals" delegates these methods to the real subject.
     // It intervenes to test for real interval type labels and identifiers,
     // as described in the "TypeHierarchy" interface.

     virtual bool is_type(Index t) const;
     virtual bool are_types(IndexSet const& T) const;
     virtual bool is_regular_type(Index t) const;
     virtual bool are_regular_types(IndexSet const& T) const;
     virtual bool is_universal_type(Index t) const;
     virtual bool is_absurd_type(Index t) const;
     virtual bool is_label(char const* l) const;
     virtual Index universal() const;
     virtual Index absurd() const;
     virtual Index identifier(char const* l) const;
     virtual char const* label(Index t) const;
     virtual bool subsumes(Index s, Index t) const;
     virtual bool comparable(Index s, Index t) const;
     virtual Index join(Index s, Index t) const;
     virtual Index meet(Index s, Index t) const;
     virtual Index accept_type(Index t) const;
     virtual void relinquish_type(Index& t) const;


     // "InheritanceHierarchy" Interface.
     //
     // "TypesWithRealIntervals" simply delegates these methods to the real
     // subject.

     virtual bool is_succession_type(Index t) const;
     virtual bool are_succession_types(IndexSet const& T) const;
     virtual bool is_leaf_succession_type(Index t) const;
     virtual bool are_sound_parents(IndexSet const& T) const;
     virtual IndexSet* succession_types() const;
     virtual IndexSet* maximal_succession_types() const;
     virtual IndexSet* children(Index t) const;
     virtual IndexSet* parents(Index t) const;
     virtual Index insert(char const* l, IndexSet const& S);
     virtual void remove(Index t);
     virtual bool operator == (InheritanceHierarchy const& types) const;
     virtual bool operator != (InheritanceHierarchy const& types) const;


private:

     KR_DECLARE_PRECONDITION_SUPPORT;
      // Precondition checking state.

     ManualTypes* _real_types;
         // Real subject defining inheritance hierarchy over the succession
         // types.

     IndexInjection _label_to_interval;
      // Injection:
         //   real interval  <-->  real interval
      //   type label           representation
      //   (char const*)        (Association<RealInterval, size_t>*)


}; // class TypesWithRealIntervals
```

```
#endif
```

B.2 GeomAscend

B.2.1 geom_ascend.hh

```
// ---------------------------------------------------------------------------
//   Order Types                                                  Teng-Wen Chang
// ---------------------------------------------------------------------------
//
// OrderTypes/geom_ascend.hh
//   - Representation of ascends on the pointset geometry (P).
//


#ifndef OG_ORDERTYPES_GEOM_ASCEND_HH
#define OG_ORDERTYPES_GEOM_ASCEND_HH


// Included Header Files.

#ifndef OG_GEOM_REGION_HH
#include "Geometry/region.hh"
#endif

#ifndef OG_PP_GEOMNAME_MANAGER_HH
#include "PreProcessor/geomname_manager.hh"
#endif
#include "TFSShells/preconditions.hh"


// Declaration of classes to be defined.

class GeomAscend;// Ascends on the pointset geometry


// ---------------------------------------------------------------------------
//   class GeomAscend
//
//   Representation of containment on the pointset geometry (P) the
//   empty pointset is specified via '-p', and the whole pointset is
//   specified via the absurd type.  For
//   example, others are specified via their unique geom_names
//
//       abox        is a point set
//       -p           is the empty set.
//
//   "GeomAscend" values are picked out by mapping a collection of
//   strings corresponding to the type labels for these objects to a
//   resource manager called "GeomNameManager".  An informational
//   ordering is supplied through the "<=" operator that interprets
//   the containment as constraints ordered by specificity.  An ascend
//   'r' subsumes another 's' iff pointset 's' contains pointset 'r'.
//   "GeomAscend" values are also manipulated by an additional
//   constructor which computes the join or meet according to the
//   given informational ordering.
//
// ---------------------------------------------------------------------------

class GeomAscend
{
public:

    // The following nested typedef distinguishes between constructors.

    typedef enum {
     geom_ascend_join, geom_ascend_meet
    } Op;

    GeomAscend();
    bool pre_GeomAscend() const
     {
         return true;
     }
```

```
  // Construct the empty pointset.

  GeomAscend(GeomAscend const& r);
  bool pre_GeomAscend(GeomAscend const& r) const
   {
       return true;
   }
   // Construct a copy of the geom_ascend 'r'.

  GeomAscend(
   Op op, GeomAscend const& r, GeomAscend const& s
  );
  bool pre_GeomAscend(
   Op op, GeomAscend const& r, GeomAscend const& s
  ) const
   {
       return op == geom_ascend_join
       || op == geom_ascend_meet ;
   }
   // If 'op == geom_ascend_join', then construct the join of 'r'
   // and 's' in the informational order "<=".  Namely, the union
   // of pointset 'r' and 's'.

   // If 'op == geom_ascend_meet', then construct the meet in
   // the informational order "<=".  Namely, the intersection of
   // pointset of 'r' and 's'.

GeomAscend(char const* l);
bool pre_GeomAscend(char const* l) const
  {
    return is_type_label(l);
  }
// Construct the interval described by the string 'l', where
// 'l' is of the form described by
// "GeomAscend::is_type_label()".
// If l contains the geomname registerred with GeomNameManager, then,
//    l is a type_label,
// otherwise, No.

GeomAscend( bool is_empty_set, Region const* g);
bool pre_GeomAscend( bool is_empty_set, Region const* g ) const
  {
    return true;
  }
// Construct the ascend described by the bool is_empty_set
// and a char* g (for geom_name)

bool pre_D_GeomAscend() const
  {
    return true;
  }
// Construct the ascend described by the bool is_empty_set
// and a char* g (for geom_name)

  static bool is_type_label(char const* l);
  static bool pre_is_type_label(char const* l)
   {
       return true;
   }
   // Return true if 'l' is a non-NULL string, and matches
   // "GeomAscend::format".  Namely, characters 'G' replaced by
   // either:
   //   + the string '-p', representing the empty geom set
   //   + a string matching regular expression
   //     '([a-z][0-9a-z_+-]*)', which is a machine generated
   //     geometry name of a finite bound


   static bool is_ascend(
    bool is_empty_set, Region const* geom
   );
   static bool pre_is_ascend(
    bool is_empty_set, Region const* geom
    )
  {
    return true;
  }
// Return true if 'is_empty_set' and 'gname' describe a well-formed
// ascend.  Namely one of:
//   + 'is_empty_set' is 'false' and 'ganme' is registerred with
//     GeomNameManager ;
```

```
//    + 'is_empty_set' is 'true', so that the ascend is an empty
//      pointset.

  static bool parse_type_label(
   char const* l,
   bool& is_empty_set, char const *& g
  );
  static bool pre_parse_type_label(
   char const* l,
   bool& is_empty_set, char const *& g
  )
   {
       return l != NULL;
   }
   // Parse the type label 'l'.  Record the parsed geom number in
   // 'is_empty_set' 'r'. Return true if 'l' matches
   // "GeomAscend::format" as for "GeomAscend::is_type_label()".

  bool is_empty_set() const;
  bool pre_is_empty_set() const
   {
       return true;
   }
   // Return true if the geom number is an empty pointset geom.

  Region const* const_geom() const;
  bool pre_const_geom() const
   {
       return !is_empty_set();
   }
   // Return the geom number of this ascend.

  char* type_label() const;
  bool pre_type_label() const
   {
       return true;
   }
   // Return a string matching "GeomAscend::format" as for
   // "GeomAscend::is_type_label()", and representing this ascend.
   // The client must dispose of the string.

  bool operator <= (GeomAscend const& r) const;
   // Return true if 'this' is less specific than 'r', i.e., the
   // boundary of 'this' is within the boudary of 'r'.

  bool operator == (GeomAscend const& r) const;
   // Return true if 'this' and 'r' describe the same ascend geom.

  bool operator != (GeomAscend const& r) const;
   // Return true if 'this' and 'r' describe different ascend geoms.


  // The following static data member provides the format for type
  // labels.  'R' indicates position of geom.  Therefore, 'R' should
  // not be used elsewhere in the format string.

  static char const* format;
   // "geom_ascend_G".


private:

   KR_DECLARE_PRECONDITION_SUPPORT;
    // Precondition checking state.

   bool _is_empty_set;
    // True if _geom is an empty pointset.

 //  Region _geom;
 char const * _gname;
    // Value of geom if it exists.


}; // class GeomAscend


#endif
```

B.2.2 geom_ascend_preprocessor.hh

```
// ---------------------------------------------------------------------------
//   Order Types                                                  Teng-Wen Chang
// ---------------------------------------------------------------------------
//
// MGSatShell/geom_ascend_preprocessor.hh
//   - Concrete class implementing a preprocessor for "GeomAscend" values
//     by extending "PreprocessorBase".
//   - Grammar Rules:
//
// const_def       -> IDENTIFIER '=' geom_expr
// geom_expr       -> geom_paren_expr [ boolean_op geom_paren_expr ] *
// boolean_op      -> 'union'
// boolean_op      -> 'intersect'
// boolean_op      -> 'diff'
// geom_paren_expr -> '(' geom_expr ')'
// geom_paren_expr -> prim_geom_expr
// prim_geom_expr  -> primitive_op '[' arg_list ']'
// prim_geom_expr  -> '$'ID
// arg_list        -> sum_expr [ ',' sum_expr ]*
// primitive_op    -> 'box'
// primitive_op    -> 'shpere'
// primitive_op    -> 'ellipse'
// primitive_op    -> 'ellipsoid'
// primitive_op    -> 'cylinder'
// primitive_op    -> 'cone'
// primitive_op    -> 'point'
// primitive_op    -> 'line'
// primitive_op    -> 'circle'
// sum_expr        -> prod_expr [ sum_op prod_expr ]*
// sum_op          -> '+'
// sum_op          -> '-'
// prod_expr       -> pow_expr [ prod_op pow_expr ]*
// prod_op         -> '*'
// prod_op         -> '/'
// pow_expr        -> trig_expr [ ** trig_expr ]*
// trig_expr       -> trig_op '(' sum_expr ')'
// trig_expr       -> paren_expr
// trig_op         -> 'sin'
// trig_op         -> 'con'
// trig_op         -> 'tan'
// trig_op         -> 'atan2'
// trig_op         -> 'exp'
// trig_op         -> 'log'
// trig_op         -> 'rand'
// trig_op         -> 'srand'
// trig_op         -> 'abs'
// trig_op         -> 'int'
// trig_op         -> 'sqrt'
// paren_expr      -> '(' sum_expr ')'
// paren_expr      -> '-' num_expr
// paren_expr      -> '+' num_expr
// paren_expr      -> num_expr
// num_expr        -> 'i'
// num_expr        -> INTEGER
// IDENTIFIER      -> [A-Z][A-Z_]*
// INTEGER         -> [-][0-9]+
// TYPEID          -> 'Solid' | 'Constant'
//


//      geom_expr -> sum_expr
//
//      geom_expr -> '-' INFINITY
//
//      real_expr -> '+' INFINITY
//
//      sum_expr -> prod_expr
//
//      sum_expr -> sum_expr '+' prod_expr
//
//      sum_expr -> sum_expr '-' prod_expr
//
//      prod_expr -> paren_expr
//
//      prod_expr -> prod_expr '*' paren_expr
//
//      prod_expr -> prod_expr '/' paren_expr
//
//      paren_expr -> REAL
```

```
//
//      paren_expr -> '-' REAL
//
//      paren_expr -> '+' REAL
//
//      paren_expr -> '(' sum_expr ')'
//
//   - Lexical Rules:
//
//      IDENTIFIER -> '[A-Z][0-9A-Za-z_+-]*'
//
//      NATURAL -> '(0|[1-9][0-9]*)'
//      REAL -> '(0|[1-9][0-9]*(d?[0-9]*))'
//
//      INFINITY -> 'i'
//
//   - Source Tracking Directives:
//
//         SYNC_LINE -> '^# +[0-9].*'
//
//         GEOM_ASCEND_LINE -> '^# +GeomAscend .*'
//
//         OTHER_LINE -> '^# +[A-Z][0-9A-Za-z_+-]* (.*)'
//


#ifndef OG_ORDERTYPES_GEOM_ASCEND_PREPROCESSOR_HH
#define OG_ORDERTYPES_GEOM_ASCEND_PREPROCESSOR_HH


// Included Header Files.

#ifndef KR_CON_INDEX_SET_HH
#include "Containers/index_set.hh"
#endif

#ifndef KR_MGSATSH_PREPROCESSOR_BASE_HH
#include "TFSShells/preprocessor_base.hh"
#endif

#ifndef OG_PP_GEOMNAME_MANAGER_HH
#include "PreProcessor/geomname_manager.hh"
#endif

#include "TFSShells/preconditions.hh"

// Declaration of classes to be defined.

union GeomAscendSymValues;// Semantic values for preprocessor.
class GeomAscendPreprocessor;// "GeomAscend" preprocessor.


// ---------------------------------------------------------------------------
//   union geomAscendSymValues
//
//   Union type defining the set of possible semantic values for symbols in
//   the "GeomAscend" preprocessor.  Semantic values point to read only
//   objects.
//
//   Semantic values updates are restricted to the "GeomAscendPreprocessor"
//   class.  This underpins a resource management policy where
//   "GeomAscendPreprocessor" owns all semantic values, and deallocates
//   resources in semantic values discarded during error recovery.
// ---------------------------------------------------------------------------

union GeomAscendSymValues
{

    struct {
    }       empty;
     // No resources.

    struct {
     char const* label;
    }       identifier;
     // Resources:
     //   labela label resource registered with the "LabelManager"
     //     singleton.
```

```
    struct {
     double value;
    }       real;
     // Resources:
     //   valuea non-NULL pointer to a Region *

    struct {
      bool empty_set;
      Region * geom;
    }       endpoint;
     // Resources:
     //   closed a boolean value, 'true' if 'value' represents a finite
     //     endpoint and 'false' if 'value' represents the sign of
     //     an endpoint at infinity; and
     //   valuea non-NULL pointer to a GNU variable length double.


}; // union GeomAscendSymValues



// ---------------------------------------------------------------------------
//   class GeomAscendPreprocessor
//
//   Concrete class implementing a preprocessor for "GeomAscend" values
//   by extending "PreprocessorBase".
// ---------------------------------------------------------------------------

class GeomAscendPreprocessor
    : public PreprocessorBase
{
public:

    GeomAscendPreprocessor();
    bool pre_GeomAscendPreprocessor() const
     {
         return true;
     }
     // Construct an unattached preprocessor.

    virtual ~GeomAscendPreprocessor();
    bool pre_D_GeomAscendPreprocessor() const
     {
         return LineSource::pre_D_LineSource();
     }
     // Deconstruct this preprocessor.


     // "LineSource" Public Interface.
     //
     // Fully implemented by "PreprocessorBase".


     // "LineSourceFilter" Public Interface.
     //
     // Fully implemented by "PreprocessorBase".


     // "PreprocessorBase" Public Interface.
     //
     // Define the preprocessor tag as "GeomAscend".

     virtual char const* preprocessor_name() const;


     // The following static data members provide format lines for error
     // reports.

     static char const* err_divide_by_zero;
      // Divide by zero.

     static char const* err_geom_eset_diff_non_eset;
      // Empty set diff non empty set.

     static char const* err_geom_trans_empty_pset;
      // tranlate an empty set.

     static char const* err_geom_trans_failed;
      // failed to tranlate a point set.
```

```
    static char const* err_geom_scale_empty_pset;
     // scale an empty set.

    static char const* err_geom_scale_failed;
     // failed to scale a point set.

    static char const* err_geom_rotatex_empty_pset;
     // rotate an empty set around X axis.

    static char const* err_geom_rotatex_failed;
     // failed to rotate a point set around X axis.

    static char const* err_geom_rotatey_empty_pset;
     // rotate an empty set around Y axis.

    static char const* err_geom_rotatey_failed;
     // failed to rotate an empty set around Y axis.

    static char const* err_geom_rotatez_empty_pset;
     // rotate an empty set around Z axis.

    static char const* err_geom_rotatez_failed;
     // failed to rotate an empty set around Z axis.

    static char const* err_existing;
     // Identifier already assigned a value.

    static char const* err_ascend;
     // Ill formed ascend.

    static char const* err_label;
     // Right hand side is not an ascend type label.

    static char const* err_syntax;
     // Syntax error in preprocessor directive.


protected:

    // "PreprocessorBase" Protected Interface.
    //
    // The protected interface provides the basis for a concrete derivation.

    virtual bool parse();


    // The following methods implement resource management on semantic values
    // in the Yacc parser.
    //

    // Several subtleties must be observed.
    //   - An identifier may be referenced many times across the collection of
    //     unconsumed tokens, therefore a single reference is held for the
    //     life of the preprocessor.  This is handled in "PreprocessorBase".
    //   - "double" values are not shared objects, therefore a simple pointer
    //     to each object is sufficient.

    double collect_real();
    bool pre_collect_real() const
     {
       char c=get_char();
       return isdigit(c) || (c == 'd');
     }
     // Collect the next unsigned real on the input buffer.  Construct
     // an "double" instance holding the value, register it as an
     // outstanding resource, and return a "const" pointer to act as a
     // handle.

    bool accept_ascend(
     char const* ident,
     bool is_empty_set, Region const* geom
    );
    bool pre_accept_ascend(
     char const* ident,
     bool is_empty_set, Region const * geom
    ) const
     {
         return ident != NULL && is_identifier(ident);
     }
     // Assign the ascend to 'ident'.  Consume the arguments.  Return
```

```
    // true if the arguments are well formed, otherwise return false and
    // collect an error report.

    bool accept_type_label(char const* label, bool& is_empty_set, Region const*
r);
    bool pre_accept_type_label(char const* label,bool& is_empty_set,Region const*
r) const
    {
        return  label != NULL && is_identifier(label);
    }
    // Assign the "GeomAscend" type label to 'ident'.  Return true if the
    // arguments are well formed, otherwise return false and collect an
    // error report.

    void reject_parse();
    bool pre_reject_parse() const
    {
        return true;
    }
    // Collect an error report indicating the parse failed.


    // Protected methods and private data members of this class are visible to
    // the parser implementation.

    friend int geom_ascend_lex(GeomAscendSymValues* lvalp, void* context);
    // Retrieve the next token recognised in the preprocessor directive
    // syntax.

    friend int geom_ascend_parse(void* context);
    // Parse the preprocessor directive.


private:

    KR_DECLARE_PRECONDITION_SUPPORT;
    // Precondition checking state.

    char const* _preprocessor_name;
    // Name of preprocessor type.


}; // class GeomAscendPreprocessor


#endif
```

B.2.3 geom_ascend_preprocessor.y

```
/*
 * ---------------------------------------------------------------------------
 *   Order Types                                                Teng-Wen Chang
 * ---------------------------------------------------------------------------
 *
 * OrderTypes/geom_ascend_preprocessor.y
 *   - Bison/Yacc grammar to specify preprocessor handling of geom ascend
 *     values.
 *   - Management of parser resources.
 *       + For each resource assigned to the parse (e.g., a string recognised
 *         in the lexical analyser), an appropriate method must be called in
 *         "TypesParse" to create a managed resource.
 *       + For each production consuming a resource carrying token, another
 *         method must be called in "TypesParse" to recover the resource.
 *         This is carried out according to the action taken on the resource:
 *           + if the resource is transfered to the reduced token, it is not
 *             relinquished, but
 *           + if the resource is not transferred, it is relinquished.
 *   - See: "OrderTypes/geom_ascend_preprocessor.hh:
 *           GeomAscendPreprocessor"
 *     for further details on managing parse resources through the enclosing
 *     "RealAscendPreprocessor" instance.
 */


%{

// Included header files.
#ifndef OG_PP_GEOMNAME_MANAGER_HH
#include "PreProcessor/geomname_manager.hh"
```

```
#endif

#ifndef OG_PP_GEOM_MANAGER_HH
#include "PreProcessor/geom_manager.hh"
#endif

#ifndef OG_GEOM_REGION_HH
#include "Geometry/region.hh"
#endif

#ifndef OG_GEOM_REGION_PRIMOP_HH
#include "Geometry/region_prim_op.hh"
#endif


#ifndef OG_ORDERTYPES_GEOM_ASCEND_PREPROCESSOR_HH
#include "OrderTypes/geom_ascend_preprocessor.hh"
#endif


// ---------------------------------------------------------------------------
//   #define YYPARSE_PARAM context
//   #define YYLEX_PARAM context
//
//   Void pointer to the encapsulating preprocessor.
// ---------------------------------------------------------------------------

#define YYPARSE_PARAM context
#define YYLEX_PARAM context


// ---------------------------------------------------------------------------
//   #define YYSTYPE GeomAscendSymValues
//
//   Union type defining the set of possible semantic values returned by the
//   scanner and the productions in the preprocessor.
// ---------------------------------------------------------------------------

#define YYSTYPE GeomAscendSymValues


// ---------------------------------------------------------------------------
//   int geom_ascend_error(char* s)
//
//   Accept the parse error.
//
//   Error reporting via the error stream is suppressed.  Instead errors are
//   stored in the preprocessor's '_err_line' data member.  This behaviour is
//   explicitly implemented in the parser.
// ---------------------------------------------------------------------------

int geom_ascend_error(char* s);

%}


/* ---------------------------------------------------------------------------
 * Bison Declarations
 */


/*
 *  Construct a reentrant parser.
 */

%pure_parser


/*
 *  Expect 0 shift/reduce conflicts.
 */

%expect 0


/*
 *  Standard tokens in the preprocessor directive syntax.
```

```
 * TWC: need to come back to here and revisit the same in GeomAsceondPreprecessor
 */

%token<identifier>IDENTIFIER
%token<identifier>GEOMNAME
%token<real>  REAL

/*
 * geometry constructor
 */
%token<empty>BOX
%token<empty>SHEET
%token<empty>LINE
%token<empty>CYLINDER
%token<empty>CAPPED_CYLINDER
%token<empty>CONE
%token<empty>CAPPED_CONE
%token<empty>CONE_FRUSTUM
%token<empty>CAPPED_CONE_FRUSTUM
%token<empty>SPHERE
%token<empty>ELLIPSOID

%token<empty>EMPTY_SET

/*
 * boolean operators
 */
%token<empty>UNION
%token<empty>DIFF
%token<empty>INTERSECT

/*
 * transformation operators
 */
%token<empty>TRANSLATE
%token<empty>SCALE
%token<empty>ROTATE_X
%token<empty>ROTATE_Y
%token<empty>ROTATE_Z


%type<empty>ascend_geom_def
%type<endpoint>geom_expr
%type<endpoint>geom_boolean_expr
%type<endpoint>geom_paren_expr
%type<endpoint>prim_geom_expr
%type<real>   sum_expr
%type<real>   prod_expr
%type<real>  paren_expr


/*
 *  Topmost production.
 */

%startascend_geom_def



/* ---------------------------------------------------------------------------
 * Grammar Rules
 */

%%

ascend_geom_def:
     IDENTIFIER '=' geom_expr
         {
        GeomAscendPreprocessor* preprocessor
            = (GeomAscendPreprocessor*) context;

        if (
            preprocessor->accept_ascend(
            $1.label,
            $3.empty_set, $3.geom
            )
        ) {
            // Accept the parse.
            YYACCEPT;
        }
        else {
```

```
            // Abort the parse.
            YYABORT;
        }
        }

        | IDENTIFIER INHERIT geom_desc
        {
        GeomAscendPreprocessor* preprocessor
            = (GeomAscendPreprocessor*) context;

        if (
            preprocessor->accept_ascend(
            $1.label,
            $3.empty_set, $3.geom
            )
        ) {
            // Accept the parse.
            YYACCEPT;
        }
        else {
            // Abort the parse.
            YYABORT;
        }
        }


    | error
        {
          GeomAscendPreprocessor* preprocessor
        = (GeomAscendPreprocessor*) context;

          // Reject the parse.
          preprocessor->reject_parse();

          // Abort the parse.
          YYABORT;
        }
    ;

geom_desc:
        IDENTIFIER
            {
            $$.label     = $1.label;
            $$.empty_set = $1.empty_set;
            $$.geom      = $1.geom;
        }
        ;
    | IDENTIFIER '&' geom_desc
            {
            $1.label = ,
        $3.empty_set, $3.geom


        }
        ;
        | IDENTIFIER '|' geom_desc
            {
        }
        ;
        | '(' geom_desc ')'
        ;

geom_expr:
    geom_boolean_expr
            {
            $$.geom = $1.geom;
            $$.empty_set = $1.empty_set;
        }
        | TRANSLATE '(' geom_expr ',' sum_expr ',' sum_expr ',' sum_expr ')'
            {
        GeomAscendPreprocessor* preprocessor
            = (GeomAscendPreprocessor*) context;

        if (! $3.empty_set) {

            if($3.geom->translate($5.value ,$7.value ,$9.value )) {

        $$.geom = $3.geom;
        $$.empty_set = $3.empty_set;

            } else {
```

```
        // error: translate failed
        preprocessor->collector()->append_count(
            GeomAscendPreprocessor::err_geom_trans_failed,
            strlen(GeomAscendPreprocessor::err_geom_trans_failed)
            );

        // Abort the parse.
        YYABORT;

            }
        } else {

            // error: try to translate an empty_pset.
        preprocessor->collector()->append_count(
            GeomAscendPreprocessor::err_geom_trans_empty_pset,
            strlen(GeomAscendPreprocessor::err_geom_trans_empty_pset)
            );

        // Abort the parse.
        YYABORT;

        }
      }
      | SCALE '(' geom_expr ',' sum_expr ',' sum_expr ',' sum_expr ',' sum_expr
')'
        {
        GeomAscendPreprocessor* preprocessor
          = (GeomAscendPreprocessor*) context;
        if (! $3.empty_set) {

            if($3.geom->scale($5.value ,$7.value ,$9.value,$11.value )){

        $$.geom = $3.geom;
        $$.empty_set = $3.empty_set;

            } else {

        // error: scale failed
        preprocessor->collector()->append_count(
            GeomAscendPreprocessor::err_geom_scale_failed,
            strlen(GeomAscendPreprocessor::err_geom_scale_failed)
            );

        // Abort the parse.
        YYABORT;
            }

        } else {

            // error: try to scale an empty_pset.

            preprocessor->collector()->append_count(
                GeomAscendPreprocessor::err_geom_scale_empty_pset,
                strlen(GeomAscendPreprocessor::err_geom_scale_empty_pset)
                );

            // Abort the parse.
            YYABORT;

        }
      }
        | ROTATE_X '(' geom_expr ',' sum_expr ',' sum_expr ',' sum_expr ','
sum_expr ')'
        {
        GeomAscendPreprocessor* preprocessor
          = (GeomAscendPreprocessor*) context;

        if (! $3.empty_set) {

            if($3.geom->rotateX($5.value ,$7.value , $9.value, $11.value)){

        $$.geom = $3.geom;
        $$.empty_set = $3.empty_set;

            } else {

        // error: rotate_x failed
        preprocessor->collector()->append_count(
            GeomAscendPreprocessor::err_geom_rotatex_failed,
            strlen(GeomAscendPreprocessor::err_geom_rotatex_failed)
```

```
        );

      // Abort the parse.
      YYABORT;
        }

      } else {

        // error: try to translate an empty_pset.
        preprocessor->collector()->append_count(
      GeomAscendPreprocessor::err_geom_rotatex_empty_pset,
      strlen(GeomAscendPreprocessor::err_geom_rotatex_empty_pset)
      );

      // Abort the parse.
        YYABORT;

      }
    }
    | ROTATE_Y '(' geom_expr ',' sum_expr ',' sum_expr ',' sum_expr ',' sum_expr
')'
        {
      GeomAscendPreprocessor* preprocessor
        = (GeomAscendPreprocessor*) context;

      if (! $3.empty_set) {

        if($3.geom->rotateY($5.value ,$7.value , $9.value, $11.value)){

      $$.geom = $3.geom;
      $$.empty_set = $3.empty_set;

        } else {

      // error: rotate_y failed
      preprocessor->collector()->append_count(
          GeomAscendPreprocessor::err_geom_rotatey_failed,
          strlen(GeomAscendPreprocessor::err_geom_rotatey_failed)
          );

      // Abort the parse.
      YYABORT;

        }

      } else {

        // error: try to translate an empty_pset.
        preprocessor->collector()->append_count(
      GeomAscendPreprocessor::err_geom_rotatey_empty_pset,
      strlen(GeomAscendPreprocessor::err_geom_rotatey_empty_pset)
      );

        // Abort the parse.
        YYABORT;

      }
    }
    | ROTATE_Z '(' geom_expr ',' sum_expr ',' sum_expr ',' sum_expr ',' sum_expr
')'
        {
      GeomAscendPreprocessor* preprocessor
        = (GeomAscendPreprocessor*) context;

      if (! $3.empty_set) {

        if($3.geom->rotateZ($5.value ,$7.value , $9.value, $11.value)){

      $$.geom = $3.geom;
      $$.empty_set = $3.empty_set;

        } else {

      // error: rotate_z failed
      preprocessor->collector()->append_count(
          GeomAscendPreprocessor::err_geom_rotatez_failed,
          strlen(GeomAscendPreprocessor::err_geom_rotatez_failed)
          );

      // Abort the parse.
      YYABORT;
```

```
          }

        } else {

          // error: try to translate an empty_pset.
          preprocessor->collector()->append_count(
        GeomAscendPreprocessor::err_geom_rotatez_empty_pset,
        strlen(GeomAscendPreprocessor::err_geom_rotatez_empty_pset)
        );

          // Abort the parse.
          YYABORT;

        }
      }
        ;

geom_boolean_expr:
     geom_paren_expr
            {
          $$.geom = $1.geom;
          $$.empty_set = $1.empty_set;
        }
        | geom_paren_expr UNION geom_expr
            {
          if($1.empty_set) {
        $$.geom = $3.geom;
        $$.empty_set = $3.empty_set;
          }
          else if($3.empty_set) {
        $$.geom = $1.geom;
        $$.empty_set = $1.empty_set;
          }
          else {
        $$.geom = Region::union_region($1.geom, $3.geom);
        $$.empty_set = false;
          }
        }
      | geom_paren_expr DIFF geom_expr
            {
          GeomAscendPreprocessor* preprocessor
        = (GeomAscendPreprocessor*) context;

          if($3.empty_set) {
        $$.geom = $1.geom;
        $$.empty_set = $1.empty_set;
          }
          else if (! $1.empty_set) {
        $$.geom = Region::difference_region($1.geom, $3.geom);
        $$.empty_set = false;
          }
          else {
        // error
        preprocessor->collector()->append_count(
            GeomAscendPreprocessor::err_geom_eset_diff_non_eset,
            strlen(GeomAscendPreprocessor::err_geom_eset_diff_non_eset)
            );

        // Abort the parse.
        YYABORT;

          }
        }
      | geom_paren_expr INTERSECT geom_expr
            {
          if( $3.empty_set || $1.empty_set ) {
        $$.geom = NULL;
        $$.empty_set = true;
          }
          else {
        $$.geom = Region::intersection_region($1.geom, $3.geom);
        $$.empty_set = false;
          }
        }
      ;

 geom_paren_expr:
      '(' geom_expr ')'
            {
          $$.geom = $2.geom;
          $$.empty_set = $2.empty_set;
```

```
        }
    | prim_geom_expr
            {
          $$.geom = $1.geom;
          $$.empty_set = $1.empty_set;
        }
    | IDENTIFIER
        {
       GeomAscendPreprocessor* preprocessor
           = (GeomAscendPreprocessor*) context;

       // Record the type label in the symbol table.
       if (
           preprocessor->accept_type_label(
          $1.label, $$.empty_set, $$.geom
           )
       ) {
           // Accept the parse.
           YYACCEPT;
       }
       else {
           // Abort the parse.
           YYABORT;
       }
        }
/*        | GEOMNAME
            {
          GeomNameManager* geomname = GeomNameManager::instance();
          $$.empty_set = false;
          $$.geom = geomname->geom($1.label);
        }
*/

     | '-'EMPTY_SET
            {
          $$.empty_set = true;
          $$.geom = NULL;
        }
     ;

prim_geom_expr:
     BOX '(' sum_expr ',' sum_expr ',' sum_expr ')'
            {
          GeomManager* g = GeomManager::instance();
          $$.geom  = box($3.value, $5.value, $7.value);
          $$.empty_set = false;
        }
     | SHEET '(' sum_expr ',' sum_expr  ')'
            {
          GeomManager* g = GeomManager::instance();
          $$.geom  = sheet($3.value, $5.value);
          $$.empty_set = false;
        }
     | LINE '(' sum_expr ',' sum_expr ',' sum_expr  ',' sum_expr ',' sum_expr ','
sum_expr ')'
            {
          GeomManager* g = GeomManager::instance();
           $$.geom  = line($3.value, $5.value, $7.value, $9.value, $11.value,
$13.value);
          $$.empty_set = false;
        }
     | CYLINDER '(' sum_expr ',' sum_expr ')'
            {
          GeomManager* g = GeomManager::instance();
          $$.geom  = cylinder($3.value, $5.value);
          $$.empty_set = false;
        }
     | CAPPED_CYLINDER '(' sum_expr ',' sum_expr ')'
            {
          GeomManager* g = GeomManager::instance();
          $$.geom  = capped_cylinder($3.value, $5.value);
          $$.empty_set = false;
        }
     | CONE '(' sum_expr ',' sum_expr ')'
            {
          GeomManager* g = GeomManager::instance();
          $$.geom  = cone($3.value, $5.value);
          $$.empty_set = false;
        }

     | CAPPED_CONE '(' sum_expr ',' sum_expr ')'
```

```
            {
            GeomManager* g = GeomManager::instance();
            $$.geom  = capped_cone($3.value, $5.value);
            $$.empty_set = false;
          }

      | CONE_FRUSTUM '(' sum_expr ',' sum_expr ',' sum_expr ')'
            {
            GeomManager* g = GeomManager::instance();
            $$.geom  = cone_frustum($3.value, $5.value, $7.value);
            $$.empty_set = false;
          }

      | CAPPED_CONE_FRUSTUM '(' sum_expr ',' sum_expr ',' sum_expr ')'
            {
            GeomManager* g = GeomManager::instance();
            $$.geom  = capped_cone_frustum($3.value, $5.value, $7.value);
            $$.empty_set = false;
          }

      | SPHERE '(' sum_expr ')'
            {
            GeomManager* g = GeomManager::instance();
            $$.geom  = sphere($3.value);
            $$.empty_set = false;
          }

      | ELLIPSOID '(' sum_expr ',' sum_expr ',' sum_expr ')'
            {
            GeomManager* g = GeomManager::instance();
            $$.geom  = ellipsoid($3.value, $5.value, $7.value);
            $$.empty_set = false;
          }
      ;

sum_expr:
      prod_expr
            {
            // Collect value from subexpression.
            $$.value = $1.value;
            }

      | sum_expr '+' prod_expr
            {
            GeomAscendPreprocessor* preprocessor
                = (GeomAscendPreprocessor*) context;

            // Compute sum of operands.
            $$.value = $1.value + $3.value ;
            }

      | sum_expr '-' prod_expr
            {
            GeomAscendPreprocessor* preprocessor
                = (GeomAscendPreprocessor*) context;

            // Compute difference of operands.
            $$.value = $1.value - $3.value;
            }
      ;


prod_expr:
      paren_expr
            {
            // Collect value from subexpression.
            $$.value = $1.value;
            }

      | prod_expr '*' paren_expr
            {
            GeomAscendPreprocessor* preprocessor
                = (GeomAscendPreprocessor*) context;

            // Compute product of operands.
            $$.value = $1.value * $3.value;
            }

      | prod_expr '/' paren_expr
            {
            GeomAscendPreprocessor* preprocessor
```

```
          = (GeomAscendPreprocessor*) context;

        // Compute ratio of operands.
        if ($3.value == 0.0 ) {
          // Divide by zero error occurred.
          preprocessor->collector()->append_count(
            GeomAscendPreprocessor::err_divide_by_zero,
            strlen(GeomAscendPreprocessor::err_divide_by_zero)
            );
          // Abort the parse.
          YYABORT;
        }
        else {
          $$.value = $1.value / $3.value;
        }
         }
    ;


paren_expr:
    REAL
        {
        // Collect value from scanner token.
        $$.value = $1.value;
         }

    | '-' REAL
        {
        GeomAscendPreprocessor* preprocessor
            = (GeomAscendPreprocessor*) context;

        // Negate value  collected from scanner token.
        $$.value *= -1.0;
         }

    | '+' REAL
        {
        // Collect value from scanner token.
        $$.value = $2.value;
         }

    | '(' sum_expr ')'
        {
        // Collect value from subexpression.
        $$.value = $2.value;
         }
    ;

%%


/* ---------------------------------------------------------------------------
 * C++ Code
 */


// ****************************************************************************
// *** int geom_ascend_lex(GeomAscendSymValues* lvalp, void* context)
// ****************************************************************************

int geom_ascend_lex(GeomAscendSymValues* lvalp, void* context)
{
    GeomAscendPreprocessor* preprocessor
     = (GeomAscendPreprocessor*) context;
    char c  // Next charcter in input buffer.
     = preprocessor->get_char();

    // Consume white space.
    while ( c == ' ' || c == '\t' )
     c = preprocessor->next_char();

    // Is this an identifier?
    if ( isalpha(c) || c == '_' ) {
      // Collect the token's label.
      lvalp->identifier.label = preprocessor->collect_identifier();

      if ( strcmp(lvalp->identifier.label, "p") == 0 )
     return EMPTY_SET;
      else if ( strcmp(lvalp->identifier.label, "union") == 0 )
     return UNION;
```

```
    else if ( strcmp(lvalp->identifier.label, "diff") == 0 )
   return DIFF;
    else if ( strcmp(lvalp->identifier.label, "intersect") == 0 )
   return INTERSECT;
    else if ( strcmp(lvalp->identifier.label, "box") == 0 )
   return BOX;
    else if ( strcmp(lvalp->identifier.label, "sheet") == 0 )
   return SHEET;
    else if ( strcmp(lvalp->identifier.label, "line") == 0 )
   return LINE;
    else if ( strcmp(lvalp->identifier.label, "cylinder") == 0 )
   return CYLINDER;
    else if ( strcmp(lvalp->identifier.label, "capped_cylinder") == 0 )
   return CAPPED_CYLINDER;
    else if ( strcmp(lvalp->identifier.label, "cone") == 0 )
   return CONE;
    else if ( strcmp(lvalp->identifier.label, "capped_cone") == 0 )
   return CAPPED_CONE;
    else if ( strcmp(lvalp->identifier.label, "cone_frustum") == 0 )
   return CONE_FRUSTUM;
    else if ( strcmp(lvalp->identifier.label, "capped_cone_frustum") == 0 )
   return CAPPED_CONE_FRUSTUM;
    else if ( strcmp(lvalp->identifier.label, "sphere") == 0 )
   return SPHERE;
    else if ( strcmp(lvalp->identifier.label, "ellipsoid") == 0 )
   return ELLIPSOID;
    else if ( strcmp(lvalp->identifier.label, "translate") == 0 )
   return TRANSLATE;
    else if ( strcmp(lvalp->identifier.label, "scale") == 0 )
   return SCALE;
    else if ( strcmp(lvalp->identifier.label, "rotate_x") == 0 )
   return ROTATE_X;
    else if ( strcmp(lvalp->identifier.label, "rotate_y") == 0 )
   return ROTATE_Y;
    else if ( strcmp(lvalp->identifier.label, "rotate_z") == 0 )
   return ROTATE_Z;
    else
   // Return as identifier.
   return IDENTIFIER;
   }

   // OR, is this an real?
   else if ( isdigit(c) || c == 'd' ) {
    // Collect real.
    lvalp->real.value = preprocessor->collect_real();

    // Return as natural number.
    return REAL;
   }

   // OR, is this end of input.
   else if ( c == '\n' || c == '\0' ) {
    return 0;
   }

   // Otherwise, return this as a character?
   else {
    int d = c;
    c = preprocessor->next_char();
    return d;
   }
} // int real_ascend_lex(RealAscendSymValues*, void*)


// ****************************************************************************
// *** int real_ascend_error(char* s)
// ****************************************************************************

int geom_ascend_error(char* s)
{
    // Error reporting via the error stream is suppressed.  Instead errors are
    // stored in the preprocessor's '_err_line' data member.  This behaviour
    // is explicitly implemented in the parser.

    // Mandatory return statement.
    return 0;
} // int real_ascend_error(char*)
```

B.2.4 types_with_geom_ascends.hh

```
// ---------------------------------------------------------------------------
//   Order Types                                                  Teng-Wen Chang
// ---------------------------------------------------------------------------
//
// OrderTypes/types_with_geom_ascends.hh
//   - Representation of ascends on the real number line (R), and proxy
//     class to extend an inheritance hierarchy with the informational order
//     of real ascends.
//


#ifndef OG_ORDERTYPES_TYPES_WITH_GEOM_ASCENDS_HH
#define OG_ORDERTYPES_TYPES_WITH_GEOM_ASCENDS_HH


// Included Header Files.

#ifndef KR_CON_INDEX_SET_HH
#include "Containers/index_set.hh"
#endif

#ifndef KR_CON_INDEX_INJECTION_HH
#include "Containers/index_injection.hh"
#endif

#ifndef KR_TFS_MANUAL_TYPES_HH
#include "FeatureStructures/manual_types.hh"
#endif

#ifndef KR_TFSSHELLS_DEFINITIONS_HH
#include "TFSShells/definitions.hh"
#endif

#include "TFSShells/preconditions.hh"


// Declaration of classes to be defined.

class TypesWithGeomAscends;// Inheritance hierarchy plus real ascends.


// ---------------------------------------------------------------------------
//   class TypesWithGeomAscends
//
//   Concrete class providing a proxy (or smart reference) to an inheritance
//   hierarchy of feature structure types.  This proxy augments the subject
//   inheritance hierarchy with the informational order of real ascends.
//
//   The "TypeHierarchy" methods "identifier()", "join()", and "meet()" act to
//   pick out real ascend types.  This role is discussed in
//   "FeatureStructures/type_hierarchy.hh".  It is accompanied by a type
//   resource management scheme that holds reference counts on real
//   ascend types.  In addition, the method "is_label()" syntactically
//   identifies labels of real ascend types.  So that the "identifier()"
//   precondition is satisfied by all syntactically correct labels.
//
//   The collection of real ascend types form a BCPO in their own right,
//   since meet is defined as smaller, for all but the empty set of ascends.
//   This structure is inserted above the universal type, and topped by the
//   absurd type.  Therefore, the inf ascend is represented by the absurd
//   type.
// ---------------------------------------------------------------------------

class TypesWithGeomAscends
    : public ManualTypes
{
public:

    TypesWithGeomAscends(ManualTypes* real_types);
    bool pre_TypesWithGeomAscends(ManualTypes const* real_types) const
        {
            return real_types != NULL;
        }
        // Construct a proxy whose real subject is the type hierarchy
        // 'real_types', and which augments the inheritance hierarchy in
```

```
        // 'real_types' by the real ascends.  Take sole ownership of the
        // real subject.

    virtual ~TypesWithGeomAscends();
    bool pre_D_TypesWithGeomAscends() const
        {
            return InheritanceHierarchy::pre_D_InheritanceHierarchy();
        }
     // Free this proxy and the real subject.  Free the representations of
     // all ascend types.


     // "TypeHierarchy" Interface.
     //
     // "TypesWithRealAscends" delegates these methods to the real subject.
     // It intervenes to test for real ascend type labels and identifiers,
     // as described in the "TypeHierarchy" interface.

     virtual bool is_type(Index t) const;
     virtual bool are_types(IndexSet const& T) const;
     virtual bool is_regular_type(Index t) const;
     virtual bool are_regular_types(IndexSet const& T) const;
     virtual bool is_universal_type(Index t) const;
     virtual bool is_absurd_type(Index t) const;
     virtual bool is_label(char const* l) const;
     virtual Index universal() const;
     virtual Index absurd() const;
     virtual Index identifier(char const* l) const;
     virtual char const* label(Index t) const;
     virtual bool subsumes(Index s, Index t) const;
     virtual bool comparable(Index s, Index t) const;
     virtual Index join(Index s, Index t) const;
     virtual Index meet(Index s, Index t) const;
     virtual Index accept_type(Index t) const;
     virtual void relinquish_type(Index& t) const;


     // "InheritanceHierarchy" Interface.
     //
     // "TypesWithRealAscends" simply delegates these methods to the real
     // subject.

     virtual bool is_succession_type(Index t) const;
     virtual bool are_succession_types(IndexSet const& T) const;
     virtual bool is_leaf_succession_type(Index t) const;
     virtual bool are_sound_parents(IndexSet const& T) const;
     virtual IndexSet* succession_types() const;
     virtual IndexSet* maximal_succession_types() const;
     virtual IndexSet* children(Index t) const;
     virtual IndexSet* parents(Index t) const;
     virtual Index insert(char const* l, IndexSet const& S);
     virtual void remove(Index t);
     virtual bool operator == (InheritanceHierarchy const& types) const;
     virtual bool operator != (InheritanceHierarchy const& types) const;


private:

     KR_DECLARE_PRECONDITION_SUPPORT;
      // Precondition checking state.

     ManualTypes* _real_types;
         // Real subject defining inheritance hierarchy over the succession
         // types.

     IndexInjection _label_to_ascend;
      // Injection:
         //   real ascend   <-->  geom ascend
      //   type label          representation
      //   (char const*)       (Association<GeomAscend, size_t>*)


}; // class TypesWithGeomAscends


#endif
```

B.3 Region

B.3.1 region.hh

```
// ------------------------------------------------------------------------------
//   Geometry                                                        Teng-Wen Chang
// ------------------------------------------------------------------------------
//
// Geometry/region.hh
//   - Representation of geometry objects, using XOX SHAPES.
//
//   - a Region is either an empty_pset/universal_pset or an non-empty_pset,
//       which is represented by the NULL/non-NULL pointer of local data _xg.
//
//   All "Region" objects constructed on heap:
//   - constructors are protected so that only the methods of this class
//     can use them;
//   - static methods provided to construct objects on the heap.
//   - see "FeatureStructures/struct_node.hh" and
//     "FeatureStructures/desc_node.hh" for examples.
//
// public:
//     static Region* geom_region(xGEOM geom);
//     static Region* union_region(Region const* a, Region const* b);
//     static Region* intersection_region(Region const* a, Region const* b);
//     static Region* difference_region(Region const* a, Region const* b);
//
// $Id$


#ifndef OG_GEOM_REGION_HH
#define OG_GEOM_REGION_HH

// Include the header files

#include "Shapes/ShapesGeom.h"

// Include the region initialisation file.
#include "region_init.hh"

class Region {
private:
  xGEOM _xg; // the geometric object

protected:

  // helper function for persistent cells
  void attach_group(xGROUP);
  bool merge(Region const*);

  // a helper function for retrieving the cell list from a group which
  // contains feature group list
  xCELLS Region::get_cells_from_grp(xGROUP grp);


  // access the internal data.
  // if do not want to modify the object, using const_pset(), otherwise,
  //   pset()
  inline xGEOM const_pset() const { return _xg; }
  inline xGEOM pset() { return _xg; }

  // create a Region class representing an empty pointset.
  Region():_xg(NULL){ }

  // a protected constructor: ensure the input is a xCELL--the end node of
  // a group graph.
  Region::Region(xCELL cell);

  // a protected constructor: ensure the input is a xGROUP.
  // the xGROUP is might be three cases:
  //  1. a feature group which contains the xCELLs.
  //  2. a feature group which has no region_attrib
  //  3. a plain group which contains a set of feature groups.
  //  4. a null group, which is an error.
  Region::Region(xGROUP grp);

 public:

  ~Region();
```

```
// ------------------------------
// The public construction methods

// this function takes a geometry object and return a Region object
// contains that geometry object.
static Region* geom_region(xGEOM geom);


// ***********
// The Boolean operators which return a new Region object as the result
// of Boolean operation.

// Union:
// this function takes two Regions (scra, scrb) and return a Region
// object contains the unioned geometric object, if sucess. otherwise,
// the result is a NULL pointer.
static Region* union_region(Region const* a, Region const* b);


// Intersection:
// this function takes two Regions (scra, scrb) and return a Region
// object contains the intersected geometric object, if sucess. otherwise,
// the result is a NULL pointer.
static Region* intersection_region(Region const* a, Region const* b);

// difference:
// this function takes two Regions (scra, scrb) and return a Region
// object contains the diffed geometric object, if sucess. otherwise,
// the result is a NULL pointer.
static Region* difference_region(Region const* a, Region const* b);



// --------------------------------
// transformation of Geometry object

// translate:
//
//   this function translate this object(_xg) via a vector (x, y, z)
//   return true if success, otherwise false.
bool translate(double x, double y, double z);

// scale:
//
//   this function scale this object(_xg) with a factor, and at a base
//    point at (px, py, pz)
//   return true if success, otherwise false.
bool scale(double factor, double px, double py, double pz);

// rotateX:
//
//   this function rotate this object(_xg) around X-axis with a radians,
//    and a basis point at (px, py, pz)
//   return true if success, otherwise false.
bool rotateX(double radians, double px, double py, double pz);

// rotateY:
//
//   this function rotate this object(_xg) around Y-axis with a radians,
//    and a basis point at (px, py, pz)
//   return true if success, otherwise false.
bool rotateY(double radians, double px, double py, double pz);

// rotateX:
//
//   this function rotate this object(_xg) around Z-axis with a radians,
//    and a basis point at (px, py, pz)
//   return true if success, otherwise false.
bool rotateZ(double radians, double px, double py, double pz);

// TWC: need more work  bool mirror(double mirrpx, double mirrpy, double mirrpz,
);

// ---------------------------------
// attribute function.
//
// attribute(name, value)

// ----------------------------------
// other functions:
```

```
  // pointset inclusion.
  //   this function return true if this object is pointset-included by 'r'.
  bool operator<= (Region const& r) const;

  // print the geometry to a file called 'name'.
  // return true if success, otherwise false.
  bool print_geom(char* name) const;


  // is_empty()
   bool is_empty() const;
} ;


#endif
```

C
Single-Fronted cottage

C.1 An ALE most general satisfier output

```
MOST GENERAL SATISFIER OF: sfc_house

sfc_house
BATHROOM [14] massing
         GEOMETRY cell_geom
                  LL cell_d
                     X [22] p_int
                     Y [20] p_int
                  UR cell_a
                     X [21] p_int
                     Y [19] p_int
DINING [10] massing
       GEOMETRY cell_geom
                LL cell_d
                   X [2] a1
                   Y [9] p_int
                UR cell_a
                   X [0] a2
                   Y [3] p_int
GEOMETRY cell_geom
         LL cell_d
            X [1] a0
            Y [12] a0
         UR cell_a
            X [0]
            Y [5] a5
HALL [6] massing
     GEOMETRY cell_geom
              LL cell_d
                 X [1]
                 Y [7] a1
              UR cell_a
                 X [2]
                 Y [4] a4
KITCHEN [13] massing
         GEOMETRY cell_geom
                  LL cell_d
                     X [26] p_int
                     Y [24] p_int
                  UR cell_a
                     X [25] p_int
                     Y [23] p_int
LOUNGE [11] massing
        GEOMETRY cell_geom
                 LL cell_d
                    X [2]
                    Y [3]
                 UR cell_a
                    X [0]
                    Y [4]
MASS_EL1 [15] massing
         GEOMETRY cell_geom
                  LL cell_d
```

```
                    X [1]
                    Y [4]
                UR cell_a
                    X [0]
                    Y [5]
MASS_EL2 [6]
MASS_EL3 [16] massing3
        GEOMETRY cell_geom
                LL [8] cell_d
                    X [2]
                    Y [7]
                UR cell_a
                    X [0]
                    Y [4]
        MASS_EL1 [18] massing
                GEOMETRY cell_geom
                        LL [8]
                        UR cell_a
                            X [0]
                            Y [9]
        MASS_EL2 [10]
        MASS_EL3 [11]
MASS_EL4 [17] massing2
        GEOMETRY cell_geom
                LL cell_d
                    X [1]
                    Y [12]
                UR cell_a
                    X [0]
                    Y [7]
        MASS_EL1 [13]
        MASS_EL2 [14]
PORCH [15]
ROOM_ROW [16]
SKILLION [17]
SLEEPING [18]
[4]  ==  [3]
,
[10]  ==  [14]
,
[13]  ==  [14]
,
[13]  ==  [10]
,
[11]  ==  [14]
,
[11]  ==  [10]
,
[11]  ==  [13]
,
[18]  ==  [14]
,
[18]  ==  [10]
,
[18]  ==  [13]
,
[18]  ==  [11]
,
[9]  ==  [7]
,
[19]  ==  [20]
,
[21]  ==  [22]
,
[23]  ==  [24]
,
[25]  ==  [26]
,
[3]  ==  [9]
,
[9]  ==  [7]
,
[4]  ==  [3]
,
[23]  ==  [24]
,
[25]  ==  [26]
,
[3]  ==  [9]
,
[19]  ==  [20]
```

[21] == [22]

D

Index of Symbols and Expressions

FU, 84	a functional unit
DU, 84	a design unit
BCPO, 94	a bounded complete partial order
Type, 94	The basic element of Typed Feature Structures, which are organised in a multiple inheritance hierarchy.
$\wedge$, 94	a join symbol, or most general common specialisation
$\vee$, 94	a meet symbol, or the most specific common generalisation
$\perp$, 94	A universal *type* is described as Bottom, at which all the *types* meet
Intro(f), 95	feature introduction
$<Type, \sqsubseteq>$, 95	an inheritance hierarchy
Feat, 95	a symbol for the feature
$\sigma \sqsubseteq \tau$, , 95	subsumption relation between σ, τ.
$Approp(f, \sigma)$, 95	An appropriateness specification over f, σ.
$\langle Q, \bar{q}, \theta, \delta \rangle$, 96	a tuple for defining a *feature structure*.
Q, 96	a finite set of nodes rooted at $\bar{q}$.
$\bar{q}$, 96	the root node.
$\theta: Q \rightarrow$ Type, 96	a total node typing function
δ: Feat $\times Q \rightarrow$ Q, 96	a partial feature value function
F @ π, 97	path value
Desc, 97	description
$\pi : \phi$, 97	path with an end node as ϕ
$\pi_1 = \pi_2$, 97	equality of two paths
$\phi \wedge \psi$, 97	'and' equation of two descriptions
$\phi \vee \psi$, 97	'or' equation of two descriptions
$\mathcal{F}$, 98	the collection of *feature structures*
$F \models \phi$, 98	satisfaction relation. called F satisfies *type* ϕ

$A \sim B$, 99	alphabetic variance
$\bar{q} \bowtie \bar{q}'$, 100	equivalence relation on $Q \cup Q'$
$F \sqcup F'$, 100	unification of F *and* F'
const(t), 110	constraints of *type t*
$\wp(s)$, 114	power set of a set s
$S \otimes t$, 114	every subset of S is subsumed by t
$\bigsqcup S$, 114	the least upper bound of S
$\Re$, 118	real numbers
$\langle \Re, \bullet \rangle$, 118	an ordered set over real numbers
$\overline{\Re_\perp}$, 118	lifted reals
$\Re_\perp$, 118	real numbers plus a bottom
$\langle \Re_\perp, \bullet \rangle$, 118	an ordered set over real numbers and a bottom
Z, 121	integers
$\langle Z, \bullet \rangle$, 121	an ordered set over integers
$\overline{Z_\perp}$, 121	lifted integers
$Z_\perp$, 121	integers plus a bottom
$\langle Z_\perp, \bullet \rangle$, 121	an ordered set over integers and a bottom
N_0, 124	the whole numbers (natural numbers plus zero)
$\langle N_0, \bullet \rangle$, 124	an ordered set over whole numbers
$\overline{N_{0_\perp}}$, 124	lifted whole numbers
$N_{0_\perp}$, 124	whole numbers plus a bottom
$\langle N_{0_\perp}, \bullet \rangle$, 124	an ordered set over whole numbers and a bottom
$\langle \Re, \leqslant \rangle$, 127	an ordered set over real numbers with "less than or equals" relation
$\Re_{\perp_\leqslant}$, 127	ascending real numbers
$\Re_\leqslant$, 127	ascending real numbers without the bottom
$\Re_\uparrow$, 127	ascending reals
$\langle \Re, \geqslant \rangle$, 130	an ordered set over real numbers with "greater than or equals" relation
$\Re_{\perp_\geqslant}$, 130	descending reals
$\Re_\geqslant$, 130	descending reals without the bottom
$\Re_\downarrow$, 130	descending reals
$\langle Z, \leqslant \rangle$, 133	an ordered set over integers with "less than or equals" relation

$Z_{\perp_{\leqslant}}$, 133	ascending integers
$Z_{\leqslant}$, 133	ascending integers without the bottom
$Z_{\uparrow}$, 133	ascending integers
$\langle Z, \geqslant \rangle$, 136	an ordered set over integers with "greater than or equals" relation
$Z_{\perp_{\geqslant}}$, 136	descending integers
$Z_{\geqslant}$, 136	descending integers without the bottom
$Z_{\downarrow}$, 136	descending integers
$\langle N_0, \leqslant \rangle$, 139	an ordered set over whole numbers with "less than or equals" relation
$N_{0_{\leqslant}}$, 139	ascending whole numbers
$N_{0_{\uparrow}}$, 139	ascending whole numbers
$\langle N_0, \geqslant \rangle$, 142	an ordered set over whole numbers with "greater than or equals" relation
$N_{0_{\perp_{\geqslant}}}$, 142	descending whole numbers
$N_{0_{\geqslant}}$, 142	descending whole numbers without the bottom
$N_{0_{\downarrow}}$, 142	descending whole numbers
$\Re_{\uparrow\downarrow}$, 145	real intersection intervals
$\langle [\Re_{\uparrow}, \Re_{\downarrow}], \supseteq \rangle$, 145	an ordered set over a set of $[\Re_{\uparrow}, \Re_{\downarrow}]$ with set interval inclusion
val(a), 145	a polymorphic function giving the corresponding number to a domain, such as $\Re_{\downarrow}$ or $Z_{\uparrow}$.
Q^{u}, 146	a set of least upper bound for every elements in Q
$Z_{\uparrow\downarrow}$, 148	integers intersection intervals
$\langle [Z_{\uparrow}, Z_{\downarrow}], \supseteq \rangle$, 148	an ordered set over a set of $[Z_{\uparrow}, Z_{\downarrow}]$ with set interval inclusion
$N_{0_{\uparrow\downarrow}}$, 151	the whole numbers intersection intervals
$\langle [N_{0_{\uparrow}}, N_{0_{\downarrow}}], \supseteq \rangle$, 151	an ordered set over a set of $[N_{0_{\uparrow}}, N_{0_{\downarrow}}]$ with set interval inclusion
$\cup^{*}$, 160	The regularised pointset union operation (Boolean operation), which is polymorphic with set union
$\cap^{*}$, 160	The regularised pointset intersection operation (Boolean operation), which is polymorphic with set intersection
$-^{*}$, 160	the regularised pointset difference operation (Boolean operation), which is polymorphic with set difference operation

CSG, 162	constructive solid geometry
PSet, 163	the set of all connected point sets
$\langle PSet, \bullet \rangle$, 163	An ordered set over *PSet* under lifted relation.
$PSet_{\perp}$, 163	the union set of *PSet* and a bottom.
$\overline{PSet_{\perp}}$, 163	lifted point-sets
$\langle PSet, \subseteq \rangle$, 166	An ordered set over *PSet* under the point-set subset relation.
$PSet_{\subseteq}$, 166	the growing point-sets
IPSet, 166	the growing point-sets
$\bigcup Q$, 167	all the points in every member of Q.
Exp, 167	expression
$\langle PSet, \supseteq \rangle$, 169	An ordered set over *PSet* under the point-set superset relation.
$PSet_{\supseteq}$, 169	the shrinking point-sets
OPSet, 169	the shrinking point-sets
$\bigcap Q$, 169	all the common points in every member of Q.
$\langle [PSet_{\subseteq}, PSet_{\supseteq}], \circledcirc \rangle$, 172	An ordered set over $[PSet_{\subseteq}, PSet_{\supseteq}]$ under inner growing outer shrinking relation
$PSet_{\circledcirc}$, 172	*IOSET* intervals
pset(a), 172	the point-set value of a
IGOSet, 172	inner growing outer shrinking point-sets
IOPSet, 172	inner-outer point-sets
$\langle [PSet_{\supseteq}, PSet_{\subseteq}], \bullet\!\!\circ \rangle$, 176	An ordered set over $[PSet_{\supseteq}, PSet_{\subseteq}]$ under outer growing inner shrinking relation
$PSet_{\bullet\!\circ}$, 176	*OGISET* intervals
OGISet, 176	outer growing inner shrinking point-sets
OIPSet, 176	outer-inner point-sets