Unfolding Design Spaces Interactively

by
Sambit Datta

Supervisor
Robert Francis Woodbury

A dissertation submitted in fulfillment
of the requirements for the Degree of Doctor of Philosophy
in the
School of Architecture, Landscape Architecture and Urban Design
at
The University of Adelaide

Adelaide, Australia
June 27, 2004

Declaration

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.

Date 27 JUNE '04

Sambit Datta

Abstract

Design is an iterative process of specifying problems, finding plausible solutions, judging the validity of solutions relative to problems and reformulating problems and solutions. Computational exploration requires formal mechanisms and human computer interaction models for supporting designing. The theory of design space exploration posits a formal substrate for representing and generating designs. To integrate the user in design space exploration, an interaction model that combines the role of the the designer and the formalism is necessary.

This thesis addresses the problem of interaction between an exploration formalism and the designer through the paradigm of *mixed-initiative*. The thesis develops a mixed-initiative interaction model for design space exploration in three layers, *domain*, *task* and *dialogue*. The domain layer supports the coordination of the designer's view of exploration in terms of problems, solutions, choices and exploration history with the concepts of state, move and structure available in the formal substrate. The dialogue layer supports communication between the designer and the formalism in terms of a shared visual notation for representing and integrating input and output from both modes of exploration. Through the dialogue layer the designer and the formalism can communicate the intermediate results of exploration. The task layer supports interaction with the operators for moving in a design space. Through the task layer the designer and the formalism can acquire, transfer and relinquish control of the exploration process to generate, navigate and synchronise exploration states. The interaction model is implemented as $\mathcal{FOLDS}$, or the *$\mathcal{F}$oldability $\mathcal{O}$f $\mathcal{L}$arge $\mathcal{D}$esign $\mathcal{S}$paces*. An example from the domain of architectural design, three-dimensional massing configurations, demonstrates the components of $\mathcal{FOLDS}$.

The mixed-initiative interaction model developed in the thesis presents a new approach for integrating the role of the designer and a description formalism in computational exploration. The model enables the designer to maintain exploration freedom in terms of formulating and reformulating problems, generating solutions, making choices and navigating the history of exploration. It permits a fine granularity of interaction through incremental turn-taking, allowing the designer and the formalism to communicate, coordinate and control each step in the process of computing exploration.

Acknowledgements

I am truly grateful to Professor Robert Woodbury, my thesis supervisor. He introduced me to the problems underlying design exploration and helped nurture my nascent ideas within a stimulating research environment in Adelaide.

Andrew Burrow and Teng-wen Chang, my colleagues in Adelaide for the many discussions on the science behind computational exploration, feature structure theory and their implications for design support. In particular, Andrew's implementation of typed feature structures, KRYOS provided a solid foundation for understanding the machinery of design space exploration. Teng-wen, with whom I shared an office for four years, helped my understanding of Genesis, SEED and SEED-Config.

The postgraduate community, staff and faculty in the School of Architecture, Landscape Architecture and Urban Design at Adelaide University, provided a generous and nurturing environment. Peter Scriver, in his role as post-graduate co-coordinator, organised valuable postgraduate seminars. Anthony Radford, Terry Williamson and Veronica Soebarto for their encouragement, support and administrative assistance during the long period of candidature. The Overseas Postgraduate Research Scholarship (OPRS) programme, Graduate Studies and International Programs of Adelaide University for providing financial and administrative support. The support staff of IPD Systems Pty. Ltd. for providing and maintaining my computing environment in Adelaide.

I would also like to acknowledge Jonah Tsai, Sheng-fen Chien (Nik), Mikako Harada, Rana Sen, Ramesh Krishnamurti and Ulrich Flemming in Carnegie-Mellon University. The staff, faculty and administration in Deakin University, who created a collegial, flexible and supportive working environment. Tim Smithers provided valuable insights on improving many deficiencies in the thesis.

Finally, Biraj and Neli for their support and understanding. Sonal, Sagar and Srija, my dear family, for being there every single day.

Contents

List of Tables ix

List of Figures x

I DESIGN SPACE EXPLORATION 1

1 Introduction 3

1.1 Design as exploration . . . 3

1.1.1 Accounts of exploration . . . 3

1.1.2 Characteristics of Exploration . . . 8

1.2 Accounts of interaction . . . 10

1.2.1 Interaction paradigms . . . 12

1.2.2 Characteristics of interaction . . . 14

1.3 Mixed-initiative interaction . . . 15

1.4 Research hypothesis . . . 19

1.5 Summary . . . 20

2 An exploration formalism 21

2.1 Entities of exploration . . . 21

2.2 Representation of exploration states . . . 22

2.2.1 Types and features . . . 23

2.2.2 Descriptions . . . 27

2.2.3 Feature structures . . . 28

2.3 Ordering of exploration structure . . . 30

2.4 Algorithms for exploration moves . . . 32

2.4.1 Incremental π-resolution . . . 32

2.4.2 Indexing and reuse . . . 34

2.4.3 Hysterical undo . . . 35

2.4.4 Design unification . . . 36
2.4.5 Design anti-unification . . . 37
2.5 Summary . . . 38

3 Mixed-initiative Interaction 39
3.1 Interaction with a description formalism . . . 39
3.1.1 Unfolding design spaces . . . 40
3.1.2 Requirements . . . 40
3.1.3 The role of mixed-initiative . . . 42
3.2 The Domain Layer . . . 43
3.2.1 Attributes of domain . . . 43
3.2.2 Extensions . . . 44
3.2.3 Role of the domain layer . . . 45
3.3 The Task Layer . . . 45
3.3.1 Attributes of task initiative . . . 46
3.3.2 Extensions . . . 46
3.3.3 Role of the task layer . . . 47
3.4 The Dialogue Layer . . . 48
3.4.1 Attributes . . . 48
3.4.2 Extensions . . . 49
3.4.3 Role of the dialogue layer . . . 50
3.5 Summary . . . 51

II MIXED-INITIATIVE AND DESIGN SPACE EXPLORATION 52

4 A mixed-initiative domain layer 54
4.1 Representation of the domain . . . 54
4.1.1 The designer's view of exploration . . . 54
4.1.2 Domain layer constructs . . . 55
4.1.3 Mapping to description formalism . . . 56
4.2 Problem State . . . 58
4.2.1 The designer's view . . . 58
4.2.2 The symbol substrate . . . 59
4.2.3 The domain layer construct . . . 60
4.3 Solution state . . . 61
4.3.1 The designer's view . . . 61
4.3.2 The symbol substrate . . . 62

4.3.3 The domain layer construct 62
4.4 Feature Node 64
4.4.1 The designer's view 64
4.4.2 The symbol substrate 64
4.4.3 The domain layer construct 65
4.5 Satisfier Space 68
4.5.1 The designer's view 68
4.5.2 The symbol substrate 69
4.5.3 The domain layer construct 69
4.6 Summary 71

5 Mixed-initiative Dialogue **72**
5.1 The dialogue layer 72
5.2 Representation of dialogue 74
5.2.1 A visual notation 76
5.2.2 Choices 77
5.2.3 Interaction with visual feature nodes 80
5.3 Integration of Dialogue 83
5.3.1 Supporting partiality 83
5.3.2 Supporting structure sharing 85
5.3.3 Supporting dialogue integration 86
5.3.4 Supporting dialogue disambiguation 87
5.3.5 Supporting multiple modes 88
5.4 Summary 90

6 A mixed-initiative task layer **91**
6.1 The Task layer 91
6.2 The task of generation 93
6.2.1 The CONSTRUCT operation 94
6.2.2 The EXTEND operation 95
6.3 The task of Navigation 96
6.3.1 The CHOOSE operation 96
6.3.2 The RETRACT operation 98
6.4 The task of synchronisation 99
6.4.1 The RECALL operation 99
6.4.2 The ERASE operation 101
6.4.3 The JOIN and MEET operations 102

6.5 Summary 104

III *FOLDS* : FOLDABILTIY OF LARGE DESIGN SPACES 106

7 Enabling mixed-initiative exploration 108

7.1 *FOLDS* 108

7.2 Domain interaction 110

7.2.1 PState and SState 110

7.2.2 FNode and SatSpace 113

7.3 Dialogue in exploration 115

7.3.1 Unfolding visual feature nodes 115

7.3.2 Implementation 118

7.4 Tasks of exploration 119

7.4.1 Extending feature nodes 120

7.4.2 Choice in exploration 122

7.4.3 Retract 126

7.4.4 Reuse of past exploration 128

7.4.5 Erasure 128

7.4.6 Joins and Meets 131

7.5 Discussion 132

7.6 Summary 134

8 Conclusions 135

8.1 Mixed-initiative Exploration 135

8.1.1 Assumptions 135

8.1.2 Research hypothesis 136

8.2 Contributions of the thesis 138

8.2.1 Feature node representation 138

8.2.2 Visual notation for feature nodes 139

8.2.3 Unfolding operations 140

8.3 Constraints on the results 141

8.4 Future directions 143

A Typed Feature Structures 145

A.1 Terminology and definitions 145

A.1.1 Type Hierarchy 145

A.1.2 Feature Structures 146

A.1.3 Type System 147
A.1.4 Descriptions 147
A.1.5 Type constraints 147
A.1.6 Incremental π-resolution 148

B UML Notation **153**
B.1 Modelling concepts 154
B.1.1 Symbols 154
B.1.2 Relationships 155
B.1.3 Diagrams 156

C Massing Configurations in $\mathcal{FOLDS}$ **158**
C.1 An inheritance hierarchy of types 158
C.2 Appropriateness specifications 163
C.3 Constraint declarations 167
C.4 Descriptions 171

D Design and Implementation Details **172**

List of Tables

7.1 A collection of path equations in the form of a conjunctive description. 113
7.2 Path descriptions from the massing example presented as a collection of disjunctive descriptions. 125

List of Figures

1.1 The Knowledge level concepts of SEED showing a simple representation of their relationships. . . . 7

2.1 The exploration entities, *state, structure, move* are mapped onto the description formalism. . . . 22
2.2 Types features, descriptions and the relationships of subsumption and unification. . 23
2.3 A representation of the entities in type hierarchy. The types stand for domain knowledge . . . 24
2.4 Another example of an inheritance hierarchy of types $\langle Type, \sqsubseteq \rangle$. . . . 25
2.5 An example of types showing how features are introduced, marked with $*$ and their inheritance by subsumption. . . . 26
2.6 An example of a feature structure. . . . 28
2.7 An illustration of feature structure unification. . . . 30
2.8 A pair of simple feature structures in a subsumption relation. . . . 31
2.9 Figure showing the computation paths in the scheme. . . . 32
2.10 he generating procedure,π-resolution captures a relation from descriptions to the satisfiers. . . . 33
2.11 Indexing and reuse are by-products of the underlying subsumption ordering. . . . 35
2.12 Exploration through information removal. . . . 36
2.13 Exploration by combining partial designs. . . . 36
2.14 Exploration by design anti-unification. . . . 37

3.1 An interaction model integrates the user and the description formalism. . . . 39
3.2 The components of a mixed-initiative formulation for interaction between the user and a description formalism. . . . 42

4.1 The designer's view of exploration. . . . 55
4.2 Mapping the designer's view of exploration to constructs in the domain layer. . . . 55
4.3 Mapping domain layer constructs to the underlying formal substrate of the design space exploration formalism. . . . 58

4.4 The problem state composes a collection of descriptions, Desc over a type system. . . 60
4.5 Types, Features, constraints and descriptions comprise the representation layer. . . . 61
4.6 Solution states compose partial designs with respect to an inheritance hierarchy of types and a partial satisfier. 63
4.7 Realisation of the design as a typed feature structure. 63
4.8 The feature node composes the relationship between the problem state and the partial satisfier. 65
4.9 Feature nodes compose operators, their arguments and the current resolution state. . 66
4.10 Representing the extrinsic attributes of the feature node, FNode. 67
4.11 The collection of feature nodes represent the satisfier space. 69
4.12 Independence of satisfier space and design space. 70

5.1 Mapping domain layer constructs to the dialogue layer through the visual feature node. 73
5.2 The elements of the visual feature node map onto the domain layer constructs. . . . 74
5.3 Feature node in directed acyclic graph notation. 75
5.4 Feature structure in AVM notation. 76
5.5 A feature value pair and a feature value map. 77
5.6 A visual feature node with coreference notation. 77
5.7 An example of a conjunct of disjunctive visual feature nodes. 78
5.8 The resultant visual feature node arising out of the resolution of disjunctive nodes by user interaction. 78
5.9 Encoding a function as a *feature-value map*. 79
5.10 Representation of a function as a visual feature node. 80
5.11 Unfolding a visual feature node representing a function. 80
5.12 An example of an imploded feature node. 81
5.13 Another example of an imploded feature node. 81
5.14 An example of substitution of a *feature-value map* with a co-reference tag. 82
5.15 Visual feature node for underspecified entity of type *massing*. 84
5.16 Visual feature node for underspecified entity of type *command*. 84
5.17 Substructures can be shared in a visual feature node. 85
5.18 Feature node of type *massing* (top) with two options *location* and *command* for disambiguation of feature LOCATION. 89

6.1 The exploration tasks are specified over a *visual feature node*. 92
6.2 The task of generation. 93
6.3 Interaction sequence of the CONSTRUCT operation. 94
6.4 Interaction with the EXTEND operation. 96

6.5 The task of navigation. 97
6.6 Interaction with the CHOOSE operation. 97
6.7 A RETRACT operation over the design space. 98
6.8 The task of synchronisation. 100
6.9 The RECALL operation uses the underlying subsumption ordering over feature nodes. 100
6.10 Erasure in design space exploration. 102
6.11 Representation of a JOIN exploration operator. 103
6.12 Representation of a MEET operation. 104

7.1 An example of the inheritance hierarchy of types. 110
7.2 Representation of massing elements. 112
7.3 A problem statement in the description language of KRYOS. 112
7.4 The figure shows a SState labelled by the type *sfc_roomrow_two* uncovered in the exploration of the problem. 114
7.5 The FNode with the label *Satspace Element*. 114
7.6 Implementation of interactive unfolding. 116
7.7 Implementation of a visual feature node in $\mathcal{FOLDS}$ 117
7.8 Interactive node operations in $\mathcal{FOLDS}$ use the signal/slot mechanism. 118
7.9 Interactive node unfolding in $\mathcal{FOLDS}$ showing in the DesignSpace Module. 119
7.10 An example of a description query. 120
7.11 The initial query is extended introducing new features. 121
7.12 The exploration is advanced by user interaction. 121
7.13 An EXTEND operation in $\mathcal{FOLDS}$ 123
7.14 The exploration of a disjunctive query description. 124
7.15 Interface to resolving disjunctions with the CHOICE operation in exploration. 126
7.16 Path retraction enables the designer to move backwards from the currently selected substructure. 127
7.17 Using the RECALL operation. 129
7.18 The ERASE operation allows the designer to remove massing entities. 130
7.19 The MEET operation allows the designer to explore multiple paths of exploration. . . 131

A.1 The generating procedure, π-resolution captures a relation from descriptions to satisfiers. 150
A.2 An antichain lattice, Ω comprises the down set of types. 151

B.1 Symbols in UML notation. 154
B.2 Relationhips in UML notation. 155
B.3 Class diagram in UML notation. 157

B.4 Sequence diagram in UML notation. 157

D.1 The implementation of the domain layer constructs. 174
D.2 The ScModule Class and its context 175
D.3 The organisation of the ScConsoleModule class 176
D.4 The organisation of the ScEntryModule class 177
D.5 $\mathcal{FOLDS}$ interface to the Kryos Feature Structure System. 178
D.6 The organisation of the ScDesignSpaceModule class 178
D.7 The ScDesignSpaceModule in $\mathcal{FOLDS}$ 179

Part I

DESIGN SPACE EXPLORATION

Part I: Design Space Exploration

"The natural sciences are concerned with how things are. Design on the other hand is concerned with how things ought to be ..."
Herbert Simon [Simon 1969, p 58].

Part I identifies the research problem, the assumptions underlying the study and the requirements for addressing the research hypothesis. It is broken into three Chapters as follows:

- Chapter 1 reviews the cognitive and computational accounts of exploration. In particular, it examines design space description formalisms and interaction environments developed for supporting computational exploration.
- Chapter 2 describes a design space description formalism. In this formalism, exploration is cast as a formal, computable process of movement in an ordered structure comprising collections of partial solutions. Algorithms for generating, ordering and reasoning over partial solutions are supported. It describes the formal concepts and features of the design space description formalism and identifies the potential for extending this paradigm of exploration with a model of interaction.
- Chapter 3 addresses the requirements for a mixed-initiative interaction model for design space exploration. Mixed-initiative supports a creative integration of user manipulation and automated services, characterised by an effort to implement deeper, more natural collaborations between users and computers.

Chapter 1

Introduction

Chapter 1 reviews the exploration model of designing and establishes the scope and meaning of the terms, *exploration*, *interaction* and *description formalism.* It identifies the requirements necessary for addressing the role of interaction in computational exploration. Based on these requirements, it proposes a research hypothesis for combining interactive and generative approaches into a mixed-initiative model of exploration.

1.1 Design as exploration

1.1.1 Accounts of exploration

The scope and meaning of the term "exploration" used in the design research literature is varied. It depends upon the theories, models and techniques proposed for its support. For example, in cognitive accounts of designing [Schön 1983, Schön 1988, Schön & Wiggins 1992] exploration is seen as a form of human cognitive activity. They are primarily concerned with explaining and interpreting the cognitive processes at play during real world designing. They contend that the relationships between seeing and moving lie at the heart of designing and that exploration emerges out of the interaction between designing (acting) and discovering (reflecting). In the context of computational exploration, it is necessary to establish a more technical meaning of the term, exploration, through a review of computational accounts of designing.

State space models

Exploration as a computational process has its provenance in the state-space search model of human problem solving. Simon [1969] proposes designing as a process of search within a state space representation of a problem. The state space represents all possible solution states of a design description. The state-space search [Newell & Simon 1972] model for human problem solving underpins the view of designing as a problem solving activity. The set of all possible problem definitions is referred to as the *problem space*. The set of all possible solutions to a particular problem space is referred to as the *solution space*. Design is a problem-solving activity, comprising the act of searching through the state space of possible solutions to a well-defined problem. It assumes that a design problem is well-defined and amenable to search. However, a major characteristic of design problems is their ill-definedness. To include ill-defined problems in state-space search, Simon [1973] characterises ill-definedness as a label for movement in a hierarchy of problems. Adapting this perspective, the design exploration process [Simon 1975] is envisioned as a series of problem definition, solution generation and testing cycles.

Rittel & Webber [1984] argue that problem-solving in planning and design are "wicked". Such problems have no definite formulation and their formulation is synonymous with the *weltanschaung* or world-view of the designer. In this class of problems, different formulations of the problem will imply different solutions. The formulation can be continuously redefined, in formal terms, they have no *stopping rule*. Instead, external factors like time or resources and the designers preconceptions frame a solution. They have no criterion for correctness: a design solution may be interpreted differently and under varying criteria. Their performance cannot be ascertained immediately by evaluation. Finally, multiple solutions can be proposed to any formulation of a design problem.

Hybs & Gero [1992] propose a process model of design comprising two distinct search spaces, behaviour space and structure space. The functional requirements of an artifact under design define the expected behaviour space. The solution is found by searching the structure space for a combination of structural elements that satisfy the expected behaviour. Further, a reformulation process enables the designer to modify the behaviour space, depending on the behavioural properties of the proposed solution.

Gero [1994*b*] presents work on several inter-related models built on the *function, behaviour, structure* or FBS framework. In this formulation, the state space representation of designs has three subspaces or abstractions, namely, the structure space, the behaviour space and the function space. State space change is accomplished through two operators, addition and substitution. Exploration is defined as an extension of state-space search through the incorporation of a modification or reformulation process in the FBS framework. Exploration is conceived here as *meta-search* over ill-structured problem formulations. The role of meta-search processes, or exploration, is to reduce such problems to problems that can be addressed by search. This model of exploration is conceived as follows,

> "Exploration may be conceived of as meta-search in that in computational terms all the state spaces which could possibly be produced by a set of exploration processes is determined a priori by the initial state space and those processes." [Gero 1994*a*, p 318]

Gero & Kazakov [1996] propose the concepts of emergence and evolution to compute the exploration processes in an evolutionary model of designing. Poon & Maher [1996] extend the process model of co-evolution described above [Hybs & Gero 1992]. In their model, two distinct movements are introduced in the design space. A horizontal movement in behaviour and structure space and a diagonal movement from behaviour to structure and vice versa. The former is classified as evolution while the latter is termed search. They define these two distinct types of movement in design spaces as a co-evolutionary model of design exploration.

Generative design

Generative design systems provide support for designing based on computational processes for specifying, generating and evaluating collections of designs. They combine process models of designing, information processing paradigms and grammars with the goal of supporting automated or semi-automated generation. Designing is conceived as a staged, iterative process of understanding problems, producing a statement of goals, finding plausible solutions and judging the validity of solutions relative to the goals.

Shape Grammars [Stiny 1980, Stiny & March 1981] propose the recognition, substitution and transformation of "indefinite sub shapes" composed of line segments and their spatial arrangements as the basis for generating designs. The crucial advance made by grammarians in design exploration research lies in two areas. First, their conception of shape composition as defined by rules of a grammar. Second, the proposition that a grammar can be used to explore a *space of designs*. Using the shape grammar formalism and its flavours, corpora of designs, [Stiny & Mitchell 1978*b*, Stiny & Mitchell 1978*a*, Krishnamurti 1980, Flemming, Coyne, Glavin, Hsi & Rychener 1989, Heisserman 1991] have been represented and generated. Krishnamurti et al propose an "arithmetic of shapes" for implementing shape rewriting systems [Krishnamurti 1980, Krishnamurti 1992, Krishnamurti & Earl 1992, Krishnamurthy & Stouffs 1993, Stouffs 1994]. Generative design [Mitchell 1977, Mitchell 1990, Flemming, Coyne, Glavin & Rychener 1988, Shih & Schmitt 1994] proposes computational formalisms for encoding design problems and mechanisms for generating solution alternatives. Flemming's work on the generation of floor layouts using rectangular dissections [Flemming 1978, Flemming 1986, Flemming, Rychener, Coyne & Glavin 1986, Flemming 1987*b*, Flemming 1987*a*, Flemming et al. 1988] is based on a grammatical view of design generation.

Embedded in the thinking on generative design systems (see for example, [Heisserman 1991]) is the use of formal mechanisms for describing and generating states in a *design space.*

Computational formalisms (such as grammars) for describing design spaces are classified as design space description formalisms in Carlson [1994]. In the generative design literature, the notion of a description formalism is the fundamental construct for supporting computational exploration.

Carlson's [1994] overview identifies several key requirements of a description formalism. First, a description formalism is a formal, executable and constructive mechanism. The ability to describe it formally implies that its assertions are testable in logical terms. The ability to execute such a formalism implies that it can be tested in terms of computational tractability and demonstrative power. The constructive nature of formalism enables users to employ the formalism to construct and compose larger systems from smaller fragments. Second, a description formalism is intended for the description of design spaces rather than a single design. Its follows that a description formalism is useful to describe collections of states of a design rather than a single state under transformation (as in the case of CAD systems). Third, a description formalism deals primarily with the formal attributes of collections of designs rather than their meaning or function. It follows from this, that in a description formalism, syntactical change will tend to dominate discussion over semantics. Fourth, a description formalism is differentiated from a design space in that a description formalism bears a class-instance relationship to the space it is used to describe. Fifth, the authoring and use of a description formalism as implemented in a computer program is an experimental process comprising *"proposing sets of statements, observing their effects and modifying the statements"* [Carlson 1994, p 123].

Exploration as a knowledge process

Exploration is distinguished from classical search theories of designing through the acceptance of the ill-structured nature of design problems. Using the term "design-as-exploration", Smithers [1992] proposes that the view of designing as a knowledge process[1] provides a better understanding of design exploration. Knowledge engineering theories of design process [Smithers 1992, Smithers, Corne & Ross 1994, Smithers 1994, Smithers 1996, Smithers 1998] provide algorithms for computing exploration. They allow for a better understanding of how technologies, particularly computational technologies, can be effectively introduced into human design practice.

In more recent work, Smithers [2000] and Smithers [2002] characterise exploration through the acceptance of imprecision, ambiguity, incompleteness and inconsistency in requirements descriptions and emphasise the role of reformulation. Exploration is the construction of a complete, consistent, precise and unambiguous requirements description from an incomplete, inconsistent, imprecise and ambiguous requirements description. This final description is seen to be satisfied by a design description. The design-as-exploration literature, while silent on the symbol level implications, promote a new understanding of the role of knowledge processes in design exploration,

> "... exploration involves the construction and incremental extension of (well structured) problem statements and associated solutions. Sometimes this takes the form of

[1]The field of knowledge engineering has developed techniques for modelling of expertise based on Newell [1982] Knowledge Level.

simply constructing problems and solutions which satisfy a subset of the requirements, but often it involves devising intermediate problems, at "tangents" to the main design problem, in an attempt to discover more about possible ways in which the design problem might be well-formed, and the kinds of solutions which would then be available." [Smithers et al. 1994, p 303–304]

The SEED project [Flemming, Coyne & Woodbury 1993, Akin, Aygen, Chang, Chien, Choi, Donia, Fenves, Flemming, Garrett, Gomez, Kiliccote, Rivard, Sen, Snyder, Tsai, Woodbury & Zhang 1997] addresses the design and implementation of computational tools to support the early phases of engineering design. The SEED knowledge level [Woodbury, Flemming, Coyne, Fenves &

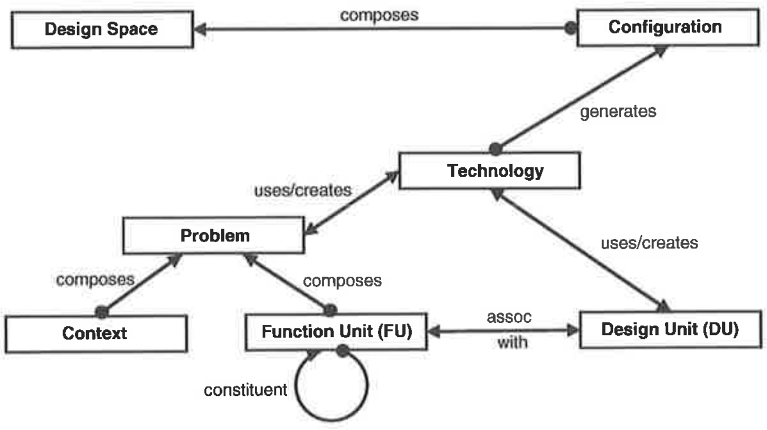


Figure 1.1: The Knowledge level concepts of SEED showing a simple representation of their relationships.

Garrett 1995] posits distinct functional units (FU) and design units (DU) as devices supporting a conceptual separation of brief and design[2]. Here, a problem is specified within a context and composed as a hierarchy of constituent function units. The Knowledge level concepts of SEED and their relationships are shown in Figure 1.1. A *function unit* is associated with a *design unit.* In the SEED Knowledge Level, the designer's view of a solution is modelled as a DESIGN_UNIT [Flemming & Woodbury 1995]. A *technology* can refine or elaborate a problem or a design and generate configurations. These configurations compose a design space. The terminology and relationships of the Knowledge Level are shared by all sub-modules of the SEED Project[3], while differing in their symbol level interpretation. The ability to refine and revise problems, to explore problem-solution pairs and generate solutions through technologies are some of the outcomes of this project relevant to supporting exploration.

[2]Details of the concepts introduced in this project are given in [Woodbury & Chang 1995*b*].

[3]See the special issue of the ASCE Journal of Architectural Engineering [Akin, Sen, Donia & Zhang 1995, Flemming & Chien 1995, Snyder, Aygen, Flemming & Tsai 1995, Woodbury & Chang 1995*b*, Flemming & Woodbury 1995] for a description of the SEED modules.

The account of design exploration as a logical process of constraint resolution [Burrow & Woodbury 1999] is of central interest to the thesis that is developed in this study. The theory of design space exploration [Woodbury, Burrow, Datta & Chang 1999], posits a formal substrate for computing exploration. They contend that design space exploration with a description formalism can be modelled as a form of *movement* in a structured *space* of designs. They argue that a collection of designs in design space require a structuring mechanism underpinning the exploration process [Burrow & Woodbury 2001]. In the case of grammars, this is the derivation process. In the case of generative design systems the structure is provided by the particular generative process embedded in the system (for example, the constraint engine embedded in SEED-Layout [Flemming, Coyne, Woodbury, Bhavnani, Chien, Chiou, Choi, Kiliccote, Stouffs, Chang, Han, Jo, Shaw & Suwa 1994]). Through this theory, they address the questions, Can a formal notion of information ordering provide a structural relationship in design space? Further, can this notion be a useful paradigm for supporting design exploration?

Burrow & Woodbury [1999] and Woodbury et al. [1999] develop a formal notion of exploration based on the specification and representation of information ordering principles in design. Using knowledge representation and constraint resolution theory, Burrow [2003] develops a formal design space structuring mechanism based on the relation of *subsumption*. In it, two design states (and recursively their subparts) are related if one subsumes the other, that is, if one contains strictly less information than the other. This scheme replaces rule-based derivation with a set of composable operators [Woodbury, Datta & Burrow 2000] for moving in design space. The distinction, between functional units and design units, as in the SEED knowledge level, is not implied in this representation scheme. The work on design space exploration is ongoing. The major concepts are reported in Woodbury et al. [1999]. This work introduces the notion of subsumption as an information ordering relation over exploration states. This particular view of exploration with a description formalism based on the subsumption relation among designs is described in greater detail in Chapter 2.

1.1.2 Characteristics of Exploration

Design as exploration is an attempt to devise schemes that are both cognitively plausible and computationally feasible. These schemes lie on a point of continuum between classical search-based techniques and open-ended metaphors of the design process. While they vary in their respective emphasis on what designing is, several common themes can be identified. From the review of the design exploration literature, the requirements for supporting exploration are captured through the following entities, problems, solutions, choices and history. These entities and the relationships between them characterise computational models of exploration. The entities of exploration, as reviewed in the literature, are as follows:

Problem formulation and reformulation.
The exploration model supports problem formulation and reformulation. Hybs & Gero [1992] argue that problems and solutions co-evolve and thus support for reformulation is an integral part of the problem definition process. Smithers [2000] emphasises the reformulation of requirements descriptions during exploration. Woodbury et al. [1999] show that problem restructuring is intrinsic in design space exploration through the property of partialness in the representation of an exploration state. In their formalism, every problem is partially specified and every solution is a partial design. Partiality is captured in the machinery of the design space formalism, hence reformulation is movement from one point in the design space to another, more specific or less specific formulation or a new formulation.

Solution generation and reuse.
A generative process for developing solutions to problems is necessary to support computational exploration [Heisserman 1991]. The generation of partial solutions and navigation of exploration paths must take into account the reuse of previous paths of exploration. Chien & Flemming [1997] show that generated solutions provide a large space of alternatives. These alternatives are part of a solution hierarchy, that is recorded in the design space and can be reused.

Choice making.
Choices made by designers during exploration avoid endless revision and resolve conflicts. Smithers [2000] identifies choice points at the intra and inter algorithm level as crucial for making choices. Chien & Flemming [1997] show that interaction models reduce cognitive overload and facilitate choice making. Schön & Wiggins [1992] consider designing as a reflective conversation that involves the recursive processes of *seeing, moving* and *seeing.* They contend that choices emerge out of the interaction between designing (acting) and discovering (reflecting).

Exploration history.
The rationale of designing, processes of coordination, communication and control are crucial to computing exploration. The rationale of exploration is captured through recording of the intermediate states of designing. These intermediate states comprising alternatives [Woodbury & Chang 1995*b*], revisions [Chien & Flemming 1997] and partial solutions [Woodbury et al. 1999] support computational exploration.

Based on the entities: problems, solutions, choice, history; three abstract invariants in the field of exploration are identified, namely *state, move* and *structure.* First, the representation of state takes two forms, problem formulation states and the possible solutions to problems, design states.

Second, initial, intermediate and final states find *reification*[4] in the exploration process through the representation of moves. Moves enable problem formulation and reformulation, navigation of solution states and the generation of new states. Together, state and move impose structure on the exploration process. This structure is reified in the the grammar literature in the process of derivation. In the AI literature, history or rationale of exploration encodes the notion of structure in the process of exploration. Thus, it follows from the requirements for supporting exploration that the description of design spaces using formal mechanisms rests on the formulation of these abstract invariant concepts. These abstract invariants, state, move and structure reify the concept of a design space, a structured space of exploration states. The design space is a multi-dimensional conceptual space that models state transitions (state) through design change (moves) and records both design moves and the resultant states (structure) [Flemming & Woodbury 1995]. In the context of this study, the term *design space* is *a structured space in which the problems, solution alternatives, the evolution of partial designs and their intermediaries are captured during the process of designing.* Given such a notion of a design space (stated in terms of state, move and structure) the scope of the term *exploration* as understood in this thesis is defined as follows:

Exploration is synonymous with movement in a structured space of partial alternatives, the design space, comprising state and structure.

Given the above, it is necessary to consider how designers interact with problems, solutions, choices and history during exploration. To establish the nature and role of the designer, accounts of interaction reported in the literature are reviewed in the next section.

1.2 Accounts of interaction

During designing, interaction in the form of communication, coordination and control between the designer and the generative mechanism is necessary. Several paradigms for human-computer interaction during designing have been proposed in the literature [Kochhar 1994]. An overview of these paradigms and their applications are summarised as follows:

Manual Paradigms.
The user is responsible for all design decisions and the system is passive with respect to the modelling process. Most CAD modelling systems fall in this category. Quadrel [1991] terms the control policy of such systems as an *open* policy, where no explicit model of control is used. These systems also support constraint-based design wherein most of the modelling is

[4]The term reification signifies the process of mapping terminology used in the design exploration literature, such as state and move into their formal analogues as developed in the thesis. Thus general concepts in the literature can be connected to specific constructs within the formalism.

done manually, except that the system attempts to satisfy a set of constraints [Borning 1977, Borning 1981, Gross, Ervin, Anderson & Fleisher 1988] as a design evolves.

Automatic Paradigms.
The computer creates a design without human assistance [Mitchell, Steadman & Liggett 1976, Galle 1981, Flemming 1986]. These generative design systems have a programmatic flavour and make limited use of direct methods for exploration. The system identifies shortcomings of an emerging design, and can automatically modify a completed design produced by the human in order to improve it [Weitzman & Wittenburg 1993]. The system makes critical design decisions and takes active participation in modifying the design. Design support takes the form of graphic inferences on partial input during design interaction with the user. Given the absence of tools for building interfaces, interaction with spatial grammar interpreters is through a command line or shell interface modelled on rule-based expert systems. Hence textual or command interfaces between the user and the system are developed with facilities for displaying, modifying and transforming shapes.

Cooperative Paradigms.
The human designer makes design decisions while the system supports detailed design refinement, generates several design alternatives, and presents these to the human designer for browsing [Friedell & Kochhar 1991, Kochhar 1994]. The system identifies shortcomings of an emerging design and performs the role of a *critic* by ensuring that the designer receives feedback on the requirements of the design. Argumentation based systems [G. Fischer & Morch 1988], such as JANUS [McCall, Fischer & Morch 1990] that utilise design rationale are examples of this paradigm of interaction.

Mixed-initiative Paradigms.
Mixed-initiative provides a sound basis for interleaving the *complementarities* of human and machine capabilities. Mixed-initiative interaction permits an integration of these complementary roles in exploration. Theories of mixed-initiative have been applied in the areas of tutoring [Carbonell 1970, Freedman 1999], AI planning [Ferguson, Allen & Miller 1996, Ferguson & Allen 1998], scheduling [Horvitz 1999] and spoken language domains [Novick & Sutton 1994, Smith & Hipp 1994, Ishizaki, Crocker & Mellish 1999], the management and coordination of software agents [Rich & Sidner 1998, Burstein, Mulvehill & Deutsch 1999], building knowledge bases [Tecuci, Boicu, Wright & Lee 1999], collaborative problem-solving [Eggleston 1999] and learning environments [Lester, Stone & Stelling 1999].

1.2.1 Interaction paradigms

Design support systems employ one or more of the paradigms to facilitate communication, coordination and control between the user and the system. The exception being the mixed-initiative paradigm. A detailed exposition of this paradigm and its potential for design support await discussion in Section 1.3 and development in Chapter 3. Before a discussion of mixed-initiative, it is necessary to examine and identify interaction requirements for formal exploration. The paradigms underlying user interaction with formal systems as published in the literature are reviewed in the next sections.

Interaction with shape grammars

User interaction models [Woodbury, Carlson & Heisserman 1988, Heisserman & Woodbury 1993] for grammatical design are a combination of the automated and cooperative paradigms. User interaction with grammar interpreters [Carlson, McKelvey & Woodbury 1991, Chase 1989] offer varying degrees of automation in the design process through mechanisms for augmenting state transformations. Heisserman develops a generative design system, GENESIS [Heisserman 1991, Heisserman 1994], that demonstrates interaction with a generative system based on boundary solid grammars. Interaction with GENESIS involves the manipulation of data structures that represent a well formed boundary solid representation augmented by labels and states. The exploration of the space of designs is performed by a boundary grammar interpreter based on rule matching and shape replacement. Matches occur on labels and sub-graphs, while replacement is a sequence of operations on the underlying boundary representation. GENESIS provides two layers of control over the derivation process through the application of rules. In the first case, the interaction with the design space is automated, the interpreter applies the set of rules without user intervention using a depth-first each strategy. In the second case, the user can choose to apply each rule and make decisions at branching points.

The manipulation of infinite subparts in conceptual design has been proposed as an interaction technique by Tapia [1996] and interaction with shape grammars is demonstrated in the GEdit [Tapia 1999] system. The ability of such systems to distinguish and recognise emergent shapes through user interaction and interactive interfaces is addressed. Chase [1999] presents a comprehensive analysis of interaction with grammatical systems. Chase describes a categorisation of interaction techniques focusing on existing grammar system implementations. Furthering this analysis, Chase [2002] develops a formal model of user interaction for developing grammars and for exploring spaces of designs. This model is essentially modelled on the cooperative paradigm described above. Chase [2002] studies the degrees of interactivity or generativity underlying grammar systems and proposes a formal model of user interaction to bring together the knowledge of how specific grammar systems organise human computer dialogue. The formal model addresses the problems of how designers interact with shape rewriting systems. The model is presented in terms of stages, entities and control. The stages of interaction correspond to the classical grammar development process. He proposes a distinction between three sets of actors, the grammar developer, the grammar user and the system. The model describes the modes of user interaction and the degrees of control possible with such systems. He identifies the interaction features of grammar implementations as modal, selection of control mode, rule definition, rule selection, object selection, presentation of results and backtracking.

The process of developing a grammar comprises two stages, grammar development and grammar application. Each of these stages involves the manipulation of the grammar by three entities, the developer, the user and the computer system. An appropriate control scenario for a given grammar application is constructed by mapping entities to stages. Three interaction scenarios are described.

As in the case of Tapia [1999], this model is specific to ruled-based shape grammars systems. However, the interleaving of control between the system and the user as proposed in this work provides a basis for developing formal models of interaction with grammar-based design systems in the cooperative paradigm.

Interaction in Generative design

Models of user interaction have been used to support generation in the SEED [Woodbury et al. 1995] research project. SEED-Config, a sub-module of SEED [Woodbury et al. 1995] supports three dimensional schematic design of building forms and technical systems. Such derivation systems are extensions of command interpreters and provide added functionality incorporating interactive user interfaces.

These systems support intermediate states, the derivation paths, visualisation of states and path branching. They employ *direct manipulation* for supporting generation and lay an emphasis on the collection of states that define the solution space, usability and the mental model of the designer. Direct manipulation interfaces [Shneiderman 1982, Shneiderman 1983] emphasise continuous object representations, physical actions and the use of rapid, incremental and reversible operations. Direct manipulation techniques [Hudson & Yeatts 1991, Shneiderman 1997] from user interface design are a natural interface for design exploration tasks. Users control and manipulate the objects of interest by interactively grabbing and pulling them.

Direct manipulation is used for interacting with grammar systems such as Tartan Worlds [Woodbury, Radford, Taplin & Coppins 1992], a generative symbol grammar system[5] and discoverForm [Carlson & Woodbury 1994]. Harada, Witkin & Baraff [1995] and Harada [1997] employ a direct manipulation technique for exploring "discrete/continuous models" based on the paradigm of physically-based modelling [Witkin, Fleischer & Barr 1987, Witkin, Gleicher & Welch 1990].

[5]Tartan Worlds gave the ability to directly manipulate designs and rules, but also provided the usual grammar control. Its worlds tended to get out of control as they proliferated. Woodbury, 2003. Personal Communication.

This technique is applied to the task of exploring design constraints interactively by direct manipulation. The discrete changes within a continuous event loop are modelled as shape transformation rules while continuity is handled directly by the users pull, push and trigger actions.

Once an exploration space has been mapped out by the underlying generative system, human-computer interaction is an integral part of navigation. Chien & Flemming [1996] addresses the problem of exploring layouts in interactive contexts. The cognitive overload imposed on the users of generative systems during exploration is analysed and navigational cues introduced in Chien & Flemming [1997]. This work employs the navigation or way finding metaphor arising in physical and information spaces to address the problem of exploring design spaces. A navigation framework for generative design systems and a software prototype for design space navigation are proposed in Chien [1998]. The design space navigation framework facilitates the growth and traversal of the design apace along five dimensions and maintains objects as well as relationships between them in the space. A key feature of the framework is the use of physical cognitive cues to develop nodes and landmarks in the space, derived from Lynch's [1960] study on imageability and way finding in cities. The lessons from the work suggest that complex multi-dimensional spaces require effective presentation and interaction through information navigation techniques. The emphasis on visualisation of design space models and information navigation extend the research from purely formal interaction to direct manipulation systems. These outcomes mirror the parallel developments in user interface tools, the emphasis on usability and the advent of software engineering methodologies for the design of human-computer interaction.

1.2.2 Characteristics of interaction

Human-computer interaction models form an essential component in developing computational tools to support designing. Current models, theories and methods for exploration support manual, automated and cooperative paradigms for integrating the user with the system. These models adopt a "black-box" approach to user interaction, where communication, coordination and control is based on the apriori division of labour between user and system. This division of labour separates the tasks to be performed between the user and the system. Exploration is achieved either through a formal generator or open-ended exploration driven by the user through some direct manipulation interface. In these approaches, the integration of user actions is mediated by the appearance of choice points in the generation, whose reference needs to be resolved. In the case of generative design systems, exploration is primarily generator driven. When direct manipulation or designer input is used, it is treated as a useful, but secondary support role. Further, interaction is stipulated by a global control policy. For example, the user can choose one of a number of possible rules for transforming the current state. The rules that apply in a current state remain under global control of the formal mechanism.

Strategies for the effective sharing of control and conflict resolution between the user and machine are necessary to support effective interaction. The mixed-initiative interaction paradigm can model a more fine grained division of labour. For example, allocating and sharing responsibility over the same task can be modelled by mixed-initiative. Further, mixed-initiative offers a more flexible mechanism for control (acquire and relinquish initiative) between the designer and the formalism. To identify how the mixed-initiative paradigm addresses these issues of joint responsibility over the same task, fine-grained control through initiative and strategies for conflict resolution, research on mixed-initiative interaction is reviewed.

1.3 Mixed-initiative interaction

The field of mixed-initiative interaction research continues to develop rapidly as new tools and techniques are established. In the next sections, the current state of the art in mixed-initiative interaction is reviewed.

The work on the mixed-initiative interaction paradigm can loosely be grouped into four classes [Cohen, Allaby, Cumbaa, Fitzgerald, Ho, Hui, Latulipe, Lu, Moussa, Pooley, Qian & Siddiqi 1998]. First, initiative is seen as the process of shifting control of conversational dialogue between the user and system. Second, initiative is seen as the coordination of joint responsibility for completing shared tasks. Third, initiative is seen as the process of directing problem-solving goals in a domain, combining aspects of both dialogue control and task coordination. Finally, initiative is defined as a collaborative process, involving turn-taking through dialogue, tasks and goals. The differences and similarities in these definitions are analysed by comparing a range of mixed-initiative application areas from task-oriented planners to tutorial dialogue systems. These theories of mixed-initiative [Cohen et al. 1998] have a natural progression of thought, from the perception of initiative as a control of the conversation, through task coordination to more complex arrangements, where initiative combines both dialogue control and task coordination. Initiative can be distinguished further, allowing for collaboration, different strengths of initiative and for multiple threads within a dialogue to be tracked simultaneously.

Allen, Ferguson & Schubert [1996] characterises mixed-initiative as the *exchange* of initiative in a flexible, opportunistic manner, *shifts* in focus of attention to meet user needs and the provision of mechanisms for maintaining *shared* implicit knowledge. Burstein & McDermott [1996] expand these characteristics to include flexible *visualisation*, *context* registration and task management support for managing shared tasks.

Haller, McRoy & Kobsa [1999] examine several mixed-initiative systems and individual efforts in designing mixed-initiative systems. These studies establish how the differing characteristics of the application areas make one or more of these definitions of initiative more useful than others in a specific context. To establish a definition of what constitutes *mixed-initiative*, researchers have identified *initiative* with the *control* of rational dialogue, the *coordination* of shared tasks and the *collaboration* between multiple autonomous agents, both human and computational.

Mixed-initiative has been used extensively in the planning domain, where users interact with software agents to produce plans. The objective is to capture and creatively reuse the derivation rationale underlying joint human and machine-based decision-making processes. Mixed-initiative is shown to [Chu-Carroll & Brown 1998] achieve better plans than either the human or machine can create alone.

Veloso [1996] and Veloso, Mulvehill & Cox [1997] employ mixed-initiative in planning to engage the user in automated planning processes. Veloso uses mixed-initiative planning as a framework in which automated and human planners interact to jointly construct plans. ForMAT is a case-based system that supports human planning through the accumulation of user-built plans, query-driven browsing of past plans, and several primitives for analysing plan functionality. Prodigy/Analogy is an automated AI planner that combines generative and case-based planning. Stored plans are annotated with plan rationale and reuse involves adaptation driven by this rationale. They integrate ForMAT and Prodigy/Analogy into a real time, message passing mixed-initiative planning system. The mixed-initiative approach consists of allowing the user to specify and link objectives that enable the system to capture and reuse plan rationale. They discuss the integration of two large systems through mixed-initiative planning.

The application of mixed-initiative is reported in the domain of scheduling [Ferguson et al. 1996]. The scheduling process requires flexible human involvement but complexity and time stress also demand substantial automated support. Scheduling tools [Cesta & D'Aloisi 1999, Kitano & Ess-Dykema 1991] consider the implications of mixed-initiative in the design of scheduling algorithms. Horvitz [1999] develops mixed-initiative user interfaces that enable users and intelligent agents to collaborate efficiently. He demonstrates the role of mixed-initiative in the domain of scheduling and meeting management.

Amant [1997*a*] present the overlapping areas of research between mixed-initiative and interaction. They combine the dialogue view of mixed-initiative with direct manipulation techniques in the domain of abstract force simulation and exploratory data analysis. They focus on the ability of an interactive environment to constrain and guide both automated agent behaviour as well as human effort. This *dialogue-based* framework unifies the different types of control and coordination initiative supported by a multi-modal system. Building on this framework, they describe a navigational metaphor [Amant 1997*b*] for integrating direct manipulation with mixed-initiative planning. In describing the domain of camera planning they state,

> "... we want to support dynamic communication of task and domain information between the system and the user, with shared control of camera placement and orientation and direction." [Amant & Cohen 1997]

Models of mixed-initiative dialogue in human-machine interaction are based on formal models of human conversation [Grice 1975]. Mixed-initiative dialogue enhances the richness of interaction by allowing more complex forms of exchange between the user and the formalism. Mixed-initiative specifies how a participant in the dialogue, either user or system, can seize or relinquish initiative. Mixed-initiative models the movement along a conversational thread through a series of topics as a *flow*. This theory identifies five types of movement operations along a conversational thread. They are going forward, changing direction, stopping or pausing, closing or repeating and interrupting [Cohen et al. 1998]. Mixed-initiative models enable flow management, namely using the current type of movement operation to acquire or relinquish control of a conversational thread. This model of mixed-initiative is used in the developed of dialogue systems [Walker & Whittaker 1990, Tsai, Reiher & Popek 1999].

Carbonell [1970] describes a mixed-initiative tutoring system, which can shift between the student asking questions and the user asking questions. Novick [Novick 1988] considers a dialogue participant to have the initiative if the participant controls the flow and structure of the interaction. Whittaker & Stenton [1988] and Walker & Whittaker [1990] equate initiative with control. They argue that as initiative passes back and forth between the discourse participants, initiative is transferred from one participant to the other. They devise rules for allocating dialogue control based on utterance types, which include assertions, commands, questions, and prompts. They analyse patterns of control shifts by applying their rules to a set of expert-client dialogues on resolving software problems. They note that the majority of control shifts are signalled by *prompts, repetitions*, or *summaries*, while in the remainder of the cases, control shifts as a result of *interruptions*. Two kinds of shifts are associated with the view of initiative as control over the flow of a conversation. First, change of control among participants through shifts in dialogue in the case of flow in the same direction. Second, shifts in dialogue signify a control shift when one of the participants changes the topic of a conversation. Guinn's [1993] computational model of dialogue and Smith's [1991] expectation driven model are two examples of formal mixed-initiative interaction models used in dialogue-based expert systems.

Freedman [1999] develops a plan-based dialogue manager, ATLAS and applies it in the domain of tutoring systems. ATLAS is based on a hierarchical task network (HTN) style reactive planner to build tutoring systems. Mixed-initiative in ATLAS allows multi modal dialogue through the integrating of natural language, text and graphics.

Ferguson & Allen [1994] investigate the coordination of tasks in the design of mixed-initiative systems. The task-oriented coordination perspective of mixed-initiative has been robust, resulting in a number of successful applications and systems, including TRAINS [Ferguson et al. 1996] and TRIPS [Ferguson & Allen 1998] in the domain of planning. Burstein & McDermott [1996] and Burstein, Ferguson & Allen [2000] address the role of participants, user and system, in the planning domain. The goal of mixed-initiative is

"...to explore productive syntheses of the complementary strengths of both humans and machines to build effective plans more quickly and with greater reliability." [Burstein & McDermott 1996]

Mixed-initiative also allows the possibility of *extended* [Allen 1999] interaction, as a series of commands, defining and discussing tasks and exploring ways to perform the task.

Ishizaki et al. [1999] examines the efficiency of mixed-initiative task coordination in a route finding application. Tecuci et al. [1999] address the domain of knowledge engineering and reports the application of mixed-initiative methods to the problem of knowledge acquisition. Their motivation for mixed-initiative lies in the fact that manual solutions to the problem of building knowledge bases is highly inefficient and automated systems for the same problem are impractical. They develop mixed-initiative methods for knowledge base development in the DISCIPLE project and provides experimental results on the feasibility of the approach. In DISCIPLE, mixed initiative is used to coordinate the tasks of rule learning, rule refinement and exception handling during knowledge-based development. Tecuci et al. [1999] establish an expert-apprentice relationship between the user and the agent, with mixed-initiative driving the learning process to support the task of knowledge acquisition. Through mixed-initiative, the agent is able to acquire multiple learning strategies, during the development of the knowledge base. In using mixed-initiative for task coordination, they observe that,

"There is the synergism between the different learning methods employed by the agent. By integrating complementary learning methods (such as inductive learning from examples, explanation-based learning, learning by analogy, learning by experimentation) in a dynamic way, the Disciple agent is able to learn from the human expert in situations in which no single strategy learning method would be sufficient." [Tecuci et al. 1999]

Mixed-initiative in DISCIPLE presents two interesting conclusions. The first is the support for knowledge acquisition, particularly the handling of incomplete knowledge bases that may be evolving dynamically through shared task coordination between human and agent. The second is the ability of the agent to use multiple strategies, particularly the ability to select a learning strategy dynamically. Lester et al. [1999] apply task-oriented mixed-initiative to develop constructionist learning environments. They develop pedagogical agents that enter into conversational dialogue with learners such that learners are able to actively participate in problem-solving exercises.

Mixed-initiative is a promising and productive model for interaction with formal systems. Its promise for design exploration lies in the understanding that human and machine can be leveraged more effectively through an integration of their complementarity. Further, that dialogue is a more effective technique for interaction than global policies for control (conflict resolution, error recovery). Finally, that joint responsibility over shared tasks can lead to better outcomes (reliable, productive) than pure division of labour.

At this stage, it is not clear how mixed-initiative can be a useful model for supporting interaction with an exploration system. The use of mixed-initiative interaction in supporting computational exploration has not been researched and thus its usefulness or otherwise has not been established. In order to investigate the possibilities of mixed-initiative in supporting exploration, the next section posits the research hypothesis. The remainder of this thesis deals with the investigation of this hypothesis for the integration of human and machine capabilities in design space exploration.

1.4 Research hypothesis

The research hypothesis of this study is based on two assumptions, both arising out of the review of the research literature on design exploration. The first assumption represents a statement of the current state of research in developing design space description formalisms for supporting exploration. The second assumption is a statement of the current state of research in understanding the role of interaction in design exploration. Given the validity of these assumptions, the research hypothesis is a statement of the new ground that is covered in this thesis. The assumption underlying the research hypothesis are as follows:

1. *The process of exploration can be formally encoded with a design space description formalism.* The description formalism specifies a set of initial states, a set of state transforming operators or moves and a structure underlying the collection of states. Chapter 2 describes such a design space description formalism. This formalism encapsulates an explicit theory of computational agency vested in the system. Hereafter, this theory is termed *design space exploration.*

2. *Integrating the role of the designer in computational exploration with a description formalism requires an interaction model.*
   As shown in this Chapter, interaction models provide a mechanism for introducing human design intent into the process of exploration. An interaction model provides a systematic exposition of how communication, coordination and control strategies enable a designer to interact with a formal system.

Given the above, the thesis that is investigated in this study is as follows:

That a mixed-initiative model of interaction presents a promising new approach for integrating the roles of the user and the description formalism in computational exploration.

Mixed-initiative is a promising new paradigm for integrating the designer with a description formalism. It is the intention of this study to demonstrate that mixed-initiative is an effective way to integrate the role of the designer in computational exploration.

Chapter 3 describes the requirements necessary for introducing mixed-initiative in the formal process of exploration. Design space exploration requires sophisticated bi-directional modes of communication, coordination and control between the designer and the description formalism. Mixed-initiative can address these requirements.

Part II develops a new model of interaction based on mixed-initiative to support design space exploration. This model addresses coordination, communication and control between the user and the formalism. Chapter 7 develops an implementation of this model. The role of the designer, the role of the formalism and the use of mixed-initiative in exploration are discussed through an example from the domain of architectural design.

Summarising, exploration is understood as movement in a space of problems and solutions, combining both formal search and human guidance. Following from the above, a model of human computer interaction for design space exploration based on mixed-initiative is formulated to address the role of *both* user and formalism for supporting exploration. This model is implemented in a prototype system and demonstrated through an example taken from the domain of architectural design.

1.5 Summary

The process of exploration can be formally encoded with design space description formalisms. Design space exploration requires a model of interaction that integrates formal generation and the actions of a human designer.

Formal systems require interaction models to enable human designers to work with exploration algorithms. Mixed-initiative is a possible candidate for addressing the interaction of the user with description formalisms during exploration. Three abstract invariants, namely *state, move* and *structure* are identified to characterise design space exploration. From the perspective of the designer, representation of state takes two forms, problem formulation and the possible solution states. Exploration moves uncover initial, intermediate and final states, supporting problem formulation and reformulation, navigation and the generation of states. State and move impose structure on the exploration process, captured in the concept of a design space. Chapter 2 describes an exploration formalism through a detailed formulation of these concepts for computational exploration.

Chapter 2

An exploration formalism

This chapter explains how the entities of exploration, namely, state, move, structure are reified in a formalism for design space exploration based on typed feature structures.

2.1 Entities of exploration

Design space exploration with description formalisms can be modelled as a form of *movement* in a structured *space* of designs. The theory of design space exploration [Woodbury et al. 1999], posits a formal substrate for computing exploration. The formalism [Woodbury et al. 1999, Burrow & Woodbury 1999, Woodbury et al. 2000] employs and develops extensions to, Carpenter's [1992] typed feature structures to account for intermediate states, exploration moves and an ordering over explored states.

The representational device of *feature structures* [Knight 1989, Kasper & Rounds 1990] and *attribute value logic* [Pollard & Moshier 1990, Franz & Jrg 1994] known from linguistic theories of generation and from the constraint programming literature underpin the logic of typed feature structures. The logic of typed feature structures brings together unification based approaches to formal grammar in computational linguistics research. For readability, formal definitions, terminology and syntax arising out of its application to design space exploration are given in Appendix A.

As a representation, typed feature structures are similar to frame-based [Minsky 1975] and terminological knowledge [Borgida, Brachman, McGuinness & Resnick 1989] representation systems. The analogy between feature structures and knowledge representation schemes comes from associating a collection of features or attributes with each node or frame. Each feature represents a slot label and the arcs themselves point to the fillers, creating a network of associations. Feature structures comprise a set of nodes, each of which is labelled by type information.

The key concepts of the description formalism relevant to the aims of the thesis are explained by mapping the entities of state, move and structure onto their symbol level representation in the description formalism. This mapping enables a clear exposition of the symbol substrate in terms of the entities of exploration. The exposition of the description formalism is organised as follows:

Representation of exploration states.
The formalism supports the representation of an exploration state through the concepts of *types, features, descriptions* and *feature structures.* In Section 2.2, the representation of exploration states in the design space exploration formalism are described.

Ordering of exploration structure.
The structure of exploration is represented through the ordering relation of *subsumption.* The concept of an ordered design space underpins the description formalism. In it, the collection of exploration states are ordered by the relation of subsumption. In Section 2.3, the ordering of exploration structure through subsumption is described.

Algorithms for computing exploration moves.
Exploration moves are cast in terms of moves in a design space upwards or downwards in an information ordering. The formalism provides a set of operators for the generation of new exploration states, modification of existing states and movement between states. In Section 2.4, the representation of exploration moves in the design space exploration formalism are described.

The exploration entities and their mapping onto the description formalism machinery is shown in Figure 2.1.

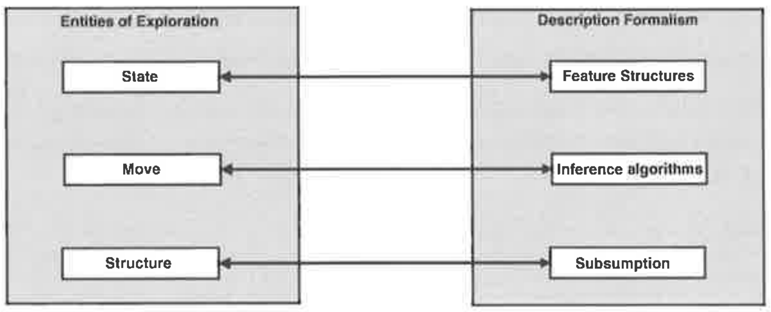


Figure 2.1: The exploration entities, *state, structure, move* are mapped onto the description formalism.

2.2 Representation of exploration states

The formalism represents exploration states through three elements from the feature structures machinery. These elements are a type hierarchy, a set or sets of feature structures and a description language for specifying constraints on types and structures. Types comprising T stand for domain knowledge of the allowable universe of discourse expressed in terse form. Structures from

F represent exploration states, in this case, physical and conceptual attributes associated with the design of buildings. Descriptions from D are constraint expressions in a formal attribute-value description language. Descriptions are used for problem formulation, constraints on types and generated structures. The relationships described here are shown in Figure 2.2. Descriptions are constraint expressions that correspond to problem descriptions [Woodbury et al. 1999, p 293]. The generation of structures from descriptions is handled by π-resolution, which awaits discussion in Section 2.4.1. The following sections develop each of these concepts supporting the representation of exploration states.

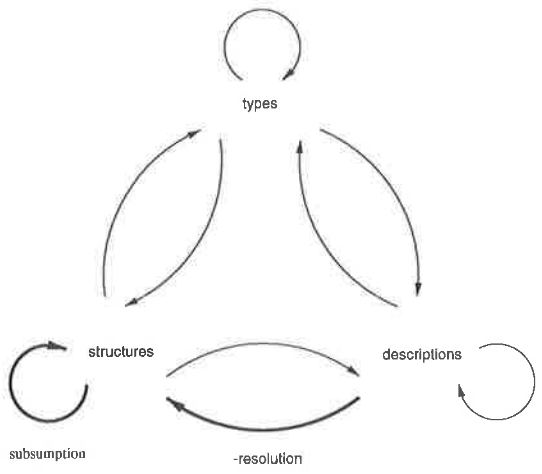


Figure 2.2: Types features, descriptions and the relationships of subsumption and unification showing the computation paths in the scheme. Types stand for domain knowledge. Structures represent models of particular designs.

2.2.1 Types and features

Types order design information in natural classes, similar to the role of *concepts* in terminological knowledge representation systems [Borgida et al. 1989, Patel-Schneider, McGuinness, Brachman, Resnick & Borgida 1991]. The inheritance hierarchy is an *ordered* collection of types, organised by an inheritance relation based on type inclusion. Types are organised into a multiple inheritance hierarchy, in which information associated with a type is extended in inheriting types, i.e., an informational ordering. Types are arranged hierarchically such that a subtype inherits all the information from its super types. A type hierarchy is a bounded complete partial order, or BCPO. Since type declarations are finite, this amounts to the restriction that every pair of types that have a common subtype have a unique most general common subtype. The existence of consistent joins and a most general type, ensure that the inheritance hierarchy of types is a BCPO. The incorporation of *types* in a feature-based formalism enriches feature logic with the polymorphism and multiple inheritance known from object oriented data models [Cardelli & Wegner 1985]. The BCPO type hierarchy, $\langle Type, \sqsubseteq \rangle$, provides a scheme of individuation for classifying knowledge based

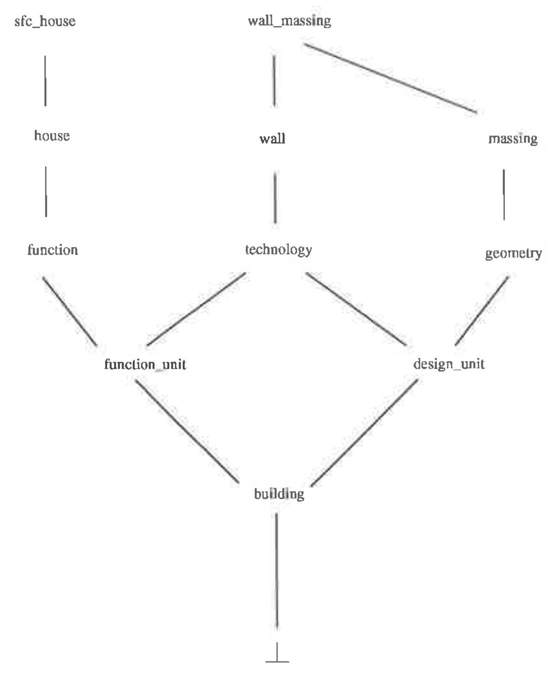


Figure 2.3: A representation of the entities in type hierarchy. The types stand for domain knowledge

on atomic properties encoded in types. Types in the BCPO can range from the degenerate type model, with one type [Carpenter 1992, p 52], to infinite order types [Chang 1999].

As an illustration, Figure 2.3 shows the knowledge level concepts of SEED (discussed in Section 1.1.1 and shown in Figure 1.1), are mapped onto a type hierarchy. The example type hierarchy comprises twelve types. The universal type is declared as the unique most general type and represented as $\perp$. The universal type is shown at the base of the type hierarchy and is called *bottom*. The type *building* has subtypes *function_unit* and *design_unit*. The type *technology* is a subtype of both these types. The relation of sub typing is transitive and the derived transitive sub typing relationship is anti-symmetric. This means that there should not be two distinct types each of which is a subtype of the other. Types are used to represent the concepts of DESIGN_UNIT and FUNCTION_UNIT. The type *function_unit* is subtyped into three types, namely *function*, *house* and *sfc_house*[1]. The *design_unit* is subtyped into *geometry* and *massing*, while the *technology* is subtyped into *wall* and *wall_massing*.

Further, maximal types and primitive data types can easily be incorporated in a type hierarchy. This is shown in Figure 2.4. Since the ordering of types in this type hierarchy is a BCPO, the lattice operations, JOIN and MEET are available over $\langle Type, \sqsubseteq \rangle$. For every set of types with a common subtype, there is most general common subtype or join. JOIN, ($\sqcup$) provides the most general common specialisation of two types. MEET, ($\sqcap$) infers the most specific common generalisation of

[1]The term "sfc" stands for "single-fronted cottage", a common traditional building type in Australia. The single-fronted cottage is used as an example to illustrate the mechanics of typed feature structures and design space exploration [Woodbury et al. 1999].

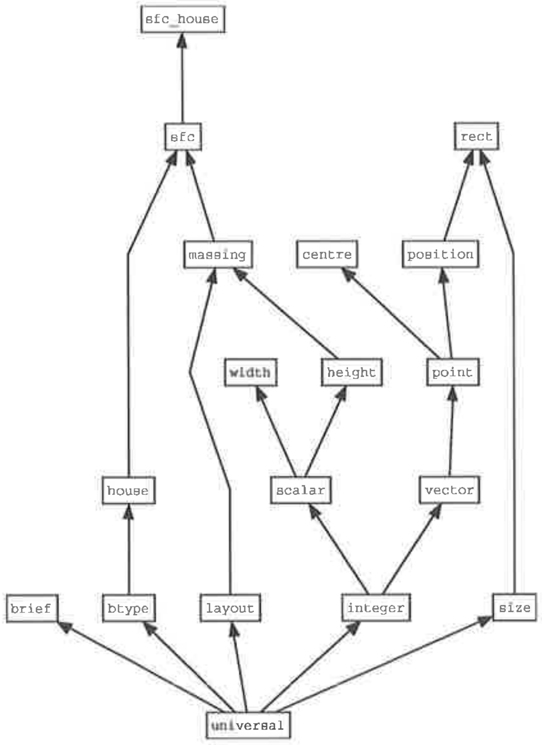


Figure 2.4: Another example of an inheritance hierarchy of types $\langle Type, \sqsubseteq \rangle$.

any two types.

Types contain attributes called *features* drawn from a set of named attributes, **Feat**. The value of a feature is *functional* rather than *relational.* This imposes a single-value restriction on features [Carpenter 1992, p 34]. A partial feature value function, δ, enforces this unique value restriction on features. Types and features are related through *appropriateness specifications.* An appropriateness specification over the inheritance hierarchy $\langle Type, \sqsubseteq \rangle$ and features **Feat** must meet the two conditions of *feature introduction* and *upward closure.* The formal definition of these conditions is given in Appendix A, Definition 1. Appropriateness conditions disallow inconsistent features by declaring which attributes are appropriate for a given type and which types are appropriate for a given attribute[2]. $Intro(f)$ is the most general type for which the feature is defined. Upward closure ensures that any type that is appropriate for a feature in a type A is at least as specific as the types that are appropriate for the features of subtypes of A. An example of the introduction of features to types and the resulting type hierarchy satisfying the appropriateness conditions is shown in Figure 2.5.

Summarising, in the design domain, types denote knowledge about empirical objects under refinement and elaboration operations. The hierarchy of types provides the fundamental ordering relation in the encoding of types. Features represent the atomic or complex attributes of the objects

[2]A detailed discussion of appropriateness and typing, is provided in [Carpenter 1992, p 86].

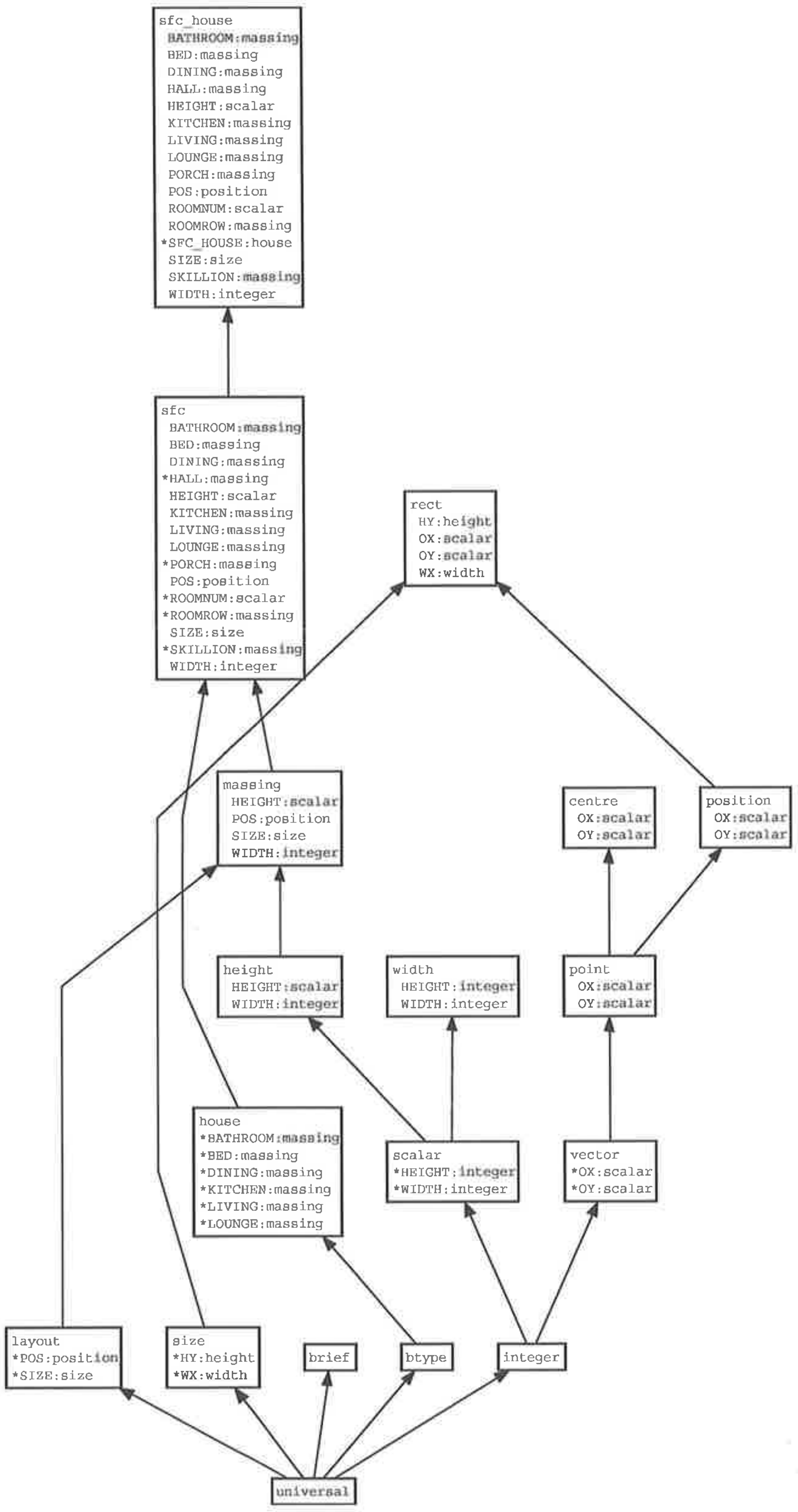


Figure 2.5: An example of types showing how features are introduced, marked with $*$ and their inheritance by subsumption.

that constitute a partial design. The inheritance hierarchy over types models conceptual design information about empirical design entities and the constraints between them.

2.2.2 Descriptions

Descriptions are constraint expressions specified in an attribute value equational language. The description language [Carpenter 1992, p 52] is given over the collection **Type** of types and **Feat** of features is the least set **Desc** such that:

- every type is a description,
- a path followed by a description is a description,
- two paths equated form a description,
- two paths disequated form a description,
- descriptions can be combined with the logical operators *and* and *or*, forming conjunctive and disjunctive descriptions respectively.

Descriptions provide a *lower bound* of specificity on the represented object. In the design space description formalism [Woodbury et al. 1999], descriptions represent problem statements and constraints on types. Like features, constraint expressions may be associated with types. Expressions associated with types impose recursive and logical constraints on types in the type hierarchy[3]. In addition to feature and type inheritance, constraints on more general types are inherited by their more specific subtypes. Constraints on types are expressed in the description language, in the form of an implication, $\sigma \Longrightarrow \phi$. Structures that carry the type σ must satisfy the constraint ϕ, which is a description.

Description may be satisfied by no structure, a finite number of structures or an arbitrarily large collection of feature structures. In designing, this is analogous to the statement that a requirement may have no solutions, a finite number of solutions or an arbitrarily large collection of solutions. Each structure that totally satisfies the description is called a *satisfier* of the description.

In the domain representation, it is sufficient to note that a constraint system is available for providing additional restrictions and relations to types. The arrangement of types by subsumption allows subtypes to inherit constraints from super types. A system of type constraints, stated in the form of descriptions and a constraint system is a total function $Cons : Type \longrightarrow Desc$ [Carpenter 1992, p 228].

[3]See the use of recursive type constraints described in [Carpenter 1992, p 228].

2.2.3 Feature structures

The ability to model partial information and intentional propositions is crucial in supporting designing. Feature structures bring the key properties of intentionality, partialness, structure sharing, and cyclicity to the representation of exploration states. A feature structure represents all that can be known about a domain object at some particular stage of computation, an exploration state. Thus a feature structure representation of an exploration state must be and is partial and intentional. An exploration state, represented as a feature structure, can be made more or less specific through inference operations. Through intentionality the existence of two states with the same information does not imply that they are the same state. Further, two distinct feature structures can represent exactly the same information.

A feature structure consists of two pieces of information and a relation between them. Firstly, every feature structure has a *type* drawn from the inheritance hierarchy, $\langle Type, \sqsubseteq\rangle$. Secondly, a feature structure is a finite, possibly empty, collection of *feature:value* pairs. A feature value pair consists of a feature and a value, where the value is either a type or a feature structure. All nodes are connected by directed arcs denoting features. One node is designated as a root node. Figure 2.6 shows a feature structure graph for a feature structure of type *entity* with features, MASS_EL and GEOM. These features have as their values two sub structures of type *massing* and type *geometry.*

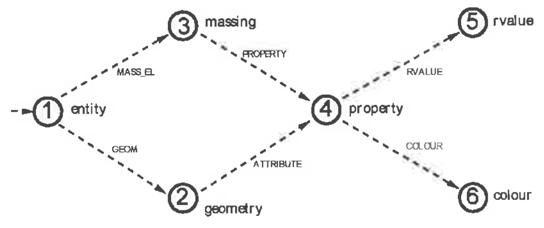


Figure 2.6: An example of a feature structure.

A path-based notion of feature structures [Carpenter 1992, p37] enables the traversal of more than one feature at a time. A *path* is a sequence of features. Let $\mathsf{Path} = \mathsf{Feat}*$ be the collections of paths, π be an element of Path, then $\delta(\pi, q)$ is the node that is reached by following the features in the path π from q. Figure 2.6 shows a feature structure in graph form. Nodes represent types and arrows represent features. The node $colour = \delta(\pi, q)$ where q is the node *entity* and π is a path comprising a sequence of features such that, $\pi =$ MASS_EL: PROPERTIES: COLOUR.

In addition to partialness and intentionality, feature structures support the sharing of structure. In Figure 2.6, the node labelled *property,* is shared by two paths, $\pi_i = \pi_j$, where $\pi_i =$ MASS_EL: PROPERTIES and $\pi_j =$ GEOM: ATTRIBUTES. Structure sharing in the representation of exploration states permits the elimination of redundant states. A formal description of the properties of feature structures is given in Appendix A.

Satisfaction

Feature structure collections and their descriptions are equivalent and are related through a *satisfaction* relation. The satisfaction relation, $\models: \mathcal{F} \rightarrow \mathsf{Desc}$ is defined with reference to descriptions [Carpenter 1992, p 53]. Every feature structure is the *most general satisfier* or MGSat of a disjunction free description [Carpenter 1992, p 56]. A description is said to be *satisfiable* if it is satisfied by at least one structure. A description ϕ *entails* a description ψ if every structure satisfying ϕ also satisfies ψ. Two descriptions are *logically equivalent* if they entail each other, or equivalently, if they are satisfied by exactly the same set of structures. A description is a reference to its most general satisfiers and thus implies a possibly empty set of pairwise incomparable feature structures.

The satisfaction relationship between descriptions and feature structures can be stated formally in terms of the functions MGSat and MGSats. Given the set of all feature structures $\mathcal{F}$, for disjunction free descriptions, $\mathsf{NonDisjDesc}$ there is a surjection MGSat : $\mathsf{NonDisjDesc} \rightarrow \mathcal{F}$. If a description is disjunction free, the named set will either be empty or contain only mutual alphabetic variants. The inclusion of disjunction into descriptions implies that there may be multiple distinct satisfiers. Formally, for the set Desc, of all descriptions MGSats : $\mathsf{Desc} \rightarrow 2^{\mathcal{F}}$ identifies sets of feature structures that are either pairwise incomparable or alphabetic variants. In other words, if a feature structure satisfies a description, then every feature structure that it subsumes also satisfies the same description [Carpenter 1992, p 55].

Unification

Underlying design space exploration is the sole generative mechanism of unification. During exploration, unification extends a query description with respect to a type constraint system [Woodbury et al. 1999, Woodbury et al. 2000]. Unification [Shieber, Uszkoreit, Pereira, Robinson & Tyson 1983, Shieber 1984, Shieber 1986] is an inference operation that computes the result of combining two pieces of information. Feature structure unification produces the most general feature structure that contains all of the information in its two arguments[4]. The *unification* of feature structures F and F', written $F \sqcup F'$, is a conjunction that incorporates the collections of feature paths in F and F' as well as their types. Informally, unification seeks the most general feature structure that is more specific than either operand.

The result of design unification can be taken as a new, more complete object that is consistent with the objects represented by the argument features. As an inference operation, unification

[4]In constraint logic programming [Smolka & Aït-Kaci 1989, Ait-Kaci, Podelski & Smolka 1992, Ait-Kaci & Cosmo 1993], unification is the central algorithm for the resolution of feature logics.

constructs the most general specialisation of two design states or fails if the structures represent inconsistent information. The unification of two design states is represented by a feature structure containing neither more nor less information than the information represented in the states being unified. Given two states, A and B, unification, written $A \sqcup B$, produces a third state C subsumed by both A and B if such a state exists, otherwise unification fails.

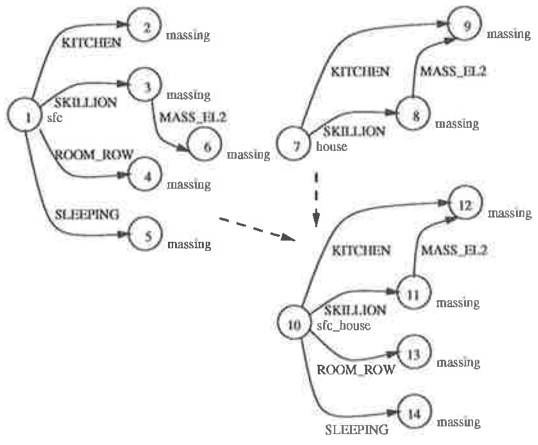


Figure 2.7: An illustration of feature structure unification: the bottom feature structure is the unification of the left and right feature structures [Burrow & Woodbury 1999].

In the example shown in Figure 2.7, the feature structures represent the decomposition of a building design: the nodes denote building entities and the edges denote functional roles in the design. By characterising feature structures as representations of partial design information, it is possible to consider both examples as partial representations of some final design. The unification of two design states simultaneously decides whether some object may exist which is consistent with the two functional decompositions. If so, unification provides an informationally minimal design state combining the information in each state. Figure 2.7 depicts two simple states which are the operands and resultant state of a unification operation. The resultant shows the new root node and the inclusion of new feature path values from both operands.

2.3 Ordering of exploration structure

The informational ordering over types is extended to exploration states by the *subsumption* relation. Subsumption as a formal inference operation is widely used for reasoning[5]. The subsumption ordering relation can answer what superclass a given class has according to its set of attributes. In the context of the exploration formalism, subsumption is a relation of implication which relates more specific to more general states of exploration. Thus, like inheritance over types, subsumption

[5]For example, description logics [Lambrix 1996] provide a reasoning operation called subsumption. Subsumption also provides a powerful tool for case-retrieval and standards processing [Hakim & Garrett 1993].

defines a partial ordering over exploration states (feature structures). This ordering of feature structures is represented as a directed graph[6]. The hierarchical graph defined by subsumption over feature structures is such that a child node may have more than one parent node. The exploration formalism provides a structuring relation based on subsumption to order collections of exploration states. This ordering relation provides an invariant structure to the design space. Here, the subsumption relation may be seen as a generalisation relation. Thus, in a given design space structured by subsumption, a subsumer state expresses a generalisation over the subsumed state. In it, two design states (and recursively their subparts) are related if one subsumes the other, that is, if one contains strictly less information than the other. This subsumption-based design space structuring mechanism is reported in Burrow's [2003] thesis. The key feature of this structuring mechanism is that the subsumption relation captures a more generic relation than the derivation relation that structures the spaces of designs generated by a grammar. A detailed view of this fact is set out in [Woodbury et al. 1999].

Formally, an exploration state represented by a feature structure F *subsumes* another F', written $F \sqsubseteq F'$, if and only if: for every path π defined in F, π is defined in F' and the type at π in F subsumes the type at π in F'; and for every pair of paths π and π' defined in F, if π and π' identify a single substructure in F then π and π' identify a single substructure in F'. Thus, the subsumption relation $\sqsubseteq$ is a pre-ordering on the collection F of design states. The subsumption relation is transitive and reflexive, but not anti-symmetric, since two states can mutually subsume each other.

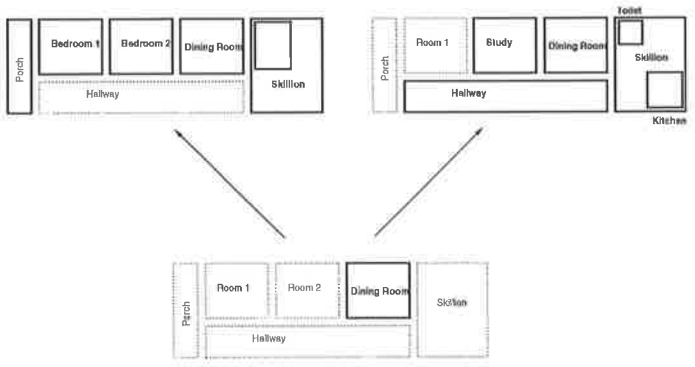


Figure 2.8: A pair of simple feature structures in a subsumption relation [Woodbury et al. 1999].

Figure 2.8 depicts a pair of simple feature structures in a subsumption relation. In the example, the bottom state of a design subsumes the top two because every functional role in the first is present in the second. Every object fulfilling multiple functional roles in the first fulfils a superset of these roles in the second. Finally, for every functional role identified in the first the object fulfilling this role in the second is of a matching or more specific type [Woodbury et al. 1999, p 296]. One

[6]See Appendix A.

advantage of representing the structure of such an ordered collection of design states is decidable subsumption. Deciding subsumption, that is, whether $F \subseteq F'$ can be accomplished in time linear to the size of F [Carpenter 1992, p42]. The formal machinery underpinning the design space description formalism is summarised in Figure 2.9. π-resolution captures the satisfiability relation from descriptions to the satisfiers and is discussed in Section 2.4.1.

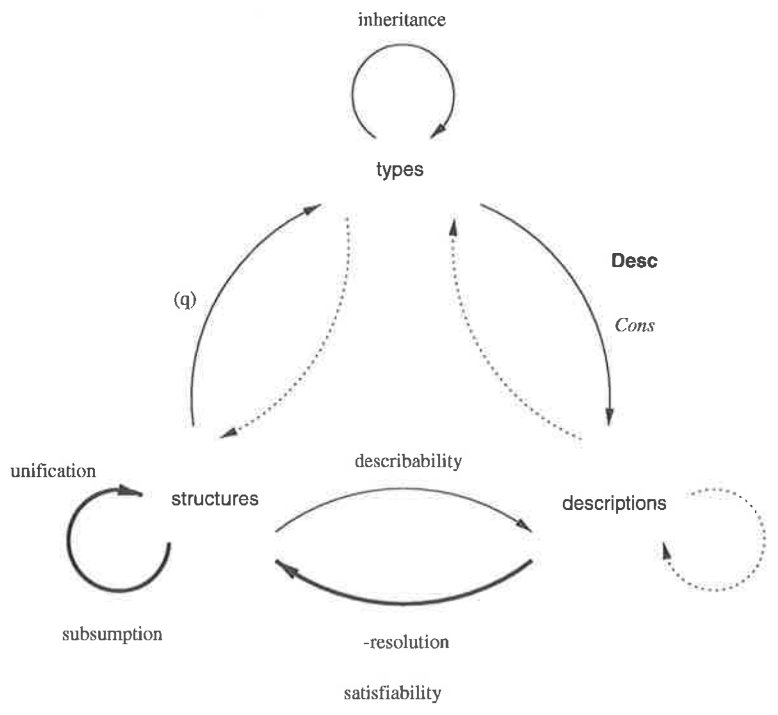


Figure 2.9: Types features, descriptions and the relationships of subsumption and unification showing the computation paths in the scheme.

2.4 Algorithms for exploration moves

Over the three elements, types T, structures F and descriptions D, are posed inference algorithms fo generating, modifying and traversing exploration states. Based on these algorithms, a set of composable operators, namely, incremental π-resolution, indexing and path reuse, design unification, design anti-unification and hysterical undo are proposed for computing exploration. Rule-based derivation as known from grammar-based theories of generation are replaced with these composable operators [Woodbury et al. 2000].

2.4.1 Incremental π-resolution

The process of generating partial structures from a description is formalised in the resolution procedure, P, called π-resolution. This is illustrated in Figure 2.2. The generating procedure

P captures a relation from descriptions to structures $P : D \rightarrow F$. Incremental π-resolution is developed by Burrow [2003] as a special case of the general approach to π-resolution described in Carpenter [1992, p 227–242]. See Appendix A for a formal definition of incremental π-resolution[7].

In design space exploration, incremental π-resolution is the computation of exploration states compatible with a given set of types and a given description. The procedure formalises the notion of generation where (partial or complete) structures are resolved into more specific structures. Woodbury et al. [1999] and Woodbury et al. [2000] develop this view of incremental π-resolution for the generation of intermediate exploration states.

The mechanics of incremental π-resolution represents each step of resolution [Burrow & Woodbury 1999]. The resolution procedure P acts incrementally by generating partially resolved structures, called *partial satisfiers*, with respect to the initial input description, D. Given the query description D, incremental π-resolution is the search across a sequence, S of feature structures $P_0 \sqsubseteq P_1 \sqsubseteq P_2 \sqsubseteq \ldots \sqsubseteq P_k$ that, at least partially, satisfy the description. The resultant feature structure P_k in each sequence is the most general satisfier of the query description. The process of generating partial satisfiers is expressed as a sequence of resolution steps. Each sequence records the resolution of a type constraint explicitly.

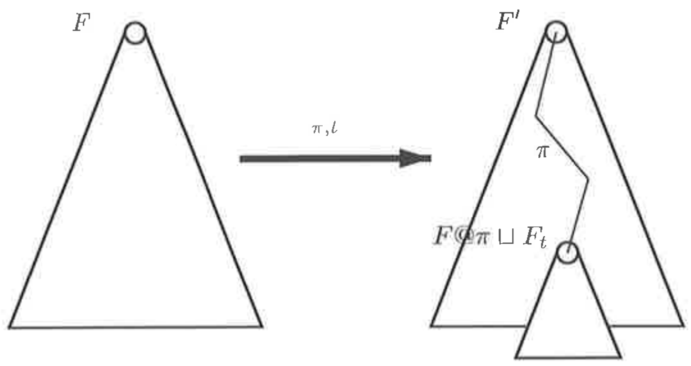


Figure 2.10: The generating procedure, π-resolution captures a relation from descriptions to the satisfiers and is the main generative mechanism in the system. The resultant feature structure in each sequence is the most general satisfier of the query description [Burrow & Woodbury 1999].

Each element of S extends its predecessor by unification with a type constraint. Since most general satisfiers may occur as collections and unification may fail, the search for resolved feature structures involves a collection of sequences. The π-resolution algorithm maps descriptions into sets of feature structures that satisfy them, taking into account a type hierarchy, including the recursive type constraints defined within it.

The mechanism of incremental π-resolution in the context of design space exploration is shown in figure 2.10. The primary generating procedure, P begins with an initial query description D. The argument π selects the substructure, and the argument t selects the constraint to resolve.

[7]See Carpenter's [1992, p 231] definition of π-resolution.

The definition of a step includes restrictions that ensure resolution steps are goal directed and well ordered against the type hierarchy. Namely, that t be a super type of the type at π and that all types more general than t are already resolved at π. The execution of a step involves unification at the substructure.

The satisfaction of the description generates a collection of sequences of partial satisfiers, S. Each step of resolution now explores the consequences of extending members of the sequence S. To support exploration, the incrementality of the generating procedure now requires user guidance at two levels. First, support for the selection of a partial satisfier from the sequence, S and second, the selection of the type constraint to resolve against. Interaction with the π-resolution operator is described in Section 6.2.2.

2.4.2 Indexing and reuse

Indexing and reuse, incorporating the retrieval of previous paths of exploration, are supported in the description formalism. The movement operations based on incremental π-resolution described in Section 2.4.1 maintain information integrity and consistency in design space. Retrieval and adaptation mechanisms suggested by the exploration formalism are structured by efficient computations in the mechanism based on the properties of describability, satisfiability, unification and subsumption. Hence, it is possible to recover the results of previous paths of exploration in a systematic manner. Indexing operates on two levels. First, the recall of paths corresponding to the *type* of the current node. In this case, the type of the node acts as an index to retrieve past exploration paths. Second, this operation deals with tracing the exploration *history* associated with the current exploration state in the design space. Indexing and reuse are by-products of the underlying subsumption ordering. In the second case, reuse corresponds to exploring the evolution and history of the current state stated in terms of the paths of exploration. This operation allows the reuse of previous exploration paths through the function $path(\pi)$. Through this operation, feature structures denoted by any path in the exploration can be retrieved for reuse. For example, as shown in Figure 2.11, the darker nodes represent earlier commitments into the design space. When the feature nodes indicated by the lighter segments are explored, the previous commitments are available for reuse based on the subsumption ordering.

In the description formalism, both cases and their indices are indistinguishable from a structure. Thus, case retrieval is achieved through navigation and query over the subsumption ordering of exploration states. Case adaptation is achieved through reuse of a path from one thread of exploration in another. A detailed discussion of indexing and reuse of cases through the mechanics of typed feature structures is set out in [Woodbury et al. 1999, Woodbury et al. 2000].

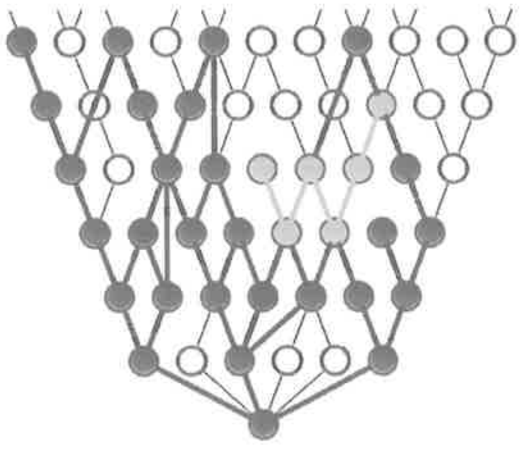


Figure 2.11: Indexing and reuse are by-products of the underlying subsumption ordering. For example, the darker nodes represent earlier commitments into the design space. When the feature nodes indicated by the lighter segments are explored, the previous commitments are available for reuse based on the subsumption ordering [Woodbury et al. 2000].

2.4.3 Hysterical undo

Conventional design support systems provide *undo, delete* mechanisms as means of information removal. Undo is the reversal of the last operation performed. Delete applies to a selection and removes objects from persistent memory. History is used as a rudimentary form of version control over the recorded list of operations. In generative systems, erasure is provided by substitution or transformation rules that replace or transform a more detailed symbol set with a more abstract one. The notion of *hysterical undo* in subsumption based exploration addresses the need for dealing with information removal in a monotonically structured design space [Woodbury et al. 1999, Woodbury et al. 2000].

The *hysterical undo* operation is a novel backward navigation technique. In a subsumption-based representation, removal corresponds to the uncovering of more general states in the implicit design space. Thus, the state from which information is removed remains unchanged in the design space while the designer's perspective shifts to an altered, less specific state. A discussion of the motivations for such a concept of erasure in a subsumption ordered design space is reported in [Woodbury et al. 2000]. Briefly, the notion of information deletion during exploration corresponds to a composite operation involving path retraction and type reduction. If a commitment is referenced by a single path, it may be deleted by retracting that path. This has the effect of purging the object and reversing the forward refinement operation performed by incremental π-resolution. Since, the reversal is performed on a currently selected context, this form of erasure can be performed across the width of the design space. The interest in using erasure as a form of movement in design space becomes apparent when removal results in multiple possible states (there may one or more than one state) to move to. Figure 2.12 shows the resulting space that might be uncovered by such an erasure operation. If a current object is referenced by a multiple paths, only one of the paths may be retracted, and the user presented with a set of features that may be retracted to a previous state.

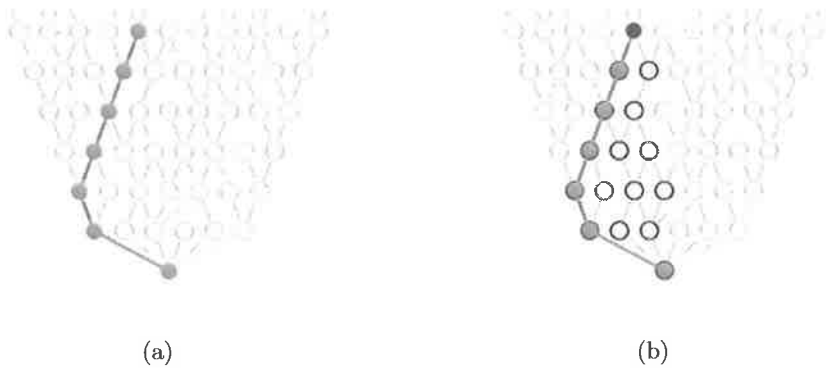


Figure 2.12: Exploration through information removal. The sequence of nodes indicate a path of exploration in (a). The application of the erasure operation on the current node yields a number of possibilities, shown in (b) [Woodbury et al. 2000].

2.4.4 Design unification

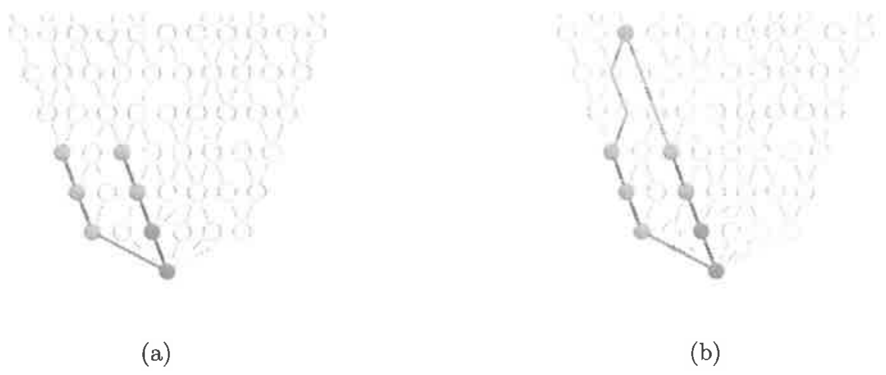


Figure 2.13: Exploration by combining partial designs. Unification extends the two derivation paths, shown in (a). An explicit join of those paths, results in a new node that combines the information of both these paths, shown in (b) [Woodbury et al. 2000].

The combination of two partial satisfiers representing designs is another form of movement operation available in design space. This combination is performed through the operation of *design unification*. Unification plays an internal role in π-resolution as the sole information combination mechanism. At that scale of operation, over closely related feature structures, it is a highly efficient operation. Since unification is defined with respect to the ordered collection of feature structures as a *least upper bound*, it is also possible to conceive unification as a formal means of combining two partial satisfiers [Woodbury et al. 2000]. Figure 2.13 illustrates what is meant by design unification. Given two derivation paths, the *design unification* of the two partial satisfiers is the result of combining the derivation steps of each path into a single partial satisfier containing the union of design commitments in the two operands. When a combination of paths is consistent with the information expressed in the type system, then design unification extends the two derivation paths to create an explicit join of those paths.

2.4.5 Design anti-unification

The recovery of the common features in two partial designs is another form of movement operation available in design space. This operation is the inverse of design unification. Anti-unification is defined with respect to design space as the most specific feature structure generalising the operands. Thus, the result of an anti-unification operation over two partial satisfiers in design space is a movement to a partial satisfier representing the *greatest lower bound* of the first two satisfiers. Since there is a single greatest lower bound no backtracking or reordering is required. By definition, exploration states in design space are guaranteed to define a greatest lower bound. The search simply tests each derivation step for satisfaction by both operands. The result of this operator is the conjunction of the shared derivation steps, which in the extreme case is simply the minimal exploration state subsuming the operands. Figure 2.14 illustrates the workings of the design anti-unification movement operation.

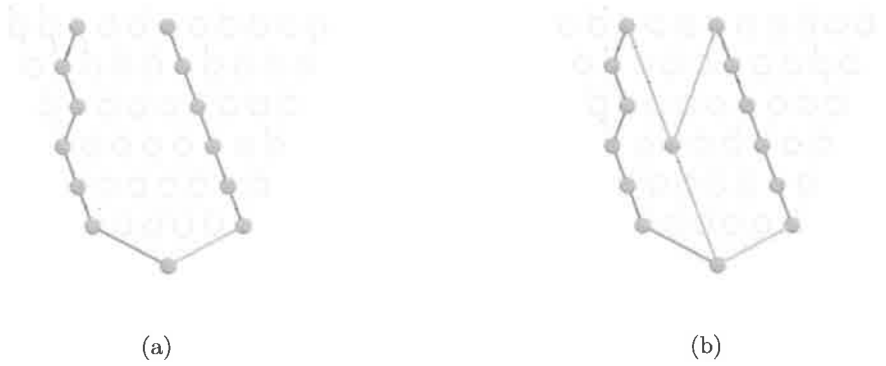

(a) (b)

Figure 2.14: Exploration by design anti-unification. The two distinct paths of exploration are shown in (a). The most general specialisation of the two paths, computed through anti-unification is shown in (b) [Woodbury et al. 2000].

2.5 Summary

This chapter describes how the entities of state, move, structure identified in chapter 1 are formally represented in a description formalism. Typed feature structures provide the formal substrate for a design space exploration formalism. First, the formalism supports the representation of exploration states. An exploration state retains the feature structure properties of intentionality and partialness. Second, the collection of exploration states, the design space, retains an invariant structure based on subsumption. Significantly, this ordering is decidable in linear time and supports several movement operations over design space. Third, the description formalisn supports a set of composable operators, namely, incremental π-resolution, indexing and reuse, design unification, design anti-unification and hysterical undo for generating, navigating and modifying exploration states. Given this representation of state move and structure in a description formalism, the requirements for user interaction with the design space exploration are addressed in the next chapter.

Chapter 3

Mixed-initiative Interaction

In this Chapter, the requirements for a model of interaction for computational exploration are developed. These requirements are addressed through a mixed-initiative formulation, interleaving human guidance with the design space description formalism. The mixed-initiative paradigm of interaction, identified in Section 1.3, presents a promising approach for addressing user interaction with the description formalism. The mixed-initiative interaction model is addressed through the following layers, a representation of the *domain*, a communication layer for *dialogue* between the user and the formalism and operations for performing the *tasks* associated with exploration.

3.1 Interaction with a description formalism

An interaction model addresses communication, coordination and control issues arising out of interaction between a designer and the formal substrate of the description formalism.

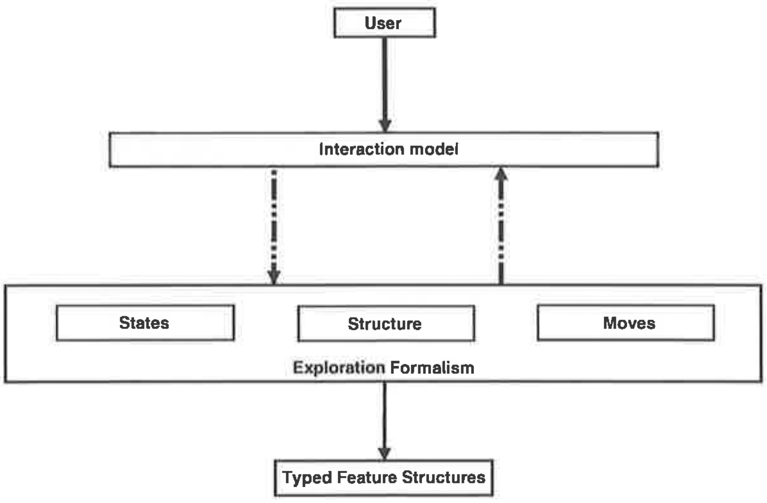


Figure 3.1: An interaction model integrates the user and the description formalism.

The description formalism described in Chapter 2, provides a rigorous formal substrate for supporting the entities of exploration, *state, move* and *structure.* To define how designers may employ the entities of the substrate at the user level, a model of interaction, as shown in Figure 3.1 is necessary. In addition to these, the requirements for communication, coordination and control of the exploration process are addressed through an interaction model.

3.1.1 Unfolding design spaces

Before identifying the requirements for an interaction model for exploration, an explanation of the conceptual nature of interaction envisioned in this thesis is necessary. Recall that exploration comprises interaction with the formal substrate: *states, structure* and *move.* The integration of the human designer with a formalism requires a uniform treatment of the role of the user and the formalism in exploration. To characterise this conceptual integration of user and formalism, the term *unfolding* is introduced in the thesis. Unfolding is defined here as the interactive process of exploring design spaces such that user actions and formal moves are seamlessly integrated. Unfolding as a metaphor thus refers to the exploration of design spaces, without distinguishing between users, human or formalism. Throughout the thesis, the term is used in a metaphorical sense and thus not defined formally.

For example, exploration can proceed by formal moves as well as user moves. In the context of exploration moves, unfolding can be further subdivided into *generation*, *navigation* and *synchronisation.* A detailed description of the characterisation of unfolding in the context of exploration moves is given in Section 6.1. A concrete discussion of the scope of the term, unfolding as used in the thesis is covered in Section 8.2.3.

To foreshadow its use in the study, the term *unfolding* simply refers to the conceptual metaphor of interaction that integrates (or does not distinguish between) a human designer with the machinery of formal exploration. Henceforth, the use of the term unfolding refers to the conceptual binding of machine and human capabilities for design space exploration. A specific realisation of unfolding, a mixed-initative interaction model for design space exploration, is developed in this study. The requirements of this model of interaction are described in the next section.

3.1.2 Requirements

An interaction model for design space exploration must address the following requirements:

- **Connect the designer's view of the exploration domain with the formal substrate.** The formalism supports the representation of an exploration state through the concepts of *types, features, descriptions* and *feature structures* as explained in Section 2.2. The interaction model must enable the designer to access these constructs in the substrate.

  Currently, the major role of the user in the domain is in two areas. First, type hierarchy specifications which encode domain knowledge into the description formalism. Second, in the authoring of descriptions that define problems to explore in the context of the former. Further, a clear distinction emerges between the creation of the type system by the user, the authoring of descriptions and the exploration of the implicature of descriptions by the formalism. To address the above, the interaction model proposes to extend the role of the user into the process of seeking goals during exploration.

The designer's view of exploration sits above the formal machinery. The interaction model must address how these are connected. At the substrate, both problems and solutions are reduced to exploration in terms of state, move and structure. Therefore the interaction model must provide support for the designer in problem formulation, solution generation, choices of alternatives and interaction with exploration history.

- **Support the designer in the tasks of computing exploration.**
  The inference algorithms of unification and constraint resolution provide entry points for developing interactive exploration operations. In particular, the incremental nature of the inference algorithms enables the designer to generate, navigate and modify intermediate states. The interaction model must enable the user to access the formal operations for computing exploration described in Chapter 2. A set of interaction level concepts that define and frame the exploration moves through mixed-initiative combining the designer's view with the formal movement operators is necessary.

- **Facilitate communication and coordination between the designer and the description formalism.**
  The user and the formalism must be able to communicate during exploration. Communication is enabled through information sharing and exchange. These must be supported through input and output mechanisms in the interaction model. During exploration, input and output may take multiple modalities. For example, spoken input, typed commands or the direct manipulation of graphical symbols. All modalities of input can be interpreted in a common symbolic representation. The same applies to different modes of output, whether generated speech, natural language explanations or graphical visualisations. Mixed-initiative must support the requirements of compensation during dialogue, for example, allowing the formal generator to compensate for errors in user input or vice versa.

  The interaction model must address how coordination between the user and the formalism is handled during exploration. Both user and formalism must have the flexibility to acquire or relinquish control of exploration tasks during exploration. A control strategy for coordinating the tasks of exploration must be supported in the interaction model.

3.1.3 The role of mixed-initiative

The role of *mixed initiative* for addressing the requirements for interaction are addressed. In the context of the discussion in Section 3.1, mixed-initiative presents a paradigm,

> *for combining a human designer and a description formalism through the specification of communication, coordination and control of the exploration process.*

Through mixed-initiative, designers can acquire initiative to provide context-dependent or situation-specific domain information that may be difficult to encode apriori in real-time problems for exploration. The description formalism is able to take or relinquish initiative during interaction to perform automated processes and repetitive tasks. The formalism can structure, provide access to and elicit formal knowledge from human designers. The formalism can archive the results of exploration, access records of past explorations, generate design rationale and documentation. Together, the designer and the formalism can share responsibility over tasks, recover gracefully from errors, reformulate problems and prune unproductive paths of exploration.

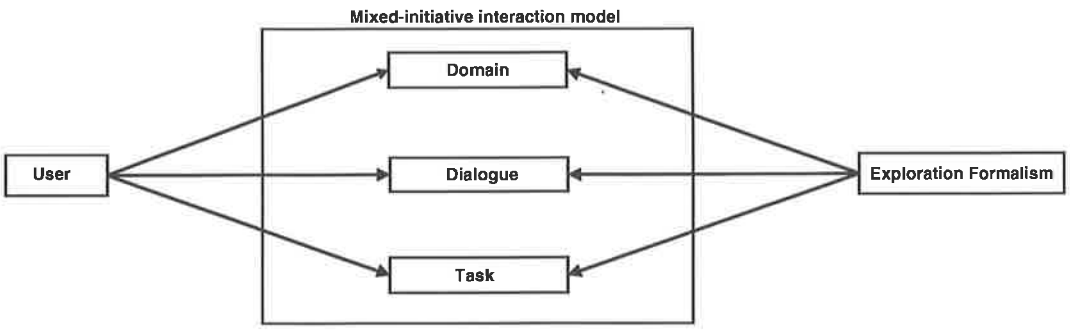


Figure 3.2: The components of a mixed-initiative formulation for interaction between the user and a description formalism.

To address the above, a three-layered model is developed for interactive exploration. The layers of the interaction model are shown in Figure 3.2. Each layer plays a distinct role for addressing the requirements for mixed-initiative exploration.

Domain Layer.
The designer's view of exploration and its connection to the formal substrate in the interaction model require the development of a *domain layer.* The role of mixed-initiative in the domain layer, the extensions necessary to implement mixed-initiative and the role of the domain layer in the interaction model are described in Section 3.2.

Task Layer.
The *task* layer aims to implement a specification for the tasks of exploration by integrating user interaction with the design space exploration formalism. The role of mixed-initiative in

the task layer, the extensions necessary to implement mixed-initiative and the role of the task layer in the interaction model are described in Section 3.3.

Dialogue Layer.
The *dialogue* layer aims to implement a common basis for extended interaction between the designer and the formalism. The role of mixed-initiative in the dialogue layer, the extensions necessary to implement mixed-initiative and the role of the dialogue layer in the interaction model are described in Section 3.4.

3.2 The Domain Layer

The *domain* layer implements the glue that connects designer level constructs (collectively identified as the entities of exploration) with parts of the description formalism that realise these constructs in the formal substrate. The domain layer contains the primitive entities about the application domain such as concepts, attributes, roles and relationships as seen from the designer's view of exploration. This view is less concerned with the formal specification of internals and more with the existence of objects and the external hooks necessary to support interactive exploration. In the following sections, the attributes of domain initiative, the extensions necessary for implementing domain initiative in design space exploration and the role of the domain layer are described.

3.2.1 Attributes of domain

The literature on mixed-initiative reviewed in Section 1.3. Mixed-initiative is an effective paradigm for addressing the process of directing problem-solving goals [Cohen et al. 1998] in a domain of discourse. Through mixed-initiative, the user and the formalism can share responsibility over domain goals. For example, Rich & Sidner [1997] and Rich & Sidner [1998] demonstrate a domain level collaboration through an interface agent that works on a plan with its user. Veloso [1996] and Veloso et al. [1997] employ a shared representation in the planning domain. Both automated and human planners are able to interact and construct plans jointly. Smith & Hipp [1994] propose a common meaning representation to achieve goals in natural language dialogue through mixed-initiative. Guinn [1996] considers initiative over mutually shared goals and how goals are solved by the participants (agent and human) in spoken dialogue systems.

Mixed-initiative over a domain goal requires both humans and automated software to share a representation of the domain of discourse. Such a common meaning representation over the domain enables users and computational agents to share domain knowledge and therefore collaborate on achieving goals jointly through interaction. In the description formalism, this implies an interaction layer that can connect the designer's view of the domain with the symbol level constructs available for computing exploration. In the next section, the extensions necessary for achieving a common meaning representation of the domain of discourse are discussed.

3.2.2 Extensions

Thus, the domain layer puts in place a set of interaction level concepts that define the designer's view of the domain of discourse. Recall the designer's view of exploration, identified in Section 1.1.2. This view comprises an account of problem formulation and solution generation. The description formalism does not distinguish problem from solution. Formal exploration abandons the problem/solution division, replacing it simply with states of exploration that may be either problems or their solutions. At the knowledge level, the distinction remains very meaningful. The interaction model must be able to bridge the gap between the knowledge level formulation of the designer's view of the exploration domain and the symbol level substrate of the description formalism. User exploration requires an explicit representation of the design requirements to be satisfied. The solutions that satisfy some or all of the requirements are then generated from the initial definitions of the problem. Therefore the designer must be able to decompose the problem into subproblems, revise the initial problem, formulate a new problem or reformulate the existing problem. In addition to working with problems, the designer must be able to revise and reformulate problems interactively and dynamically during exploration. The system must be able to assist the designer in the generation of solutions that satisfy the requirements and present these solutions to the designer in a structured manner. Thus user exploration encompasses the formulation of requirements and the generation of solutions based on these requirements. The structure of exploration is represented through the ordering relation of *subsumption*. The concept of an ordered design space underpins the description formalism. In it, the collection of exploration states are ordered by the relation of subsumption. In Section 2.3, the ordering of exploration structure through subsumption is described. The subsumption ordering over the collection of partial satisfiers provides an entry point for navigating the design space of partial satisfiers. Choices, their connections and the developing history of explicitly discovered design alternatives must be accessible to the designer through interaction with the structure of exploration. The interaction model must provide the designer with a view of design space structure. From the designer's perspective such a model must capture at least the elements of the history of design exploration. As Burrow & Woodbury [2001] argue, history is necessarily the primary device for supporting exploration. In their account, a *trajectory* records the co-option of design features and their assignment to roles. The designer must be able to exploit this history through navigation and recombination of the paths of exploration. In the description formalism, subsumption preserves information specificity relations in design space. The interaction model must provide an account of the intentional choices made by the designer during problem formulation and solution generation. Thus support for navigating the history of exploration is necessary.

The domain layer provides a sound representation of problem states, partial designs, choice-making and exploration space. These constructs encapsulate the entities of exploration from the designer's perspective and connect the designer to the formal substrate. Following the discussion of interaction requirements and the extensions necessary to address these requirements above, the role of the mixed-initiative domain layer in the interaction model for design space exploration is discussed.

3.2.3 Role of the domain layer

The role of the domain layer in the mixed-initiative model of exploration is as follows:

Designer's view of the domain.
Explicit support for the user by supporting the designer view of exploration in the form of problems, solutions, choices and history. The domain layer provides concepts for the representation of problems, their reformulation and the generation of alternative solutions from the user's perspective. This shared representation acts as a layer of mediation between the user and the formalism. The common representation of the domain layer mediates between the designer's view of exploration comprising problems, solutions, choices and history; and a design space representation aimed at efficient generation, indexing and recall.

Joint responsibility over goals.
Joint responsibility over domain goals is a major requirement of the mixed-initiative paradigm. The domain layer enables both the designer and the formalism to maintain context and share responsibility over goals in the domain of exploration. The description formalism supports a generic view of exploration. The domain layer allows the designer to tailor this machinery to specific goals in a domain of discourse.

The detailed development of the domain layer of the mixed-initiative interaction model addressing the above is developed in Chapter 4.

3.3 The Task Layer

The *task layer* addresses user access to the formal operations for computing exploration as described in Chapter 2. The task layer puts in place a set of interaction level concepts that define and frame the exploration tasks through mixed-initiative combining the designer's view with the formal movement operators. In the following sections, the attributes of mixed-initiative in the task layer, the extensions necessary for implementing task initiative in design space exploration and the role of the task layer in interactive exploration are described.

3.3.1 Attributes of task initiative

The coordination of tasks based on mixed-initiative interaction reviewed in Section 1.3 is reported in a number of applications. In the domain of planning, the TRAINS [Ferguson et al. 1996] and TRIPS [Ferguson & Allen 1998] implementations demonstrate how mixed-initiative coordinates tasks between the user and system using joint responsibility over a shared task. Mixed-initiative task formulation enables domain goals to be achieved more quickly and with greater reliability [Burstein & McDermott 1996]. Tecuci et al. [1999] report the application of mixed-initiative to the task of knowledge acquisition in knowledge engineering. They note that manual solutions to the problem of building knowledge bases remain highly inefficient while purely automated systems for the task are impractical. Mixed-initiative provides a feasible alternative combining the advantages of manual and automated methods. Novick & Sutton [1997] develop a multi-factor model of initiative where choice of task determines what the conversation is about and choice of outcome allocates the decision or action necessary to achieve the task. Task initiative provides a number of benefits in realising interaction with formal systems.

First, it enables a productive syntheses of the complementary strengths of both humans and machines. The mixed-initiative formulation enables a combination of human and machine competencies over a shared task. Joint responsibility over a shared task is more productive than a pure division of labour between user and machine. Through mixed-initiative, brute force portions of a single task can be allocated to the machine, and soft tasks such as conflict resolution, error recovery and compensation are allocated to the human. Second, mixed-initiative enables a more robust and efficient framework for achieving goals than that possible through a purely automated or purely manual approach. Third, the goals to be achieved in a domain may have incomplete specifications, change dynamically, and evolve simultaneously with the execution of the task. In such situations, mixed-initiative in the task layer permits the human or the system to allocate or cede control of a task or initiate a new task based on changing goals.

In the description formalism, this implies an interaction layer that can distribute an exploration task over two components, the designer's level and the symbol level operators. In the next section, the extensions necessary for achieving a common meaning representation of the domain of discourse are discussed.

3.3.2 Extensions

As outlined above, a set of formal operators for computing exploration tasks is supported in the formalism. To employ them in exploration, the movement algorithms need to be stated in terms of the designer's domain of discourse. The extensions necessary to implement mixed-initiative in the task layer are divided into three principal constructs, namely, *generation*, *navigation* and *synchronisation*. The generation task corresponds to the process of constructing problems, reformulating problems and generating partial solutions from these problem statements. The description formalism provides the machinery necessary for representing the generative process through the incremental π-resolution algorithm. Incrementality of this process provides an entry point for incorporating the designer into the process of π-resolution. The construction of problems, their reformulation and the generation of partial solutions are addressed in the task layer.

The navigation task corresponds to retracting attributes of a partial design and making choices on possible alternatives. The description formalism provides the machinery for representing alternatives through disjunctive descriptions. These alternatives introduce non-determinism into a purely formal exploration process. The formulation of navigation in the task layer permits the use of mixed-initiative in resolving exploration non-determinism. Through mixed-initiative, the user is able to select one of a number of possible alternatives that arise from formal resolution.

The ability to exploit the structure and history of designing is an important task in supporting exploration. The synchronisation task covers the process of indexing and reuse of exploration results. Through mixed-initiative the designer is able to access the formal movement operations of recall, hysterical undo and the unification or anti-unification of two partial solutions.

Mixed-initiative in the task layer must permit the user to access the formal exploration operators, integrate system-driven and user-driven moves and enable the user and the formalism to share joint responsibility. During exploration, system-driven moves represent the formal operations for moving in design space. Designer-driven moves represent operations that access and compose the moves available in the symbol substrate.

The task layer provides a sound representation for sharing the tasks of generation, navigation and synchronisation over problem states, partial designs, choice-making and exploration space. Each of these constructs encapsulate a task of exploration from the designer's perspective and connect the designer to the formal moves available in the formalism. Following the discussion of interaction requirements and the extensions necessary to address these requirements above, the role of the mixed-initiative task layer in the interaction model for design space exploration is discussed.

3.3.3 Role of the task layer

Mixed-initiative provides a formulation of the the role of the user, the role of the formalism in generating, navigating and synchronising the results of exploration. The conjunction of the tasks of exploration at the level of design intention with formal moves of exploration is developed in the task layer of the mixed-initiative model. This layer allows the designer to exchange, compose and coordinate a range of exploration tasks in association with the formal movement algorithms. The role of the task layer in the mixed-initiative model of exploration is as follows:

Support construction and reformulation of problems.
The task layer incorporates the user in the construction of problems, and the generation of partial solutions. The domain layer provides concepts for the representation of problems, their reformulation and the generation of alternative solutions from the user's perspective. In the task of generation, interaction comprises the specification of problems and the incremental generation of partial solutions.

Support navigation of problems and solutions.
The user must be able to navigate both problem states and partial satisfiers. Navigation corresponds to interaction with the operations for movement along paths of exploration. The task layer supports the ability of the designer to make choices at branch points in exploration.

Support synchronisation of exploration results.
The designer and the formalism must be able to synchronise previous exploration paths. The task layer supports the synchronisation of exploration results between the designer's actions comprising problems, solutions, choices and history; and the description formalism tasks of indexing and reuse, undo, unification and anti-unification based on the history of exploration.

The mixed-initiative task layer for supporting exploration is developed in Chapter 6.

3.4 The Dialogue Layer

The dialogue layer provides support for communication, coordination and control of an exploration process between the formalism and the designer.

In the following sections, the attributes of mixed-initiative in the dialogue layer, the extensions necessary for implementing dialogue initiative in design space exploration and the role of mixed-initiative dialogue in interactive exploration are described.

3.4.1 Attributes

The key notion underpinning dialogue representation in mixed-initiative interaction is conversational structure. Communication and coordination between the user and the system is established through a shared representation of discourse. A common representation of dialogue enables the participants in the dialogue to negotiate reference and confirm mutual understanding. A conversational model of dialogue enables the possibility of *extended* interaction [Allen 1999] between the user and the system. Grice's [1975] maxims of rational conversation is one such formulation. The literature on conversational structure is large and its review is beyond the present scope. It suffices here to have a model suitable for organising mixed-initiative interaction with a computer. For this, rational conversation is a good model and Grice an exemplar.

Grice [1989] observes that human dialogue is characterised by rationality, cooperation, common purpose and direction. He states,

> "Our talk exchanges do not normally consist of a succession of disconnected remarks, and would not be rational if they did. They are characteristically, to some degree at least, cooperative efforts; and each participant recognises in them to some extent, a common purpose or set of purposes, or at least a mutually accepted direction." [Grice 1989]

Based on this observation of human conversation, he formulates a set of conversational maxims. The first is *quality*, implying truthful and accurate information. The second is *quantity*, neither more nor less information than is required for the dialogue. The third is *relation*, only information appropriate to the task is considered. Finally, *manner*, clear and unambiguous information is necessary for conversation. Grice's model of rational conversation forms the basis for addressing communication, coordination and control issues in the interaction model.

The concept of turn-taking is based on shifting, tracking and allocating a thread of control among dialogue participants, machine and human. Novick & Sutton [1994] propose a computational model of dialogue that utilises meta-locutionary acts, such as give-turn, clarify, and confirm-mutual. Rich & Sidner [1998] use mixed-initiative to acquire and transfer control during dialogue between a collaborative interface agent and its user. Chu-Carroll & Brown [1997*b*] and Chu-Carroll & Brown [1997*a*] present a model for tracking initiative in dialogue between participants. Hartrum & DeLoach [1999] propose two types of turn-taking in mixed-initiative dialogue, transactional and incremental. The transaction-based model corresponds to a single query request and the response to the query. The incremental model corresponds to several agents, sharing and writing data to a common resource. The computational agent displays the information on the screen and dynamically updates it as other agents (human and computational) submit incremental changes.

3.4.2 Extensions

This section sketches how the attributes of mixed-initiative dialogue are addressed in the dialogue layer of the interaction model. The dialogue layer must support communication and coordination between the designer and the formalism. Representing the input and output modalities of dialogue between the user and the formalism is a key requirement of mixed-initiative in the dialogue layer. The user or the formalism must be able to communicate and coordinate the input and output modalities during dialogue. The input and output modalities of the description formalism are clearly defined in the terms of the formal substrate. Descriptions and partial satisfiers, the base representation of input and output from the formalism are, ipso facto, given in terms of feature structures. The dialogue layer must provide an account of the input and output modalities of the designer. A common representation that can unify both modalities of exploration is necessary.

A transparent exposition of the modalities of the formalism in a visual manner is one way of expressing and integrating the input and output modalities of both user and formalism. This approach is similar to the work of Piela [1989] in the ASCEND modelling system [Piela, McKelvey & Westerberg 1993]. ASCEND provides a visual and direct manipulation interface for developing and testing incremental constraint programs in the domain of process engineering. In the ICE project [Zeller & Snelting 1995, Zeller 1997] an interactive front end enables the user to construct configuration threads through the addition and modification of configuration constraints. The Oz Explorer [Schulte 1997] is another visual constraint programming tool for supporting the user driven development of constraint programs. A tree visualisation of the constraint problem is the central metaphor for exploration of any constraint node in the tree. The user can tailor and program user guided search engines over this tree for the development of constraint programs. FEGRAMED [Kiefer & Fettig 1995] employs a fully interactive front end that presents the user with a customised view of feature structures. This feature structure editor can be used for developing and maintaining feature structures in constraint based systems. The dialogue layer realises mixed-initiative through a shared visual representation of input and output modalities that is accessible to both user and formalism. This mutually shared context enables the negotiation of reference and provides the glue that binds user actions with the formal substrate.

Given a sound representation of communication and coordination, it is necessary to address the sharing of control over the thread of exploration. An incremental model of turn-taking between the user and the formalism enables both participants in dialogue to acquire, shift and allocate control of the exploration process. The dialogue layer must support a robust structure of turn-taking between the user and the formalism. Admitting Grice's conversational maxims, a tight coupling of user actions with the formal substrate is one way of implementing a control model based on turn-taking. Incrementality and turn-taking enable the best joint interpretation of input and output modalities between the designer and the formalism during exploration.

3.4.3 Role of the dialogue layer

Mixed-initiative in the dialogue layer provides support for communication, coordination and control of exploration between the user and the formalism. Communication and coordination during exploration are addressed through the representation of dialogue. Control of the exploration process is addressed through a model of turn-taking based on rational conversation. The role of the dialogue layer in the mixed-initiative model of exploration is as follows:

Support the representation of dialogue.
A representation of the communication and coordination of input and output between the designer and the formalism. A transparent exposition of the input and output modalities of the formalism is one way of representing dialogue.

Support the integration of dialogue.
A model of turn-taking integrates dialogue between the designer and the description formalism. Through turn-taking, control of a thread of exploration can shift between the user and the formalism. The role of the incremental model of turn-taking is to enable both the user and the formalism to acquire, relinquish, shift and allocate control of the exploration process.

The mixed-initiative dialogue layer for supporting exploration is developed in Chapter 5.

3.5 Summary

This chapter identifies the requirements necessary for developing a model of interaction for computational exploration. To address these requirements, the mixed-initiative paradigm of interaction is proposed for supporting design space exploration. A three-layered mixed-initiative interaction model for integrating user interaction with the design space exploration formalism is developed. In this model, the role of the user is explicitly realised and connected to the formal substrate through the three layers, *domain, task* and *dialogue.* Each layer occupies a distinct role in implementing mixed-initiative and addresses the requirements for the mixed-initiative interaction model. The *domain* layer provides the glue that connects designer level constructs (collectively identified as the entities of exploration) with parts of the description formalism that realise these constructs in the formal substrate. The *task* layer weaves the user with the exploration operations of the description formalism. The *dialogue* layer provides a common basis for extended interaction between the designer and the formalism. In Part II of this thesis, each of the layers of the mixed-initiative interaction model for design space exploration is developed.

Part II

MIXED-INITIATIVE AND DESIGN SPACE EXPLORATION

Part II: Mixed-initiative design space exploration

"Our larger interest in mixed-initiative planning systems grows out of some observations of the strengths and weaknesses of both human and automated planning systems as they have been used.... Humans are ... better at formulating the planning tasks.... Machines are better at systematic searches of the spaces of possible plans..."
Ferguson and Allen [Ferguson & Allen 1994, p 44].

Part II develops a mixed initiative model of interaction based on the requirements of mixed-initiative for supporting design space exploration. It posits the developments of interface level constructs in each layer of the model. The notation of the unified modelling language, UML is used to describe each construct, its connection to the concepts in the formal substrate and its role in mixed-initiative exploration. The exposition is broken into three Chapters as follows:

- Chapter 4 describes the domain layer of the mixed-initiative interaction model. Four domain constructs corresponding to problems, solutions, choices and exploration history are developed. The connection of these constructs to the description formalism are described.

- Chapter 5 develops the dialogue layer of the mixed-initiative interaction model. A single construct, the visual feature node is described. This construct encapsulates communication, coordination and control between the designer and the description formalism. The interaction logic necessary for the dialogue layer of mixed-initiative interaction is addressed through a visual notation for representing feature nodes graphically. Interactions with design states through the direct manipulation of visual feature nodes is described.

- Chapter 6 presents the task layer of the mixed-initiative interaction model. The task layer comprises a collection of interface level constructs for facilitating exploration in terms of the tasks of construction, navigation and synchronisation of exploration states. The integration of exploration tasks with the movement operators in the description formalism is described.

Chapter 4

A mixed-initiative domain layer

This chapter describes the domain layer of the mixed-initiative interaction model. As proposed in Section 3.2, the domain layer captures the designer's view of exploration and ties this view to the formal substrate of the description formalism.

4.1 Representation of the domain

From the designer's perspective, the representation of the domain must account for and connect onto the concepts with which the design space formalism are conceived. A difficulty of explanation arise in this task. The elements of the domain layer collapse into and find explanation in the sparse symbol-level machinery below. One formal device in the substrate serves several concepts in the domain layer. To address these difficulty, it is necessary to maintain three levels in the exposition of the domain layer: the designer's view of the components of exploration, the formal substrate underpinning these views and finally, domain layer concepts that map the user level concepts onto the formal components of the design space formalism.

4.1.1 The designer's view of exploration

The designer's view of exploration, identified in Section 1.1.2, comprises an account of problems, solutions, choices, their connections and the developing space of explicitly discovered design alternatives. The designer's model of exploration comprises problems, solutions, choices and history (their connections and the resulting explicit design space). The problem formulation and reformulation cycle, the solution generation and reuse cycle, the intentional choices of the designer and the rationale of exploration in the form of a history are captured in this view. The representation of the designer's view is shown in Figure 4.1.

Looking up to the designer's model, the domain layer accounts for the major entities of exploration as understood in designing, namely, problems, solutions, choices and history. Looking down

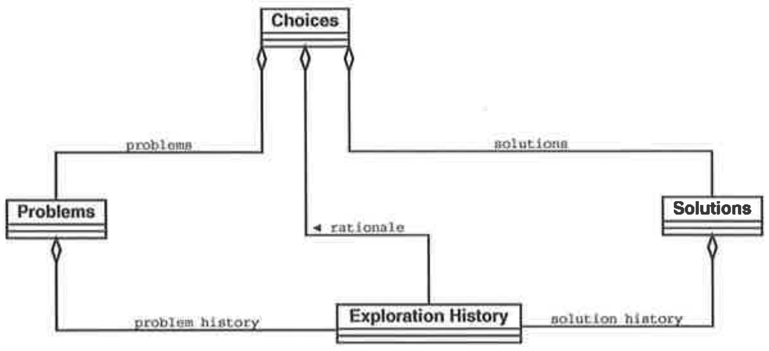


Figure 4.1: The designer's view of exploration can be captured through a representation of the following entities, problems, solutions, choices and history. The problem formulation and reformulation cycle, the solution generation and reuse cycle, the intentional choices of the designer and the rationale of exploration in the form of a history are captured in this view.

to the formal substrate, the domain layer calls on the formal machinery to compute the tasks of exploration asked of it by the designer's model. In this fashion, the domain layer captures both the intentionality of the designer and is fundamentally tied to the rigours of the description formalism. The constructs in the domain layer corresponding to the designer's view of exploration and their relations are explained in the next section.

4.1.2 Domain layer constructs

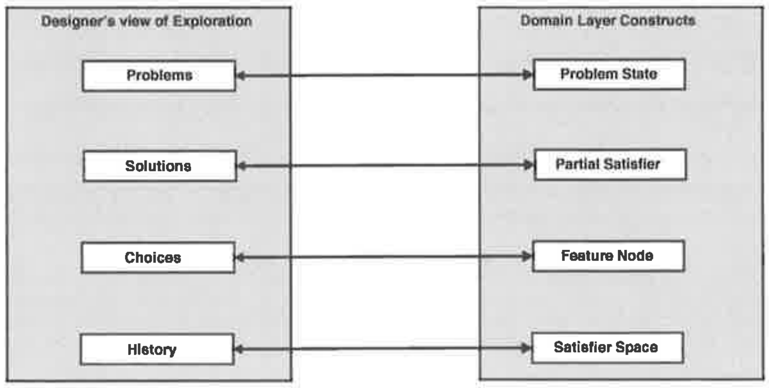


Figure 4.2: Mapping the designer's view of exploration to constructs in the domain layer.

The first task is to map each of the exploration entities into a corresponding construct in the domain layer. As shown in Figure 4.2, this mapping, which corresponds to the domain layer, captures the designer's view of exploration in a knowledge-level representation. The representation acts as an intermediary between the exploration entities and the underlying formal machinery of typed feature structures. This mapping into an intermediary representation comprises four components, namely, *problem state*, *solution state*, *feature node* and *satisfier space.*

The domain constructs in the mapping are defined as follows:

Problem state.
A problem state corresponds to the designer's view of problem formulation. The problem state connects to the formal substrate and supports reformulation. This construct is explained in Section 4.2.

Solution state.
A solution state corresponds to the initial, intermediate and final designs satisfying a problem. The solution state connects the designer's view of a solution to its representation in the formal substrate. The solution state supports generation and reuse and these are explained in Section 4.3.

Feature Node.
The connections between a problem and its possible solutions (partial or complete) are encapsulated in a feature node. A feature node composes a problem state and the uncovered (possible) solutions to the problem state. Through this composition, the feature node represents the intentional choices made by the designer. The feature node connects to the formal substrate and supports choice-making, reformulation, generation and navigation during the exploration process. These are explained in Section 4.4.

Satisfier space.
The intentionality or rationale of exploration expressed by the designer's movement during exploration are captured by the satisfier space. The satisfier space is composed as the collection of feature nodes. The satisfier space connects to the formal substrate concept of a design space. Since, satisfier spaces contain ancestor and progeny feature nodes, a feature node is referred to as a *Satspace Element*, in the context of satisfier space. The satisfier space records exploration history and the choices made by the designer. These are explained in Section 4.5.

These constructs capture a designer model of exploration without imposing the algorithmic and symbol level implications of the formal substrate. Problems become *problem states.* Solutions become a relation between problem states and *partial satisfiers.* Choices become *feature nodes.* Exploration history is captured in a *satisfier space.* However, it is necessary to explain these at a second level, that answers to both the designer's model and to the formal substrate below. In the next section, the mapping from the domain layer constructs to the symbol structures of the underlying machinery is made explicit.

4.1.3 Mapping to description formalism

To enable mixed-initiative, it is necessary to build intermediary relations between the components of exploration and the symbol structures that represent them. The formal concepts of *types, features, constraints* and *descriptions*, explained in Chapter 2, provide the basis for realising the domain layer constructs of the interaction model. Mapping the designer's view onto the typed feature structure machinery is the basis for the manipulation of types, features, structures and descriptions.

This mapping between the designer's view of the entities of exploration and the formal concepts that enable its computation is crucial to supporting mixed-initiative in the domain layer as described in Section 3.2.3. This layer enables the shifting, allocation and tracking of domain initiative in the interaction model through the intermediary concepts outlined above.

The designer's view of exploration are linked into substrate concepts using domain layer concepts described above. To clarify these linkages between domain layer concepts and the underlying feature structure machinery, a path notation is used. This notation provides a concise handle to illustrate how domain concepts reach onto the typed feature structure machinery. The intention of the notation is two-fold. First, to achieve a uniform description of all aspects of the domain layer. Second, to use domain layer constructs as a filter through which only the relevant parts of the typed feature structure machinery can be seen. The path notation comprises the following elements,

Domain layer construct.
This element captures the four domain layer constructs. A *Problem State* is signified by the element, PState. A *Partial Satisfier* is represented as PSat. A *Satspace Element* is a Fnode. A *Satisfier Space* is a SatSpace.

Path connector.
A connection between domain layer constructs is indicated by the path connector, represented by "•". The connections to the substrate are indicated by the path symbol, ":". Through the path connectors, access to the relevant parts of the typed feature structure machinery, through the domain layer constructs is clearly shown.

Substrate concept.
This element captures the constructs in the underlying description formalism onto which the domain layer constructs are mapped. They are represented by TFSConstruct, the prefix "TFS" indicating that they belong to the description formalism.

Through the above notation, the mapping between the domain layer constructs and the underlying formal machinery of typed feature structures is clearly established. Given the preponderance of technical terms in the formalism, the path notation is used to clarify the distinctions between layers whenever necessary. At all other times, symbol level concepts are taken to imply an exposition at the formal substrate level, while domain layer terms are taken to imply an exposition at the level of the designer.

The mapping from the domain layer to the description formalism is visually stated through the UML notation[1] in Figure 4.3. The domain layer mediates between a designer model of exploration

[1]The UML, (Unified Modelling Language) notation is used throughout the thesis to express concepts and their

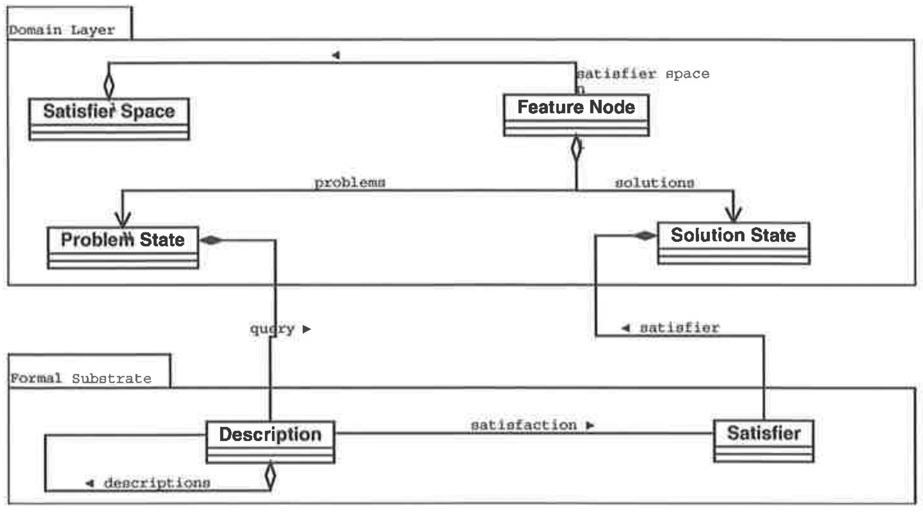


Figure 4.3: Mapping domain layer constructs to the underlying formal substrate of the design space exploration formalism.

comprising problems, solutions, choices and history; and a design space representation aimed at efficient generation, indexing and recall. In the domain layer, *Problem State* represents a design problem. More specifically, the notion of a problem is cast in terms of the machinery of design space exploration, namely, constraint collections written in the form of descriptions. *Solution State* corresponds to partial design solutions. More specifically, the notion of partial, intermediate and final solutions to a problem is cast as the collections of partial satisfiers of a description. Further, *Feature Node* models the connection between problems and the solutions uncovered by designer choices during exploration. This construct introduces the intentionality of the designer into the domain and is crucial for mixed-initiative in the domain layer. Finally, *Satisfier Space* models the history of design exploration as recorded in the collections of feature nodes.

These constructs are developed in greater detail, following a three level exposition, the designer's view, the symbol substrate, and the mapping of the two based on the domain layer. The intention of the exposition is to make transparent the granularity of mixed-initiative in the interaction model at the domain layer.

4.2 Problem State

4.2.1 The designer's view

To a designer, problems comprise both design requirements and desired properties of an artifact. Problems may be hierarchical, that is in addition to requirements and properties they may comprise sub-problems, which themselves may be similarly recursive. Designers revise problems as aspects

relationships in a visual format. To facilitate the reader, a consise summary of the UML notation is given in Appendix B.

of a design situation reveal themselves through exploration, the conception of the actual problem being solved may change. In a designer's problem space, work is done by a combination of problem formulation (specifying (adding) requirements, attributes and sub-problems) and problem revision (removing or modifying the same).

4.2.2 The symbol substrate

At the substrate, no distinction is made between a problem and a solution. This bears explanation. Unlike knowledge level designs such as that of SEED [Flemming & Woodbury 1995] described in Section 1.1.1 such distinctions neither exist, nor are meaningful in the formal substrate once the particular mapping to typed feature structures is made. For example, from the perspective of problem specification, a requirement for daylighting might be stated in different revisions of the problem as the requirement itself, the specification of a certain area of transparency in a part of the building envelope, or as a window design itself. In terms of future exploration for a solution, each of these would act in exactly the same way, that is, as a constraint on the explorations that the system can enact. At the model level (the designer's level) this blurring of the underlying formal bounds is addressed through constructs that distinguish between problem and solution.

At the symbol substrate, problems are specified through two main constructs of the typed feature structure mechanism: a type hierarchy and a description. First, the design requirements, specifications and properties of an artifact to be designed can be specified by the elements of the InheritanceHierarchy[2]. The specification of a problem amounts to the construction of an inheritance hierarchy of types, a collection of feature declarations introducing and appropriate for those types and constraints on types. Thus, the specification of the type hierarchy implicitly constitutes a set of problem formulations that a designer can visit during the course of a given exploration.

Second, a problem to be solved can also be expressed using descriptions drawn from Desc with respect to a type hierarchy, $\langle Type, \sqsubseteq \rangle$. An idiom of problem spaces is immediately apparent: problem specification can be distributed between the description and the type hierarchy. This happens because, in the substrate, a type can be a member of a description (or a description in and of itself)[3]. A type being used in the description of a problem amounts to a declaration that the designer will be satisfied with solutions that arise from this type. In the designer's terms, this is declaring that past experience will suffice here, for example, a standard bathroom layout might be all that is sought. Present here is the representation of problems through type hierarchy construction and the authoring of descriptions. A description of how the typed feature structure algorithms apply to these structures awaits the description of solution states and feature nodes.

[2]The elements of the type hierarchy are described in Section 2.2.1. A formal definition of these elements is given in Definition 4.

[3]See Section 2.2.2 for a discussion of the relationship between types and descriptions.

4.2.3 The domain layer construct

In the domain layer, problems find reification as *problem state* objects, written as PState. A PState represents problems and relations among subproblems. Thus, the problem state construct specifically adds intentionality concerning problems. As shown in Figure 4.4, the domain layer construct A PState maps to the type hierarchy and to descriptions. From the perspective of an

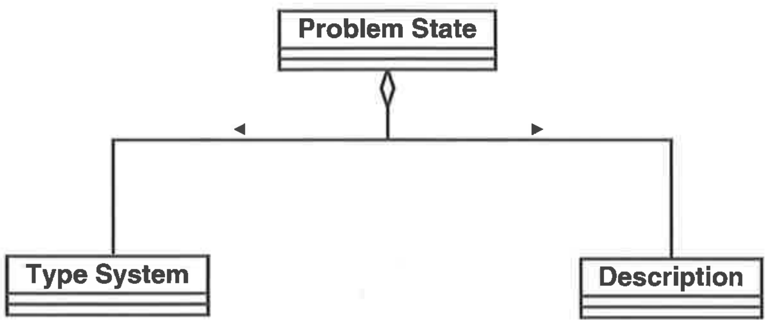


Figure 4.4: The problem state composes a collection of descriptions, Desc over a type system.

exploration process, a design problem can be expressed as a type inheritance hierarchy. Typically these would be such forms that have been ossified by past experience in design space. In this case, the problem would be available as a problem state, specified by its trivial description as a type in the type hierarchy. Following the path notation, the mapping of a problem state to the type system can be explained as follows:

$$\mathsf{PState} \equiv \bullet\mathsf{PState} : \mathsf{InheritanceHierarchy} : \mathsf{Type} \tag{4.1}$$

A problem can also be specified as a description specifying certain forms of spatial relations or constraints on types based on the description language (descriptions are described in Section 2.2.2). The connection between a PState and the formal substrate of descriptions drawn from Desc is given as follows:

$$\mathsf{PState} \equiv \bullet\mathsf{PState} : \mathsf{Desc} : \mathsf{Description} \tag{4.2}$$

In this case, the exploration of the initial description would amount to designer interaction with the the domain layer construct, a PState. Through interaction with a PState, the designer can define design requirements either within the type system or through a collection of descriptions. Acting through a PState, a designer iterates through problem formulation/reformulation cycles by reformulating descriptions, adding new descriptions or monotonic changes to the type system itself. Figure 4.5 extends Figure 4.4 by expanding a problem state to reveal its connections with typed feature structures. It follows that underlying the problem state is the full machinery of typed feature structures. It is within this machinery that problem exploration occurs. Summarising, the problem state is defined as a type or a description over a type hierarchy of types. Thus problems

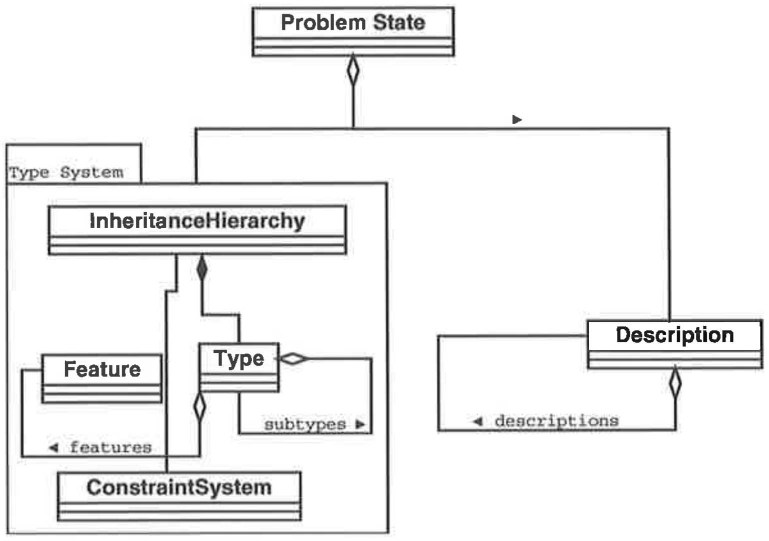


Figure 4.5: Types, Features, constraints and descriptions comprise the representation layer for defining, decomposing and revising problems in the domain layer.

and requirements are constructed over the InheritanceHierarchy. The interaction level construct that encapsulates this problem formulation and reformulation process is the PState. This process is further elaborated in the explanation of *feature nodes* in Section 4.4.

4.3 Solution state

4.3.1 The designer's view

In the domain of design, problems and requirements have multiple solutions. This is analogous to the statement that a requirement may have no solutions, a finite number of solutions or an arbitrarily large collection of solutions. In the exploration model, the solution generation and reuse cycle comprises an iterative process of interaction between the designer and the formalism. As Chien & Flemming [1997] demonstrate, in a designer's solution space, work is done by a combination of generation from problem specifications, navigation of partial solutions and solution revision (removing or modifying the same). Design units, as described in Section 1.1.1, represent a physical model as it is elaborated and records the relation amongst functional requirements and characteristics of a physical structure that satisfy these requirements. To a designer, a solution is a component in the spatial or physical structure of a building and has an identifiable spatial boundary. Thus solutions describe physical and geometric characteristics of structures satisfying the problem description. Further, generated solutions engender a large space of alternatives. These alternatives form a solution hierarchy and as designers revise solutions, a revision history of solutions is recorded. Support for the actions of the designer in making choices about solution alternatives and solution revision is necessary.

4.3.2 The symbol substrate

In the description formalism, the view a "solution" to a problem is the given by the notion of satisfaction. Satisfaction, described in Section 2.2.3, implies the existence of one or more possible resolutions of a problem as typed feature structures. In the formal substrate, a design solution is realised as a typed feature structure, FeatureStructure. Descriptions may be satisfied by no structure, a finite number of structures or an arbitrarily large collection of feature structures.

The steps in the problem/design satisfaction relation are realised as incremental π-resolution states, is termed a *partial satisfier*[4] represented as TFSPartialSatisfier. These component of the substrate represent initial, intermediate (partial) and fully resolved solutions of a given description. Notably, the designer's concept of an alternative partial design is represented in the symbol substrate by a partial satisfier. A TFSPartialSatisfiers composes descriptions and feature structures through the satisfaction relation, Satisfaction, $\models$: Desc $\rightarrow$ FeatureStructure as follows:

$$\mathsf{TFSPartialSatisfier} \equiv \mathsf{Desc} : \mathsf{FeatureStructure} \tag{4.3}$$

The label PSat is used as a shorthand term for representing the relationship between satisfiers and descriptions in the substrate. A PSat composes a collection of TFSPartialSatisfiers

$$\mathsf{PSat} \equiv \mathsf{TFSPartialSatisfier} \tag{4.4}$$

Further, descriptions may themselves be statements of a solution. A feature structure that totally satisfies the given description is termed a *satisfier* of the description and implies a fully resolved feature structure, TFSSatisfier with respect to the InheritanceHierarchy.

$$\mathsf{TFSSatisfier} \equiv \mathsf{Desc} : \mathsf{InheritanceHierarchy} \tag{4.5}$$

The notion of solutions in the formal machinery of design space exploration is given either as a collection of intermediate partial satisfiers with respect to a problem description or a fully resolved feature structure with respect to an inheritance hierarchy of types, InheritanceHierarchy.

4.3.3 The domain layer construct

In the domain layer, the view of a solution is encapsulated in *solution state* objects, written as SState. As an object, a solution state composes both resolved designs and partial or intermediate satisfiers. The constituents of a solution state are shown in Figure 4.6.

To a designer, a solution state provides a view on a developing design. The realisation of the design as a typed feature structure and the satisfaction relation between problems and solutions as an incremental π-resolution state is shown in Figure 4.7. Within it, one can see first, a single

[4]See Section 2.4.1 for a discussion of *partial satisfiers*. .

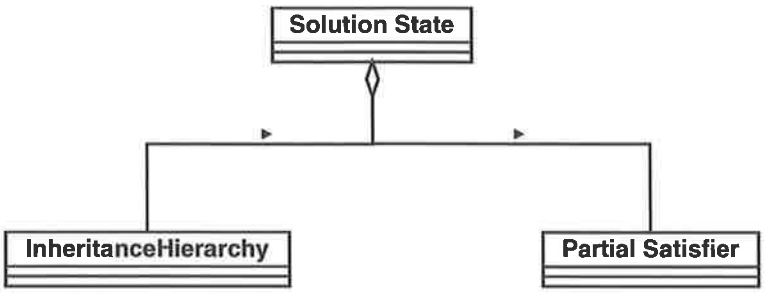


Figure 4.6: Solution states compose partial designs with respect to an inheritance hierarchy of types and a partial satisfier.

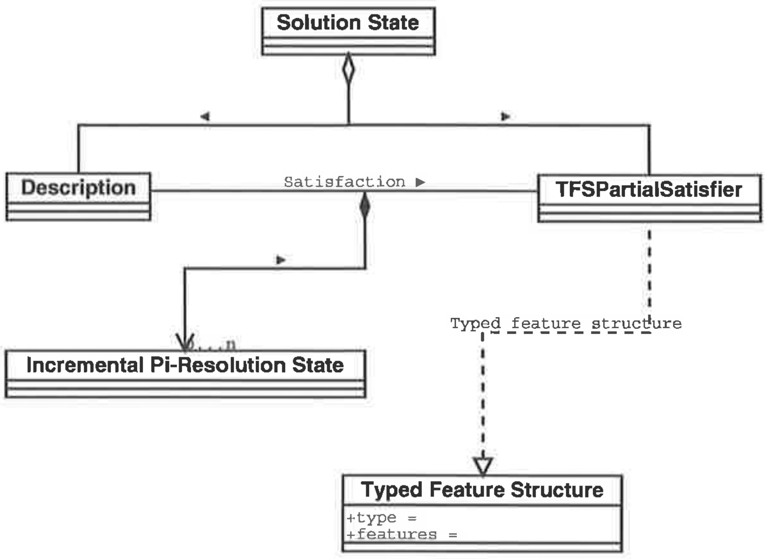


Figure 4.7: Realisation of the design as a typed feature structure and the satisfaction relation between descriptions and solutions as an incremental π-resolution state.

possible response to a problem; and second, a trace of how the problem is satisfied in the form of partial designs through incremental stepwise refinement. In the first instance, the realisation of a fully resolved design as a SState is its conversion from a description to a fully resolved feature structure with respect to an inheritance hierarchy, InheritanceHierarchy, of types. In the second instance, a SState represents a partial solution. The solution state construct specifically adds intentionality concerning solutions to a problem specification.

The connection of the solution state, SState to the formal substrate is given as follows:

$$\mathsf{SState} \equiv \bullet\mathsf{SState} \bullet \mathsf{PState} : \mathsf{Desc} : \mathsf{TFSPartialSatisfier} : \mathsf{FeatureStructure} \tag{4.6}$$

As shown in the path notation above, a solution state models typed feature structures. Ingrained in the notion of a solution state are problem states, the space of their solutions defined by the satisfaction relation and the incremental π-resolution states defined by the formal resolution process.

Summarising, the domain layer construct SState represents the notion of a design solution. In it, are embedded the symbol substrate concepts of description, the satisfaction of a description as satisfiers and the trace of intermediate solutions as partial satisfiers.

4.4 Feature Node

4.4.1 The designer's view

In the domain layer, a problem is represented as a PState. A partial or complete solution to a problem is represented as a SState. These constructs capture the distinction between problems and solutions. However, there exist strong dependencies between the problem formulation process (type specifications, description authoring) and solution generation process of exploration (satisfaction, resolution). The designer's view of exploration encompasses both these processes.

For example, in SEED [Flemming & Woodbury 1995], the relationship between problems and solutions is captured as a design state. The design state comprises a set of function units and the collection of design units that allocate them. The design state captures both the process of generating designs from problem specifications as well as the creation of new problems. Thus, in a design state, designs and problems, are elaborated dynamically. A design state also records the relation amongst functional requirements and characteristics of a physical structure that satisfy these requirements. A domain layer construct to address the relationships arising out of problem formulation/reformulation process and solution generation/reuse process is necessary.

Supporting choice-making is of particular importance in exploration. Through the feature node construct, the actions of the designer in making choices on which threads of exploration to work on can be addressed.

4.4.2 The symbol substrate

In the symbol substrate, the actions of the designer are supported through the notions of exploration non-determinism, incrementality in the resolution process, the explicit recording of resolution steps and access to the design space movement operators. Exploration non-determinism arises when descriptions contain disjunctions in them. Disjuncts in a problem description are handled in the formal substrate through the concept of a *conjunct of disjuncts* [Burrow 1999]. The generation process presents the user with all possible alternative combinations of satisfiers arising out of disjunctive descriptions. The symbol level construct that implements this concept is the DescNode. In it, all descriptions are collected in a node representation. The formal substrate provides a conversion from alternate problem formulations in the form of disjuncts to a *conjunct of disjuncts*, represented as a DescNode.

The resolution of descriptions comprises an incremental process of π-resolution as described in Section 2.4.1. The resolution procedure acts incrementally by generating partially resolved structures. The process of generating partial satisfiers is expressed as a sequence of resolution steps, where each step records the resolution of a type constraint explicitly. The symbol level concept that implements the elements of a sequence of π-resolution states is the SatNode. In it are captured the partial satisfier that represents a partial solution (the SState in the domain layer) as well as the type constraints that remain to be resolved. In addition to the construction of descriptions and the generation of partial satisfiers by π-resolution, several operators are available in the substrate for supporting movement in design space. Section 2.4 covers these design space operations, namely, indexing and reuse, hysterical undo, design unification and design anti-unification. They operate on partial satisfiers in a SatNode to generate new SatNodes. At the user level, it is necessary to enable the designer to access these substrate operators.

4.4.3 The domain layer construct

Problems and solutions are explicitly captured in PState and SState. A feature node, FNode, encapsulates the designer's interaction with the formalism by coupling user actions with the elements of the underlying symbol level. The FNode records what choices a designer might make and how a designer would make such choices, that is, design intention. User choices with respect to problem alternatives, incremental generation and the design space navigation are addressed in the feature node. The feature node, FNode captures the relationship between a problem state, PState and an alternative design that is a partial solution to the problem, SState. The composition of the relationship between a problem state and its partial satisfiers are shown in Figure 4.8. Through

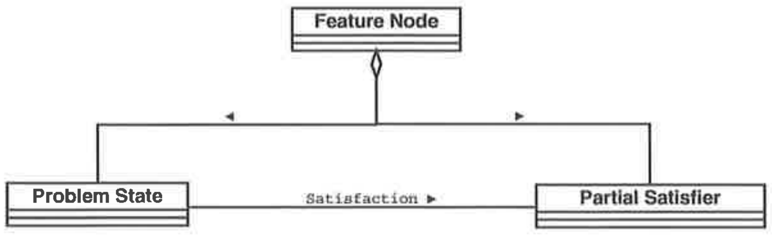


Figure 4.8: The feature node composes the relationship between the problem state and the partial satisfier.

a FNode, the user accesses the typed feature structure machinery and its contained structures: problem states and their partial satisfiers.

User choices and actions on problem formulations are defined through a DescNode. A DescNode composes typed feature structure descriptions. The connection between a FNode and a DescNode in path form is as follows:

$$\mathsf{FNode} \equiv \bullet\mathsf{FNode} \bullet \mathsf{PState} : \mathsf{DescNode} : \mathsf{Desc} \tag{4.7}$$

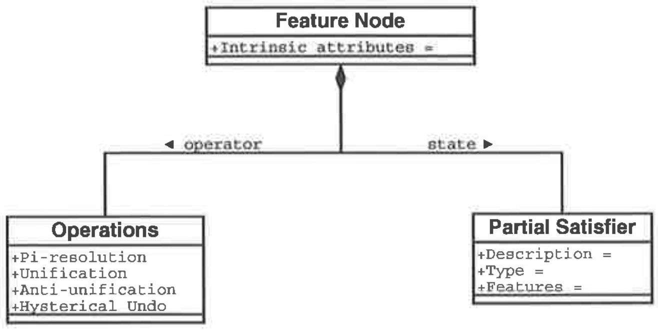


Figure 4.9: Feature nodes compose operators, their arguments and the current resolution state. The design space operators, Operations, are accessible to the user through the intrinsic attributes of a feature node. The state of the current resolution state is represented by a partial satisfier.

Through the FNode, the user and the formalism participate in a mixed-initiative problem formulation and reformulation process. Problem specifications are specified by the user through interaction with a PState. These in turn compose disjunctive and non-disjunctive statements in the description language of typed feature feature structures. Through the FNode, the user and the formalism participate in a process of incremental generation of partial solutions of a problem statement. User choices and actions on partial solutions are defined through a SatNode. A SatNode composes typed feature structures in the underlying formalism. User guidance in the generative process is supported at two levels. First, in the selection of a SatNode from the a collection of possible solutions. Second, in the specification of the next step of resolution. The connection between a FNode and a SatNode is given in the path form as follows:

$$\mathsf{FNode} \equiv \bullet\mathsf{FNode} \bullet \mathsf{SState} : \mathsf{SatNode} : \mathsf{PSat} : \mathsf{FeatureStructure} \tag{4.8}$$

TheFNode captures a mixed-initiative formulation of choice-making between the designer and the formalism. The FNode enables the user to make choices at a particular point in the problem formulation and solution generation process. Further, the choices made by the designer and the results of choice making are recorded as feature nodes.

This mode of interaction is consolidated further by conjoining designer actions and formal moves together as the Operations of exploration. These operations enable the designer and the formalism to share joint responsibility for design space navigation. The movement algorithms of the formalism support feature node navigation. In the FNode, formal moves are cast as *intrinsic attributes* of a feature node. They are intrinsic because they mirror the moves described in Section 2.4. Therefore, Intrinsics are Operations providing direct access to the underlying the arguments and operators of the design space exploration machinery.

An example of this interaction is shown in Figure 4.9. The designer can access a partial satisfier and apply an operator from Operations to extend the partial satisfier, PSat.

With reference to a Fnode and the formal operators in the substrate, Operations can be written as equivalent to the following path,

$$\mathsf{Operations} \equiv \bullet\mathsf{FNode} \bullet \mathsf{Intrinsics} : \mathsf{PSat} : \mathsf{FeatureStructure} \tag{4.9}$$

Through this formulation, FNode • Intrinsics capture design moves that mirror the operators, arguments and states already available in the formal substrate. This connection to the underlying exploration machinery makes no claims about supporting the contingent intentionality of designer actions. To be truly mixed-initiative, feature nodes need to support operations that enable the designer to manipulate the entities of a feature node.

Supporting contingent user interaction, actions with no analogue in the substrate of the design space exploration, are crucial for mixed-initiative exploration. The interactive manipulation of a feature node by the designer require an additional set of operations. These operations are cast as *extrinsic* attributes of a FNode. Extrinsics capture the class of Operations that permit the user flexible and extensible interaction with the elements of a FNode. An example of an extrinsic operation is the ability to navigate the contents of a feature node by direct manipulation. A detailed elaboration of extrinsics awaits discussion in Chapter 5. With reference to a FNode and the interaction operators external to the formal substrate, Extrinsics can be written as equivalent to the following path,

$$\mathsf{Operations} \equiv \bullet\mathsf{FNode} \bullet \mathsf{Extrinsics} : \mathsf{FNode} \tag{4.10}$$

In this manner, FNode • Extrinsics are defined recursively over feature nodes. The mapping of extrinsic attributes of a feature node is explained in Figure 4.4.3.

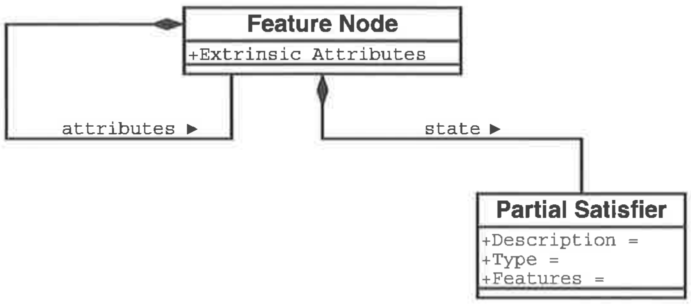


Figure 4.10: The extrinsic attributes of the feature node, FNode, represent the behavioural aspects of the feature node contingent upon user interaction but with no analogue in the formal substrate.

Summarising, FNode • Intrinsics enable the user to access formal moves available in the symbol substrate. FNode•Extrinsics provide a hook to account for the contingent aspects of user interaction.

The representation of a FNode makes it possible to access initial requirements, PState, intermediate and final solutions, SState and the operators of exploration, Operations. Operators are further classed into Intrinsics and Extrinsics, both being attributes of a FNode. Given a feature node, Intrinsics allow the designer to access the formal (intrinsic) operators in the formal substrate. Extrinsics permit the manipulation of feature nodes with no analogue in the formal substrate. These are cast as behavioural (extrinsic) attributes of a FNode. The node-attribute formulation enables the interleaving of system-driven and user-driven moves for mixed-initiative interaction.

Summarising, the FNode construct proposes a principled formulation for supporting the process of exploring problems, solutions in conjunction. Throughout this domain layer construct, the operations and states of exploration are brought under a common conceptual frame.

4.5 Satisfier Space

4.5.1 The designer's view

A model of the designer's view of design space completes the domain layer for supporting mixed-initiative exploration. The clearest exposition of the designer's actions in design space is described in Chien & Flemming's [1996] model of navigation. This model is discussed in Section 1.2.1. They construct a notion of navigation based on nodes and edges where nodes represent design states and edges map their relationships in a design space. User navigation of design spaces is through the traversal of paths and landmarks defined over the navigation structure. This structure enables the designer to orient and maintain context during exploration, make choices and visually browse the history of exploration (alternatives, revisions) recorded in design space [Chien & Flemming 1997].

Burrow & Woodbury [2001] treat the history of exploration as the primary device for teleological explanations of designs. The symbol substrate provides the relation of subsumption amongst designs and this relation is available to a designer through the concept of a *satisfier space*. This construct provides a unified model for representing the set of problems, subproblems, problem revisions and associated designs that a designer actually considers. Problems need not be fixed. Designs can be partial or complete with respect to the initial problem formulation. A designer may make varied choices that imply different kinds of design space operations. All are captured in the satisfier space. The satisfier space floats above design space structure to tell the story of what a designer actually did in design space. From the designer's perspective such a model must capture at least the history of design exploration. The history of choices made and intentions expressed by the designer during exploration are captured by the satisfier space.

A *Satisfier Space* composes a set of ancestor and progeny nodes, Fnode, recording the history of exploration, as uncovered by the designer's actions. Symbolically, the satisfier space is simply a tree of visited design possibilities. Each node in the satisfier space is a feature node which connects to the underlying design space machine.

4.5.2 The symbol substrate

The exploration formalism provides a structuring relation based on subsumption to order collections of exploration states. Looking down to the typed feature structure machinery, the satisfier space connects, via feature nodes, to points in the underlying subsumption-ordered design space. The ordering of exploration structure in the symbol substrate is described in Section 2.3. In the substrate, subsumption defines a partial ordering over exploration states (feature structures) and this ordering of feature structures is represented as a hierarchical graph.

The design space representation records change in formal terms aimed at efficient design creation, indexing and retrieval. In it, the subsumption relation provides a global, principled way for keeping track of additions, deletions and other forms of change as the exploration progresses.

4.5.3 The domain layer construct

In the domain layer, the subsumption-ordered design space is explored through interaction with feature nodes. Choice and history are recorded as a collection of feature nodes. The recording of this interaction process is captured in the domain layer construct, SatSpace. Thus, in contrast to the design space below, the satisfier space is not ordered by subsumption, but by user choice and intentional history. While subsumption accounts for information specificity, the satisfier space, as a collection of feature nodes developed through exploration moves, accounts for choices and the history of exploration.

The relationship between a satisfier space and its constituent feature nodes is shown in Figure 4.11. A FNode is an element of the SatSpace. The label SatSpaceEl can be used interchangeably

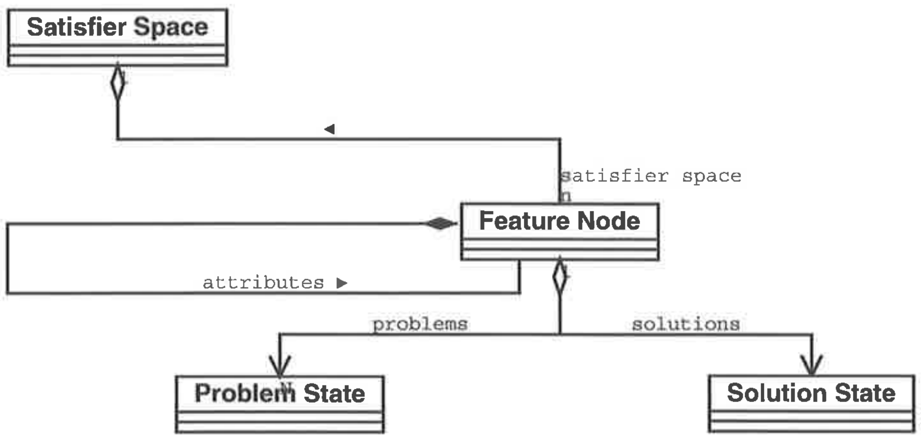


Figure 4.11: The collection of feature nodes, developed through exploration moves, represent the satisfier space. User choices and exploration history are recorded in the satisfier space.

to represent feature nodes in SatSpace. Each SatSpaceEl captures the mapping between problem state, PState, their solutions as SState objects and the record of their connection to the underlying design space as uncovered by user choice. The formulation of a satisfier space, SatSpace as a domain layer construct provides two key benefits. First, the designer can defer formal movement operations to the design space below. This deferral preserves the key invariant structure in design space, subsumption. For example, the incremental π-resolution preserves information monotonicity.

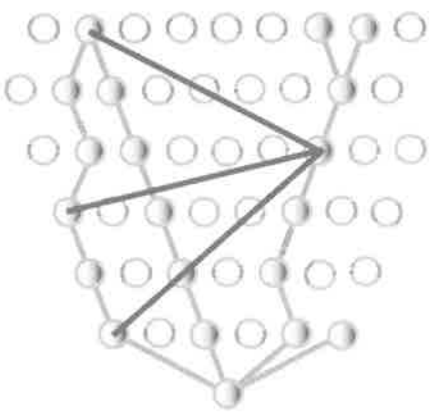


Figure 4.12: Independence of satisfier space and design space. The design space represents the space of all possible exploration states, structured by subsumption. The satisfier space represents the set of exploration states traced by exploration.

Second, the satisfier space, SatSpace is independent of the design space. Design moves have multiple implications in the underlying design space. For example, the application of a π-resolution operation may affect more than one state in the design space. In the satisfier space, the user only sees the branching (through choice and history) course of intentional exploration. The application of a π-resolution operation results in the creation of a new feature node extending the current thread of exploration. The independence of satisfier space and design space structure is informally shown in Figure 4.12. The satisfier space captures intentional moves at the user level floating above the subsumption-ordered design space.

Summarising, the domain layer construct SatSpace, provides a principled representation of user actions in exploration. It allows access to the structure of the subsumption relation in design space for exploration operations. At the same time, it allows the intentional moves by the user, in the form of choices and history, to be recorded in a principled manner.

4.6 Summary

This chapter develops the domain layer of the mixed-initiative interaction model for design exploration. This layer constructs a designer's view of exploration comprising problems, solutions, choices and history over the symbol level representation of design space exploration. The domain layer constructs are *Problem state*, *Solution state*, *Feature node* and *Satisfier space.* In each of these constructs, the case for mixed-initiative is made through a three level exposition, the designer's view, the symbol system view and the mapping of the two through the domain layer constructs, PState, SState, FNode and SatSpace. *Problem states* represent design problems. *Solution states* represent partial design solutions. *Feature Nodes* compose problem formulation and solution generation processes and support the exploration operations. *Satisfier space* records designer choices and encapsulate the history of exploration. Chapter 5 goes on to describe the components of the dialogue layer of the mixed-initiative interaction model for design space exploration.

Chapter 5

Mixed-initiative Dialogue

This chapter develops the dialogue layer of the mixed-initiative interaction model for design space exploration. Following the requirements identified in Section 3.4, the dialogue layer provides a communication and control interface for conversational dialogue between the user and the formalism.

5.1 The dialogue layer

The requirements of a dialogue layer based on Grice's [1989] axioms of rational conversation are proposed in Section 3.4. Control and communication between the user and the generative formalism are addressed from the standpoint of dialogue representation and dialogue integration as follows,

- **Dialogue representation.**
  A common symbolic representation is proposed for supporting the modalities of input and output during mixed-initiative exploration. Exploration dialogue between the user and the formalism is represented through a visual notation based on the domain layer entity, FNode identified in Section 4.4. The visual notation expresses the input and output modalities from both the user and the formalism. The visual representation of feature nodes is developed in Section 5.2.

- **Dialogue integration.**
  A model of turn-taking is proposed for integrating the different modalities of action available to both the user and the formalism during exploration. Dialogue integration is expressed through the specification of interaction behaviour for turn-taking. Through interaction with the intrinsic and extrinsic attributes of a FNode, the user and the formalism are able to acquire, relinquish, shift and allocate control of the exploration process. The integration of dialogue through mixed-initiative is developed in Section 5.3.

The domain layer concepts (see Section 4.1), provide access to the machinery of typed feature structures and, *ipso facto*, represent the results of exploration and generated partial designs. The

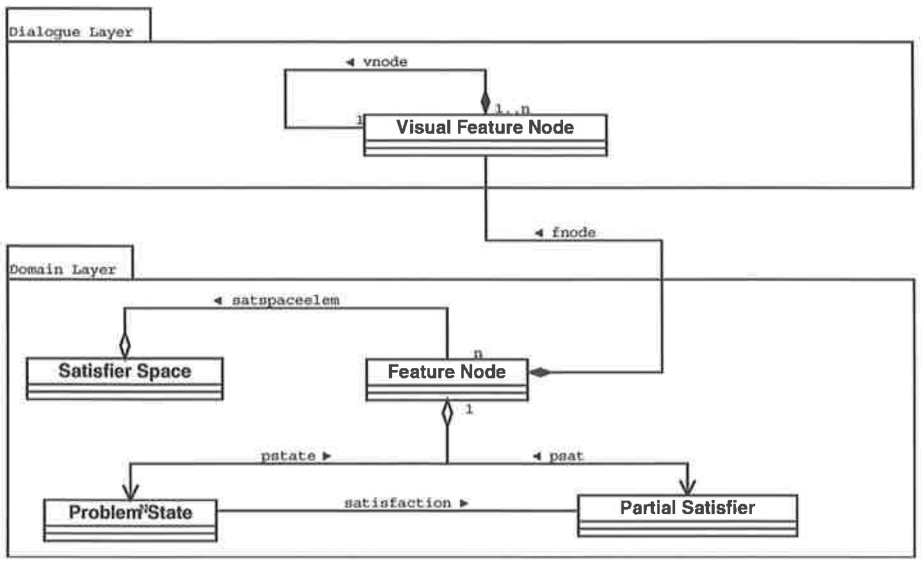


Figure 5.1: Mapping domain layer constructs to the dialogue layer through the visual feature node.

dialogue layer extends the representation of domain layer concepts built upon typed feature structures to represent dialogue. Typed feature structures provide clearly defined common semantics for representing dialogue for *both* the designer's model of exploration and the formal substrate. The domain layer builds a set of useful concepts on top of the formal substrate. Thus, the representation of dialogue can be formulated in terms of the domain layer concepts. The components of the domain layer, problems, solutions, choices and history are recorded in the FNode construct. The FNode construct (see Section 4.4) encapsulates a principled formulation of the designer's view onto the description formalism.

The domain layer constructs are made transparently visible to the user by introducing the concept of a visual feature node, VNode, in the dialogue layer. A visual feature node represents dialogue between the user and the formalism. Through the construct of the visual feature node, a principled formulation of mixed-initiative conversational structure is established between user and formalism. This formulation supports a number of key properties, identified abstractly in Section 3.4 and are based on Grice's model of rational conversation. The relationship between the domain layer constructs and the visual feature node is shown in Figure 5.1.

The concept of a visual feature node enables a principled formulation of dialogue representation. As shown in Figure 5.1, the visual feature node, VNode provides a common frame for representing mixed-initiative dialogue in the interaction model. Two distinct views are mapped onto the same representation. First, the results of exploration initiated by the description formalism are available as partial satisfiers (incorporating types, features, descriptions) of a VNode. Second, the representation of the results of user manipulation are available as feature nodes (incorporating problem states, choices and functions, interaction history) through the extrinsic attributes of the VNode.

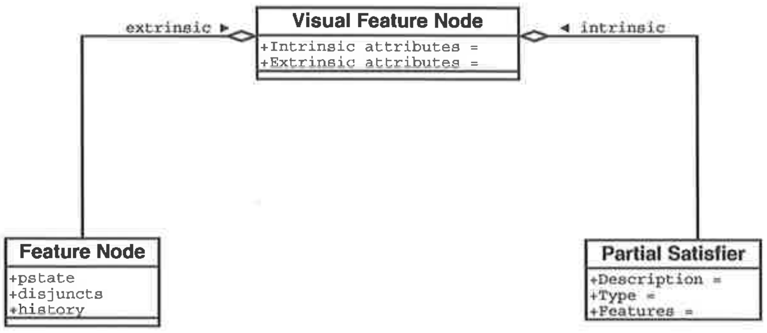


Figure 5.2: The elements of the visual feature node map onto the domain layer constructs. The intrinsic attributes represent the formal features along which the exploration may proceed. The extrinsic attributes represent designer moves.

The representation of dialogue based on rationality, cooperation, common purpose and direction through the visual feature node is described in Section 5.2. The conjunction of the visual representation and the model of turn-taking are described in Section 5.3.

5.2 Representation of dialogue

Formally, all the elements of a typed feature structure are simply features. Operationally, these feature collections are interpreted as part-whole and property hierarchies – both common means of representation. The domain layer concepts represent intentional actions (choice and history), problems and their partial solutions (partial satisfiers, alternatives) during exploration. In this section, a visual representation of feature structures is used in representing exploration dialogue.

The representation of feature structures is in three forms, text (in the form of descriptions), graph notation [Carpenter 1992, p 37] and the attribute-value matrix notation [Pollard & Sag 1987], hereafter AVM. Each of these forms map feature structures into different parts of the description formalism.

For example, the type system is directly expressed in structured textual form, as are constraints and problem formulations. Textual descriptions in a description language, through the satisfaction and describability theorems, make feature structures interchangeable with descriptions. Given this equivalence, descriptions and feature structures can both be written in textual notation, following the logical description language, *Desc*, described in Appendix A. Textual descriptions are useful for persistence, storage and coding of problem statements.

Feature structures, are rooted, labelled graphs. Their automata-like and graph-like character can be cast into a graph representation comprising nodes and edges. In this view, nodes and edges of a graph are taken to represent the *type* and *attributes* of a feature node. In a typed feature structure, Q, the set of nodes represent domain objects and edges represent functional connections.

Similarly, a feature node can be conceptualised as a form of directed graph as depicted in Figure 5.3. For example, node 4 represents a general property class. Two examples of features are shown, where the attributes COLOUR, RVALUE of node *property* point to nodes 5 and 6. The nodes are enclosed in

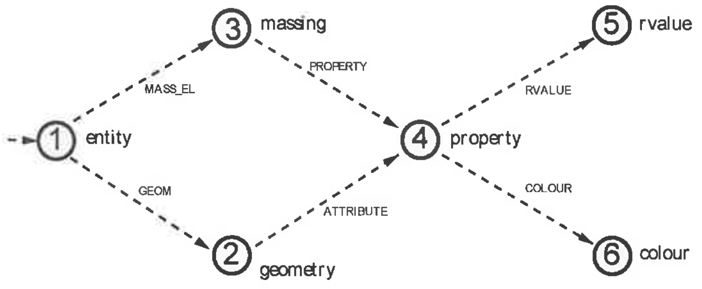


Figure 5.3: Feature node in directed acyclic graph DAG notation. The numbered nodes represent structures annotated by their type labels. The edges represent features annotated by their feature attributes. The values of attributes are other nodes.

circles, with the arrow pointing at the root, types appear in boldface next to their nodes. Features, in small caps, label the connective arcs between nodes. The graph representation is useful for depicting small dialogue fragments and their relations. When the typed feature structures become very large, it is difficult to understand the nodes and track relationships in this notation. Further, this approach does not capture the semantics of a feature node and the behaviour of individual elements of the notation.

The analogy between feature nodes and frames provides a notation for visualising large collections of feature structures. Each node is a frame, the features on arcs represent slot labels, and the arcs themselves point to the slot fillers. This frame-based notation is the standard notation for visualising feature structures used in the description of linguistic fragments and discourse representation [Pollard & Sag 1987, Carpenter 1992]. The notation is called *attribute-value matrix* or AVM notation. The AVM notation provides a direct method for representing feature nodes and their formal and behavioural attributes. Figure 5.4 illustrates the components of the AVM notation. It depicts the same feature node shown in Figure 5.3. Each node is represented with the frame delimiters "[" and "]". The frame is annotated with the *type* of the node. Thus far, feature structures have described the results of generation. This visual representation of feature structures paves the way for introducing them as the mode of user manipulation. The potential of representing the visual feature node using AVM notation is described in the next section.

$${}_{\textit{entity}}\left[\begin{array}{ll} \text{MASS\_EL}: & {}_{\textit{massing}}\left[\begin{array}{ll}\text{PROPERTIES}: & {}_{\textit{property}}\left[\begin{array}{ll}\text{RVALUE}: & \text{rvalue}\\ \text{COLOUR}: & \text{colour}\end{array}\right]\end{array}\right] \\ \text{GEOM}: & {}_{\textit{geometry}}\left[\begin{array}{ll}\text{ATTRIBUTES}: & {}_{\textit{property}}\left[\begin{array}{ll}\text{RVALUE}: & \text{rvalue}\\ \text{COLOUR}: & \text{colour}\end{array}\right]\end{array}\right]\end{array}\right]$$

Figure 5.4: Feature structure in AVM notation. The feature stucture of type, *entity* with features, MASS_EL and GEOM. These features have two substructures of type *massing* and type *geometry*.

5.2.1 A visual notation

The AVM notation visually describes feature structures. A *visual feature node* maps the intrinsic and extrinsic attributes of a feature node, FNode onto elements of the AVM notation. This mapping annotates the visual feature node with the type, feature names, feature values and coreferences taken from the underlying partial satisfier. The connection between a visual feature node, VNode and a FNode is given as follows:

$$\mathsf{VNode} \equiv \mathsf{VNode} \bullet \mathsf{FNode} \tag{5.1}$$

The VNode composes and aggregates elements of the underlying representation, shown in 5.1. In a VNode, the values of a feature may be atomic, complex or another feature node. The value of a visual feature node VNode, is given either by a *feature value pair* or *feature-value map*. The smallest element of the visual feature node is the *feature value pair* or *feature-value pair*. The *feature-value pair* represents the relation between a feature and its value. This is shown in Figure 5.5.

The *feature-value map* specifies the relation between a feature node and its sub nodes. For example, in Figure 5.5, the *feature-value map* represents the functional relationship between the GEOM and its value. A *feature-value map* is enclosed by the delimiters "[" and "]" and annotated by the type label drawn from its partial satisfier. In the example, the partial satisfier is of type, *geometry*. Feature nodes support recursive containment. Thus, the value of a feature node may be another feature node. The attribute-value notation is easily adapted for a visual representation of a *feature-value map* as follows: the *feature-value map* can be conceptualised as a recursive container of entities of type *feature-value pair*.

In Figure 5.5, the *feature-value pair*, represents the functional relation between the feature, MASS_EL and its a value, which is minimally the type *massing*. The value of MASS_EL may also be complex, such as a query description, resolution step or function application or external complex datatype. The value of MASS_EL may also be another feature structure.

In a visual feature node, VNode, two or more paths can share the same information. This is called *structure sharing*. Paths engaging in structure sharing are called reentrant. Shared structure in a visual feature node is represented by *co-references*, also called *tags*. The co-reference $\boxed{n}$

$$\text{MASS\_EL} : \text{massing}$$

(a)

$${}_{entity}\left[\text{GEOM} : {}_{geometry}\left[\text{ATTRIBUTES} : {}_{property}\left[\boxed{1}\right]\right]\right]$$

(b)

Figure 5.5: A feature value pair (a) and a feature value map is shown in (b). The pair ATTRIBUTES : *property*, is indicated by a co-reference tag, indexed by the number 1 to a node outside of the diagram fragment shown.

denotes an index value where n is the identity of the node that is shared between one or more feature structures. Co-references and their denotation by indices is straightforward in the AVM notation. As shown in Figure 5.6, reentrancy, or structure sharing is indicated by reference tags such as $\boxed{4}$. The slots are the features and the values are written next to them.

$${}_{entity}\left[\begin{array}{ll} \text{MASS\_EL} : & {}_{massing}\left[\text{PROPERTIES} : \boxed{1}\; {}_{property}\left[\begin{array}{ll}\text{RVALUE} : & \text{rvalue}\\ \text{COLOUR} : & \text{colour}\end{array}\right]\right] \\ \text{GEOM} : & {}_{geometry}\left[\text{ATTRIBUTES} : {}_{property}\left[\boxed{1}\right]\right]\end{array}\right]$$

Figure 5.6: A visual feature node incorporating coreference notation. The shared feature structure of type *property*, is indicated by the coreference tag, indexed by the number 1.

5.2.2 Choices

The notation supports user choices in the VNode through the representation of alternatives, resolution steps and function applications.

The notation incorporates the operators of the description language, *conjunctions* and *disjunctions.* Conjuncts and disjuncts in feature structures are denoted using the same notation as feature value pairs. In place of the feature labels, the labels *conjunct* and *disjunct* are used, with the values as feature structures. This common representation can be scaled to represent the conjunction of disjuncts and the disjunction of conjuncts. In linguistic attribute-value formalisms [Pollard & Sag 1987, Pollard & Moshier 1990], conjuncts and disjuncts are denoted by special delimiters, such as "{" and "}" and their edge names are either omitted, suppressed or obscured. This is not necessary in this interactive representation.

An example of a conjunct of disjunctive visual feature nodes is shown in Figure 5.7. User

$$
{}_{entity}\begin{bmatrix}
\text{MASS\_EL} : & {}_{conjunct}\begin{bmatrix} \text{CONJUNCT} : & {}_{disjunct}\begin{bmatrix} \text{DISJUNCT\_1} : {}_{geometry}\begin{bmatrix}\text{geom1}\end{bmatrix} \\ \text{DISJUNCT\_2} : {}_{geometry}\begin{bmatrix}\text{geom2}\end{bmatrix} \\ \text{DISJUNCT\_3} : {}_{geometry}\begin{bmatrix}\text{geom3}\end{bmatrix} \\ \text{DISJUNCT\_4} : {}_{geometry}\begin{bmatrix}\text{geom4}\end{bmatrix} \end{bmatrix} \end{bmatrix} \\
\text{GEOM} : & {}_{conjunct}\begin{bmatrix} \text{CONJUNCT} : & {}_{disjunct}\begin{bmatrix} \text{DISJUNCT\_5} : {}_{geometry}\begin{bmatrix}\text{geom5}\end{bmatrix} \\ \text{DISJUNCT\_6} : {}_{geometry}\begin{bmatrix}\text{geom6}\end{bmatrix} \\ \text{DISJUNCT\_7} : {}_{geometry}\begin{bmatrix}\text{geom7}\end{bmatrix} \\ \text{DISJUNCT\_8} : {}_{geometry}\begin{bmatrix}\text{geom8}\end{bmatrix} \end{bmatrix} \end{bmatrix}
\end{bmatrix}
$$

Figure 5.7: An example of a conjunct of disjunctive visual feature nodes. The disjunctive nodes are accessed by a node of type *conjunct* and represented as a *feature-value map* and each disjunct is represented as a *feature-value pair*, whose features are defined by DISJUNCT_*n* where *n* is an index over disjuncts.

interaction is necessary to resolve the structures associated with the features MASS_EL and GEOM of the feature structure of type, *entity*. The disjunctive nodes of type *disjunct* are represented as a *feature-value map*. Each *feature-value map* is accessed by a feature CONJUNCT, of type *conjunct*. Each *feature-value map* has four possible disjuncts, which are represented as a *feature-value pair*, whose features are defined by DISJUNCT_*n* where *n* is an index over disjuncts.

$$
{}_{entity}\begin{bmatrix}
\text{MASS\_EL} : {}_{geometry}\begin{bmatrix}\text{geom2}\end{bmatrix} \\
\text{GEOM} : {}_{geometry}\begin{bmatrix}\text{geom7}\end{bmatrix}
\end{bmatrix}
$$

Figure 5.8: The resultant visual feature node arising out of the resolution of disjunctive nodes by user interaction.

The user can choose a single conjunct for each of features of the node *entity* shown in Figure 5.7. For example, if the designer selects the disjuncts, DISJUNCT_2 and DISJUNCT_7 shown in Figure 5.8, as the appropriate values of MASS_EL and GEOM respectively, the node that results will carry the structure shown in Figure 5.8. Through the dialogue layer construct, VNode, it is possible to expose the internal representation of descriptions (problem states), partial satisfiers (solutions) and alternatives (choices) for user interaction. Since the VNode is recursively defined, a collection of choices and user interaction history is expressed as a collection of visual feature nodes. Thus far, the notation has shown how the intrinsic attributes of a feature node FNode can be visually represented for direct manipulation by the designer. It is also possible to represent the attributes of a FNode that are extrinsic to the formalism, using the same notation.

Functions

This representation of functions, commands and their arguments extends the visual feature structure notation for user interaction. The behaviour of functions and commands during interaction, namely function application, function unification and function unfolding[1] can be added to the interaction.

$$append(X,Y) \Longleftrightarrow {}_{append}\begin{bmatrix} \text{ARG1} : {}_{arg}\begin{bmatrix} \text{X} \end{bmatrix} \\ \text{ARG2} : {}_{arg}\begin{bmatrix} \text{Y} \end{bmatrix} \end{bmatrix}$$

Figure 5.9: Encoding a function as a *feature-value map*. The function *append(X, Y)* which concatenates values can be represented as the *feature-value map* of type, *append* and the two features ARG1 and ARG2. The features, ARG1 and ARG2 encode the values X and Y as two feature value pairs.

For example, a function can be represented in the visual feature node. Functions can be encoded within the *feature-value map* representation such that the functor annotates the *feature-value map* and the arguments are features. An example of the duality of a function and its arguments with a feature node representation is shown in Figure 5.9. This representation allows the feature structure to encode traditional command languages found in geometry-based design systems. The specification of a command or function then returns a value, which can be atomic, complex or a feature structure. The use of feature structures to encode functions can also be used to pass commands[2].

The expressiveness of feature structure command representations needs to address the possibility of cyclic feature structures and structure sharing. A cycle arises when following a non-empty sequence of features out of a node leads back to that node, which is a useful property in the finite modelling of knowledge [Carpenter 1992, p 51-p 52]. A recognition mechanism is necessary to interrupt infinite loops in a visual feature node representation for commands. The restriction on commands is that structure sharing is not considered a valid part of the command syntax. If co-references do occur, the structures they represent are copied uniquely within each command.

Another way of visualising functions within feature structures is to encode the functional definition as the value of a *feature-value pair* . In this scheme, for a function *append(X, Y)* with two

[1]the term unfolding is defined in Section 3.1.1. Here, it is used in a restrictive sense of exploring the arguments of a function as described above.

[2]Programming languages like LIFE [Ait-Kaci & Cosmo 1993] use types for commands

arguments,there exists a type *function*, such that its value is a function definition with the syntax, *append(X, Y)*.

$${}_{design\_unit}\begin{bmatrix} \text{GEOM: geom1} \\ \text{TRANSFORM: } \langle translate(a,b,c) \rangle \end{bmatrix}$$

Figure 5.10: The function *translate(a,b,c)* is represented as a *feature-value pair* and contained within a visual feature node, with feature TRANSFORM and value, *translate(a,b,c)*.

An example of a procedural function in visual form is shown in Figure 5.10. User interaction on this node involves three possible behaviours. First, the application of the function to an appropriate node results in a new node, consistent with the application. Second, the unification of a functional node with an appropriate feature, results in a new feature structure, following the laws of unification for typed feature structures described in Section 2.2.3. Finally, the function can be unfolded into its constituent subparts following the interaction defined above and its values subject to exploration. An example of the latter is shown in Figure 5.11. The unfolding of a functional representation shows that the feature node representation of the function, *translate(a,b,c)* is of type *translate* and the arguments are the three feature value pairs, TX, TY and TZ.

$${}_{design\_unit}\begin{bmatrix} \text{GEOM: geom1} & \\ \text{TRANSFORM:} & {}_{translate}\begin{bmatrix} \text{TX :} & {}_{arg}\begin{bmatrix} \text{a} \end{bmatrix} \\ \text{TY :} & {}_{arg}\begin{bmatrix} \text{b} \end{bmatrix} \\ \text{TZ :} & {}_{arg}\begin{bmatrix} \text{c} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

Figure 5.11: An unfolding of a functional representation shows the feature structure notation of the function, *translate(a,b,c)*. The type of the function is *translate*. The arguments are unfolded into the three feature value pairs, TX, TY and TZ.

5.2.3 Interaction with visual feature nodes

The visual feature node representation is extended to incorporate behaviours that admit user level actions extrinsic to the formalism. Interaction with a large collection of visual feature nodes requires functionality for panning, zooming in and out of context, search and the expansion of tags. Visual feature nodes are nested entities. User navigation of a large collection of feature nodes is enhanced by functionality for zooming in and out of nodes, imploding nested nodes and the expansion of coreference tags. If the nesting is very deep or broad, panning functionality provides the ability to scroll through the whole feature node. Zoom and implode interaction behaviour provide functionality for controlling depth nesting. By using this functionality the user can fold (implode) and unfold (zoom) feature nodes. The user obtains information about substructures of a node by zooming into them. Further, zooming into the selected substructure enables a reorientation of context such that the selected node becomes the new *root node* of exploration. An example of unfolding substructures by user interaction is given in Figure 5.12.

Zooming and imploding nodes

$${}_{\textit{entity}}\left[\text{GEOM1} : {}_{\textit{geometry}}\left[\text{GEOM2} : {}_{\textit{geometry}}\left[\text{GEOM3} : {}_{\textit{geometry}}\left[\text{GEOM4} : {}_{\textit{geometry}}\left[\boxed{+}\right]\right]\right]\right]\right]$$

Figure 5.12: An example of an imploded feature node hiding the contained substructures. The symbol "+" indicates that the substructures of the *feature-value map* of type *entity* are closed. User interaction on this node is necessary to open these hidden structures.

Feature nodes can be nested recursively to arbitrary levels containing many substructures. The interactive mechanism accounts for folding the nested substructures of a *feature-value map* to hide their underlying notation and for unfolding the imploded structure to see the details of a feature value map. In the visual representation, this is realised by allowing users to open or close substructures visually through the the symbol $\boxed{+}$.

$${}_{\textit{entity}}\left[\begin{array}{l}\text{MASS\_EL} : {}_{\textit{massing}}\left[\text{PROPERTIES} : \boxed{1}\ {}_{\textit{property}}\left[\boxed{+}\right]\right] \\ \text{GEOM} : {}_{\textit{geometry}}\left[\boxed{+}\right]\end{array}\right]$$

Figure 5.13: Another example of an imploded feature node. The symbol "+" indicates that the *feature-value map* of type *property* and *geometry* contains nested subnodes that can be unfolded by user interaction.

The unfolding symbol, $\boxed{+}$ shows up in two different situations. A restriction may be placed on the depth of display of a *feature-value map*. Any substructure in a *feature-value map* that exceeds that depth, is represented by the symbol, $\boxed{+}$. This is automatically managed by the dialogue layer and the nesting levels set through preferences. The user can also manipulate the *feature-value map* interactively. The *feature-value map* will be shown as folded, until it is explicitly unfolded. Thus, the $\boxed{+}$ symbols on the *feature-value map* coming from depth restriction are generated and removed dynamically while the user navigates a feature structure. In contrast, the maps that are unfolded manually need explicit interaction to change their display. This enables the user to control the level of detail shown, while zooming and imploding very large feature node collections.

Pan functionality is provided by enclosing the feature structures in scroll bars. This is a standard means for providing canvas real estate. User interaction with the scroll bars allows context to be shifted horizontally and vertically.

Interaction with co-reference tags

Path equality in structures is one of the oldest information structuring concepts in computational design[3]. This notion is captured through structure sharing between feature structures at the formal substrate.

$$
{}_{\textit{entity}}\left[\begin{array}{ll}
\text{MASS\_EL}: & {}_{\textit{massing}}\left[\text{PROPERTIES}: \boxed{1}\; {}_{\textit{property}}\left[\begin{array}{ll}\text{RVALUE}: & \text{rvalue}\\ \text{COLOUR}: & \text{colour}\end{array}\right]\right]\\
\text{GEOM}: & {}_{\textit{geometry}}\left[\text{ATTRIBUTES}: {}_{\textit{property}}\left[\boxed{1}\right]\right]\\
\text{FUNCTION\_UNIT}: & \text{fu}\\
\text{DESIGN\_UNIT}: & \text{du}
\end{array}\right]
$$

Figure 5.14: An example of substitution of a *feature-value map* with a co-reference tag, $\boxed{1}$. The co-reference tag is an index to the nested partial satisfier of type *property* that is shared by the features, PROPERTIES and ATTRIBUTES.

From the perspective of the designer, path equality in the underlying formalism is visually displayed through co-reference tags. Thus, the appearance of co-reference tags in the visual representation indicate nodes that are strictly structure shared partial satisfiers. In the context of exploration, these correspond to entities that are feature values that by definition, are feature structures.

Co-reference tags are used in two ways, both shown in Figure 5.14. Firstly, a co-reference is used to annotate a *feature-value pair* that structure-shares a *feature-value map*. Secondly, it is used to simplify the visual representation of partial satisfiers, by simple substitution of the shared *feature-value map* by the co-reference tag, denoted by $\boxed{n}$. The co-reference tag can be substituted by the partial satisfier it denotes by user interaction. If the partial satisfier is represented, the co-reference appears outside the *feature-value map* , as shown in the value of PROPERTIES. If the co-reference is used to refer to the partial satisfier, it appears inside the *feature-value map* as shown in the value of ATTRIBUTES.

The user can move within a collection of nodes using the find operation. Search for a nested node using the find operation is supported by the use of tags. A successful search results in the matching *feature-value pair* being panned into focus. The user can specify *find* function to search only for features, types, co-references or atomic values. In the case of co-reference search, the tag number is used to locate the partial satisfier corresponding to the specified tag. This is useful in dialogue situations when structure shared partial satisfiers are widely separated. Tags

[3]The first instance was Sutherland [1963].

mark structure shared nodes and these can be expanded *in-situ* following the general convention of appearing in the first location in which they are introduced. When tag expansion occurs during dialogue, the co-references are updated and the partial satisfiers that they represent are redisplayed. These extensions are discussed in the next sections.

Having defined the visual feature node, its representation and its support for interaction, it is now possible to address how this construct supports mixed-initiative in the dialogue layer. In the next section, the integration of exploration dialogue between the user and formalism through the visual feature node construct is described.

5.3 Integration of Dialogue

The second requirement, dialogue integration, combining two modalities into a common frame, is described in this section. Modes of input and output, for example, typed commands, generated structures or the direct manipulation of graphical symbols need integration during dialogue. This section describes how the visual notation for representing dialogue can support the integration of these modalities.

5.3.1 Supporting partiality

As shown in Section 5.2.1, a visual feature node, VNode inherits key behavioural properties of typed feature structures. Thus, given the well-defined semantics of types, features and descriptions, the visual feature node supports partiality of input and output. The representation framework of visual feature nodes, type annotated frame delimiters, *feature-value pair/ feature-value map* and co-reference tags carry the specification of partial information. Partial information during dialogue provides the opportunity for both user and formalism to underspecify, relying on the extension of dialogue through turn-taking. A partially specified exploration move is represented as an underspecified visual feature node. In this situation, a subset of feature-value pairs is not instantiated, following the intensional nature of feature structure representation. Instead, they are assigned a certain *type*, corresponding to the semantics of the move. This bears explanation through an example of turn-taking on a partial specification.

An example of supporting partiality in mixed-initiative dialogue is explained in Figure 5.15. In this example, a given description can be integrated with a massing of type *geometry*, the resultant is assigned an underspecified location feature, whose value is required to be of type *geom*. The description, *massing* is assigned to the visual feature node shown. In this scheme, it is possible to state during exploration that there exists an entity of *massing* with three features PROPERTIES, OBJECT and LOCATION.

In the visual feature node representation, it is possible to specify the features, PROPERTIES of type *property* and OBJECT of type *geometry* but not the feature LOCATION, which is assigned a

$$
{}_{\textit{massing}}\!\left[\begin{array}{ll}
\text{PROPERTIES} & \boxed{1}\ {}_{\textit{property}}\!\left[\begin{array}{ll} \text{STYLE} & \text{continuous} \\ \text{COLOUR} & \text{yellow} \\ \text{LABEL} & \text{mass} \end{array}\right] \\
\text{OBJECT} & {}_{\textit{geometry}}\!\left[\begin{array}{ll} \text{INSTANCE} & \text{geom} \\ \text{ATTRIBUTES} & \boxed{1} \end{array}\right] \\
\text{LOCATION} & {}_{\textit{point}}[\,]
\end{array}\right]
$$

Figure 5.15: Visual feature node for underspecified entity of type *massing*.

generic type, *point*. Thus, the visual node captures the idea that there exists a geometric object of type *massing* and that its feature LOCATION is constrained to be of type *point*. Given the equivalence of structures and descriptions, the underspecified visual feature node shown in Figure 5.15 can be interpreted as a formal command for the generation of massing elements. More importantly, the possibilities for locating the massing element are open to mixed-initiative specification by the user's action on a visual feature node or through the resolution of formal constraints.

$$
{}_{\textit{command}}\!\left[\begin{array}{ll}
\text{LOCATION} & {}_{\textit{point}}\!\left[\begin{array}{ll}
\text{COORD\_X} & {}_{\textit{real}}\!\left[\begin{array}{ll} \text{VALUE} & 625.6 \end{array}\right] \\
\text{COORD\_Y} & {}_{\textit{real}}\!\left[\begin{array}{ll} \text{VALUE} & 125.6 \end{array}\right] \\
\text{COORD\_Z} & {}_{\textit{real}}\!\left[\begin{array}{ll} \text{VALUE} & 3.6 \end{array}\right]
\end{array}\right]
\end{array}\right]
$$

Figure 5.16: Visual feature node for underspecified entity of type *command*.

This is explained in Figure 5.16 with another underspecified feature node of type *command*, whose feature LOCATION is constrained to be of type *point* with specified coordinates. Note, that this is only true under the assumption that feature nodes can carry maximal values such as the coordinates of a location[4]. It follows from the above, that if *massing* from Figure 5.15 is *compatible* with *command* from Figure 5.16, the unification of the two will provide a feature structure whose location feature LOCATION will result in the more specific of the two values. To be compatible in type, the result must be the meet or a subtype of the meet of the two argument types. Hence, the result of a typed unification is a more specific feature structure or atom drawn from the type hierarchy. Thus, in a mixed-initiative scenario, the user might provide a location value for an underspecified geometry generated by the formalism. Alternatively, the user might specify a geometry for which the description formalism provides one or a number of possible locations in a current problem state. Through turn-taking, both the location and geometry of an element in a

[4]In the design of GENESIS, Heisserman [1991, p 133] notes that the inability to specify an intensional model of geometric information remains a major drawback for interactive systems for generative design.

solution state can be resolved.

5.3.2 Supporting structure sharing

$$
{}_{\textit{massing}}\begin{bmatrix}
\text{PROPERTIES} & \boxed{1}\ {}_{\textit{property}}\begin{bmatrix} \text{STYLE} & \text{continuous} \\ \text{COLOUR} & \text{yellow} \\ \text{LABEL} & \text{mass} \end{bmatrix} \\
\text{OBJECT} & {}_{\textit{geometry}}\begin{bmatrix} \text{INSTANCE} & \text{geom} \\ \text{ATTRIBUTES} & \boxed{1} \end{bmatrix} \\
\text{LOCATION} & {}_{\textit{point}}[\,]
\end{bmatrix}
$$

Figure 5.17: Substructures can be shared in a visual feature node. For example, the PROPERTIES of *massing* are the same as the ATTRIBUTES of *geometry*. Further, the specification of *property* can be changed either through the PROPERTIES of *massing* or through the ATTRIBUTES of *geometry*.

Structure sharing is another fundamental property of typed features that plays a significant role during mixed-initiative dialogue. Visual feature nodes enable the user to develop specific exploration paths in great detail and then provide the resultant feature structure to the generator. The generator can reuse the resultant feature structure multiple times in other feature node contexts through structure sharing. Recall that feature nodes compose feature structures (Section 4.4). Thus, at the visual feature node level, dialogue constructs can take advantage of structure sharing in their underlying feature structures.

Feature structures can be shared across a design space. By reusing of shared feature structures, the user can *converge* information from other exploration paths into the current path of exploration. This bears explanation. For example, there may be two bathrooms in two distinct house designs that are identical. The feature node collections that represent the exploration steps of the first and second designs are distinct paths in the satisfier space. The problem states and choices made by the designer in both paths are also distinct states. However, a portion of the solution state, subsequent to exploration, is identical, in this case, the bathroom design.

In design space, this fact would result in the existence of two distinct feature structures containing the same information (feature structures are intentional). The feature node allows the designer to structure share the bathroom solution of the second design with the first. Once this equivalence is declared, the exploration structure of the first is converged into the exploration structure of the second design. It is important to note that two distinct visual feature nodes can represent two distinct exploration threads in satisfier space. Through structure-sharing dialogue, feature structures can be shared, reused and converged in design space.

Another result of structure-shared feature nodes is the fact that no single value has a unique status. For example, the creation of symbolic links in a unix file system, creates the illusion of sharing structure, but merely points to a different node in the file system. In feature structures, all structure-shared nodes have equal presence.

In Figure 5.17, the PROPERTIES of *massing* are the same as the ATTRIBUTES of *geometry*. Further, the specification of type *property* can be reformulated either through the feature PROPERTIES of type *massing* or through the feature ATTRIBUTES of type *geometry*. During exploration, the generative component can instantiate a massing element, with an unspecified feature ORIENTATION. Subsequently, the user might specify an orientation by direct manipulation, by drawing an arrow that has a value, *direction* for its feature, ANGLE. The process of typed feature structure unification enables the explorer to structure share the value of feature ANGLE of the feature structure of type *direction* with the value of ORIENTATION. Thus, when the two features are unified successfully, the resultant feature node of *massing* has a new value for the feature, ORIENTATION. This value is now given by the more specific value of the feature, ANGLE. Note that the property of structure sharing can be nested. Feature nodes are recursively contained through the *feature-value map* and *feature-value pair* relations of the notation. Structure sharing and its representation using co-reference tags presents the notation with the ability to avoid redundant substructures[5].

5.3.3 Supporting dialogue integration

The visual feature node addresses problems associated with the integration of input from the differing modalities of exploration. While the input of each individual mode of exploration can be assigned meanings, the problem of combining each input into an integrated meaning remains.

From the perspective of the formal substrate, integration is realised by conducting the dialogue in terms of the formal mechanisms available for information combination. For example, consider the unification algorithm, described in Section 2.2.3. Feature structure unification[6] is an operation that determines the consistency of two pieces of partial information, and if they are consistent, combines them into a single result. Within the formalism, this operation is used to address dialogue integration. Unification is an appropriate basis for mixed-initiative integration as it can combine complementary input or redundant input from both modalities of exploration. Further, in the case of contradictory inputs, unification can rule out the possibility of integration. A feature node consists of a collection of feature value pairs. The value of a feature may be an atom, a variable or another feature value map. When two maps are unified, a composite map containing all of the feature specifications from each component structure is formed. This is subject to the restriction that any feature common to both feature structures must not clash in value. If the values of a common feature are atoms, they must be identical. If one is a variable, it becomes bound to the

[5]In the design of GRAMMATICA, Carlson [1993] notes that the ability to detect and represent duplicates remains a major challenge for interactive exploration of design spaces.

[6]See Carpenter [Carpenter 1992, p 45] for a formal discussion of unification.

value of the corresponding feature in the other feature structure. If both are variables, they become bound together, constraining them to always receive the same value. If the values themselves are feature structures, then the unification operation is applied recursively.

Similarly, the formal mechanism for extending partial satisfiers incrementally, π-resolution, is used to extend dialogue. During exploration, this enables a given step of formal exploration to be compatible with a given step of user manipulation. If two dialogue fragments are compatible, then the two inputs can be combined together into a single result. The compatibility of dialogue fragments is captured through the unification operation. Note that the order in which feature value pairs are displayed in a visual feature node is flexible. If conflicts arise during dialogue, the position of a *feature-value pair* can be reordered to reveal inconsistent features. From the perspective of the user, the results of the application of exploration moves are seen in the dialogue layer as visual feature nodes. User interaction with feature nodes forms the basis for dialogue integration. In this manner, the user and the formalism are able to integrate their actions to act jointly on exploration problems.

5.3.4 Supporting dialogue disambiguation

Mutual disambiguation is another property supported in the visual feature node. An exploration move that is partially specified is open to multiple interpretations. In such a situation, a collection of many feature-value pairs may be available for unification. For example, if a given description can be integrated with a massing of type *geom*, it can be assigned an underspecified LOCATION feature, whose value is required of be of type *geom* as shown in the previous discussion of partiality in Figure 5.15. In the description node, *massing* can also be assigned a feature node of type *location* from a number of sources. For instance, the location might be constrained to be adjacent to a previously created entity of *massing*. Thus, it is possible that there exists not one, but a number of feature nodes of compatible type with type *massing* with the feature, LOCATION. The dialogue layer provides a mechanism for mutual disambiguation, such that it is possible to specify the feature, LOCATION of *any* compatible type at the current state of exploration to disambiguate the choice. Given multiple options for interpreting the value of the feature node of type *location* shown in Figure 5.15, a disambiguation process is a necessary attribute of dialogue to resolve non-determinism. The same process of dialogue disambiguation applies to disjunctive nodes. Given a number of possible alternative choices, the designer can disambiguate a disjunction by interaction with the visual feature node representing the disjuncts.

For example, in a user driven query, the formalism might compute a range of possible feature nodes that are extensions of type *location*. The dialogue layer makes them available to the user as a collection of visual feature nodes. Alternatively, for an explorer-driven step, for instance, a command to create an entity of type *massing*, the type *location* can be disambiguated by the user through interactive browsing of the current state, selection of a current point, command input, structure-sharing or a constraint specification.

It follows from the above that when conflicts or multiple choices arise during exploration, mixed-initiative in the dialogue layer can provide for mutual disambiguation through the visual notation.

In the visual feature node, this process of disambiguation is *mutual*. This is, it can be interpreted either as a formal command for the generation of massing elements by the explorer or a user-driven process wherein the possibilities for locating the massing element are open to exploration through the specification by the user's action on the visual feature nodes.

Figure 5.18 shows a feature node of type *massing*, whose feature LOCATION is constrained to be of type *point*. The feature value can be resolved either by the user or by the formal generator. It follows that if the location feature of *massing* is compatible with one or more locations of compatible *location*, the unification of the two can be resolved through a process of mutual disambiguation.

In the example of disambiguation shown here, there are several partial interpretations, one for massing and two for location. Since, either of the two locations might be equally valid for the unification to succeed, only a process of mutual disambiguation can isolate the valid choice of *location*. The dialogue layer allows *either* of the above and does not distinguish between the modality of exploration, direct specification by the user or constrained search by the generator. In each case, the unification-based integration strategy ensures that mixed-initiative exploration compensates for exploration non-determinism through type constraints on the values of features. Further, the restrictions imposed on these values ensure that the exploration maintains integrity and consistency during the process of disambiguation.

5.3.5 Supporting multiple modes

Visual feature nodes provide support for multiple modes of exploration. An exploration process can enter into an explicit "mode" of operation in which a specific type of operation is repeated until either the mode is changed or incorrect input is entered. This enables an exploration process to block out input that is inconsistent with the first specification. The modality consists of an initial input specification by either the generator or the user and all subsequent moves are filtered through unification with this mode. The mode is set by an initial input specifying a feature structure. Subsequent moves result in the creation of more specific feature structures that unify with the specified mode. This constrains and restricts the dialogue to entities that are refinements or extensions of the feature structure specification setting the mode. When there is no interpretation that unifies with the one initially specified, the "mode" is ended. The setting of multiple modalities introduces a stronger form of mixed-initiative, where the output of the processes are directed by the initial mode. For example, the generator can enter into a mode for creating entities of type, *massing*, whose OBJECT value is constrained to be of type *column*. This results in a more specific

$${}_{massing}\begin{bmatrix} \text{PROPERTIES}_{property}[\,] \\ \text{OBJECT}_{geometry}[\,] \\ \text{LOCATION}_{point}[\,] \end{bmatrix}$$

(a)

$${}_{location}\begin{bmatrix} \text{LOCATION} & {}_{point}\begin{bmatrix} \text{COORD\_X} & {}_{real}\begin{bmatrix}\text{VALUE} & 125.0\end{bmatrix} \\ \text{COORD\_Y} & {}_{real}\begin{bmatrix}\text{VALUE} & 25.0\end{bmatrix} \\ \text{COORD\_Z} & {}_{real}\begin{bmatrix}\text{VALUE} & 3.5\end{bmatrix} \end{bmatrix} \end{bmatrix}$$

(b)

$${}_{command}\begin{bmatrix} \text{LOCATION} & {}_{point}\begin{bmatrix} \text{COORD\_X} & {}_{real}\begin{bmatrix}\text{VALUE} & 625.6\end{bmatrix} \\ \text{COORD\_Y} & {}_{real}\begin{bmatrix}\text{VALUE} & 125.6\end{bmatrix} \\ \text{COORD\_Z} & {}_{real}\begin{bmatrix}\text{VALUE} & 3.6\end{bmatrix} \end{bmatrix} \end{bmatrix}$$

(c)

$${}_{massing}\begin{bmatrix} \text{PROPERTIES}_{property}[\,] & \\ \text{OBJECT}_{geometry}[\,] & \\ \text{LOCATION} & {}_{point}\begin{bmatrix} \text{COORD\_X} & {}_{real}\begin{bmatrix}\text{VALUE} & 1125.6\end{bmatrix} \\ \text{COORD\_Y} & {}_{real}\begin{bmatrix}\text{VALUE} & 125.6\end{bmatrix} \\ \text{COORD\_Z} & {}_{real}\begin{bmatrix}\text{VALUE} & 13.6\end{bmatrix} \end{bmatrix} \end{bmatrix}$$

(d)

Figure 5.18: Feature node of type *massing* (top) with two options *location* and *command* for disambiguation of feature LOCATION.

feature structure, which serves as the initial specification for the mode, which will be subsequently unified with future input, either from the generator, user or both. For example, the user could move the mouse to a desired location in the state and lock in the location. This will result in the creation of a massing entity with OBJECT of type *column* and LOCATION of type *point*. The modality enables the initial input to be unified with the subsequent input resulting in a structure that represents a more specific type of entity. Further, subsequent exploration moves directed from the user's mouse actions on the feature nodes, creating feature structures of type *point*, will result in the creation of massing units with OBJECT value *column* and each LOCATION value of *point*. When the user enters an input that results in the failure of unification, the mode ends, and initiative is returned to its default mode.

5.4 Summary

This Chapter develops the dialogue layer of the interaction model for design space exploration. Two requirements posed in developing human-computer dialogue, *dialogue representation* and *dialogue integration* are addressed using the attribute-value matrix notation for typed feature structures. Mixed-initiative dialogue in design space exploration is addressed through the development of a visual notation for representing problems, solutions, choices and history.

The notation is extended into interaction objects by specifying interaction logic for unfolding the components of the visual notation. The feature structure representation, and interaction logic are brought together in the dialogue layer construct, the *visual feature node*. The process of mixed-initiative dialogue during exploration is implemented using visual feature nodes. Through this construct, the user is able to participate in a dialogue with the description formalism to construct problems, navigate solutions, make choices and record the history of exploration. Visual feature nodes implement the display, feedback and propagation of dialogue during exploration. They are implemented as user interface objects in $\mathcal{FOLDS}$, a detailed discussion of which awaits description in Chapter 7, Section 7.1 on Page 108. The next chapter addresses the task layer of the mixed-initiative interaction model for supporting design space exploration.

Chapter 6

A mixed-initiative task layer

The third and final layer of the mixed-initiative model is the task layer. The specification of the task layer completes the development of the mixed-initiative interaction model for design space exploration. The task layer permits the user to access the formal design space exploration movement algorithms in terms of the designer's domain concepts specified in the domain layer and through a model of mixed-initiative dialogue specified in the dialogue layer.

6.1 The Task layer

The domain layer constructs, problem states, solution states, feature nodes and satisfier spaces encapsulate the entities of exploration from the designer's perspective. The dialogue layer construct, the *visual feature node*, provides a model of turn taking between the formalism and the designer. Based on these constructs, the *task layer* addresses user access to the formal operations for computing exploration described in Section 2.4 and enables the designer to generate, navigate and synchronise a range of *design moves*.

As noted in the requirements of the interaction model in Section 3.1.2, mixed-initiative must permit the user to access the formal exploration operators, integrate system-driven and user-driven moves and enable the user and the formalism to share joint responsibility.

During exploration, system-driven moves represent the operations that modify the *intrinsic* attributes of a feature node. The unfolding[1] of the intrinsic properties of a visual feature node is based on the movement operators of the exploration formalism described in Section 2.4. The formal moves operate on a partial satisfier PSat through the visual feature node, VNode.

Designer-driven moves represent operations that modify the intrinsic and extrinsic attributes of a visual feature node. The unfolding of the intrinsic attributes of a visual feature node is based on interaction with formal design moves. The unfolding of extrinsic properties of the feature node is

[1]the term unfolding is defined in Section 3.1.1.

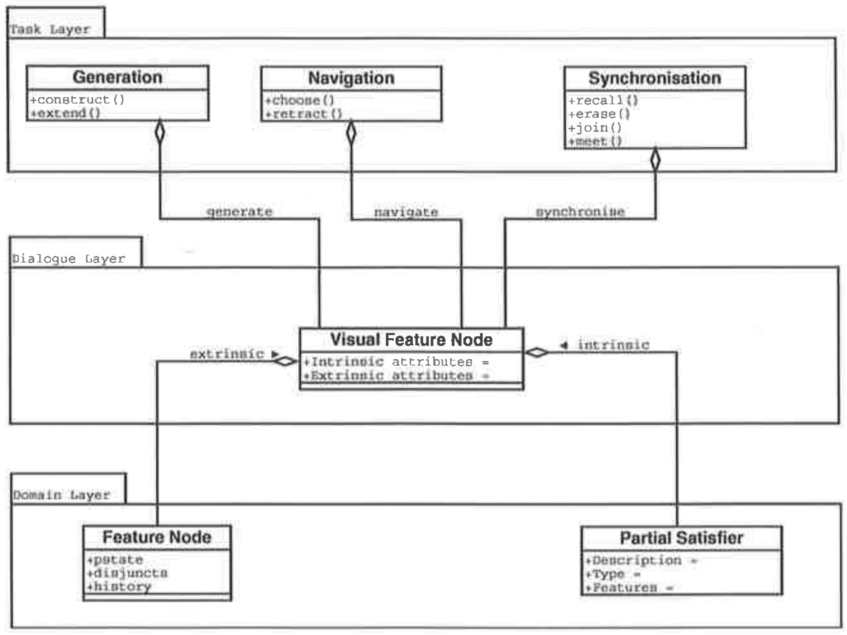


Figure 6.1: The exploration tasks are specified over a *visual feature node.* The elements of the visual feature node map onto the domain layer constructs.

based on interaction with the interface components of a visual feature node, VNode. The designer operates on the visual interface components of the feature node to affect change during exploration and these changes are cast as extrinsic to the representation. To incorporate mixed-initiative, it is necessary to integrate both types of moves in the task layer.

This is addressed by treating both types of operations under a common conceptual metaphor, termed *unfolding.* Visual feature nodes can be *unfolded* by formal moves as well as their behavioural properties during exploration. The task layer permits the designer and the formalism to unfold a visual feature node during exploration. The task layer for mixed-initiative unfolding is subdivided into node *generation*, node *navigation* and node *synchronisation.* This mapping is diagrammed in Figure 6.1.

The generation task corresponds to the process of creating visual feature nodes from initial problem statements. The navigation task corresponds to movement along the attributes of a visual feature node. Navigation comprises movement along both intrinsic (formal) and extrinsic (behavioural) attributes of a feature node. The synchronisation task covers the process of unifying two feature nodes, suppression of features and erasure of nodes.

Each task layer construct comprises specific types of operations for node unfolding. Under each operation, the role of the user, the role of the formalism and the resultant changes to the exploration space are examined.

- **Generate a visual feature node,** VNode
  The task of generation comprises interaction with the CONSTRUCT and EXTEND operations. The construct operation involves the specification of a query description. The query is parsed and converted into a visual feature node for further exploration.

- **Navigate a visual feature node** VNode
  Navigation corresponds to movement along the paths of the feature node, VNode. Path navigation comprises finding and moving between paths in the current node through search, query and sequential moves. The path navigation process comprises interaction with the operations, CHOOSE, RETRACT.

- **Synchronise paths, P_i and P_j.**
  Given two extant paths, P_i and P_j in the exploration, synchronisation comprises interaction with four operations, RECALL, ERASE, JOIN, MEET. The RECALL operation permits the reuse of previously explored paths. The ERASE operation allows the designer to perform deletion and hysterical undo of paths. The JOIN operation computes the specialisation of two distinct exploration paths. The MEET operation computes the generalisation of two exploration paths.

6.2 The task of generation

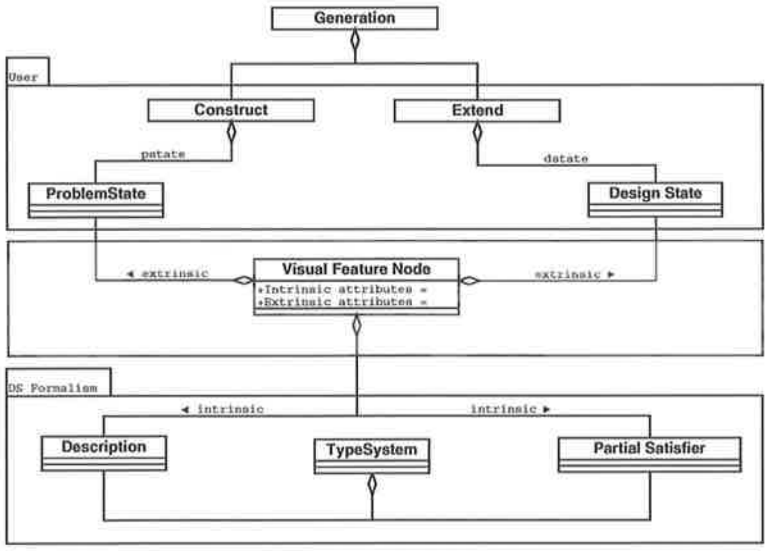


Figure 6.2: The task of generation comprises the construction of problem states through CONSTRUCT and the extension of partial design states through EXTEND. The *visual feature node* maps the generation task between the user and the design space formalism.

The *visual feature node* maps the generation task between the user and the description formalism as shown in Figure 6.2. The operators involved in the task of generation are described in the next sections.

6.2.1 The construct operation

The generation of a visual feature node comprises the conversion of an initial problem statement as a query ϕ, into a problem state $\bar{d}$ and its subsequent extension. A problem state is initially defined in the attribute-value description language, as described in Section 4.2. A visual feature node composes the problem state, $\bar{d}$, as the first element of a satisfier space, SatSpace. The task of generation comprises the construction of problem states through CONSTRUCT operation.

The initial feature node is constructed using the CONSTRUCT operation. The CONSTRUCT operation comprises taking the initial query ϕ converting it into a problem state $\bar{d}$ and computing its visual feature node representation, VNode. The attributes of VNode are displayed as a *feature-value pair*. The sequence of interactions involving the designer and the formalism in the feature node construction process is shown in Figure 6.3.

The designer represents the problem state through a query description, ϕ in the attribute-value description language described in Chapter 2. The explorer then converts ϕ using the satisfiability algorithm into a problem state, $\bar{d}$. The result of this step of exploration is the creation of a satisfier space and its corresponding visual feature node as defined in Chapter 4. The mixed-initiative dialogue layer displays $\bar{d}$ as the visual feature node, *VNode*. This node serves as the basis for sharing context and problem focus for the next steps of exploration.

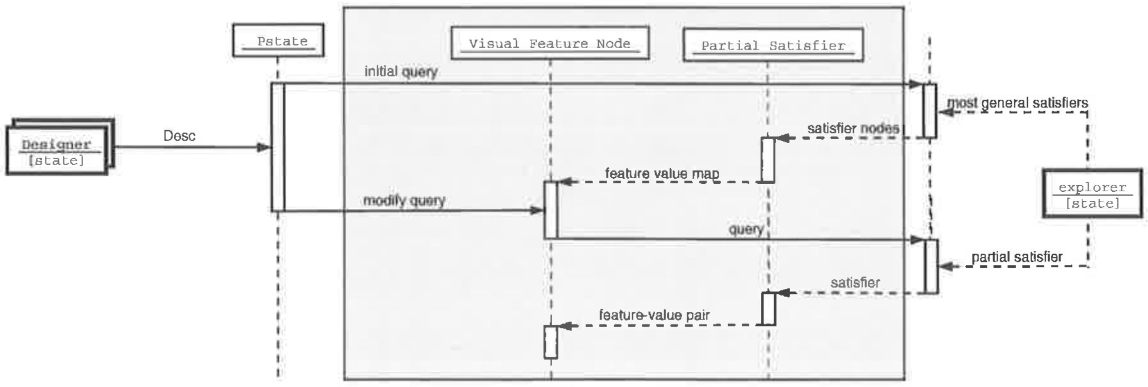


Figure 6.3: Interaction sequence of the CONSTRUCT operation. The designer specifies an initial problem statement to the description formalism. The formalism returns a collection of partial satisfiers. The vertical bars indicate shifts in control.

The designer controls the visual feature node and its surrounding context through the dialogue layer. Dialogue with this node allows designers to display and explore the visual attributes associated with a feature node and their values.

For example, as shown in Figure 6.3, mixed-initiative interaction with the CONSTRUCT operation begins with the specification of a description in the form of an initial query (problem) by the designer. The explorer then constructs a a satisfier node, PSat. The satisfier node represents the specification of the problem through the *feature-value map*. Through the *feature-value map* , the most general satisfier is mapped to a visual feature node, VNode and control handed back to the designer. The user can unfold this node along a valid path, generate a new description query, ϕ or reformulate the problem description. In this way, the visual feature node becomes the context for subsequent tasks of exploration.

6.2.2 The extend operation

Once a feature node has been constructed as described in Section 6.2.1, the explorer retrieves the last element in the path sequence and returns its root node as a partial satisfier. This node is converted to a visual feature node and the designer can interact with any one of the attributes of the visual feature node, defined in chapter 5.

The exploration of an initial problem state, $\bar{d}$, to partial design states is performed through the EXTEND operation. The generative algorithm underlying stepwise extension of a *feature-value pair* of the visual feature node is incremental π-resolution. The incrementality of π-resolution, described in Section 2.4.1, provides natural entry points for user interaction. The designer interacts with the resolution process, by selecting a *feature-value pair* and seeking to extend it to the next state. Exploration advances by stepwise operations on the *feature-value map* displayed as a visual feature node. At each step, the explorer (description formalism) constructs a sequence of partial satisfiers, PSat through incremental π-resolution. This permits the explorer to introduce new nodes in design space corresponding to the steps of extension specified by the designer. These steps of extension are performed over a *feature-value map*. Path extension by the designer comprises a sequence of extension steps corresponding to the selection of a *feature-value pair*, and the selection of a type constraint as a direction for extension. The mixed-initiative interaction with the EXTEND operations is shown in Figure 6.4.

The designer selects a *feature-value pair* and requests an EXTEND operation on this structure. Given this *feature-value pair*, the explorer traces a path through the subspace of possible states that are consistent with it. Since the EXTEND operation is synonymous with an incremental π-resolution step, the set of legal types and associated recursive type constraints to which the selected node can be refined along this attribute are presented to the designer as a list of types and constraints. Thus, given the existence of a set of legal types, the designer can choose one of the subsuming types and present the explorer with a legal operation that extends the current node. If the EXTEND operation is successful, the new *feature-value map* is displayed. The dialogue between the designer and the explorer continues through this type of turn-taking until the partial satisfier is fully resolved. Decisions are not subject to a global inference strategy, but are goal-directed in that each resolution

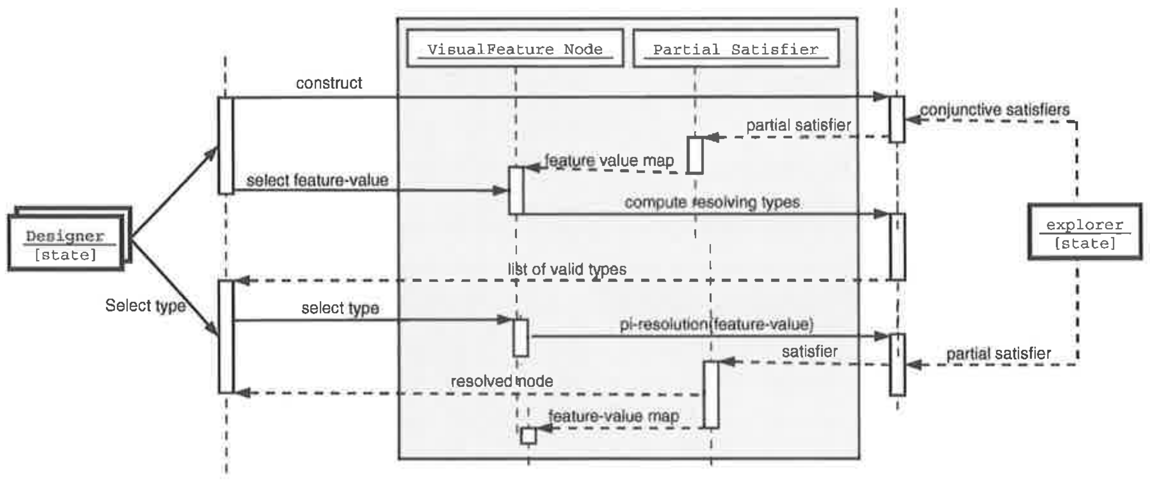


Figure 6.4: Interaction with the EXTEND operation. The designer selects a *feature-value pair* and requests a move to the next state. The explorer returns the list of available types to which the *feature-value pair* can be extended as a collection of types from $\langle \mathsf{Type}, \sqsubseteq \rangle$. The designer then selects a type from the selection set and returns the type to the explorer. The explorer computes the partial satisfier and returns the value to the mixed-initiative dialogue layer for further exploration until the node is fully resolved. The solid arrows indicate flow of the designer's interaction, while the dotted arrows indicate the flow of the explorer's actions.

step introduces new constraints and opens up possible spaces. Once the next legal operation has been decided by the two-stage dialogue described above, the explorer extends the computation to the next state and the interaction loop described above is repeated.

6.3 The task of Navigation

Navigation corresponds to movement along the attributes (paths) of a visual feature node. Navigation comprises the incremental *movement* from the rooted *partial satisfier* to another along a defined path. Navigation operations enable the designer to locate, identify and move through the collection of constructed nodes and their paths. Navigation operations support forward and backward traversal from a node in satisfier space along any one attribute of the node. For example, a step forward along a feature path in the current node will bring the exploration to the next feature node. A step backward on a feature at the current node will return the focus to the previous node. The operations for navigating a rooted partial satisfier node PSat are CHOOSE and RETRACT. They are shown in Figure 6.3.

6.3.1 The choose operation

The CHOOSE operation supports the resolution of branching conditions that arise during exploration. Support for choice at branch points is a key element for dealing with exploration nondeterminism.

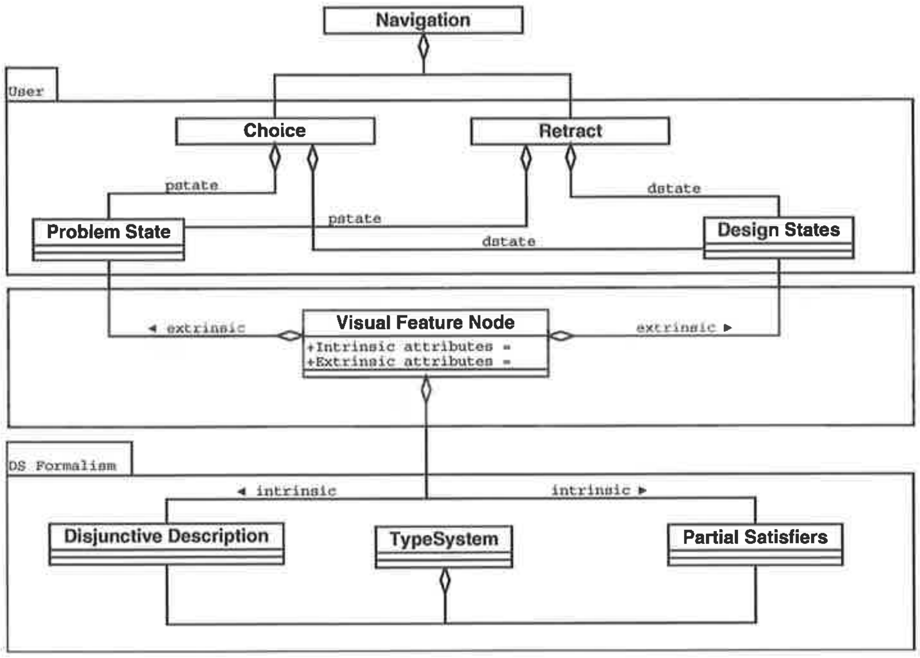


Figure 6.5: The task of navigation comprises movement through through the operations of CHOICE and RETRACT. The *visual feature node* maps the navigation task between the user and the design space formalism.

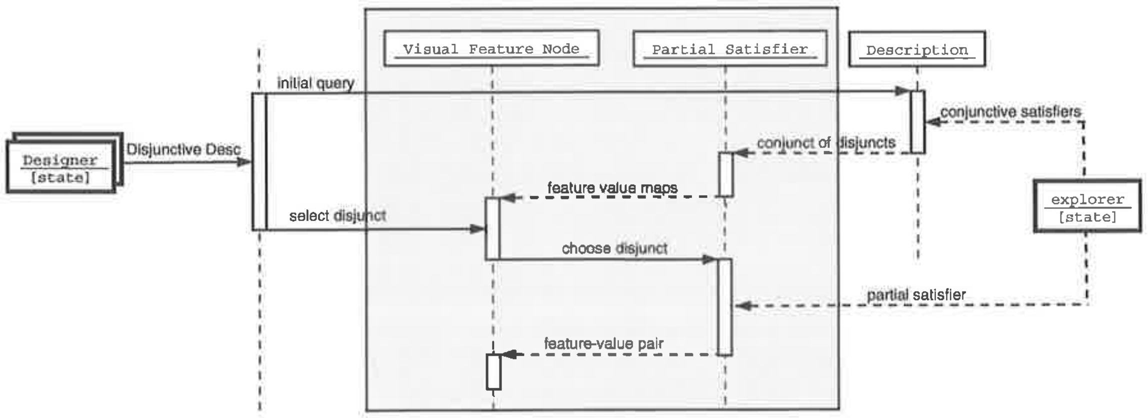


Figure 6.6: The CHOOSE operation. The designer specifies a disjunctive query description, $\phi \bigvee \psi$. The explorer returns the most general satisfiers as a conjunct of disjuncts. The conjunct is displayed as a collection of feature nodes, each representing the disjunctive portion of the query. The vertical bars indicate explicit focus of control. The shaded box indicates areas of mixed-initiative. The horizontal lines indicate the direction and flow of control.

Problems can be specified in the form of *disjunctive* descriptions such as, $\phi \bigvee \psi$. Such a disjunction creates alternatives in the path sequence, requiring user intervention. In such a disjunctive problem specification, the possible solutions to the problem description are represented as a *conjunct of disjuncts* in the visual feature node (see Section 5.2.2). The explorer presents all the choice points of branching to the designer. The designer resolves the non-determinism by choosing one of the disjuncts. Figure 6.6 shows the sequence of interaction involving the resolution of non-determinism through mixed-initiative dialogue.

Further, choices made by the designer are recorded as visual feature nodes in SatSpace. Hence the intentionality of the history of exploration is made transparent for future exploration. In the example figure, this amounts to returning to the visual feature node and choosing an alternate disjunct. Choice enables the designer to engage the power of the formalism but maintain control over the generation of alternatives and their exploration.

6.3.2 The retract operation

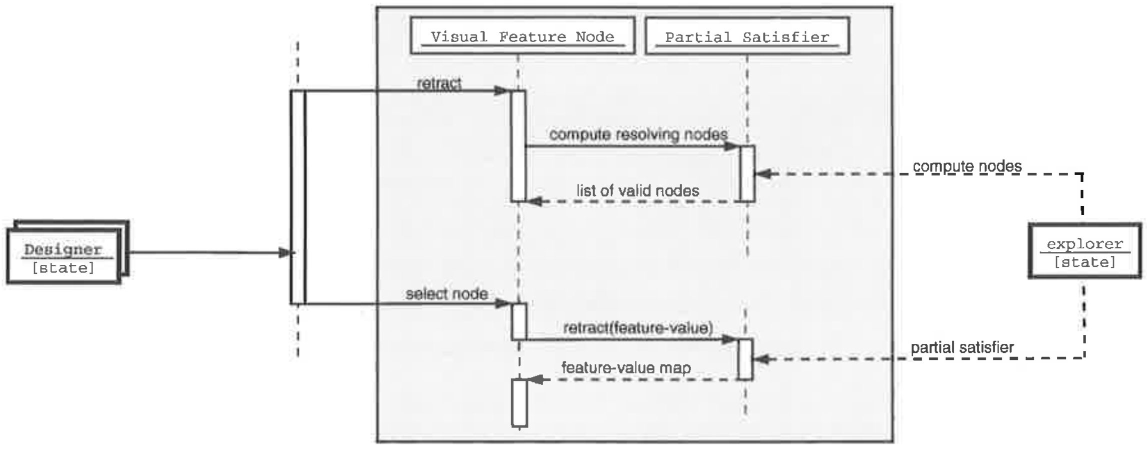


Figure 6.7: A RETRACT operation over the design space. The designer specifies a visual feature node to retract. The explorer computes the list of nodes that satisfy the retract operation on the map. The designer selects a valid node from the returned list. The explorer computes the partial satisfier and returns the feature value map corresponding to a retract operation on the node.

The RETRACT operation enables a designer to reverse the effects of an EXTEND operation on the current node. The RETRACT operation is also the exploration equivalent of backing up the current active path to the previous state. The path sequence in satisfier space enables the explorer to perform a RETRACT operation on a substructure node in the partial satisfier and retract the current state to any of the previous subsuming states of the current node. All paths from a state to its immediate subsuming states and recursively to all subsuming states are accessible in design space. Retraction permits the designer to access this ordering at the satisfier space. Thus retraction corresponds to a form of information abstraction, providing the means to move from a more detailed feature value to a less detailed one. Through RETRACT, the designer can move to any previous *feature-value map* available in the path sequence. The designer can move back along the current path sequence at the finest level of granularity, that is, an incremental π-resolution step. Alternatively, the retraction can be to any node in SatSpace that is conceptually a more general subsumer of the current node. Thus backwards movement in satisfier space is a useful operation for moving either to any previous π-resolution state along a path sequence or to an arbitrary state representing a more abstract state of the current node. As a concrete example of backward movement in the design space consider the exploration of an abstract massing model of a house of type *house* that contains a substructure node of type, *kitchen.* The designer begins with an overall massing model of the house and creates an exploration fragment to detail the kitchen. The designer can move backward along the resolution path of the kitchen exploration or change the massing model of the house, which subsumes the kitchen exploration.

The generation and navigation operations operate at the level of a single path of exploration. It is possible[2] to operate at the level of the design space structuring mechanism, subsumption. Large grained interaction over design space is covered in the next section.

6.4 The task of synchronisation

The satisfier space is the collection of feature nodes traced by interaction. The design space is the collection of all possible subsumption ordered states of exploration. Synchronisation addresses the problem of unfolding the results of two distinct exploration paths in the satisfier space with respect to their underlying ordering in the design space. In the formalism, exploration is organised by logical components related through the property of subsumption. Subsumption ordering in design space maintains information integrity. Through subsumption, it is possible to recover the results of previous explorations. This may happen either along different attributes of the node or along the same attribute of different nodes. The task of synchronisation is shown in Figure 6.4. These operations address the problem of synchronising two paths into a single one. The *interleaving* of path unfolding through synchronisation enables the designer to make large moves, shift between parts of a design and retrieve or re-use previous paths of exploration. The operations for supporting synchronisation are RECALL, ERASE, JOIN and MEET.

6.4.1 The recall operation

The RECALL operation enables the designer to retrieve exploration fragments from related path sequences to assist in the current exploration. For example, the RECALL operation allows the designer to access problems and solutions of past explorations. The type of a node in an exploration

[2]See the formal operations in Section 2.4 and the domain layer construct, SatSpace in Chapter 4.

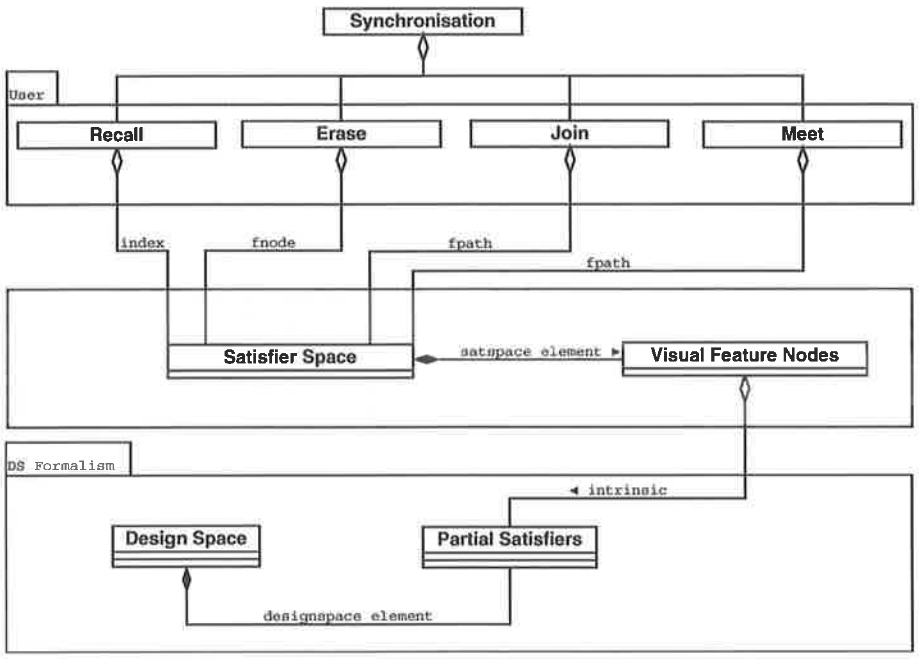


Figure 6.8: The task of synchronisation comprises movement through through the operations of RECALL, ERASE, JOIN AND MEET. The *visual feature node* maps the synchronisation task between the user and the design space formalism.

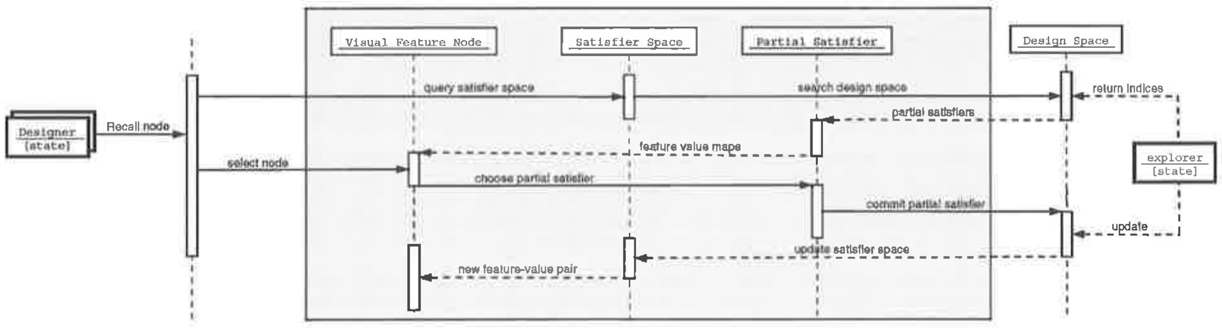


Figure 6.9: The RECALL operation uses the underlying subsumption ordering over feature nodes.

path acts as an index for recall. The use of a feature structure type as an index to a collection of design cases is covered in Section 2.4.2. A detailed view of indexing and recall is set out in [Woodbury et al. 1999].

An example of recall is set out in Figure 6.9. The designer can query the satisfier space, SatSpace for possible exploration paths corresponding to the type of the current node. The formalism returns the indices (types) of possible nodes in design space matching the query as a *feature-value map*. The designer can browse this collection and select a *feature-value pair* that satisfies the needs of the current path. The formalism then commits this selection as a PSat in design space, updates the satisfier space and presents the recalled PSat as a new *feature-value pair*.

This operation supports the recall of an entire path of exploration for a given partial satisfier in design space. This corresponds to a notion exploring object evolution and design history. For example, in the example of recall illustrated in Figure 6.9, the designer can select a node and request its exploration history. In this case, the formalism can reconstruct the path sequence of an index and return the history as a *feature-value map*. The designer can then unfold the resultant visual feature node through interaction to trace its progeny and ancestor nodes in SatSpace.

6.4.2 The erase operation

Support for erasure is central to designing. Conventional design support systems provide *undo*, *delete* and *history* mechanisms for dealing with erasure. Undo is the reversal of the last operation performed. Delete applies to a selection and removes objects from persistent memory. History records a list of operations and is used as a rudimentary form of version control. These systems support only linear forms of erasure [Norman 1988] and do not support arbitrary levels of undo [Cooper 1999]. In generative systems, this process is cast as the substitution of a more detailed symbol by a more abstract one. For example, the grid definition process in Stiny & Mitchell's [1978*b*] Palladian grammar is an example of such substitution.

In design space exploration erasure corresponds to movement in the backward direction [Woodbury et al. 2000]. Here, the separate treatment of undo, delete and history are replaced by a notion of information abstraction. Section 2.4.3 provides an account of the movement operation *hysterical undo*. In subsumption ordered design spaces, erasure fulfils the conventional model of undo as well as the computation of less specific instances of the current feature node, which may not have been retrieved previously.

The ERASE operation is a novel backward navigation technique that conceptually extends simple retraction. In a subsumption-based representation, ERASE pertains to the uncovering of more general states in the implicit design space, not so much to the removal of parts of a state. The ERASE operation corresponds to the movement from the current *feature-value pair* to a less specific one. The user interaction with the ERASE operation involves a mixed-initiative dialogue, similar

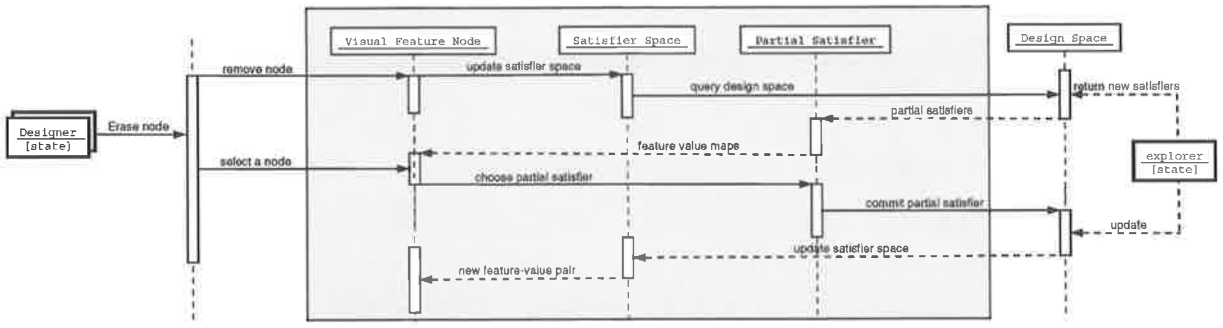


Figure 6.10: The selected node is removed from the satisfier space. This removal yields a change in underlying design space, which is reflected as a set of possible, less specific partial satisfiers. The designer selects one of the satisfiers and the ubsumption ordering is updated to relfect the erasure.

to the previous operators. The user selects the ERASE operation and applies it to a *feature-value pair* in the current feature node. The explorer then determines the erasure possibilities outlined above. An example of this operation in the task layer is illustrated in Figure 6.10. The designer selects a node in SatSpace for removal. The visual feature node marked for removal is deleted from SatSpace. The control then reverts the description formalism. The design space is queried for all satisfiers that subsume the erased node and a *feature-value map* corresponding to the partial satisfiers that subsume the node are returned to the designer. The designer then selects a less specific node that corresponds to the notion of information abstraction and commits this to design space. The state from which information would be "erased" remains unchanged in the design space while the designer's perspective shifts to the the state that corresponds to the notion of information abstraction. This concept provides a clean view of information deletion within the formal properties of the explored space.The effect of an ERASE operation is to shift the focus of the exploration to another node in design space.

6.4.3 The join and meet operations

The JOIN operation corresponds to computing the *unification* of two distinct exploration paths. The MEET operation operation corresponds to computing the *subsumption* of two exploration paths.

The JOIN operation provides access to the formal movement operator for design-unification. This operator is discussed in Section 2.4.4. Design-unification is defined with respect to the subsumption ordered collection of feature structures as a *least upper bound*. Given two exploration paths, the *join* of the two partial satisfiers is the result of combining the derivation steps of each path into a single partial satisfier containing the union of design commitments in the two operands. Figure 6.11 shows an example of exploration using the JOIN operator.

A user interacts with the JOIN operation by selecting two paths from different states. If unification succeeds, the result is a single structure containing all information accessible along either

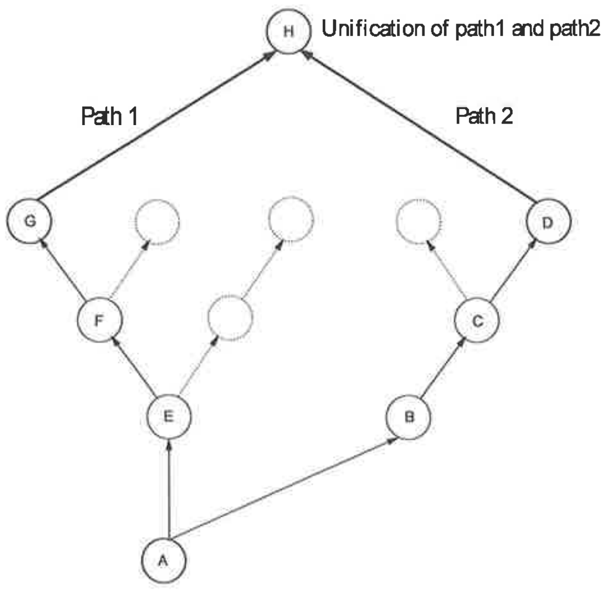


Figure 6.11: Representation of a JOIN exploration operator. Path $A : E : F : G$ and Path $A : B : C : D$ have a valid join. The feature node H represents the specialisation of the values of paths, $Path1$ and $Path2$.

path in the two argument structures. If the paths are simply the root nodes of two feature structures, unification, if successful, results in a single structure combining them both and the resulting structure would take its place in the design space above each of the argument structures. The interactive JOIN procedure is useful in achieving the reuse of design fragments, particularly where the two operand design states embody work on distinct aspects of a common problem.

Figure 6.11 depicts an illustration of the JOIN operation over nodes in a subsumption relation. The example represents the decomposition of a single node A into two distinct explorations, signified by the paths $A : E : F : G$ and $A : B : C : D$. The paths leading to these nodes denote building entities and the edges denote functional roles in the design. Both paths are partial alternate explorations of some final design. The designer can decide to merge these two alternatives into a final design using the JOIN operation. The final outcome represents the specialisation of *Path 1* and *Path 2* into a single design, denoted by the node H.

The properties of the MEET operation over two paths in the present a mechanism for synchronisation in the downward direction. This operator is discussed in Section 2.4.5. Design anti-unification is defined over two partial designs as the most specific feature structure generalising the operands — the *greatest lower bound.* The subsumption ordering over design space guarantees a greatest lower bound. The result of a MEET operation over two paths is the conjunction of the shared

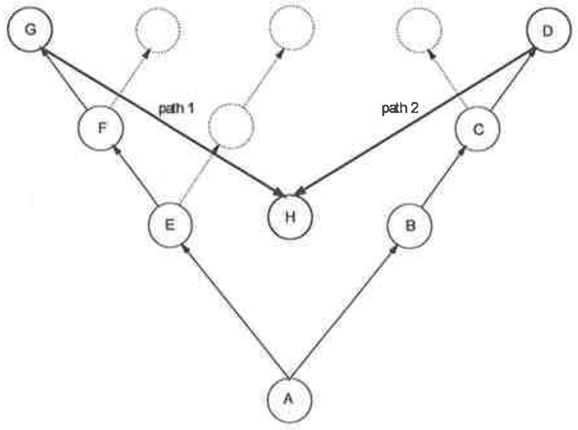


Figure 6.12: Representation of a MEET operation. Paths $A : E : F : G$ and $A : B : C : D$ have a valid meet. The feature node H represents the generalisation of $Path1$ and $Path2$.

derivation steps, which in the extreme case is simply the minimal design state. Contrary to JOIN, MEET is guaranteed to succeed.

Figure 6.12 shows an example of synchronisation through the MEET operation. The example represents the decomposition of a single node into two distinct explorations, signified by the paths $A : E : F : G$ and $A : B : C : D$. The paths leading to the nodes represent two distinct and partial alternate explorations. The designer can decide to compute the generalisation of these two alternatives into a single design using the MEET operation. The final outcome represents the collapsing of the two paths, denoted by *Path1* and *Path2* into a single design, denoted by the node H.

As illustrated above, MEET presents a sophisticated means for large scale information synchronisation. The explorer traverses the subsumption ordering and moves the computation through mixed initiative interaction with the exploration formalism. The designer can choose to continue the exploration process from the resultant state in the subsumption ordering.

6.5 Summary

This Chapter develops the task layer of the mixed-initiative interaction model for design space exploration. The constructs of the task layer provide access to the formal design space movement algorithms (Section 2.4 on page 32). Exploration moves are cast in terms of the interaction metaphor, *unfolding*. Three user level constructs for unfolding, construction, navigation and synchronisation of visual feature nodes are described. Through the unfolding of visual feature nodes, the designer constructs problems, navigates possible solutions to these problems and synchronises nodes in the satisfier space. The specification of the task layer completes the development of the mixed-initiative interaction model for design space exploration.

Part III

FOLDS : FOLDABILTIY OF LARGE DESIGN SPACES

Part III: On the Foldabiltiy of Large Design Spaces

"In the act of design we bring forth the objects and regularities in the world of our concern. We are engaged in an activity of interpretation that creates both possibilities and blindness. As we work within the domain we have defined, we are blind to the context from which it was carved and open to the new possibilities it generates. These new possibilities create a new openness for design, and the process repeats in an endless circle."
Winograd and Flores [Winograd & Flores 1987, p 178].

Part III presents an implementation of the mixed-initiative interaction model, a demonstration of mixed-initiative unfolding of design spaces and the results of the study. It is divided into the following chapters below:

- Chapter 7 discusses the design and implementation of $\mathcal{FOLDS}$, a prototype implementation of the mixed-initiative interaction model for design space exploration. $\mathcal{FOLDS}$ is used to demonstrate mixed-initiative exploration through an example of massing configurations in the context of architectural design.

- Chapter 8 presents the results of the study, the scope and limitations of the mixed-initiative model and a discussion of future work for addressing the limitations of the current work in design space exploration.

- The appendices, Appendix A, Appendix B, Appendix C and Appendix D, contain background material on the typed feature structure implementation underlying the $\mathcal{FOLDS}$ system, a collection of formal terms and definitions used in the thesis, a brief summary of the UML notation, the description of the massing case and notes on the design and implementation of $\mathcal{FOLDS}$.

Chapter 7

Enabling mixed-initiative exploration

To examine how the model of mixed-initiative interaction developed in the thesis can support exploration, an illustrative example of a concrete application in the domain of massing design is presented in this Chapter. The application comprises the implementation of the prototype, $\mathcal{FOLDS}$ and its connection to KRYOS, a collection of class libraries implementing the formal substrate of design space exploration. Through the example implementation, mixed-initiative interaction between the designer and the formalism in massing exploration are discussed.

7.1 $\mathcal{FOLDS}$

$\mathcal{FOLDS}$ or the *$\mathcal{F}$oldability $\mathcal{O}$f $\mathcal{L}$arge $\mathcal{D}$esign $\mathcal{S}$paces*, is an implementation of the mixed-initiative interaction model proposed in this thesis. $\mathcal{FOLDS}$ provides the interaction infrastructure for mixed-initiative exploration through a suite of user interface modules that harness the domain, dialogue and task layers of the mixed-initiative interaction model. Through these layers, the designer can interact with the formal machinery of design space exploration. Notes on the design and implementation of $\mathcal{FOLDS}$ are given in Appendix D.

KRYOS is an implementation of the formal machinery of design space exploration [Burrow 2003]. It provides the formal substrate of the description formalism presented in Chapter 2. KRYOS maintains the relationships between constructs in the description formalism, provides object management and maintains the integrity of design space structure. Notes on the technical architecture of KRYOS are described in Appendix D.

Several kinds of configuration problems arising in design have been identified in the literature on configuration design. They include the following: *layout* dealing with composition of spatial layouts in two dimensions [Flemming et al. 1988, Flemming & Chien 1995, Akin & Sen 1996]; *massing* dealing with abstract composition of solids [Flemming 1990, Woodbury & Griffith 1993, Datta & Woodbury 1998]; *enclosure* dealing with external-internal building envelopes [Woodbury & Chang 1995*a*] and *structure* dealing with structural support systems [Fenves, Rivard & Gomez 1995].

Each kind of configuration problem requires different representation and algorithms. Massing, the configuration of abstract three dimensional entities during the early phases of design is the focus of this study. Although all configuration problems share the requirement for a design space representation, each would require different type hierarchies and descriptions to be represented in *FOLDS*. For example, exploration of structural configurations would require the mapping of the domain layer constructs onto type hierarchies, descriptions and other domain specific tools suitable for supporting structural configuration.

To illustrate how mixed-initiative supports exploration, the scope of the example problem is restricted to the domain of massing configuration. In conceptual architectural design, massing enables a designer to rapidly develop design schemes and explore possible alternatives in the form of abstract compositional forms in three dimensions. The notion of massing configuration addressed in the example is based on the detailed overview of massing given is the SEED-Config Knowledge level [Woodbury & Chang 1995*b*].

Mixed-initiative exploration of massing configurations of single-fronted cottages is explained in *FOLDS*. The example builds upon the kinds of configuration design addressed in SEED-Config [Woodbury & Chang 1995*b*] and the layout configurations of single-fronted cottages reported in Woodbury et al. [1999]. A detailed specification of the massing configuration example is given in Appendix C.

Mixed-initiative interaction between the designer and KRYOS (formal substrate) through *FOLDS* (interaction model) is described through the illustrative example from the domain. The demonstration addresses the states, structure and moves of exploration, the interaction process between user and formalism and the role of mixed-initiative in supporting interaction.

The example of mixed-initiative exploration is described through the following :

- **Domain interaction.**
  The domain layer implements four constructs, problem states, solution states, feature nodes and satisfier space for tying the designer's view of exploration (the domain) to the formal substrate of KRYOS. Mixed-initiative in the domain layer of *FOLDS* is described in Section 7.2.

- **Dialogue in Exploration.**
  FOLDS implements the dialogue layer construct, visual feature node. Through interaction with intrinsic and extrinsic attributes of a visual feature node, *FOLDS* enables mixed-initiative coordination and communication between the designer and KRYOS. Mixed-initiative in exploration dialogue through *FOLDS* is described in Section 7.3.

- **Exploration tasks.**
  FOLDS supports supports mixed-initiative in the task layer through the construction, navigation and synchronisation of visual feature nodes. Examples of mixed-initiative in the task layer of *FOLDS* are described in Section 7.4.

7.2 Domain interaction

The domain layer comprises the constructs, PState, SState, FNode, and SatSpace. Together, they reify problems, partial solutions, choices and history. Through this layer, $\mathcal{FOLDS}$ accesses the Kryos substrate constructs, type hierarchy, appropriateness specifications, constraint system, generators and design space.

7.2.1 PState and SState

The process of defining and initialising a PState involves interaction with elements of a Kryos TypeSystem. The designer writes the kryos TypeSystem components comprising the type definitions, *types*, the appropriateness specifications, feats and constraints, *cons* respectively. Each of these components are recorded in text files. A detailed description of these files with respect to the example discussed here is given in Appendix C.

Type Hierarchy

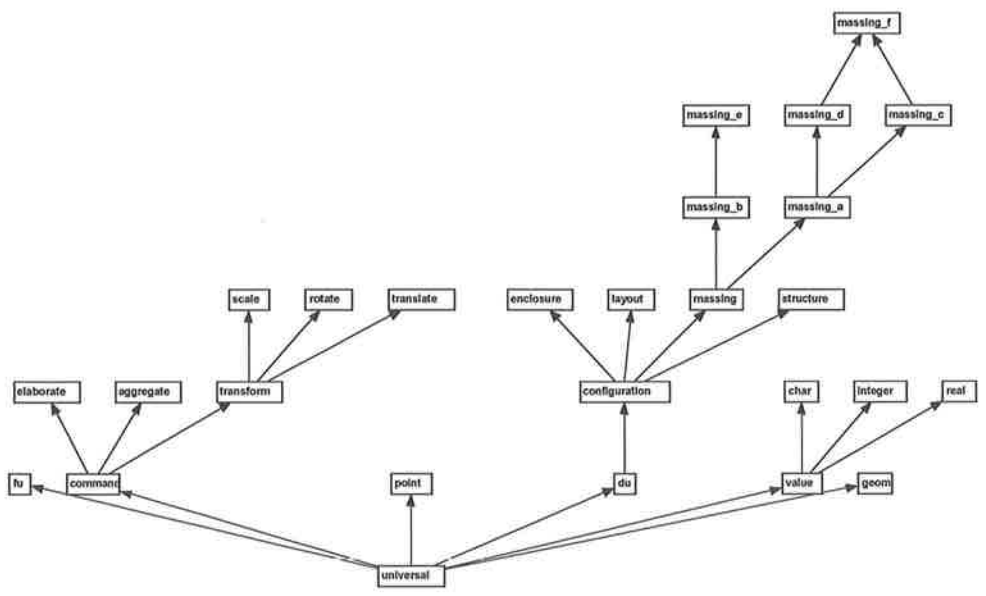


Figure 7.1: The inheritance hierarchy of types. This hierarchy extends the example presented in Figure 2.3 for interactive exploration. The extended hierarchy provides new additional types for supporting massing design exploration including the types: *command, geom, fu, du.*

A PState is mapped onto the type hierarchy as shown in Section 4.2, Equation 4.1. An inheritance hierarchy comprising a list of 43 types representing the domain is shown in Figure 7.2.1.

In the type hierarchy, above the type *universal*, two new types are introduced. The first type *fu*, corresponds to function-units and the second type *du* corresponds to design-units in the SEED Knowledge Level. The models corresponding to the type *du* can be thought of as designs for buildings; those corresponding to the type *fu* as architectural briefs. Immediately above *du*, the type *configuration* is introduced. The *configuration* of a design comprises the subtypes *layout, massing, structure, enclosure.* All of these types are abstract, serving to refine the type *bottom* and do not introduce any features, but such abstract types can be used in constraint expressions. A massing element is a refinement of the type *configuration* which is a subtype of type *du* or design_unit. This example concentrates on the exploration of massing designs. Thus the type *massing*, corresponding to abstract volumes that configure an overall building mass, are discussed in greater detail here.

A detailed representation of massing elements is shown in Figure 7.2. The type *massing*, inherits two features from *du*, DU_LABEL, GEOM. It introduces three types of features, namely MASS_LABEL, MASS_POS, FU. The feature MASS_LABEL serves to identify the geometric design_unit with a name. The feature MASS_POS provides the massing with a positional co-ordinate of type *point.* The feature FU is a hook to the functional roles that the massing element may play during exploration. Massing configurations can recursively contain other massings. The type *massing* has a sequence of subtypes *massing_a, massing_b...* in which *massing_a* $\sqsubseteq$ *massing_b.* Each massing of type *massing_n* introduces a feature MASSEL_N which denotes a sub-massing of *massing.* Thus a feature structure representing a massing of type *massing_f* is a design configuration of $a+b+c+d+e$ sub-massings. The type *fu* has a single sub-type *house* at which features corresponding to functions are introduced. These features carry only functional information, in contrast to the the features introduced in the type, *sfc*, representing a *single fronted cottage.* These features, corresponding to a porch, hall, row of rooms and skillion, denote the particular spatial organisation of single-fronted cottages without making commitment to actual addition or dimensions. Type *house* inherits from both *building* and *massing.* The role of this type is to specialise the generic types, *building* and *massing* by introducing features that correspond to explorations of residential buildings. The type *sfc_house* inherits features from both *sfc* and *house.* Finally, the type *sfc* extends the type, *sfc_house.*

Description queries

The definition of a designer's view of a problem is enunciated in Section 4.2, Equation 4.2. A PState is specified through s a textual description with respect to the InheritanceHierarchy as explained in the previous section. For example, consider the problem statement shown in Figure 7.3.

Problem statements in the description language are loaded and stored in the construct PState. The example problem formulation shown in Figure 7.3 comprises a collection of path equations stated in the form of a conjunctive description. For each valid disjunction-free description of this form, there exists a corresponding PState. With reference to the massing example, the designer

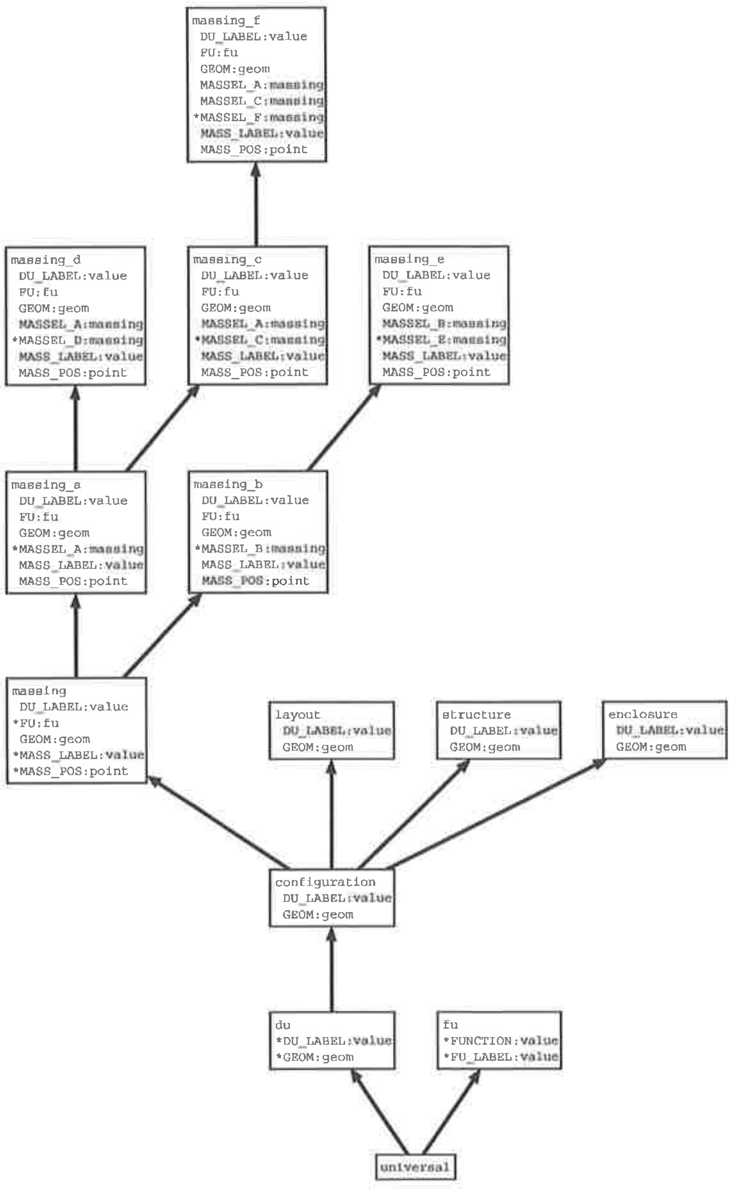


Figure 7.2: Representation of massing elements. A type hierarchy fragment showing their inheritance from the type *du* and the introduced features. The asterisk symbol marks features that are introduced on that type.

```
(ENTITY˙A FU : massing˙a) &
(ENTITY˙A FU == ENTITY˙B FU)
```

Figure 7.3: A problem statement in the description language of KRYOS.

wishes to explore massings with the following constraints on the problem domain, the feature path ENTITY_A FU must be of type *massing_a* and that the former path must be structure-shared with the path ENTITY_B FU. This statement of the problem is analogous to the query,

What are the possible massing configurations in design space that satisfy the constraints on their features ?

The designer can compose such queries and pass it to KRYOS through the *FOLDS* interface. If the description is well-formed, it is parsed by KRYOS into a feature node with the label, DescNode. The successful parse of a description is loaded into *FOLDS* as a PState. This parsed description is then avaialable for exploration by designer.

Consider the more detailed example shown in Table 7.1. Here, the problem to be explored is encoded as a collection of path descriptions. Each path is connected by a conjunction, denoted by the symbol &. Such a PState represents the initial element of an exploration path. The conversion of this statement to a SState is enabled by user interaction. The domain layer construct SState represents the notion of a partial design solution in *FOLDS*. In it, are embedded the symbol substrate concepts of description, the satisfaction of a description as satisfiers and the trace of intermediate solutions as partial satisfiers. Since problems and solutions are composed as feature nodes in the designer's view, this process is explained in the next section.

```
(SFC'HOUSE GEOM POS : point &
      SFC'HOUSE GEOM COMMAND  : command) &
(        FU      : fu
&        DU      : du
&        DU DU'LABEL == FU FU'LABEL
&        DU GEOM COMMAND == SFC'HOUSE GEOM COMMAND
&        PORCH SFC'PORCH : du
&        ROOMROW ENTITY'A DU GEOM == SFC'HOUSE GEOM
&        ROOMROW ENTITY'B DU GEOM == DU GEOM)
```

Table 7.1: A collection of path equations in the form of a conjunctive description.

7.2.2 FNode and SatSpace

The PState is a compiled representation corresponding to the above description and is displayed in the interface through a feature node FNode with the label, *Satspace Element* as shown in Figure 7.4. Thus, a problem state, PState is the initial representation of a problem. It appears to the user in the *FOLDS* interface as the first element with the label, *Satspace Element.* It is on this structure that the process of modification and reformulation of problems occurs. Once a TypeSystem and a DescNode are successfully parsed, the designer regains control of the dialogue and can either construct a SatSpace and continue with the next state of exploration or formulate additional problems.

A *Satspace Element* labels a single exploration state, the FNode. An FNode composes a PState and SState. The first element of a FNode is a PState as shown in Figure 7.4. By interaction with

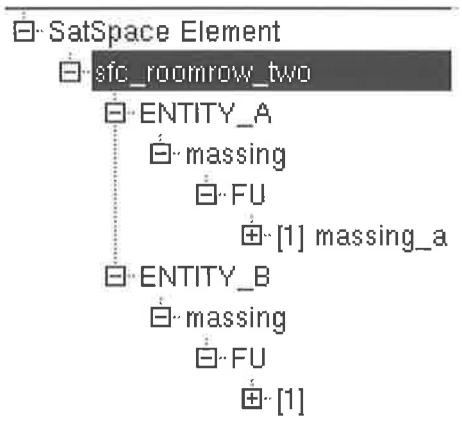


Figure 7.4: The label **SatSpaceElement** represents the feature node corresponding to the **PState**. The figure shows a **SState** labelled by the type *sfc_roomrow_two* uncovered in the exploration of the problem.

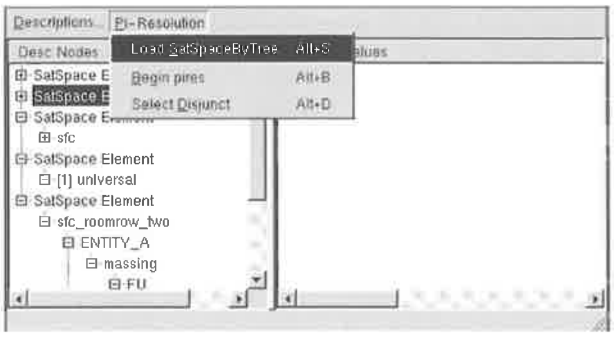


Figure 7.5: The **FNode** with the label *Satspace Element* can be loaded into the interface after parsing the query description. Each element represents the most general satisfier of a query, and can be subject to interactive unfolding from this point upwards.

this element, the designer can either modify (reformulate) the PState or generate a new problem state. The rest of the elements of a FNode are entities representing SState nodes. These are the partial satisfiers (solution states) of the FNode. By interaction with these elements, the designer can unfold the possible solution states of the problem. The elements composed by label *Satspace Element* become the subject of mixed-initiative dialogue with KRYOS via π-resolution and other movement operations.

The SatSpace is the largest and most complex object in the domain layer. The technical details and its relation to the formal substrate of subsumption ordered design space are given in Section 4.5 and in Appendix A. The design space descriptions are written as ASCII text files[1] and accessible through project names. The interface module of $\mathcal{FOLDS}$ is described in Section D. This interface provides the interface hooks to initialise the Kryos system, load in project definitions, construct the design space and begin the process of exploration. This module provides the interface for loading a TypeSystem. Construction of feature nodes is mediated by an iterator-like object, i.e., a SatSpacePath. Such an object is a path of computations from the statement of the problem, PState to a partial satisfier. However, like all generative processes, a starting condition is necessary. In $\mathcal{FOLDS}$, exploration begins with an *empty path*, the path from the problem state PState to its trivial conversion to a partial satisfier, PSat. Given the empty path, the user can explore the partial satisfiers on this path by iterating through the possible elements of this path using the exploration operations. The label SatSpace Element provides a hook to its SatSpace. So given an exploration path, the user can explore element on this path by interaction.

In order to advance the exploration, the designer performs an incremental π-resolution operation on this structure. The substructures corresponding to the results of the π-resolution operation are returned by the formalism as feature nodes. The designer then selects a substructure, SatNode to the node at PORCH. This node can be selected from the feature-value graph shown in Figure 7.2.2 or from the massing elements shown in Figure 7.15. If the designer picked a type t from the list of legal *type_resolutions* shown in Figure 7.9, then the exploration is advanced to the next element in the satisfier space.

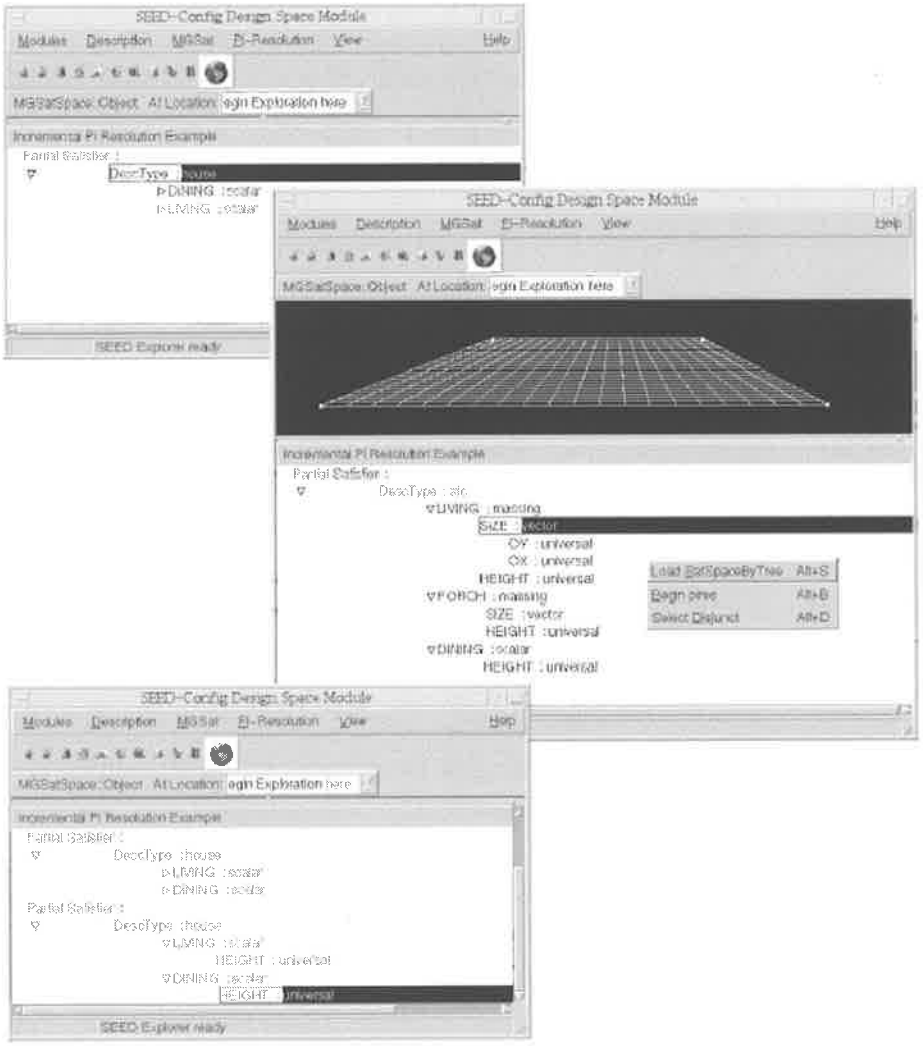
7.3 Dialogue in exploration

7.3.1 Unfolding visual feature nodes

As identified in Chapter 4, the visual feature node provides a sound representation for the communication and coordination of mixed-initiative dialogue. Two partial satisfiers of a query description from KRYOS and their interpretation as visual feature nodes in $\mathcal{FOLDS}$ are shown in Figure 7.7.

Interaction with the visual feature node involves three possible behaviours. First, the feature

[1]An example of these files are given in Appendix C.


SEED-Config Design Space Module
Modules Description MGSat PI-Resolution View Help
MGSatSpace::Object At Location: egin Exploration here
Incremental PI Resolution Example
SEED Explorer ready
Load SatSpaceByTree Alt+S
Begin pine Alt+B
Select Disjunct Alt+D

```
Partial Satisfier :
▽          DescType :house
                ▷DINING :scalar
                ▷LIVING :scalar
```

```
Partial Satisfier :
▽          DescType :sic
                ▽LIVING :massing
                     SIZE :vector
                          OY :universal
                          OX :universal
                     HEIGHT :universal
                ▽PORCH :massing
                     SIZE :vector
                     HEIGHT :universal
                ▽DINING :scalar
                     HEIGHT :universal
```

```
Partial Satisfier :
▽          DescType :house
                ▷LIVING :scalar
                ▷DINING :scalar
Partial Satisfier :
▽          DescType :house
                ▽LIVING :scalar
                          HEIGHT :universal
                ▽DINING :scalar
                          HEIGHT :universal
```

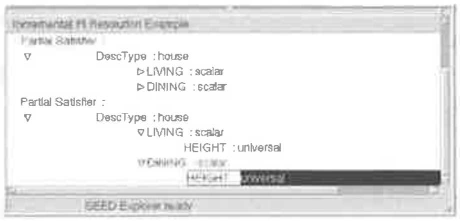
Figure 7.6: Implementation of interactive unfolding of a *feature-value map* using the visual representation of a feature node.

$${}_{\textit{Partial Satisfier}}\left[\textsc{DescType}: {}_{\textit{house}}\left[\begin{array}{ll}\text{LIVING}: & \text{scalar}\\ \text{DINING}: & \text{scalar}\end{array}\right]\right]$$

(a)

$${}_{\textit{Partial Satisfier}}\left[\textsc{DescType}: {}_{\textit{house}}\left[\begin{array}{ll}\text{LIVING}: & {}_{\textit{scalar}}\left[\text{HEIGHT}: \quad \text{universal}\right]\\ \text{DINING}: & {}_{\textit{scalar}}\left[\text{HEIGHT}: \quad \text{universal}\right]\end{array}\right]\right]$$

(b)



```
Incremental FI Resolution Example
Partial Satisfier :
 ∇            DescType  : house
                   ▷LIVING  : scalar
                   ▷DINING  : scalar
Partial Satisfier :
 ∇            DescType  : house
                   ∇LIVING  : scalar
                           HEIGHT  : universal
                   ∇DINING  : scalar
                           HEIGHT  universal
SEED Explorer ready
```

(c)

Figure 7.7: Implementation of a visual feature node in $\mathcal{FOLDS}$. The notation in (a) and (b), are implemented as the interactive nodes shown in (c).

node can be unfolded into its constituent subparts following the standard interaction and its values. Second, an unfolding operation can be applied to a feature-value pair. Third, the unification of a type with a feature node can be specified. An example of interactive node unfolding in $\mathcal{FOLDS}$ is shown in Figure 7.2.2. In this example, a description query returns a partial satisfier of type *sfc* comprising three feature-value pairs, LIVING, PORCH, DINING and their most general substructure nodes. The elements can be expanded and imploded using the triangular arrows, while the selected *feature-value pair*, SIZE of type *vector* can be subject to exploration through mixed-initiative.

7.3.2 Implementation

Mixed-initiative unfolding between the designer and the explorer is implemented in $\mathcal{FOLDS}$ using the signal/slot protocol for communication between the designer and the generative system. This protocol is a straight forward adaptation of the message passing system, *signal-and-slots* [Dalheimer 1999] available in QT[2]. This communication method is combined with the visual feature node interaction to form the basis for the interactive path operations in $\mathcal{FOLDS}$. The signal/slot mechanism, used for bi-directional communication between interface operations on a FNode and KRYOS, is shown in figure 7.8.

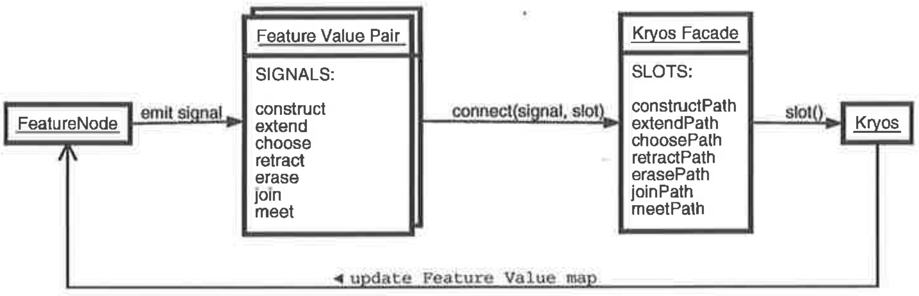


Figure 7.8: Interactive node operations in $\mathcal{FOLDS}$ use the signal/slot mechanism for communication with KRYOS. User interaction on a node, emits signals to the selected *feature-value pair*. Each *feature-value pair* signal is connected to a slot that communicates with KRYOS. The results of the operation update the initial feature node with the new feature value pair.

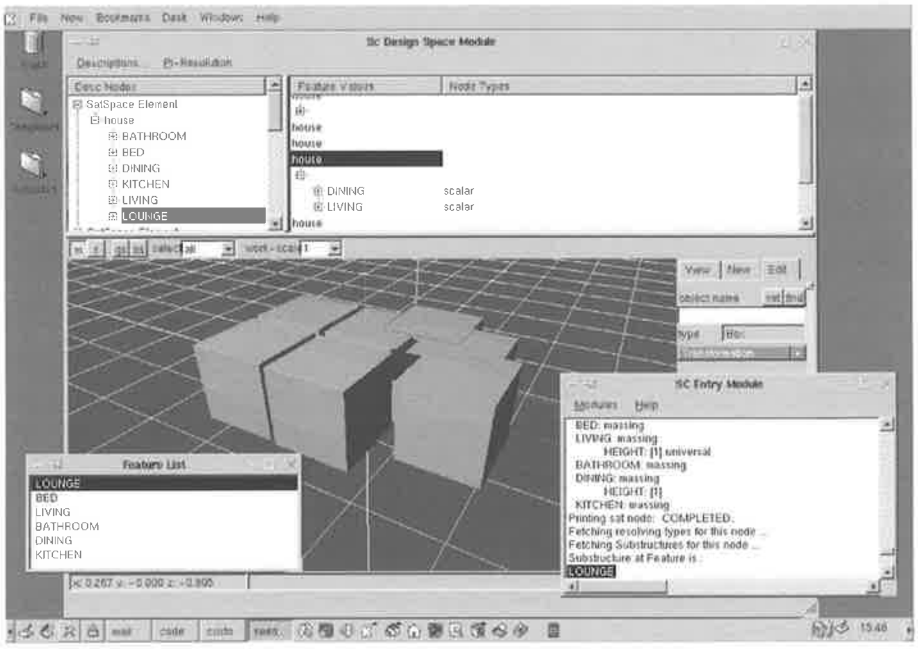
A *signal* is a type of message that is emitted when a particular event occurs, either initiated by the designer at the interface or by the generative system. A *slot* is a function that is called in response to a particular signal. Signals and slots are coupled together using the *connect*(*signal*, *slot*). If the signal is interesting to two or more objects, the *connect* function can couple the signal to slots in all the objects. Manipulation of a *feature-value pair* results in a signal. When a signal is emitted, the slots connected to it are executed immediately in KRYOS. The results of the operation are then

[2]An elegant notification mechanism for communication between software components is available in the Qt distribution based on signals and slots. Signals are emitted by objects when they change their state in a way that may be interesting to the outside world. Slots are normal member functions used for receiving signals. A single slot can receive multiple signals and a signal can be broadcast to multiple slots.

returned to the feature node as an updated *feature-value map*. Technical details of the signal-slot mechanism and dialogue through visual feature nodes are described in Appendix D. A deeper understanding of dialogue awaits the discussion of interaction with the exploration operations in the next section.

7.4 Tasks of exploration

The exploration operations enable the *generation, navigation* and *synchronisation* of visual feature nodes in a satisfier space, SatSpace of candidate massing configurations.



```
 BED: massing
 LIVING: massing
       HEIGHT: [1] universal
 BATHROOM: massing
 DINING: massing
       HEIGHT: [1]
 KITCHEN: massing
Printing sat node:  COMPLETED.
Fetching resolving types for this node ...
Fetching Substructures for this node ...
Substructure at Feature is :
LOUNGE
```

Figure 7.9: Interactive node unfolding in $\mathcal{FOLDS}$ showing in the DesignSpace Module.

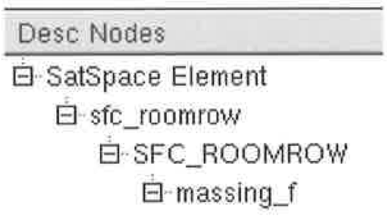
The Design Space module shown in Figure 7.9 is the main interface to exploration operations in $\mathcal{FOLDS}$. Its comprises a graphical view of the feature value nodes. The feature nodes are shown in first level to the left and the substructure nodes are shown in the right. During interaction with the movement operations, type resolutions or disjunctive points are shown in smaller pop up widgets. The central portion of the module displays the massing designs graphically. Camera, shading and transformation controls are provided for viewing massing elements. The console linked to the unfolding process provides feedback during exploration.

The functionality of $\mathcal{FOLDS}$ in supporting exploration tasks through mixed-initiative is described here. First, the process of feature node construction describes the problem formulation and extension process through interaction with the CONSTRUCT and EXTEND operations. Second, the navigation of the feature node using the CHOICE and RETRACT operations are described.

Finally, the synchronisation of feature nodes in satisfier space, supported through interaction with the RECALL, ERASE, JOIN and MEET operations is discussed.

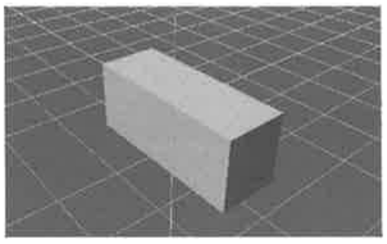
7.4.1 Extending feature nodes

Recall that feature node navigation is defined in Section 6.3 as the incremental *movement* from the rooted *partial satisfier* to another along a defined path. The exploration path is advanced by small-grained, monotone operations over partial satisfiers. As for all paths, for each element in the exploration path of massing designs of single fronted cottages, there exists a feature graph corresponding to its contents. Exploration is advanced by navigation operations on the feature-value graph of the satisfier, PSat. The operations available for unfolding a given visual feature node are CHOOSE and RETRACT.



Desc Nodes
- SatSpace Element
  - sfc_roomrow
    - SFC_ROOMROW
      - massing_f

(a)

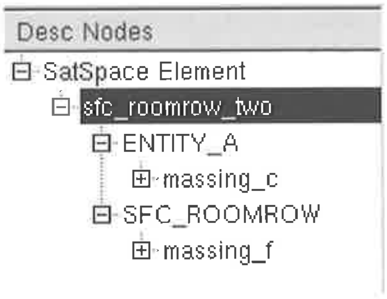


(b)

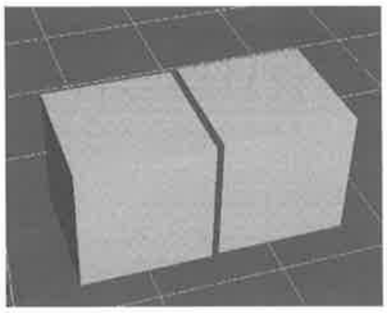
Figure 7.10: The description query, (SFC_ROOMROW : *massing_f*), its corresponding PState, labelled as a *DescNode* (a) and massing representation of type, *sfc_roomrow*, (b).

The EXTEND operation is used to unfold the implicature of a conjunctive description query. The designer can simultaneously perceive visual change in the substructures of the developing exploration path and trigger changes in the functional feature node representation. Both forms are indices to the underlying design space of massing solutions. As an example, consider the initial query shown in Figure 7.10 for massing exploration.

The designer instantiates the feature SFC_ROOMROW and constrains it to be of type, *massing_f*. Using the CONSTRUCT operation described in the previous section, its corresponding FNode is provided for user interaction. The user instantiates the feature to the massing geometry, and KRYOS infers its type to be at least as specific as *sfc_roomrow*. From this starting point, the EXTEND operation is used to extend the thread specified by the type *sfc_roomrow*. The thread of exploration shown in Figure 7.10 can now be extended by user interaction. In Figure 7.11, the initial query is extended by adding the feature, ENTITY_A and committing the massing type, *massing_c* as its value. The designer is then returned the visual feature node corresponding to the advance in the computation associated with the extension of the query. Figure 7.12 shows the introduction of the feature, ENTITY_C and committing the massing type, *massing_d* as its value.



(a)

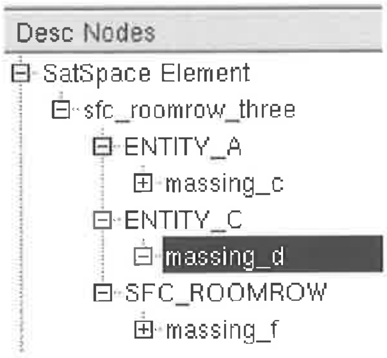


(b)

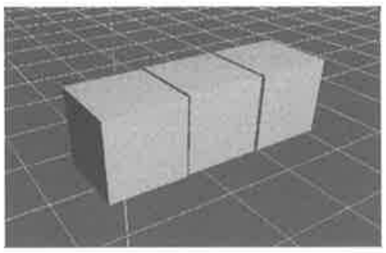
Figure 7.11: The initial query is extended by introducing the feature, ENTITY_A with value of type, *massing_c*. Its corresponding *DescNode* is shown in (a) and the massing representation is extended to be of type, *sfc_roomrow_two*, shown in (b).

The mixed-initiative dialogue enables the designer to incrementally extend the computation path until no further exploration through EXTEND is possible.

Note that though exploration begins at a point in the satisfier space specified by the type *sfc_roomrow*, it is possible to move to other points in the satisfier space. From this point, it is possible to direct the exploration through either query reformulation, or substructure extension. Through mixed-initiative, these possibilities enable richer forms of interaction to explore the motivating example of single-fronted cottage massing configurations.



(a)



(b)

Figure 7.12: The exploration is advanced by introducing feature, ENTITY_C with value of type, *massing_d*. Its corresponding *DescNode* is shown in (a) and the massing representation is extended to be of type, *sfc_roomrow_three*, shown in (b).

To evaluate the range of interactions further, consider the state of the interface shown in Figure 7.13. The visual feature node generated in its attribute-value matrix notation of type *sfc* is as follows :

```
sfc
DINING: massing
        HEIGHT: [3] integer
        SIZE: vector
             OX: [4] integer
             OY: [4]
LIVING: massing
        HEIGHT: [3]
        SIZE: vector
              OX: [5] integer
              OY: [5]
PORCH: massing
       HEIGHT: [3]
       SIZE: vector
            OX: [5]
            OY: [5]
```

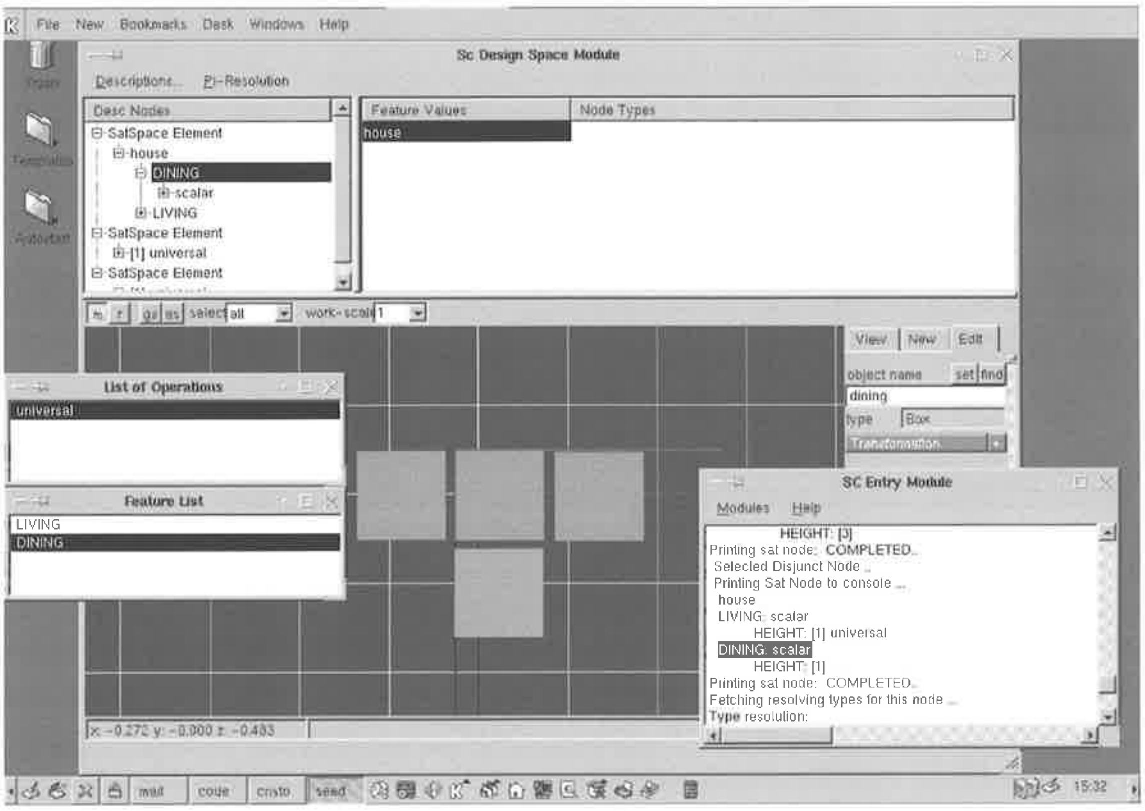
In this example, the **PState** comprises three features LIVING, DINING, PORCH and their massing values. Note the co-reference tags that indicate the heights of each of the geometries representing the type *massing* are structure shared. Further note that the sizes, here denoting the layout dimensions, are also the same. Given these constraints, the massing explorations generated from this problem statement will maintain the type constraints associated with the type *sfc*.

Of specific interest is the fact that the designer is not restricted to exploration within these constraints. The designer can return to a previous **PState** of the current exploration and change the constraints to generate a new set of partial satisfiers. Further, the designer can modify a generated partial solution of type **SState** and use this entity as a starting point for another cycle of generation. Thus, the task of exploration through mixed-initiative allows the designer to modify, change and retract not only the initial **PState**, but also use the generated collection of partial satisfiers to advance the exploration.

While the distinction between problems and solutions is maintained in the domain layer, the nature of the formalism permits interchange between these distinctions. Thus, mixed-initiative enables the freedom to reformulate and change the nature of generation without losing the advantages of the underlying order structure. The next section shows how the CHOICE operation permits even more freedom in exploration in the case of disjunctive problem statements.

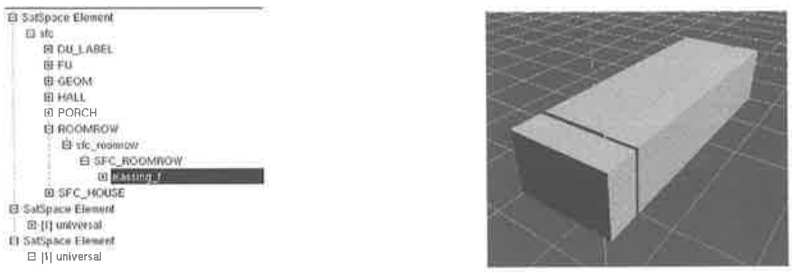
7.4.2 Choice in exploration

Disjunctive problem statements comprise the composition of queries as explained above. However, the disjunction operator, "|" provides natural choice points to the selection of solutions in $\mathcal{FOLDS}$. As an example, consider the problem shown in Table 7.2.



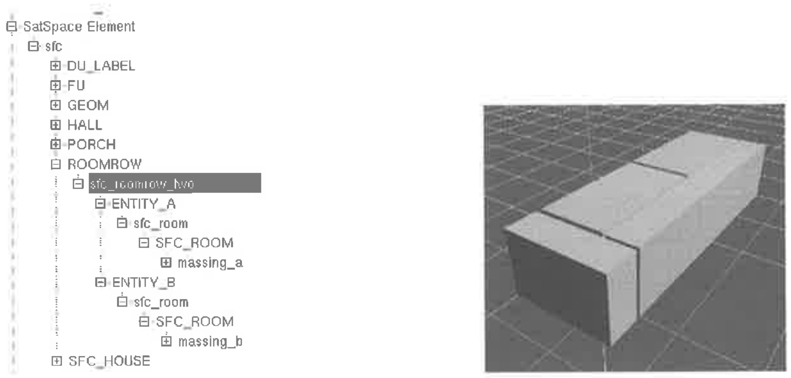
```
          HEIGHT: [3]
Printing sat node:  COMPLETED..
 Selected Disjunct Node ..
 Printing Sat Node to console ...
  house
  LIVING: scalar
        HEIGHT: [1] universal
  DINING: scalar
        HEIGHT: [1]
Printing sat node:  COMPLETED..
Fetching resolving types for this node ...
Type resolution:
```

Figure 7.13: An EXTEND operation in $\mathcal{FOLDS}$. Note that a description node of type *universal* is introduced as a massing element. This element is extended by interaction to be of type *house.* A list of features, DINING, LIVING of type *scalar* and the constraints on them (both share the value of the feature HEIGHT) are incorporated into the satisfier. These features are assigned type *scalar* as this type represents the most general or least specific resolution of the features under the current problem statement.



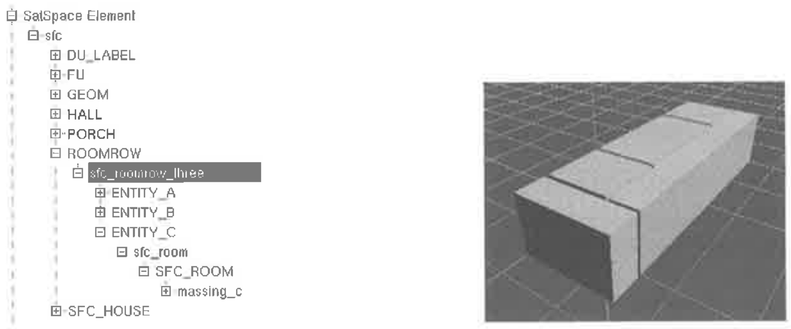
```
SatSpace Element
  sfc
    DU_LABEL
    FU
    GEOM
    HALL
    PORCH
    ROOMROW
      sfc_roomrow
        SFC_ROOMROW
          massing_f
    SFC_HOUSE
SatSpace Element
  [1] universal
SatSpace Element
  [1] universal
```

(a)



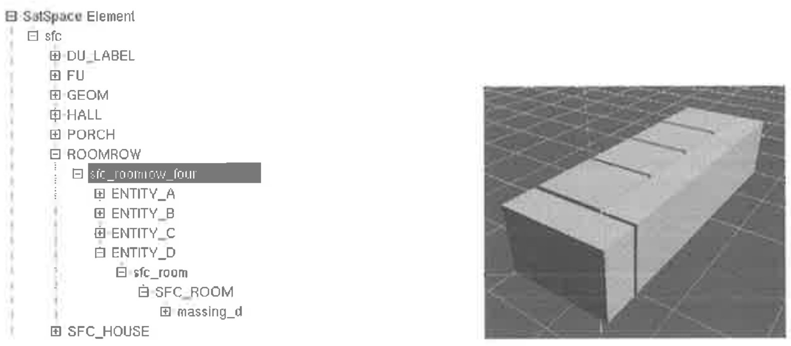
```
SatSpace Element
  sfc
    DU_LABEL
    FU
    GEOM
    HALL
    PORCH
    ROOMROW
      sfc_roomrow_two
        ENTITY_A
          sfc_room
            SFC_ROOM
              massing_a
        ENTITY_B
          sfc_room
            SFC_ROOM
              massing_b
    SFC_HOUSE
```

(b)



```
SatSpace Element
  sfc
    DU_LABEL
    FU
    GEOM
    HALL
    PORCH
    ROOMROW
      sfc_roomrow_three
        ENTITY_A
        ENTITY_B
        ENTITY_C
          sfc_room
            SFC_ROOM
              massing_c
    SFC_HOUSE
```

(c)



```
SatSpace Element
  sfc
    DU_LABEL
    FU
    GEOM
    HALL
    PORCH
    ROOMROW
      sfc_roomrow_four
        ENTITY_A
        ENTITY_B
        ENTITY_C
        ENTITY_D
          sfc_room
            SFC_ROOM
              massing_d
    SFC_HOUSE
```

(d)

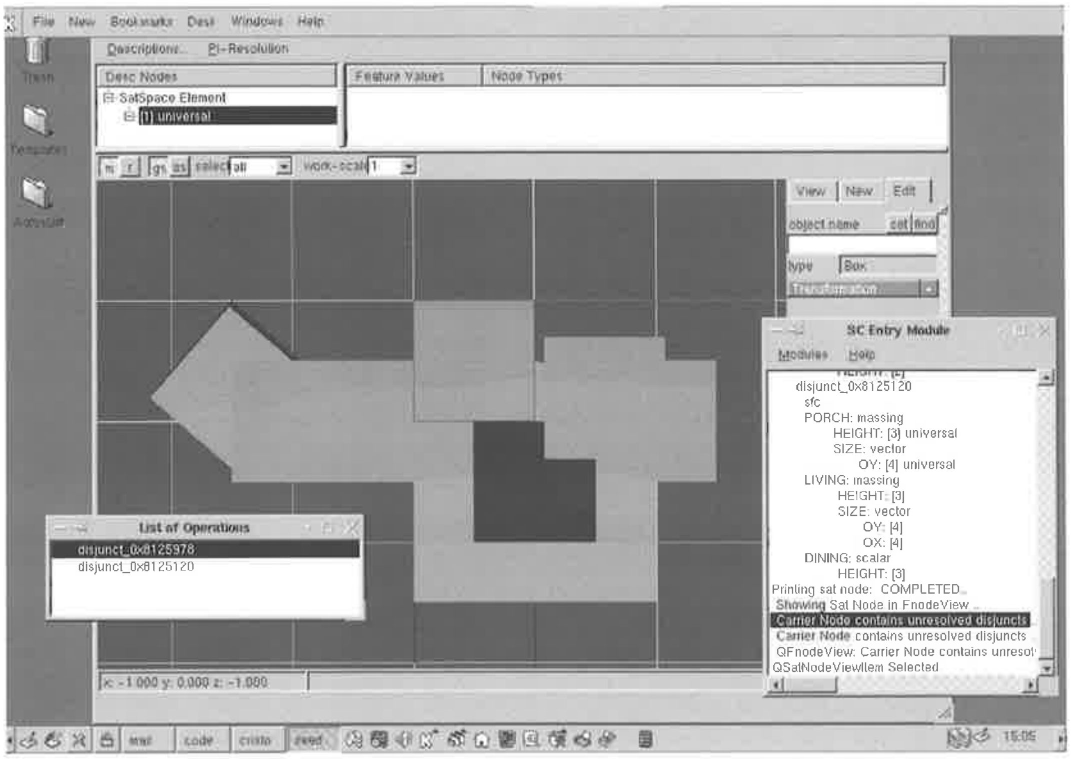
Figure 7.14: The exploration of a disjunctive query description. The values of the conjuncts within the disjunct are shown in the visual feature node representation in the left. Their corresponding design states with massing representations and type extensions are shown in the right.

```
(SFC'HOUSE GEOM POS : point &                                        1
      SFC'HOUSE GEOM COMMAND  : command) &                           2
      (      DU DU'LABEL == FU FU'LABEL                              3
      &       DU GEOM COMMAND == SFC'HOUSE GEOM COMMAND              4
      &       PORCH SFC'PORCH : massing'a                            5
      &       PORCH SFC'PORCH MASSEL'A DU GEOM == DU GEOM) &         6
                                                                     7
      (      SFC'HOUSE ROOMROW ENTITY'A MASSEL'A == SFC'HOUSE GEOM ) —   8
                                                                     9
      (      SFC'HOUSE ROOMROW ENTITY'A DU GEOM == SFC'HOUSE GEOM    10
      &       SFC'HOUSE ROOMROW ENTITY'B DU GEOM == DU GEOM ) —      11
                                                                     12
      (      SFC'HOUSE ROOMROW ENTITY'A DU GEOM == SFC'HOUSE GEOM    13
      &       SFC'HOUSE ROOMROW ENTITY'B DU GEOM == DU GEOM          14
      &       SFC'HOUSE ROOMROW ENTITY'C DU GEOM == DU GEOM ) —      15
                                                                     16
      (      ROOMROW ENTITY'A MASSEL'A == SFC'HOUSE GEOM             17
      &       SFC'HOUSE ROOMROW ENTITY'B DU GEOM == DU GEOM          18
      &       SFC'HOUSE ROOMROW ENTITY'C DU GEOM == DU GEOM          19
      &       SFC'HOUSE ROOMROW ENTITY'D DU GEOM == DU GEOM)         20
```

Table 7.2: Path descriptions from the massing example presented as a collection of disjunctive descriptions. Disjunctions signify non-deterministic choice points in the satisfier spaces. The designer constrains the massing exploration to the paths shown above and provides the constraints on features through structure sharing and path equality.

In the above PState, the feature SFC_HOUSE is constrained by two path equations. Line #1 of Table 7.2, constrains the exploration path SFC_HOUSE GEOM POS to be of type *point*. This is analogous to the natural language statement,
The position of the geometries of this type must be given as a point in the geometry window.

Thus any geometry that satisfies the feature SFC_HOUSE is associated with a *point*, provided by the designer through the geometry interface. Line #2 states that the exploration path, SFC_HOUSE GEOM COMMAND is constrained to be a command type of *command*. The path SFC_HOUSE PORCH has the type *massing_a* and its extension path MASSEL_A DU GEOM is structure shared with the current path DU GEOM. The next part of the query is a collection of four disjunctions describing the constraints on the entities that are introduced in the path SFC_HOUSE ROOMROW. Line # 7, 10, 13, 17 in Table 7.2 equate the structure at the end of path SFC_HOUSE ROOMROW ENTITY_A MASSEL_A with the current structure DU GEOM. The rest of the query constrains the formalism to generate alternate arrangements for the value of ROOMROW comprising one, two, three or four massings. The disjunctive choice points are maintained within the SatSpaceEl representation, such that they can be displayed in a graphical interface for user intervention as shown in Figure 7.15. In this figure, two possible resolution paths exist. The user must choose between the two disjuncts in order to resolve the non-determinism.



```
disjunct_0x8125978
disjunct_0x8125120
```

```
disjunct_0x8125120
  sfc
  PORCH: massing
        HEIGHT: [3] universal
        SIZE: vector
              OY: [4] universal
  LIVING: massing
         HEIGHT: [3]
         SIZE: vector
               OY: [4]
               OX: [4]
  DINING: scalar
         HEIGHT: [3]
Printing sat node:  COMPLETED
 Showing Sat Node in FnodeView
 Carrier Node contains unresolved disjuncts
 Carrier Node contains unresolved disjuncts
 QFnodeView: Carrier Node contains unresol
QSatNodeViewItem Selected
x: -1.000 y: 0.000 z: -1.000
```

Figure 7.15: Interface to resolving disjunctions with the CHOICE operation in exploration. Disjunctive problems present the user to with the alternative possiblities to resolve.

In the example above, mixed-initiative in $\mathcal{FOLDS}$ presents a natural way to resolve the non-determinism arising out of the resolution of disjunctive problems, that is, problems with many alternative arrangements. This is necessarily an explanatory example: the scalability of the proposition remains to be tested in very large sets of disjunctions. One solution to very large sets of disjunction is foreshadowed in the discussion on incrementality in Section 2.4.1.

7.4.3 Retract

As described in Section 6.3 on Page 96, the RETRACT operation enables the exploration to move to *any* previous *feature-value map* available in the path sequence. In the case of the massing example, path retraction provides the means to move from a more detailed feature value to a less detailed one. Consider the exploration of a massing model of a house as shown in Figure 7.12. The designer begins with an overall massing model of the house of type *sfc*, creates an exploration fragment to detail the ROOMROW to the type, *sfc_roomrow_three*. At this point in the exploration, it may be necessary to abstract away some of the exploration details and redirect the exploration along a different path. The RETRACT operation enables a designer to reverse the effects of the most recent EXTEND operation on the current node. Further, the RETRACT operation is also the exploration equivalent of the conventional model of undo, which is the action of backing up the current path to the previous state. The "path sequence" in the design space is the set of feature nodes that are more abstract (less specific in terms of information) than the current node.

Using this operation, the designer can then return to change the exploration of the massing model, and modify aspects of the earlier ROOMROW exploration and retract the massing to the type *sfc_roomrow*, which is a less specific type. This act of moving to less specific states during exploration is illustrated in Figure 7.16.

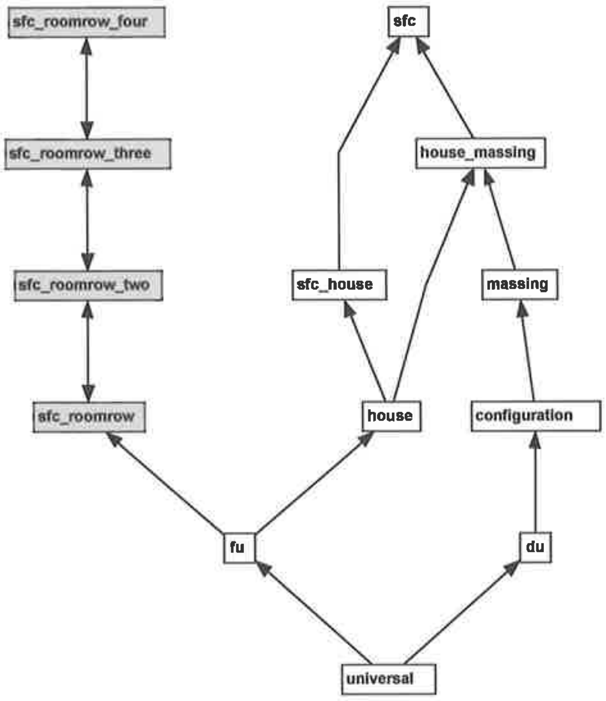


Figure 7.16: Path retraction enables the designer to move from the currently selected substructure to the set of states that are strictly less specific in terms of information than the current substructure. The feature node at the end of the path SFC_HOUSE ROOMROW can be retracted along the types *sfc_roomrow_three, sfc_roomrow_two, sfc_roomrow.*

Mixed-initiative enables the explorer to perform a RETRACT operation on the substructure node, denoted by the feature path SFC_HOUSE ROOMROW in the partial satisfier. This operation allows the user to move back along the current path sequence at the finest level of granularity, that is, an incremental π-resolution step. Thus, the massing entities denoted by the types *sfc_roomrow_two, sfc_roomrow_three, sfc_roomrow_four* are available to the designer. The designer can choose to move backwards along this path using the RETRACT operation to move to the less specific states denoted by the types, *sfc_roomrow, sfc_roomrow_two, sfc_roomrow.*

7.4.4 Reuse of past exploration

In the massing example thus far, exploration moves along a single thread of exploration are described. As discussed in Section 6.4, synchronisation operations provide the designer with the ability to integrate two distinct threads of exploration in the design space into a unified state.

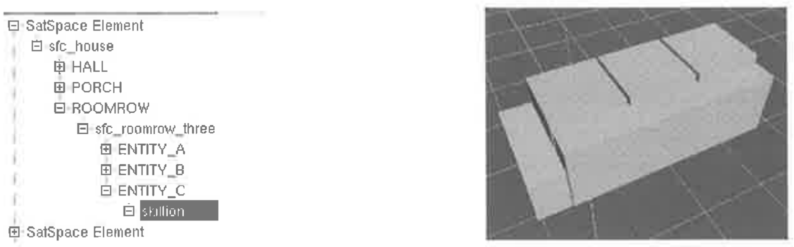
The RECALL operation allows the designer to access the problems and solutions of past explorations which have been stored in a database of exploration histories for the given design project. In the case of the massing example discussed above, note the descriptions in Figure 7.17. They represent distinct queries over the massing type system undertaken in different sessions. It is possible, using the RECALL operation to query the design space, seeking to find exploration paths that might satisfy the specification for the current path SFC_HOUSE ROOMROW ENTITY_C SKILLION. Using unification, mixed-initiative dialogue can incorporate past exploration paths into the current path of exploration. Since the monotone operations based on incremental π-resolution maintain information integrity and consistency in the space, it is possible to engage the RECALL operation to recover the results of previous explorations. In another thread of exploration, the SKILLION is elaborated in the exploration of type *house.* The RECALL operation is used to extend the path ROOMROW ENTITY_C to a skillion. The result of the recall is shown in Figure 7.17.

The RECALL operation can operate on two levels. First, the recall of paths corresponding to the *type* of the curent node, which acts as an index for recall. The use of a feature structure as an index to a collection of design cases is covered in [Woodbury et al. 1999]. Second, this operation deals with tracing the exploration *history* of a current node in the design space. This corresponds to a notion of object evolution and design history of the current node. This operation allows the dialogue layer to enable extended interactions across and between sessions, between different parts of the design space and between different designers.

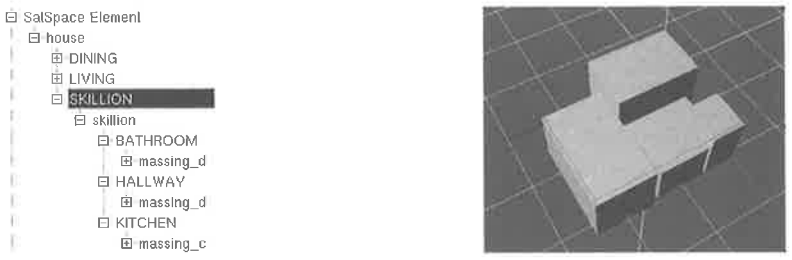
7.4.5 Erasure

The definition of information removal, or *erasure* [Woodbury et al. 2000] is a novel backward navigation technique [Woodbury et al. 2000]. The ERASE operation conceptually extends simple retraction as shown in Figure 7.16. The ERASE operation allows the designer to perform non-destructive undo, destructive deletion and suppression of features in the design space.

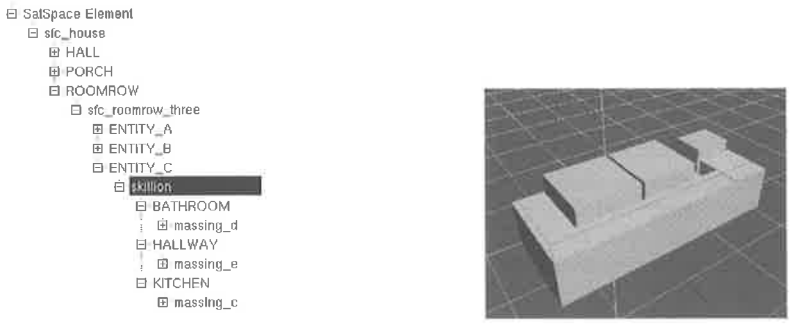
Figure 7.18 shows how the ERASE operation is used in massing exploration to set the value of the



(a)

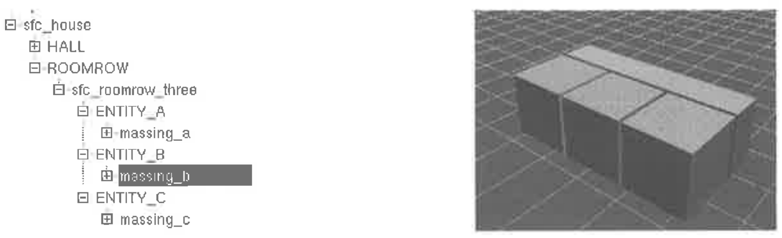


(b)



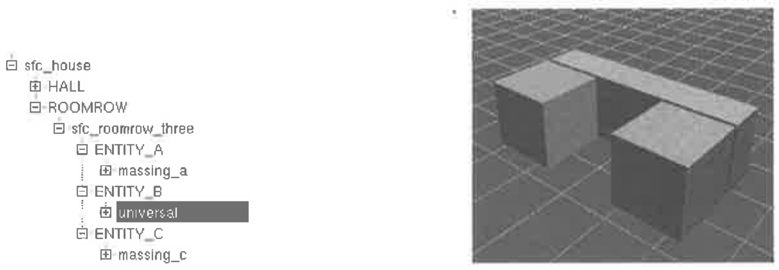
(c)

Figure 7.17: Using the RECALL operation. The exploration of type *sfc_house* is developed with the features PORCH, HALLWAY, ROOMROW shown in (a). In another thread of exploration, the SKILLION is elaborated in the exploration of type *house* shown in (b). The RECALL operation is used to extend the path featroomrow entity_c to a skillion by combining (a) and (b). The result of the recall is shown (c).



```
⊟ sfc_house
  ⊞ HALL
  ⊟ ROOMROW
    ⊟ sfc_roomrow_three
      ⊟ ENTITY_A
        ⊞ massing_a
      ⊟ ENTITY_B
        ⊞ massing_b
      ⊟ ENTITY_C
        ⊞ massing_c
```

(a)



```
⊟ sfc_house
  ⊞ HALL
  ⊟ ROOMROW
    ⊟ sfc_roomrow_three
      ⊟ ENTITY_A
        ⊞ massing_a
      ⊟ ENTITY_B
        ⊞ universal
      ⊟ ENTITY_C
        ⊞ massing_c
```

(b)

Figure 7.18: The ERASE operation allows the designer to remove massing entities from a configuration. The exploration of type *sfc_house* is developed with the features PORCH, HALLWAY, ROOMROW shown in (a). The massing entity corresponding to the ENTITY_B is deleted from the composition leaving the massing and corresponding feature node shown in (b).

feature path ROOMROW ENTITY_B to the type *universal* to suppress its effect on the massing. The intensional nature of the feature nodes is apparent in this operation, as erasure is achieved by feature *suppression* rather than removal. The only difference is that the new states are are conceptually less specific than the feature nodes from which they are computed. An implementation of erasure in the general case as outlined in Section 2.4.3 awaits further research.

7.4.6 Joins and Meets

Since unification is defined with respect to the ordered collection of feature structures as a *least upper bound*, it is also possible to conceive of unification as a formal means for combining two partial designs signified as exploration paths in a coherent, single exploration path. The JOIN operation provides a means for computing the *unification* of two distinct exploration paths. In the implementation of $\mathcal{FOLDS}$, JOIN failed consistently to result in a least upper bound. While JOIN remains a valid synchronisation operation, it requires stringent type compatibility in order to succeed. The unification of partial designs through JOIN requires further work, both at the level of the substrate as well as in its implementation in $\mathcal{FOLDS}$. The MEET operation corresponds to computing the *subsumption* of two exploration paths.

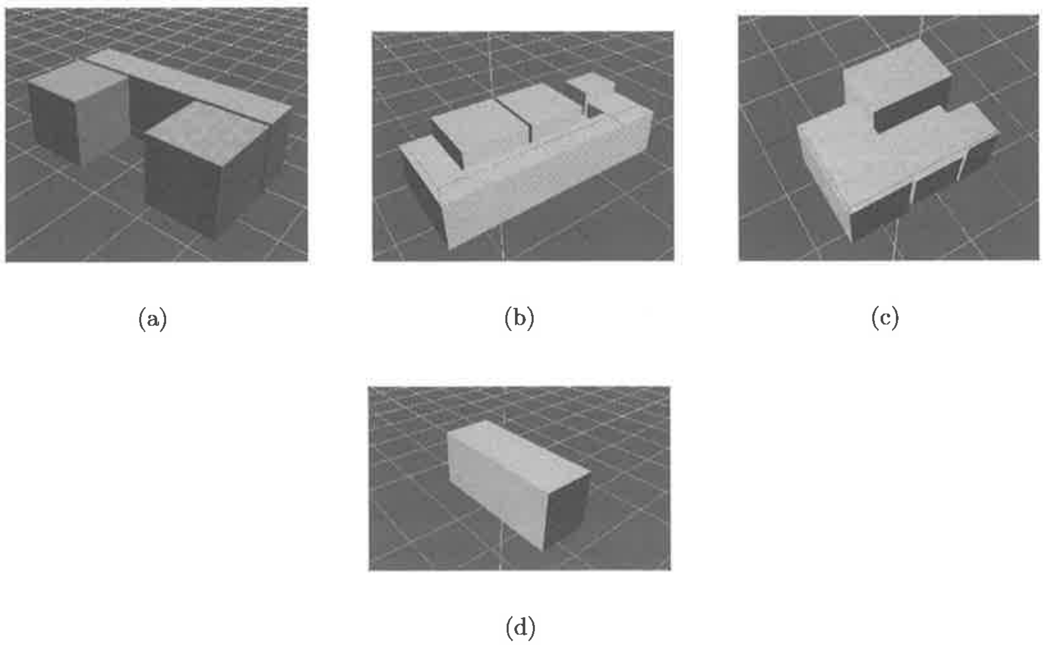

(a) (b) (c) (d)

Figure 7.19: The MEET operation allows the designer to explore massing entities from a configuration through synchronisation of multiple paths of exploration. The MEET of the configurations shown in (a), (b) and (c) is the configuration of type *sfc* shown in (d).

Figure 7.19 depicts massing states structures in a subsumption relation. The example represents the decomposition of a massing design using the MEET operation. The upper nodes denote building entities from separate exploration. The lower node denotes their respective reductions to single entity.

The MEET operation presents a sophisticated means for large scale information synchronisation. A user interacts with the MEET operator by selecting two paths, each from a different state as shown in Figure 7.19. The explorer then traverses the subsumption ordering and moves the computation to the meet of the selected states. The user can then choose to continue the exploration process as the result is a design state automatically placed somewhere in the subsumption ordering. If the two paths are the root nodes of states in the design space, anti-unification produces the most specific subsumer of those two states.

7.5 Discussion

From the implementation of $\mathcal{FOLDS}$ and an example demonstration in the exploration of massing designs for single-fronted cottages, it is now possible to discuss the findings arising out of the prototype implementation. The findings are as follows:

- A substantial effort is required to author a valid type system in terms of specifying types, features and constraints on types. In the current implementation, a type system is constructed manually with a text editor and this process remains error-prone. To make this process interactive, extending the domain layer to the authoring of type systems is necessary.

- The clear distinction between problems and solutions identified in the domain layer is blurred. In the example discussed above, following the SEED Knowledge level, functional decompositions and design solutions are specified in different parts of the type hierarchy. This is borne out by the example to be an unnecessary construction for design space exploration. As implemented in $\mathcal{FOLDS}$, problem formulation and solution generation are distinct processes. However, in the mixed-initiative environment, these categories are interchangeable. This is a reflection of real world designing, where solutions and problems remain intertwined in complex ways. Further, in the formal substrate, no distinctions are made between problems and solutions.

- Mixed-initiative permits the satisfier space to be independent from the design space. The real power of this independence is revealed in the making of choices and the recording of the history of exploration. A trail of exploration rationale is built up in the satisfier space and these are easily accessible to the designer. This permits the designer to experiment and explore with a range of problem statements, without losing the threads of intentional exploration. Mixed-initiative permits the complexity of design space (large subsumption ordered graph) to be encapsulated in satisfier space (folded intentional tree). This proved to be the most successful aspect of the implementation.

- A loss of generality comes with the tree formulation of the SatSpace. This is a trade off with respect to ease of use and implementation. This loss of generality in satisfier space means that the implications of redundant states through structure sharing in the case of non-trivial type hierarchies (the massing example has only 43 types) is not currently known and awaits a construction of larger automated type systems. However, this is not a problem in theory: given the subsumption ordering of design space, redundant states would be automatically detected[3].

- The dialogue layer construct of visual feature nodes is posited in Chapter 5 as a straight forward adaptation of feature structure notation. As borne out by the example implementation, it demonstrates the mixed-initiative nature of interaction but is a limiting case for exploration. The visual feature nodes generate and control the configuration of massing designs. The visualisation of these designs is achieved through a straight forward mapping into the three-dimensional viewer in $\mathcal{FOLDS}$. A more compelling interface metaphor, for example the direct manipulation of graphic entities, is necessary for seamless exploration. As far as possible, dialogue needs to be construed in terms of the language designers understand best, two and three dimensional visual representations, not in terms of symbol level abstractions and their visualisation. The lack of direct manipulation of massing entities to further exploration remains a limitation in $\mathcal{FOLDS}$.

- The tasks of exploration in $\mathcal{FOLDS}$ mirror the movement algorithms of the formalism. All of these are conceptual variations of information ordering and inference operations available at the symbol level. Currently, the implementation is truly mixed-initiative in three operations, EXTEND, CHOICE AND RETRACT. At the time of writing, ERASE is performed without mixed-initiative interaction. As explained in the example, the feature path is suppresed in terms of the formulation enunciated in Chapter 2, Section 2.4.3. The complexity of mixed-initiative erasure at the user level requires further research.

- The case of JOIN proved problematic to demonstrate through mixed-initiative. The theoretical soundness and completeness of this large grained synchronisation operation has been argued in the thesis (See for example, Section 2.4.4 and Section 6.4.3). However, the demonstration of its validity in the general case in $\mathcal{FOLDS}$ awaits further research into the nature of synchronisation from the perspective of implementation. The successful implementation and demonstration of JOIN requires careful and automated construction of the TypeSystem, in relation to type checking for computing compatibility of joins. In the manual example illustrated here, this proved difficult to achieve as types became specialised and incompatible fairly early in the process.

[3]This is foreshadowed in Woodbury et al. [2000] discussion of duplicate detection

- The case of MEET proved to be the only large scale operation demonstrated through mixed-initiative. This is attributed to the fact that the formalism consistently returns the *greatest lower bound*, which in the example is the simple massing of type *sfc_roomrow*. Reconstructing the intermediate nodes in the path to the greatest lower bound would make this operation more compelling as a metaphor for exploration.

- The $\mathcal{FOLDS}$ implementation opens a new area in terms of query and retrieval interfaces for recall operations. In the case of RECALL, simple substitution rather than inference is used in the current example due to the lack of persistence and storage mechanisms for exploration results. However, given a persistent data storage of past explorations, mixed-initiative is potentially an effective mechanism for search, match and retrieval of past explorations. Taken together, a type hierarchy editor, a visual description authoring environment and a repository of exploration results present a potentially powerful environment for developing a new form of systematic design reuse.

7.6 Summary

This chapter describes a prototype implementation, $\mathcal{FOLDS}$, of the mixed-initiative interaction model developed in Part II. The $\mathcal{FOLDS}$ prototype is implemented over the KRYOS, an implementation of the formal machinery of design space exploration. $\mathcal{FOLDS}$ demonstrates mixed-initiative interaction between the formalism and the designer. An example of mixed-initiative exploration in the domain of three dimensional configurations of massing designs is described. Through this example, the role of the mixed-initiative interaction model during exploration is demonstrated. In the next chapter, the conclusions of the study with reference to the hypotheses investigated, the development of the mixed-initiative interaction model, the prototype implementation and directions for future work are presented.

Chapter 8

Conclusions

This study investigates the role of mixed-initiative interaction in design space exploration. In this Chapter, the results of this investigation, the constraints on the results and directions for further work are described.

8.1 Mixed-initiative Exploration

The research hypotheses on the role of mixed-initiative in design space exploration, posed in Section 1.4 on page 19 can now be addressed.

8.1.1 Assumptions

The assumptions underlying the study are as follows:

1. *The process of computational exploration can be formally encoded with a design space description formalism.*
   The design space exploration formalism provides a sound basis for representing the concepts of state, move and structure. First, types, features and descriptions taken together provide well founded support for representing problem and solution states. The state representation supports the formal properties of intentionality, partialness, structure sharing, and cyclicity as discussed in Section 2.2.3. Second, the inference algorithms over this representation support a principled notion of exploration moves that enumerate partial states under a subsumption ordering. Moves generate new states, navigate and modify existing states. These moves are discussed in Section 2.4. Finally, underpinning the exploration of partial satisfiers is an ordering based on subsumption. In Chapter 2, a description formalism for representing the formal substrate comprising state, move and structure is described. The work on the formal substrate has been reported in Woodbury et al. [1999], Burrow & Woodbury [1999], Woodbury

et al. [2000] and in Chang's [1999] and Burrow's [2003] theses. The implementation of the formal theory of design space exploration is available in KRYOS [Burrow 1999].

2. *Integrating the role of the designer in computational exploration with a description formalism requires an interaction paradigm.*
Interaction paradigms provide a mechanism for introducing human design intent into computational exploration. Such a paradigm provides a systematic exposition of how communication, coordination and control strategies enable a designer to interact with a formal system. Manual, automated and cooperative paradigms for integrating the user with the system have been proposed in the literature. These models are described in Section 1.2. These accounts adopt a "black-box" approach to user interaction, where communication, coordination and control is based on the *apriori* division of labour between user and system. These paradigms posit a neat separation of the tasks to be performed between the user and the system under a global control policy. The mixed-initiative interaction paradigm models a more fine grained division of labour, allocating and sharing control over the same task jointly between the user and the system. This fine grained division of control offers a more flexible mechanism for acquiring and relinquishing initiative between the designer and the formalism.

8.1.2 Research hypothesis

The *mixed-initiative* formulation for supporting user interaction with a formalism is a useful paradigm for design space exploration. To identify how the mixed-initiative paradigm of interaction can address the problem of computational exploration, the following hypothesis was investigated:

That a mixed-initiative model of interaction presents a promising new approach for integrating the roles of the user and the description formalism in computational exploration.

This hypothesis posits the need for a mixed-initiative model of interaction for supporting design space exploration. The requirements necessary for addressing mixed-initiation exploration are identified in Chapter 3. To integrate the designer's view of exploration identified in chapter 1 with the entities supported by the description formalism described in Chapter 2, an interaction model for supporting mixed-initiative in developed in Part II. To address the requirements of mixed-initiative, the thesis develops an interaction model comprising the following: a representation of the *domain*, a communication layer for *dialogue* between the user and the formalism and operations for performing the *tasks* associated with exploration. The interaction model proposes *feature node unfolding* as the interaction paradigm for integrating the role of the user and the role of the formalism in design space exploration. Feature node unfolding is implemented in $\mathcal{FOLDS}$. The prototype comprises interfaces for constructing feature nodes, a navigator of interacting with visual representations of partial satisfiers, a viewer for visualising geometry in partial states and a communication mechanism for coordination and control of dialogue between the designer and the formalism. Mixed-initiative interaction is demonstrated through an example in the domain of architectural design: the exploration of three-dimensional massing configurations.

The fundamental contribution of mixed-initiative interaction to design space exploration is that *it enables the designer to maintain exploration freedom, preserves the underlying structure of exploration and permits a finer granularity of dialogue.*

These characteristics of the mixed-initiative formulation of interaction can be summarised as follows:

1. **Maintenance of exploration freedom.**
   Mixed-initiative maintains freedom for incorporating the intentional actions of the designer at any state of exploration. The *satisfier space* provides a unified model for representing the set of problems, subproblems, problem revisions and associated designs that a designer actually considers. Problems need not be fixed. Designs can be partial or complete with respect to the initial problem formulation. A designer may make varied choices that imply different kinds of design space operations. All are captured in the satisfier space. Symbolically, the satisfier space is a tree of visited design possibilities. The independence of the satisfier space from design space is developed in Section 4.5 and demonstrated in the example of mixed-initiative exploration outlined in Section 7.2.

2. **Preservation of order.**
   Mixed-initiative enables *order preserving* exploration. The structure of exploration is represented through the ordering relation of *subsumption*. The concept of an ordered design space underpins interaction between the designer and the description formalism. In it, the collection of exploration states are ordered by the relation of subsumption. Exploration moves are cast in terms of moves in a design space upwards or downwards in an information ordering. In Section 2.3, the ordering of exploration structure through subsumption is described. Mixed-initiative provides a principled way for keeping track of additions, deletions and other forms of change as the exploration progresses without negating the underlying subsumption ordering of possible states.

3. **Granularity of interaction.**
   Mixed-initiative permits a finer granularity of interaction between between the designer and the formalism. It supports incrementality and turn-taking in the process of exploration dialogue. Through incrementality, emphasis is shifted from the the final results of exploration to its intermediate constructive steps. Through turn-taking, the formal movement algorithms are made accessible to the designer. The incremental, turn-taking model of interaction permits a sound treatment of exploration non-determinism, where disjunctions in description queries, alternative constraints, conflicts and errors can be resolved by user intervention.

8.2 Contributions of the thesis

The mixed-initiative interaction model for design space exploration is predicated on the development of three interface level constructs: the feature node representation (Section 4.4) in the domain layer, a visual notation representing feature nodes (Section 5.2.1) in the dialogue layer, and a set of unfolding operations over feature nodes (Section 6.1) in the task layer. The contributions of the thesis are as follows:

1. **Feature node representation.**
   An interface construct for composing both the substrate concepts and the designer's view of exploration in a common representation.

2. **Visual notation for feature nodes.**
   A visual notation for presenting the input and output modalities of both the formalism and the user in an integrated manner.

3. **Unfolding operations.**
   Operations for unfolding feature nodes both by formal computation (intrinsic attributes) and by user interaction (extrinsic attributes).

8.2.1 Feature node representation

The *feature node* is an interface construct for composing the designer's view of exploration and the substrate concepts available in the formalism into a common representation. The designer's view of exploration is in terms of problems, solutions, choices and history. In this view, the process of exploration is elaborated as problem formulation and reformulation, solution generation and reuse, choice-making over alternatives and revisions and the ability to use the rationale or history of exploration. The symbol substrate provides a formalism stated in terms of the logical language of typed feature structures. It is a generic theory supporting the computational concepts of information ordering, partiality, intensionality, structure sharing and satisfiability. The symbol level is construed in terms of types, features, descriptions and resolution algorithms. This approach to search and exploration through mixed-initiative is reported in Datta & Woodbury [2000] and Datta & Woodbury [2001].

The role of the domain layer in the mixed-initiative model of exploration (described in Section 3.2.3) is to support the designer's view of the domain, provide a shared representation that mediates between the designer's view and the formal substrate and support joint responsibility over domain goals. The feature node representation addresses these requirements as follows:

- **Support for the designer's view of the domain.**
  The feature node, FNode expresses the relationship between a problem state, PState and an

alternative design that is a partial solution to the problem, SState and enables the designer to access the processes of problem formulation, solution generation and design space navigation. The FNode captures what choices a designer might make and how a designer would make such choices, that is, design intention. Finally, a *Satisfier Space* composes a set of ancestor and progeny feature nodes recording the history of exploration, as uncovered by the designer's actions. The satisfier space is a a tree of visited design possibilities such that each element in the satisfier space is a feature node, Fnode, which connects to the underlying design space machine.

- **Joint responsibility over goals.**
  The goals of exploration are bounded by the domain of discourse encoded in the TypeSystem. The designer can specify problems as descriptions which appear in the interface as feature nodes. The results of formal exploration are also cast in the same representation. An example of such a domain is given in Section 7.2. The first element of a FNode is a PState (see Figure 7.5). By interaction with this element, the designer can either modify (reformulate) the PState or generate a new problem state. The rest of the elements of a FNode are entities representing SState nodes generated by the formalism. These are the partial satisfiers (solution states) of the FNode. By interaction with these elements, the designer can unfold the possible solution states of the current problem. In this manner, through feature nodes, both the user and the formalism share joint responsibility in advancing the goals of exploration.

8.2.2 Visual notation for feature nodes

The visual notation presents the input and output modalities of both the formalism and the user in an integrated manner. The visual representation of a feature node is based on extensions to the AVM notation developed in computational lingustics. While the AVM notation visually describes feature structures, the notation introduces typed feature structures as the medium of dialogue and a mode of user manipulation.

The role of the dialogue layer in the mixed-initiative model of exploration (described in Section 3.4.3) is to support the representation (input and output) and integration (turn-taking) of dialogue between user and formalism. The visual notation for feature nodes addresses these requirements as follows:

- **Support for the representation of dialogue.**
  The intrinsic attributes of a feature node, FNode are mapped onto elements of the AVM notation. This mapping annotates the feature node with the type, feature names, feature values and co references taken from the underlying symbol substrate. Through the intrinsic attributes of a feature node, the formalism is able to represent its input and output modalities.

The notation is extended into interaction objects by specifying interaction logic for representing the extrinsic attributes of a feature node. The feature structure representation and the interaction logic are brought together in the dialogue layer construct, the *Visual feature node.* The designer's input and output modalities are handled through the extrinsic attributes of a feature node. The connection between a visual feature node, VNode and a FNode using the AVM notation is described in Section 5.2.1.

- **Support for the integration of dialogue.**
  The process of mixed-initiative dialogue during exploration is implemented using visual feature nodes. Through this interface construct, the user is able to participate in a dialogue with the description formalism. The dialogue layer supports a model of incremental turn-taking. Turn-taking allows the designer to manipulate the output from the formalism through visual means. It permits the formalism to manipulate the input from the designer in terms of typed feature structures. The integration of dialogue through user interface objects is implemented in $\mathcal{FOLDS}$ and is described in Section 7.3.1.

The graphical notation for mixed-initiative dialogue developed here is reported in Datta & Woodbury [2002] and Datta [2002].

8.2.3 Unfolding operations

Operations for unfolding feature nodes both by formal computation (intrinsic attributes) and by user interaction (extrinsic attributes) are supported. To incorporate mixed-initiative, it is necessary to integrate system-driven and designer-driven moves in the task layer. This is addressed by treating both types of operations under a common conceptual metaphor, termed, *unfolding.* Visual feature nodes can be *unfolded* by formal and behavioural properties during exploration.

During exploration, system-driven moves modify the *intrinsic* attributes of a feature node. The unfolding of the intrinsic properties of a visual feature node is based on the movement operators of the exploration formalism described in Section 2.4. The formal portion of an unfolding operation operates on a partial satisfier PSat through the visual feature node, VNode.

Designer-driven moves represent the unfolding of the extrinsic attributes of a visual feature node. The unfolding of extrinsic properties of the feature node is based on interaction behaviour of a visual feature node, VNode. These behaviours are described in Section 5.3. The designer operates on the visual elements of the feature node to affect change during exploration and these changes are cast as extrinsic to the representation.

The mixed-initiative unfolding of feature nodes comprises the *generation, navigation* and *synchronisation* of feature nodes. The role of the task layer in the mixed-initiative model of exploration (described in Section 3.3.3) is to support the construction and reformulation of problems, the navigation of problems and solutions and the synchronisation of exploration results. Unfolding operations over feature nodes address these requirements as follows:

- **Support for problem formulation and solution generation.**
  The designer represents the problem state through a query description, ϕ in the attribute-value description language. The explorer converts ϕ using the satisfiability algorithm into a problem state, $\bar{d}$. The CONSTRUCT operation takes a problem state $\bar{d}$, computes its visual feature node representation, VNode, and displays it for user input. Through the EXTEND operation, the designer can interact with the resolution process, by selecting a *feature-value pair* and seeking to extend it to the next state. Exploration is advanced by stepwise operations on the *feature-value map* displayed as a visual feature node. At each step, the explorer (description formalism) constructs a sequence of partial satisfiers, PSat through incremental π-resolution. The sequence of interactions for unfolding a feature node through the CONSTRUCT and EXTEND operations are described in Section 6.2.

- **Support for the navigation of problems and solutions.**
  This operation of unfolding corresponds to the task of navigating attributes of a visual feature node. Navigation in the design space is supported through the operations of CHOICE, RETRACT. Navigation operations enable the designer to locate, identify and move through the collection of generated nodes and their paths. The sequence of interactions for unfolding a feature node through the CHOOSE and RETRACT operations are described in Section 6.3.

- **Support for the synchronisation of exploration results.**
  Support for synchronisation addresses the problem of unfolding the results of two distinct exploration paths in the satisfier space with respect to their underlying ordering in the design space. Synchronisation of distinct exploration paths is supported through the operations of RECALL, ERASE, MEET, JOIN. The sequence of interactions for unfolding a feature node through these operations are described in Section 6.4.

8.3 Constraints on the results

A number of constraints that limit the scope of the results reported in this study are outlined.

Restriction on subsumption-ordered design spaces.

The framework of design space navigation proposed in Chien [1998] uses a five-dimensional sructure to integrate problems, designs, versions and alternatives. During exploration, objects can be added to any of the dimensions of the design space, an arbitrary number of times. As reported in Chien & Flemming [1997], this formulation leads to computational intractability in keeping track of relationships between components of the design space.

This problem is partly alleviated in this study by keeping the conception of a design space simple: using a formal structure in defining the design space based on a restricted but well understood set of properties. New nodes are added along a single dimension in a strictly monotonic fashion based on the relation of *subsumption*. To limit its complexity, one restriction enforced on the formal model of a design space is to separate the subsumption ordering of the underlying design space from user interaction. This is done through by mediating the interaction through the SatSpace, which represents the tree of visited nodes rather than the full design space. Thus the mixed-initiative model operates over a strictly monotonic, single-dimensional design space. No claims can be made on non-monotonic, multi-dimensional and alternate conceptions of design space.

Restriction to functional decomposition

The scenario of designing outlined in this thesis is restricted to abstract, functional attributes. The semantics of a design, the relation between conceptual and graphical domains is not addressed. Two approaches that establish explicit relations in design semantics are proposed in Klein & Pineda [1990] and in Harada et al. [1995]. The work reported in this study is limited by two assumptions. First, designs are assumed to be strictly feature structure-like. Second type information, for example, the types *massing* and *house* correspond to mutually exclusive decompositions. Thus the mixed-initiative model of interaction operates on designs based on functional view of exploration. No claims can be made on the manipulation of visual and geometric properties of designs.

Restriction to massing configurations

The mixed-initiative interaction model as implemented in $\mathcal{FOLDS}$ is limited to the demonstration of a single class of configuration problem, namely massing. The SEED [Flemming et al. 1993, Akin et al. 1995, Flemming & Chien 1995, Akin et al. 1997] project provides a discussion of work done in supporting a comprehensive class of configuration problems, particularly the relationships between different configuration modules. The mixed-initiative model of interaction described in this thesis is restricted to exploring massing configurations. While the mixed-initiative interaction model is more general than the example used to demonstrate it, no claims can be made about other configuration tasks such as *structure* or the relationships between configuration problems such as *layout* and *enclosure*. This requires the implementation of additional cases in these types of configuration design.

Restriction on unfolding

In $\mathcal{FOLDS}$, the representation of massing elements is restricted to a one-to-one mapping between geometric entities and partial satisfiers. The continuous transformation of a generated geometry, that modifies or conflicts with the constrained *bounds* of the functional specification is not permitted. Unfolding is restricted to feature nodes and the manipulation of geometry in design states is treated as external to the formal exploration process. The mixed initiative model does not address the relationship between the formal movement algorithms and model transformation operations external to π-resolution. No claims are made on the handling of geometric manipulation during exploration.

8.4 Future directions

Further extensions to the model of mixed-initiative exploration is necessary to address the limitations discussed above. Four potential areas in design space exploration that remain are as follows:

- **Interaction with type systems.**
  Domain knowledge is encoded through the specification of types, features and constraints. Each component is manually edited in its own file, using a standard text buffer, which is then loaded onto the interface using a shell console. The KRYOS feature structures system parses this data into the TypeSystems. This process becomes complex and error-prone as more types, features and constraints are added onto the system. The graph visualisation program, DAVINCI [Fröhlich & Werner 1994, Fröhlich & Werner 1995] is used to correct the complex relationships between types, features and constraints. To facilitate the authoring of scalable TypeSystems, more comprehensive support for writing type systems is necessary. An interactive editor would facilitate the crucial process.

- **Description processor.**
  The authoring of descriptions comprises the processes of writing, displaying, browsing and parsing description fragments. In the massing design example, a substantial effort is spent in formulating and reformulating description queries. Description authoring comprises a two-stage process. Valid syntactically correct description fragments in $\mathcal{FOLDS}$ are parsed into description nodes in KRYOS while invalid descriptions are labelled with the type *absurd* and returned for reformulation. Valid description nodes are then labelled in $\mathcal{FOLDS}$ by dumping their memory address into strings from the raw data provided by the KRYOS feature structures system. An interactive description processor that maintains the persistence of descriptions by naming them, would enhance formulation and reformulation cycle of description authoring in $\mathcal{FOLDS}$.

- **Visualisation of design spaces.**
  The development of algorithms for visualising the components of design space remains an ares for future work. The work of Tecuci et al. [1999] in the visualisation of evolving knowledge

bases using mixed-initiative methods presents a possible way. The development of history-rich tools for interaction [Wexelblat 1999, Wexelblat & Maes 1999] and mechanisms for implementing and tracing object evolution [Feijo & Lehtola 1996] offer new possibilities for the support for tracing the rationale of exploration.

- **Incorporation of concrete domains.**
  Further progress on the theoretical characterisation of concrete domains within the theory of typed feature structures is necessary. Burrow foreshadows how this may be done. In Burrow [1999], integer intervals are incorporated into type hierarchies using OrderTypes. Chang's [1999] *Geometric Feature Structures* extends this idea of *order types* [Burrow 2003] to represent geometric information directly within the framework of typed feature structures. Handling interaction with such types of concrete domain information remains a topic of future research.

Appendix A

Typed Feature Structures

Typed feature structures provide a representation for design space exploration in which both the action of exploration and the structure of a design space are given a sound theoretical basis. Typed feature structures provide well-founded support for an object-and-relations view of representation and within that view support intentionality, partialness, structure sharing, and cyclicity. Typed feature structures can tersely express and solve simple finite-domain generation of alternative mappings of function into form in the building domain [Woodbury et al. 1999]. Feature structures provide a representation for the constraints on the design space, relates those constraints to the generation of designs and supports intermediate, partial representations of design states.

A deep understanding of typed feature structures requires an appreciation of its formal mechanics. Carpenter [Carpenter 1992] provides a complete description of typed feature structures, including many proofs on the logic of typed feature structures. This appendix describes the formal definitions and terminology underpinning the exploration formalism described in Chapter 2.

A.1 Terminology and definitions

A.1.1 Type Hierarchy

The finite set of types is organised into a structure according to information specificity. The type σ subsumes type τ (written as $\sigma \sqsubseteq \tau$) if type τ contains strictly more information than type σ. Type σ is a super-type of τ ; τ a subtype of σ. Beyond the partial order properties implied by type subsumption (which is not same relation as the feature structure subsumption relation), a feature structure type hierarchy is required to have two additional properties. First, for any set of types there is at most one type that is directly subsumed by all types in the set. A set of types with a common subtype is said to be bounded or consistent. This condition on the type hierarchy amounts to saying that there must be a unique most general subtype for any consistent set of types in the hierarchy. Second, there must be a most general type (conventionally called *Bottom* and written

$\perp$) at which all types meet. This condition is implied by the first if the sets of consistent types includes the empty set. Together with the partial order conditions of transitivity, anti-symmetry and reflexivity, these conditions create what is called a *bounded complete partial order*, refereed to a BCPO.

With types are associated features, drawn from a finite set Feat of features. The function $Intro : \mathsf{Feat} \rightarrow \mathsf{Type}$ defines for every feature a unique most general type at which that feature is introduced into the type hierarchy. All sub-types of $Intro(f)$ contain f and f is said to be *appropriate* for $Intro(f)$ and its successor sub-types. Sub-type feature inclusion and being a complete function in Feat implies that any feature that is multiply inherited from two or more super-types is, in fact, the same feature, thus some typical ambiguities of multiple inheritance do not arise. The partial function $Approp : \mathsf{Feat} \times \mathsf{Type} \rightarrow \mathsf{Type}$ gives a type restriction on the values of a particular feature: $Approp(f, \tau)$ is the most general type that a value of feature f in type τ can have. On $Approp$ are placed the conditions of upward closure mentioned above and that feature values can only become more specific in subtypes, that is, if for two types σ and τ, $Approp(f, \sigma)$ is defined and $\sigma \sqsubseteq \tau$ then $Approp(f, \tau)$ is also defined and $Approp(f, \sigma) \sqsubseteq Approp(f, \tau)$.

Definition 1 (Appropriateness Specification [Carpenter 1992]) *An appropriateness specification* over the inheritance hierarchy $\langle Type, \sqsubseteq\rangle$ *and features* Feat *is a partial function Approp:* $\mathsf{Feat} \times \mathsf{Type} \mapsto \mathsf{Type}$ *that meets the following conditions:*

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

Upward Closure *if* $Approp(f, \sigma)$ *is defined and* $\sigma \sqsubseteq \gamma$*, then* $Approp(f, \gamma)$ *is also defined and* $Approp(f, \sigma) \sqsubseteq Approp(f, \gamma)$

A.1.2 Feature Structures

Given an inheritance hierarchy, a BCPO of types $\langle \mathsf{Type}, \sqsubseteq\rangle$ and a set of features Feat, a typed feature structure is formally defined as,

Definition 2 (Feature Structures [Carpenter 1992]) *A* feature structure *is the tuple* $F = \langle Q, \bar{q}, \theta, \delta\rangle$*, where*

- Q *is a finite set of* nodes,
- $\bar{q} \in Q$ *is the* root node,
- $\theta : Q \mapsto \mathsf{Type}$ *is the node* typing *function,*
- $\delta : \mathsf{Feat} \times Q \rightarrow Q$ *is the partial* feature value *function, and*

- *Q is the smallest set such that $\bar{q} \in Q$ and $q \in Q \wedge q' = \delta(f, q) \rightarrow q' \in Q$.*

Definition 3 (Resolved Feature Structure [Carpenter 1992]) *A feature structure F is resolved if and only if*

$$\forall \pi \in \mathsf{Path}, \forall t \in \mathsf{Type} \quad : \quad t \leq_{\mathsf{Type}} \tau(F@\pi) \rightarrow F@\pi \textit{ satisfies } \mathrm{Cons}(t) \tag{A.1}$$

Definition 3 describes the properties of a fully resolved feature structure. However, resolved feature structures are the endpoints in a constraint resolution process. Since the design space is explicitly concerned with partiality it will include intermediate stages in the resolution process as design states

A.1.3 Type System

A type system is composed of the objects *types, features, constraints and descriptions.*

Definition 4 (Type System) *A* TypeSystem *is the quadruple* $\langle \langle Type, \sqsubseteq \rangle, \mathsf{Feat}, \mathsf{Cons}, \mathsf{Desc} \rangle$ *where*

- $\langle Type, \sqsubseteq \rangle$ *is an inheritance hierarchy of types,*
- Feat *is a finite set of features,*
- Cons *is a constraint system and*
- Desc *is a description language.*

A.1.4 Descriptions

Definition 5 (Generator) *The set of* generators *over a* TypeSystem *and collection of descriptions* Desc *is the least set* Gen *such that*

- $\sigma \in \mathsf{Gen}$ *if* $\sigma \in \mathsf{Desc}$
- $\pi : \phi \in \mathsf{Gen}$ *if* $\pi \in \mathsf{Path}, \phi \in \mathsf{Desc}$
- $\pi_1 \doteq \pi_2 \in \mathsf{Gen}$ *if* $\pi_1, \pi_2 \in \mathsf{Path}$
- $\phi \vee \psi, \phi \wedge \psi \in \mathsf{Gen}$ *if* $\phi, \psi \in \mathsf{Gen}$

A.1.5 Type constraints

Given a type inheritance hierarchy $\langle Type, \sqsubseteq \rangle$, and an appropriateness specification, *Approp* a constraint system can be defined as

Definition 6 (Constraint System [Carpenter 1992]) *A* constraint system *is a total function* $\mathsf{Cons} : \mathsf{Type} \mapsto \mathsf{Desc}$.

The type t of a feature structure F asserts only that the represented object is an instance of t or some subtype. By stating restrictions on the feature structures of each type, we assert additional properties on the feature structures and therefore, by implication, about represented objects. The *constraint system* expresses restrictions on a substructure, according to its type, which determine legal labellings over a finite but arbitrary collection of paths.

Let *Desc* be the resulting collection of descriptions and $\mathcal{G}$ be the resulting collection of feature structures. A constraint system associates every type σ with a description ϕ in *Desc*. A feature structure of type σ must *satisfy* the constraint ϕ,

$$\sigma \Rightarrow \phi,$$

where ϕ is an arbitrary description. Taken in the form of an implication, a feature structure $\mathcal{F}$, satisfies a system of constraints if every one of its substructures satisfies the constraints on its type. An algorithm for constraint resolution determines whether a feature structure satisfies a constraint system. A substructure satisfies the constraint system if it satisfies the type constraints for its type and all super-types. A feature structure is *resolved* if it satisfies the constraint system at every substructure.

The process of constraint resolution constructs and orders these partially resolved feature structures. It is a complex recursive process — as substructures are resolved new substructures and type labellings are introduced that require further resolution steps.

A type constraint description at a substructure is satisfied if a most general satisfier subsumes the constrained substructure. Therefore, constraint resolution is the search for the most general feature structure subsumed by both a constraint satisfier and the current substructure.

A.1.6 Incremental π-resolution

The fundamental structure construction process is called π-resolution [Carpenter 1992, p 230]. It is a non-deterministic search for solutions to a query description stated formally in terms of paths. A solution is a feature structure, which satisfies a *description* and the *constraint system.* Starting with the most general satisfier(s) of the query, all of the solutions to the query can be effectively enumerated.

Given a type system and a description, a π-resolution procedure enumerates the most general feature structures satisfying the conjunction of description and type constraints. In design, non-deterministic resolution is more useful in the generation of branch points in the design space than for retrieving the set of solutions. Design processes are intrinsically processes of incremental construction, where the representation of the intermediate states of constraint resolution is essential. Hence, incrementality and nondeterminism in the resolution process are critical to supporting exploration. The following definition extends the formalism to address these difficulties. A non-deterministic rewriting procedure that *enumerates* solutions to a given constraint system based on a strongly typed system is proposed by Burrow [Burrow 2003]. The incremental π-resolution procedure is a special case of the approach described Carpenter's *Typed Feature Structures* [Carpenter 1992, p 227–242].

Definition 7 (Incremental π-Resolution [Burrow & Woodbury 1999]) *Given F a feature structure and π, a path to F, such that $F@\pi$ is defined, a function recording resolution steps Δ, and a type t such that $t \leq_{\mathsf{Type}} \tau(F@\pi) \wedge t' \leq_{\mathsf{Type}} t \rightarrow t' \in \Delta(F@\pi)$, we take an* incremental π-resolution step

$$\langle F, \Delta \rangle \overset{\pi,t}{\Longrightarrow} \langle F', \Delta' \rangle$$

iff

$$\exists F_t \in \mathrm{MGSats}(\mathrm{Cons}(t)) \;:\; F'@\pi \sim F@\pi \sqcup F_t \tag{A.2}$$

$$t \in \Delta'(F'@\pi) \tag{A.3}$$

$$\forall \pi' \in \mathsf{Path} \;:\; F@\pi' \textit{ defined } \rightarrow \Delta(F@\pi') \subseteq \Delta'(F'@\pi') \tag{A.4}$$

In incremental π-resolution, each decision point in the enumeration is available as an intermediate result. An intermediate state satisfies the invariant that every node in the partially resolved satisfier has been resolved against a down-set of types whose join subsumes the target type. The elements of AntichainLattice [Burrow 2003] represent intermediate states in the summation of type constraints. An intermediate state satisfies the invariant that every node in the partially resolved satisfier has been resolved against a down-set of types, called the AntichainLattice, whose join subsumes the target type. The elements of AntichainLattice [Burrow 2003] represent intermediate states in the summation of type constraints. The subsumption ordering of these objects form a lattice structure called an antichain lattice. An antichain lattice is shown in Figure A.2. An antichain is the set of mutually incomparable elements in a partially ordered set, *poset.*

The *solution* is the result of a sequence of extension steps corresponding to the satisfaction of constraints, which are organised into an inheritance hierarchy of types. Since π-resolution proceeds by extension, the resultant search space can be order embedded into the information ordering over feature structures. Since constraints are drawn from an inheritance hierarchy, alternatives may be organised according to notions of abstraction. Thus, if feature structures are employed to represent functional decompositions, π-resolution non-determinism allows exploration in terms of alternative functional decompositions.

The operation in Definition 7 is one step in an incremental π-resolution search. During the search process the current partial satisfier evolves via numerous such resolution steps with the addition of information from the constraint system. Each step acts on a substructure of the current partial satisfier. This is depicted in Figure A.1. The argument π selects the substructure, and the argument t selects the constraint to resolve. The definition of a step includes restrictions that ensure resolution steps are goal directed and well ordered with respect to the type hierarchy. Namely, that t be a super-type of the type at π and that all types more general than t are already resolved at π. The execution of a step involves unification at the substructure.

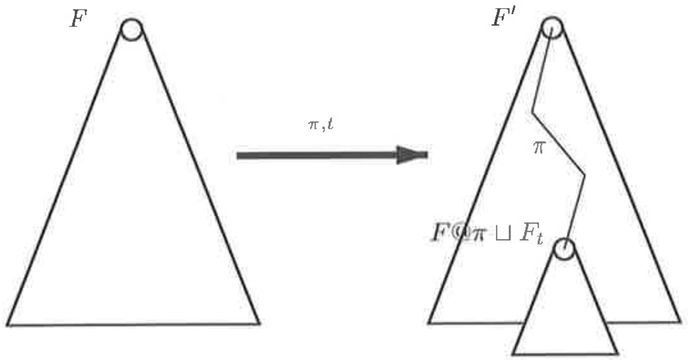


Figure A.1: The generating procedure, π-resolution captures a relation from descriptions to the satisfiers and is the main generative mechanism in the system. The resultant feature structure in each sequence is the most general satisfier of the query description [Burrow & Woodbury 1999].

Definition 7 differs from [Carpenter 1992, p 231] in its granularity, and in explicitly recording the resolution of each type constraint. Rather than resolve the conjunction of constraints from types subsuming $\tau(F@\pi)$, a sequence of steps accumulates this same down-set and records the progress in Δ. Given a query description D, π-*resolution* is the construction of sequences of feature structures $P_0 \sqsubseteq P_1 \sqsubseteq P_2 \sqsubseteq \ldots \sqsubseteq P_k$. The initial feature structure in each sequence is a most general satisfier of the query description. The sequence represents the inclusion of type information in the form of constraints — each element extends its predecessor by unification with a type constraint. Since most general satisfiers may occur as collections and unification may fail, the search for resolved feature structures involves a collection of sequences.

An example of incremental π-resolution is given by compiling the following disjunctive description over the type hierarchy,

```
(LIVING HEIGHT == DINING HEIGHT) |
        (LIVING HEIGHT == DINING HEIGHT
        & LIVING SIZE OY == PORCH SIZE OY
        & LIVING SIZE OX == LIVING SIZE OY
        & PORCH HEIGHT == LIVING HEIGHT)
```

A partial satisfier is constructed from the query and this node becomes the subject of an incremental π-resolution.

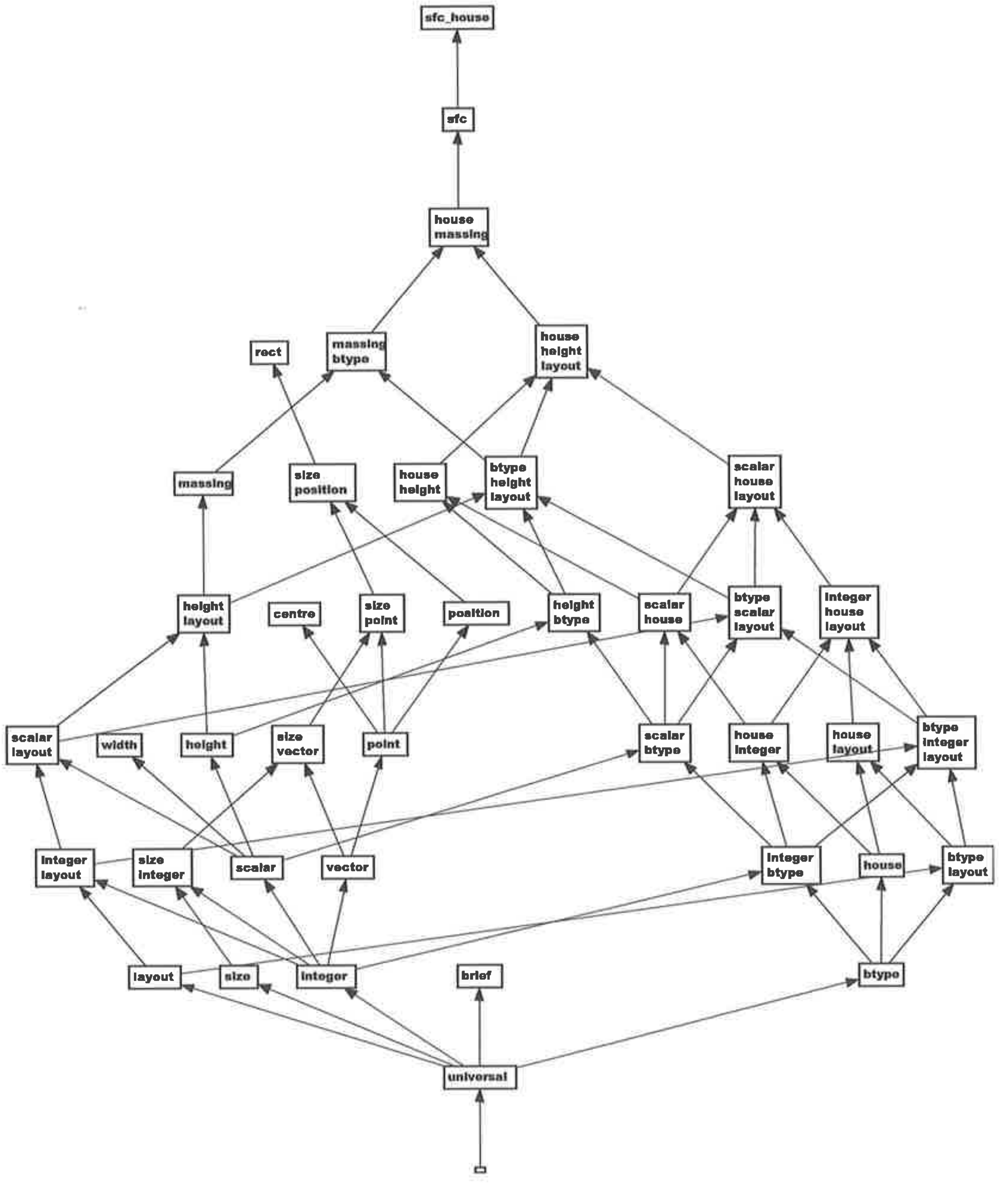


Figure A.2: An antichain lattice, Ω comprises the down set of types.

```
[1] universal
[1] :
  conjunct_0x812b498
    disjunct_0x80bb070
      house
      DINING: scalar
              HEIGHT: [2] universal
      LIVING: scalar
              HEIGHT: [2]
    disjunct_0x80c5ff8
      sfc
      DINING: scalar
              HEIGHT: [3] universal
      LIVING: massing
              HEIGHT: [3]
              SIZE: vector
                    OX: [4] universal
                    OY: [4]
      PORCH: massing
             HEIGHT: [3]
             SIZE: vector
                   OY: [4]
```

Appendix B

UML Notation

The Unified Modelling Language [Fowler, Scott & Jacobson 1997, Jacobson, Booch & Rumbaugh 1998], UML, is a formally defined, object-oriented modelling notation. It aims to provide a standard notation for modelling systems, particularly software intensive systems where an object-oriented implementation is anticipated. It is independent of programming language or development techniques.

In the UML, a model is a complete representation of a system from a particular viewpoint, that is, an aggregation of a set of views from a specific perspective. At the same time, systems are composed of many nearly independent models representing many independent subsystems, each of which can be treated as a model. Thus, a system is implicitly represented by a top-level model with subsidiary models representing the subsystems.

In UML, a model is made up of diagrams and text. The UML specification[1] describes diagrams as "views of a model", each representing a particular perspective that the overall model integrates. Thus a UML model is an abstraction of a system, and models the concepts, relationships, behaviours and interactions in the system. Models are made up of model elements. Models and model elements are rendered graphically in diagrams.

In UML notation, model concepts are expressed as symbols icons. Relationships are expressed by adorned lines, with semantic content. The way the model concepts connect provides the meaning of the model. Thus, underlying the graphics of the model are the specifications of model elements, a mix of formal and informal elements. Diagrams are two dimensional and text is used to annotate the diagrams. Their are three kinds of relationships in diagrams, namely,

- connection.
  Adorned lines connect icons and symbols, forming connecting paths.

- containment.

[1]Object Management Group [online]. 1999. UML specification, Version 1.3. Available from www.omg.org.

Enclosed shapes such as boxes and circles contain symbols, icons and lines.

- visual attachment.
  Elements close together, such as a name above a line or a number next to a box imply that they apply to that element.

B.1 Modelling concepts

In this study, a subset of symbols from the UML notation is used to express modelling concepts. This subset is presented in this section. The reader can refer to these definitions for understanding the diagrams that are presented in UML notation in the thesis.

B.1.1 Symbols

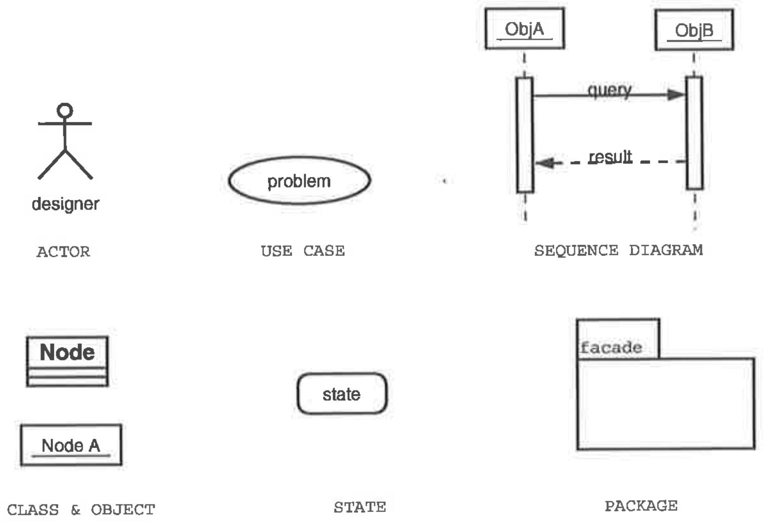


Figure B.1: Symbols in UML notation.

Actor

An *actor* is something or someone outside the system that interacts directly with the system.

Use case and Sequence diagrams

A *use case* is a sequence of interactions between an actor and the system.

A pattern of interaction among objects is shown on an interaction diagram. Interaction diagrams come in two forms based on the same underlying information but each emphasising a particular aspect of it: sequence diagrams and collaboration diagrams.

A *sequence diagram* shows an interaction arranged in time sequence. In particular, it shows the objects participating in the interaction by their *lifelines* and the *messages* that they exchange arranged in time sequence. It does not show the associations among the objects. A *collaboration* is a collection of objects that interact to implement behaviour. A collaboration is used to specify the realisation of a use case.

Class and Object

A *class* is an abstraction of a set of possible objects that share the same attributes, operations and relationships. An *object* is an instance of a class.

Package

A *Package* is a UML container used to organise model elements.

State

A *state* is the situation or status of an object as the result of an interaction. A state can model ongoing activity.

B.1.2 Relationships

In UML, lines are used to express dynamic connections between model elements, relationships between model elements and interactions.

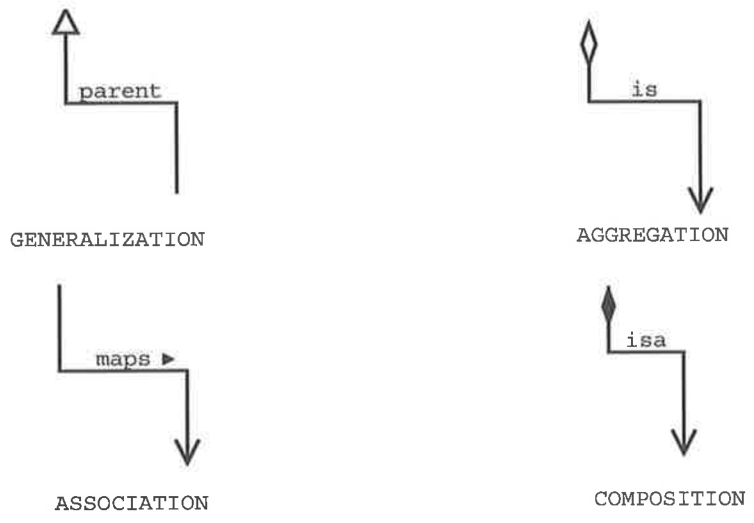


Figure B.2: Relationhips in UML notation.

Generalisation

A *generalisation* is a relation between two elements in which one is a more general form of the other.

Association

An *association* maps one object to another set of objects. Aggregation and composition are special forms of association. A plain form of association shows the relationship between peers.

Aggregation

An *aggregation* represents a part-whole relationship between one object and several subparts. One element is the whole and the other(s) are parts.

Composition

A *composition* is an aggregation that exhibits strong ownership on its parts, that is, the parts belong to the composing object.

B.1.3 Diagrams

In UML, diagrams are the graphical presentation of semantic models. In this study, two diagram types, the class diagram and the sequence diagram are used.

Class diagram

A class is the descriptor for a set of objects with similar structure, behaviour, and relationships. UML provides annotation for declaring classes and specifying their properties, as well as using classes in various ways. The *class diagram* is a static structural representation of the interfaces, packages and relationships the comprise a model specification. *Use case* diagrams show actors, use cases and their relationships.

Interaction diagrams

Interaction diagrams are used for modelling dynamic situations. A *sequence* diagram shows an interaction arranged in a time sequence comprising objects, the messages that pass between them and the interaction that occurs. A sequence diagram has a list of participating objects, an object lifeline, the time-ordered visual framework for message exchange between objects exchange between objects.

A *collaboration* diagram shows the passing of objects between objects focusing on the order of their execution.

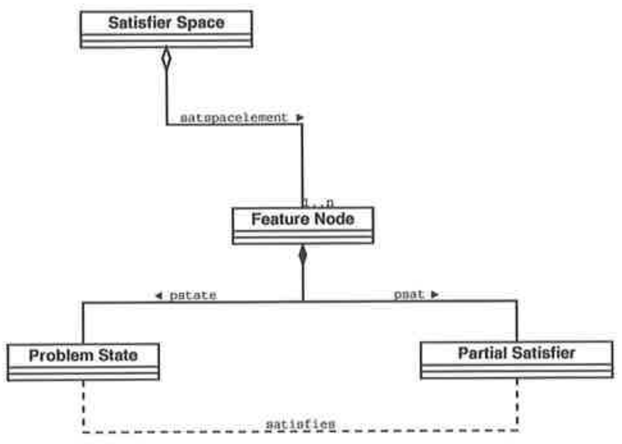


Figure B.3: Class diagram in UML notation.

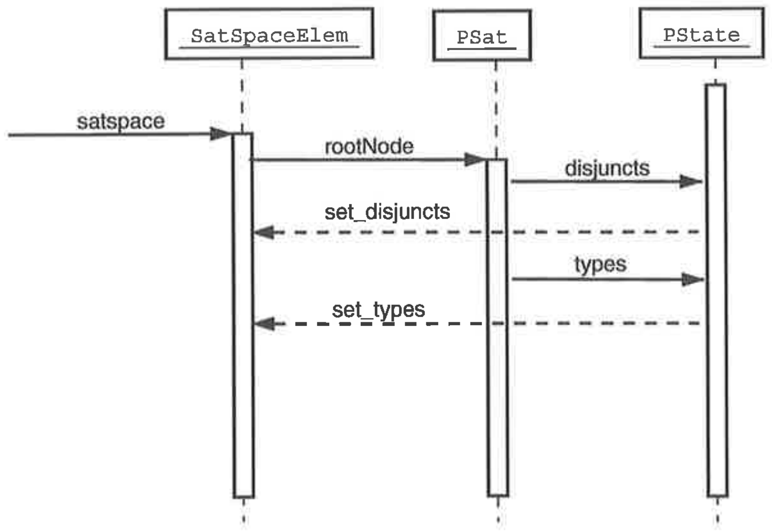


Figure B.4: Sequence diagram in UML notation.

Appendix C

Massing Configurations in *FOLDS*

The massing configuration problem described in Chapter 7 is based on the creation of a TypeSystem (defined in Section A.1.3) in KRYOS. The definition comprises the declaration of types, features, constraints and descriptions. Examples of these components of the massing TypeSystem are given in this Appendix.

C.1 An inheritance hierarchy of types

The specification of the inheritance hierarchy of the massing configuration problem is given as follows:

```
%%%*************************************************************%%
%%%  Kryos : type hierarchy $Id: sfc.types,v 1.1.1.1 1999/12/22
%%%  05:19:29 akids Exp $

%%%*************************************************************%%


% math types

integer
    inherit universal;

scalar
    inherit universal;

% basic spatial types
```

```
point
    inherit universal;


position
    inherit point;


centre
    inherit point;


geom
        inherit universal;

%%%*****************************************************************%%
%%%
%%%  SEED KNOWLEDGE LEVEL TYPES
%%%
%%%*****************************************************************%%

btype
    inherit universal;


brief
    inherit universal;


function
        inherit universal;
du
  inherit universal;


fu
 inherit universal;


configuration
 inherit du;


%a single geometry
```

```
massing
  inherit configuration;

layout
  inherit configuration;

solid
        inherit scalar;
wall
    inherit scalar;
column
    inherit scalar;
slab
    inherit scalar;

%massing types
massing_a
         inherit massing;
massing_b
         inherit massing_a;
massing_c
         inherit massing_b;
massing_d
         inherit massing_c;
massing_e
         inherit massing_d;
massing_f
         inherit massing_e;


% a house is a functional specification related to a formal model
house
  inherit fu;

%derived sfc types
skillion
```

```
        inherit massing;
sfc_house
  inherit house;

%single units
sfc_hall
        inherit fu;
sfc_room
        inherit fu;
sfc_roomrow
        inherit fu;
sfc_porch
        inherit fu;
sfc_skillion
        inherit skillion;
sfc_kitchen
        inherit fu;

sfc_bath
        inherit fu;

% double units
sfc_roomrow_two
        inherit sfc_roomrow;

sfc_roomrow_three
        inherit sfc_roomrow_two;

sfc_roomrow_four
        inherit sfc_roomrow_three;

% massing exploration types

house_massing inherit house & massing;

%sfc_massing inherit sfc & massing;
```

```
sfc  inherit sfc_house & house_massing;

%command components

command
        inherit universal;

arg_list
    inherit universal;

empty_arg_list
    inherit arg_list;

nonempty_arg_list
    inherit arg_list;
```

C.2 Appropriateness specifications

The specification of the appropriateness specifications (feature introduction, as explained in Section A.1.1) of the massing configuration problem is given as follows:

```
%%%*******************************************************************%%
%%% Appropriateness Specifications
%%%*******************************************************************%%
%         a hook to base datatypes
scalar
        intro        SCALAR : integer;

%         basic spatial type
point
        intro        OX : scalar
            &        OY : scalar
            &        OZ : scalar;

%        A complex data type for geometric elements
geom
        intro            POS        : point
            &        COMMAND        : command;

%        Design Unit representation
du
        intro        DU_LABEL : scalar
            &            GEOM : geom;

%        Function Unit representation
fu
        intro        FU_LABEL : scalar
            &        FUNCTION : function;

%         massing types introduce a function unit and a position
massing
        intro        MASS_LABEL : scalar
            &          MASS_POS : position
            &                FU : fu;
```

```
%         building types are described with massing
%        => functional names and positions
house
        intro           HOUSE : massing_a
            &          LIVING : du
            &          DINING : du
            &        SKILLION : skillion
            &          LOUNGE : du
            &             BED : du;

%         a single fronted cottage, i.e. sfc is a type of house
%        introducing some features
sfc_house
        intro       SFC_HOUSE : fu
            &           PORCH : sfc_porch
            &         ROOMROW : sfc_roomrow
            &         ROOMNUM : scalar
            &            HALL : sfc_hall;

%        Types of massing

sfc_roomrow
        intro SFC_ROOMROW : massing;

sfc_hall
        intro SFC_HALL : massing;

sfc_porch
        intro SFC_PORCH : massing;

sfc_room
        intro SFC_ROOM : massing;

skillion
        intro        BATHROOM : massing
            &         HALLWAY : massing
```

```
         &         KITCHEN : massing;


sfc_skillion
        intro SFC_SKILLION : massing;


sfc_kitchen
        intro SFC_KITCHEN : massing;


sfc_bath
        intro SFC_BATH : massing;


%single functional labels with 2 du


sfc_roomrow_two
        intro         ENTITY_A : sfc_room
            &         ENTITY_B : sfc_room;



%        single functional labels with 3 du
sfc_roomrow_three
        intro         ENTITY_C : sfc_room;


sfc_roomrow_four
        intro         ENTITY_D : sfc_room;


%         massing types are expressed as features with design units
%         attached to them
massing_a
        intro MASSEL_A         : massing;


massing_b
        intro MASSEL_B         : massing;


massing_c
        intro MASSEL_C         : massing;


massing_d
```

```
        intro MASSEL_D         : massing;


massing_e
        intro MASSEL_E         : massing;


massing_f
        intro MASSEL_F         : massing;


%         command features
command
    intro ARG_LIST         : nonempty_arg_list;
%        recursive types for creating lists
nonempty_arg_list
    intro     ARG         : scalar
        & AL_TAIL         : arg_list;
```

C.3 Constraint declarations

The specification of constraints on types and features of the massing configuration problem is given as follows:

```
%%%**************************************************************%%
%%% Constraint System
%%%**************************************************************%%
scalar
        cons        SCALAR : integer;

% basic spatial types
point
        cons        OX : scalar
           &        OY : scalar
           &        OZ : scalar;

%         Geom constraints
geom
        cons            POS        : point
           &        COMMAND        : command;

%         Function unit constraints
%          the fu label is the same as the massing label
fu
        cons        FU_LABEL        : scalar
           &        FUNCTION        : function;


du
        % the du label is the same as the massing label
        cons         DU_LABEL         :         fu
        &        GEOM                :        geom;

%          massing types introduce a function unit and a position
massing
        cons        MASS_LABEL        : scalar
           &          MASS_POS        : position
           &            FU        : fu;
```

```
%        constraints on the house type
house
        cons          HOUSE        : massing
          &        BATHROOM        : du
          &          LIVING        : du
          &           DINING        : du
          &         KITCHEN        : du
          &          LOUNGE        : du
          &             BED        : du;

%        constraints on single fronted cottage type
sfc_house
        cons        SFC_HOUSE : du
          &             PORCH : sfc_porch
          &           ROOMROW :
                         ( sfc_roomrow        |
                           sfc_roomrow_two    |
                           sfc_roomrow_three  |
                           sfc_roomrow_four
                         )

          &           ROOMNUM :
                         ( scalar |
                           integer
                         )

          &           SKILLION SFC_SKILLION GEOM
                      POS : point
          &              HALL : sfc_hall;


%        single functional labels with du
sfc_roomrow
        cons SFC_ROOMROW : du;


sfc_hall
```

```
        cons SFC_HALL : du;


sfc_porch
        cons SFC_PORCH : du;


sfc_room
        cons SFC_ROOM : du;


sfc_skillion
        cons SFC_SKILLION : du;


sfc_kitchen
        cons SFC_KITCHEN : du;


sfc_bath
        cons SFC_BATH : du;

%        single functional labels with 2 du
sfc_roomrow_two
        cons        ENTITY_A : sfc_room
           &        ENTITY_B : sfc_room;

%        single functional labels with 3 du
sfc_roomrow_three
        cons        ENTITY_C : sfc_room;


sfc_roomrow_four
        cons        ENTITY_D : sfc_room;

%         massing types are expressed as features with design units
%        attached to them
massing_a
        cons         MASSEL_A : massing;


massing_b
        cons         MASSEL_B : massing;
```

```
massing_c
        cons          MASSEL_C : massing;


massing_d
        cons          MASSEL_D : massing;


massing_e
        cons          MASSEL_E : massing;


massing_f
        cons          MASSEL_F : massing;

% command features
command
            cons          ARG_LIST : nonempty_arg_list;

%        constraints on list ltype
nonempty_arg_list
            cons              ARG : scalar
          &        AL_TAIL : arg_list;

%EOF
```

C.4 Descriptions

The specification of initial descriptions is explained in Section A.1.4. An example of the output of parsing a generator from the massing configuration problem is given as follows:

```
sfc˙house
  SFC˙HOUSE: geom
            DU: [1] du
                DU˙HEIGHT: [2] universal
                DU˙LABEL: [3] fu
                DU˙LENGTH: integer
            FU: [3]
            COMMAND: command
            POSITION: centre
  ROOMROW: sfc˙roomrow˙two
          ENTITY˙B: geom
                    DU: du
                        DU˙HEIGHT: [2]
          ENTITY˙A: geom
                    DU: [1]
  PORCH: sfc˙porch
        SFC˙PORCH: [1]
```

Appendix D

Design and Implementation Details

In Part III, a software prototype, $\mathcal{FOLDS}$, is used to demonstrate the mixed-initiative interaction model for design space exploration. This appendix presents the design of the software components underlying $\mathcal{FOLDS}$. The domain objects and interactions between the software components are presented visually using the notation of the Unified Modelling Language [Erich Gamma & Vlissides 1995], UML. As with the description of the interaction model in Part II, the notation of UML permits a sufficient level of abstraction to describe the entities and their interactions without burdening the description with the symbol level implementation in C++.

The software components of the mixed-initiative exploration system, $\mathcal{FOLDS}$, developed in this thesis comprise the following:

- KRYOS
  KRYOS is an implementation of Carpenter's typed feature structures, designed and developed by Burrow [2003] in C++. The libraries implement the design space exploration machinery underlying the description formalism described in Chapter 2. KRYOS [Burrow 1999] comprises five libraries. Containers is a library of basic data structures. Patterns] a library implementing reusable design patterns [Erich Gamma & Vlissides 1995]. Orders is a library of order structures. FeatureStructures is a library comprising parsers, data structures for representing and reasoning with feature structures. TFSShells provides commandline shell programs for interaction with the exploration machinery, such as the incremental π-resolution of descriptions.

- the QT GUI toolkit
  The $\mathcal{FOLDS}$ interfaces and interaction framework are implemented using the multi-platform C++ graphical user interface toolkit, Qt[1]. The interactive components of the mixed-initiative model are implemented in C++ using the QT libraries for the front end and OpenGL for 3D graphical interaction. The QT GUI toolkit provides the basic building blocks for elementary

[1]Available from Troll Tech, http://www.troll.no/

user interface widgets, basic data structures and higher level application components for interface design. The Qt Class hierarchies, documentation and other details are available in the QT web site, *http://www.troll.no/*.

- Geometry visualisation is supported in *FOLDS* using the open source MindsEye 3D rendering and modeling package[2] based on the OpenGL [Neider, Davis & Woo 1993] 3D graphics library.

The design framework of *FOLDS* comprises the following software layers:

Facade The Facade layer of *FOLDS* enables the encapsulation of the components described above, namely, KRYOS, the QT GUI libraries and the MindsEye. For example, the **KrFacade** class provides a unified interface to elements of the KRYOS libraries.

Module The module layer is a grouping abstraction that provides a framework for developing user-centred views of the context. In *FOLDS* , a module comprises a context and a view, which implement specific functional classes associated with the mixed-initiative interaction model.

Explorer The Explorer is the front end layer of *FOLDS*. It serves as an aggregation of a collection of modules.

Each of the components of the above software layers of *FOLDS* are described in the following sections.

Facade

The **Facade** layer describes the interface to the components of the kernel of the design space exploration machinery. The design criteria for this interface is the use of the object-oriented *façade* design pattern as described in Erich Gamma & Vlissides [1995]. A façade pattern provides a unified interface to a set of interfaces in a subsystem and defines a higher-level interface that makes the subsystem easier to use. The façade pattern simplifies access to a related set of objects by providing a single *façade* object that all objects outside the set use to communicate with the set. The implementation of communication between the designer and the formal substrate take place through the singleton class, *KrFaçade*. The relationships between the formal components of KRYOS and *FOLDS* are shown in Figure D.1. The subcomponents of the **Facade** layer provide access to the KRYOS Feature Structure System.

The UML notation in this figure represents how the *KrFaçade* implements the façade design pattern and communicates with the substrate of KRYOS objects, *InheritanceHierarchy, ConstraintSystem, AppropSpecification*, comprising the **TypeSystem** constructed in Section 7.2 on Page 110.

[2]Available from http://mindseye.sourceforge.net/main.html

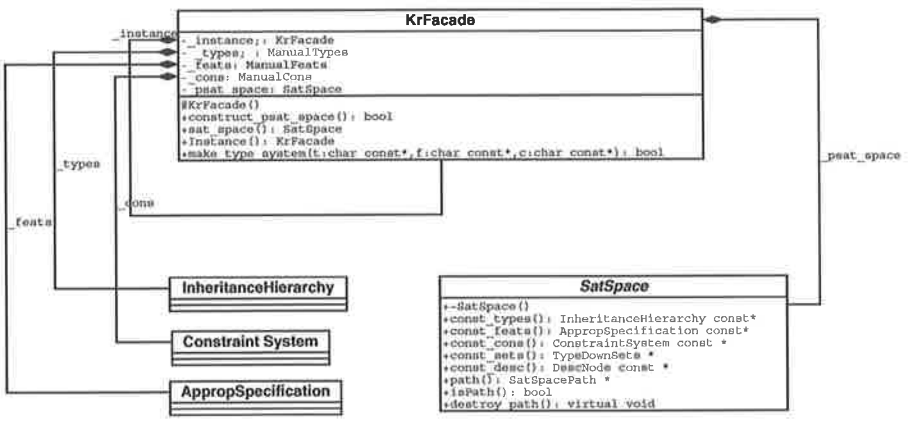


Figure D.1: The implementation of the domain layer constructs, their connection to $\mathcal{FOLDS}$, the communication between the designer and the generative design system take place through the singleton, *KrFaçade.* The notation in this figure shows how *KrFaçade* implements the façade design pattern and communicates with the underlying objects, *InheritanceHierarchy, ConstraintSystem, AppropSpecification, SatSpace* in the *kryos feature structures system.*

All communication with the Kryos feature structures system in $\mathcal{FOLDS}$ uses this single interface object, KrFacade, that encapsulates the underlying complexity of KRYOS within it. Any communication between the kryos libraries and $\mathcal{FOLDS}$ is done through a collection of messages to KrFacade, which communicates with the underlying implementation and returns the results to $\mathcal{FOLDS}$. The Façade pattern ensures that the KrFacade object acts as an intermediary for method calls between $\mathcal{FOLDS}$ objects and other external objects not known to the $\mathcal{FOLDS}$ objects. $\mathcal{FOLDS}$ sends the files to KRYOS for parsing. KRYOS parses the files and if the specification is error-free, creates a satisfier space corresponding to the problem domain and signals the designer to continue.

Module

The interfaces of $\mathcal{FOLDS}$ are organised into independent subsystems called *modules.* Modules are a grouping abstraction that aggregate the design space interaction machinery into specific functional entities, composing a *view* and a *context.* In $\mathcal{FOLDS}$, the ScModule Class is the basic root class in the dialogue interfaces. First, the ScModule connects the formal domain and the operations for exploration to the user. Second, the ScModule provides the base class for implementing the interaction between the resolution machinery and the user. Figure D.2 shows the composition of an ScModule class that provides the interface to the kernel facade classes and access to a loadable collection of prototype modules and a console.

Further, the ScModule implements operations for registering and unregistering a prototype module [Erich Gamma & Vlissides 1995, p 121]. This design makes the $\mathcal{FOLDS}$ system extensible and flexible. For example, external modules for evaluation and building performance tasks, can be dynamically incorporated into $\mathcal{FOLDS}$. Alternatively, the system can be enhanced simply by

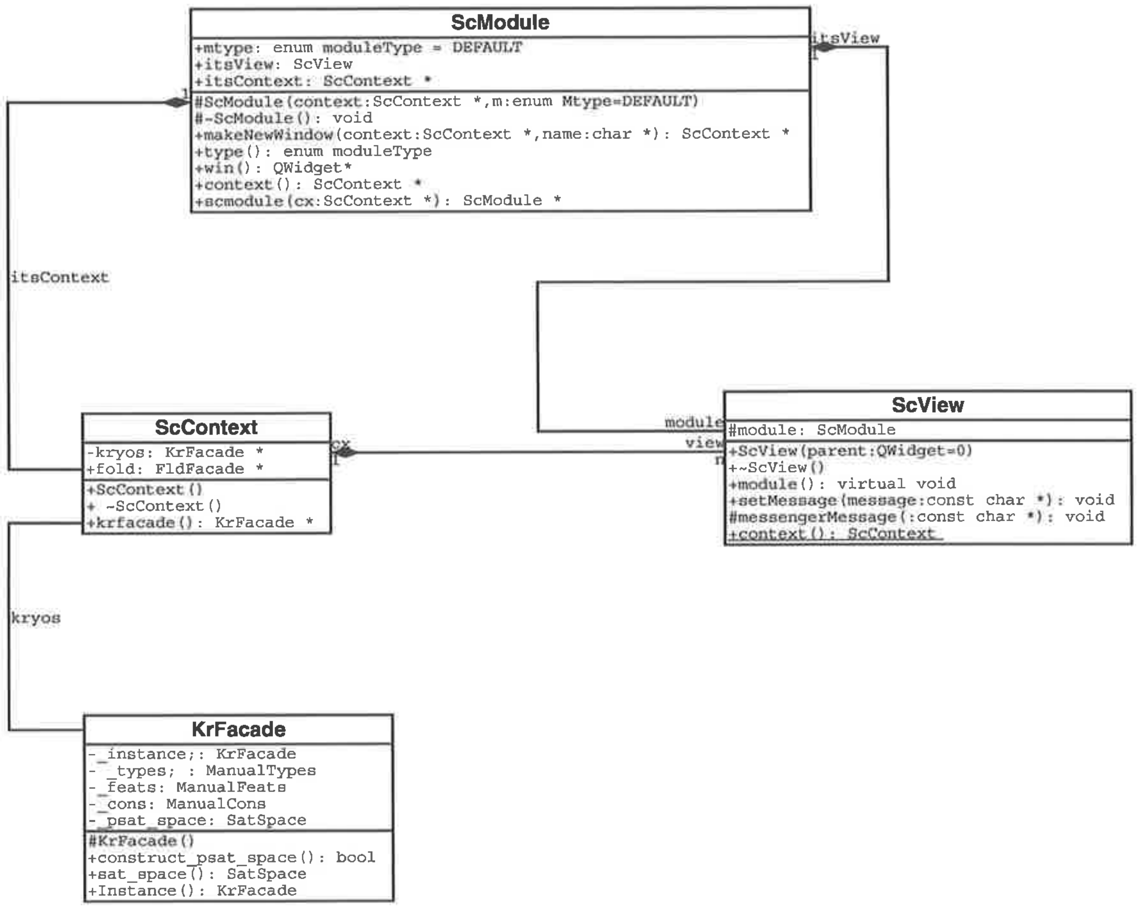


Figure D.2: The ScModule Class and its context

extending the functionality of the ScModule.

Explorer

The Explorer is the front end of *FOLDS*, comprising an aggregation of modules. Three modules of the explorer, ScConsoleModule, ScEntryModule and DesignSpaceModule are described here.

ScConsoleModule

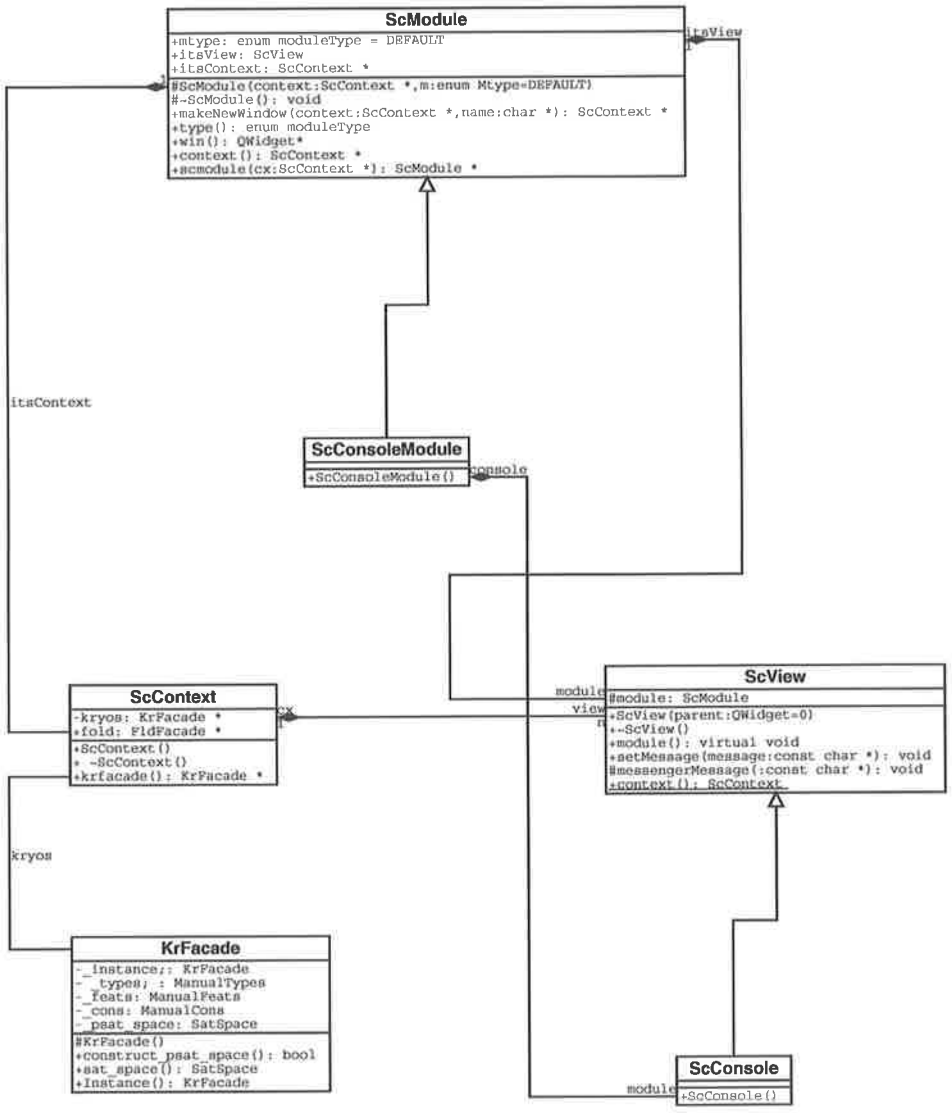


Figure D.3: The organisation of the ScConsoleModule class

The ScConsoleModule, shown in UML notation in Figure D.3, is the container module that loads all the context necessary for design space exploration. Other modules of SEED, as well as external libraries are available to the explorer through this module.

ScEntryModule

The entry module, shown in Figure D.4, provides provides the interface hooks to initialise the Kryos system, load in project definitions, construct the design space and begin the process of exploration.

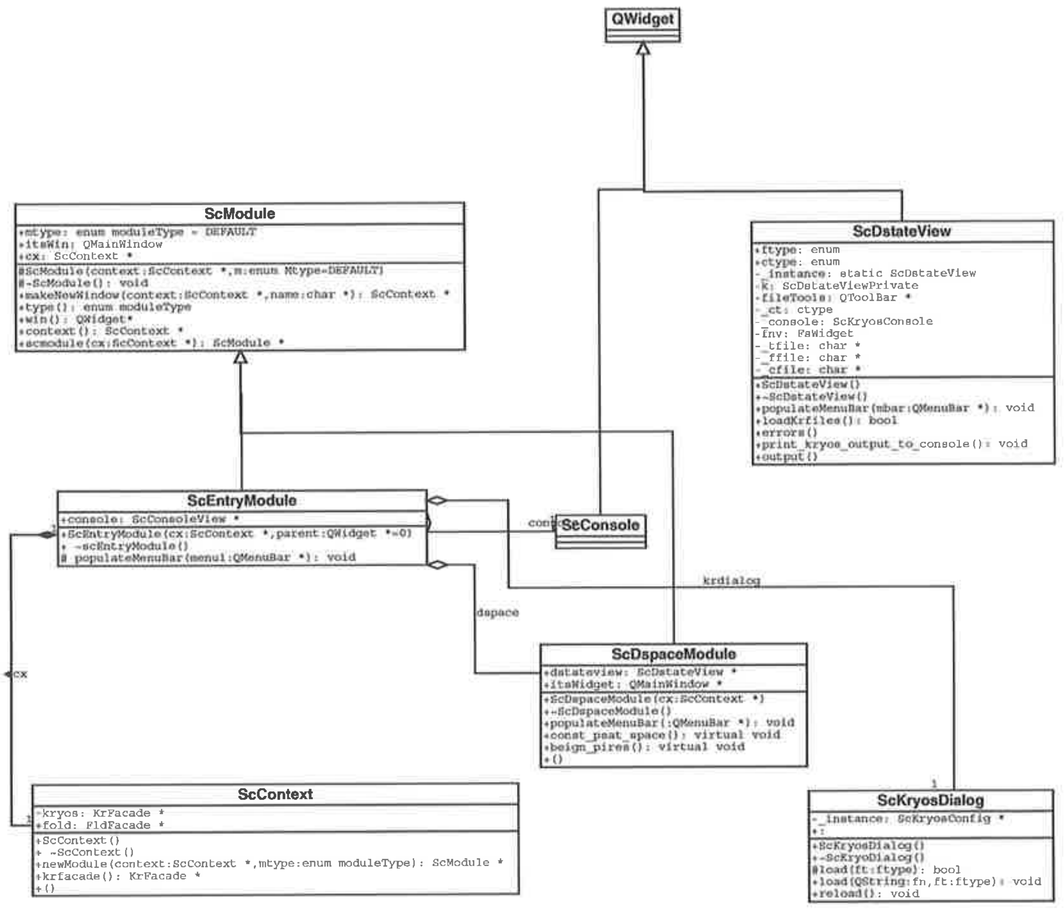
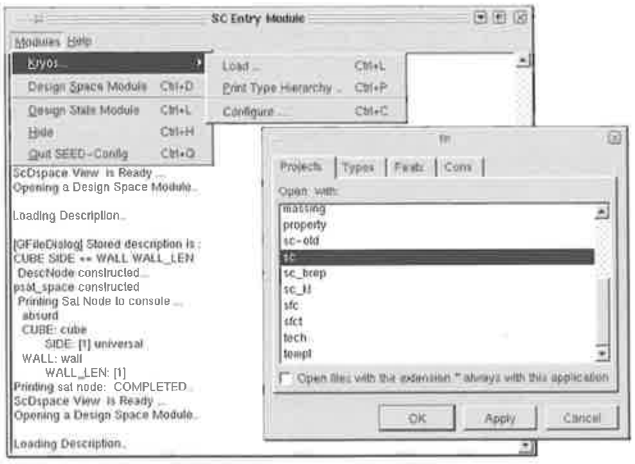

QWidget
ScModule
+mtype: enum moduleType = DEFAULT
+itsWin: QMainWindow
+cx: ScContext *
#ScModule(context:ScContext *,m:enum Mtype=DEFAULT)
#~ScModule(): void
+makeNewWindow(context:ScContext *,name:char *): ScContext *
+type(): enum moduleType
+win(): QWidget*
+context(): ScContext *
+scmodule(cx:ScContext *): ScModule *
ScDstateView
+ftype: enum
+ctype: enum
-_instance: static ScDstateView
-k: ScDstateViewPrivate
+fileTools: QToolBar *
-_ct: ctype
-_console: ScKryosConsole
-Env: FsWidget
-_tfile: char *
-_ffile: char *
-_cfile: char *
+ScDstateView()
+~ScDstateView()
+populateMenuBar(mbar:QMenuBar *): void
+loadKrfiles(): bool
+errors()
+print_kryos_output_to_console(): void
+output()
ScEntryModule
+console: ScConsoleView *
+ScEntryModule(cx:ScContext *,parent:QWidget *=0)
+ ~scEntryModule()
# populateMenuBar(menu1:QMenuBar *): void
con
ScConsole
krdialog
dspace
cx
1
ScDspaceModule
+dstateview: ScDstateView *
+itsWidget: QMainWindow *
+ScDspaceModule(cx:ScContext *)
+~ScDspaceModule()
+populateMenuBar(:QMenuBar *): void
+const_pest_space(): virtual void
+beign_pires(): virtual void
+()
ScContext
-kryos: KrFacade *
+fold: FldFacade *
+ScContext()
+ ~ScContext()
+newModule(context:ScContext *,mtype:enum moduleType): ScModule *
+krfacade(): KrFacade *
+()
ScKryosDialog
-_instance: ScKryosConfig *
+:
+ScKryosDialog()
+~ScKryoDialog()
#load(ft:ftype): bool
+load(QString:fn,ft:ftype): void
+reload(): void

Figure D.4: The organisation of the ScEntryModule class

The console module, shown in Figure D.5, described above provides a messaging interface for communication between *FOLDS* and KRYOS.

ScDesignSpaceModule

The software components described thus far implement the interfaces necessary to connect *FOLDS* with KRYOS, to create a flexible and extensible module design and to implement a communication mechanism between the user and the underlying formalism. The ScDesignSpaceModule shown in Figure D.6, implements the dialogue and task layers of the interaction model directly and permits mixed-initiative exploration. In this module, contexts and views that permit interaction with the visual notation, and interaction with the tasks of exploration are implemented.



```
ScDspace View is Ready ...
Opening a Design Space Module...

Loading Description...

[GFileDialog] Stored description is :
CUBE SIDE ++ WALL WALL_LEN
 DescNode constructed...
psat_space constructed
 Printing Sat Node to console ...
  absurd
  CUBE: cube
      SIDE: [1] universal
  WALL: wall
      WALL_LEN: [1]
Printing sat node:  COMPLETED...
ScDspace View is Ready ...
Opening a Design Space Module...

Loading Description...
```

Figure D.5: *FOLDS* interface to the Kryos Feature Structure System. The SC Entry Module provides provides the interface hooks to initialise the Kryos system, load in project definitions, construct the design space and begin the process of exploration. The console provides a messaging interface for communication between *FOLDS* and KRYOS.

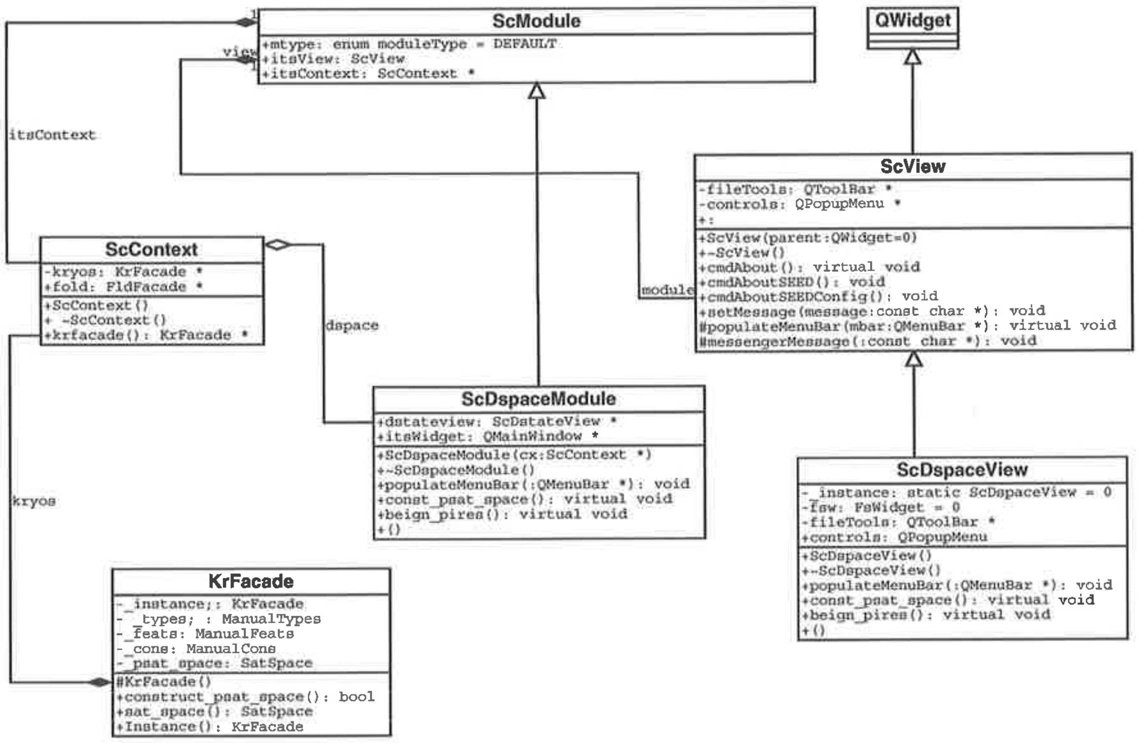
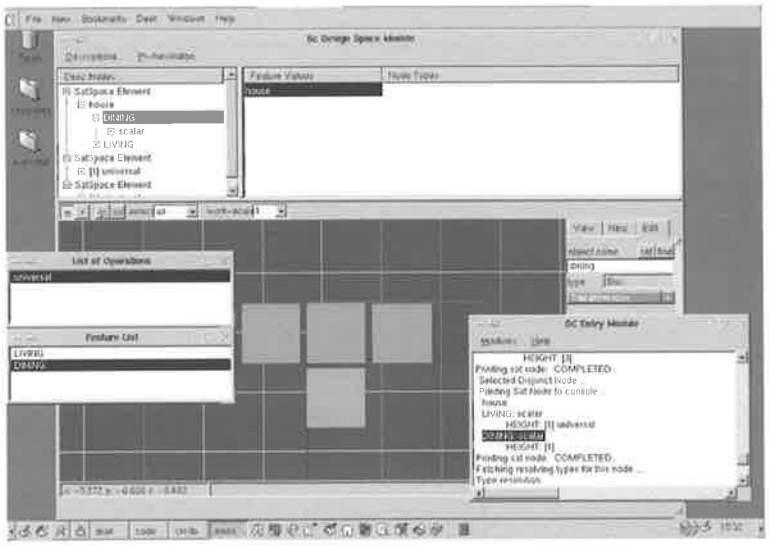

ScModule
+mtype: enum moduleType = DEFAULT
+itsView: ScView
+itsContext: ScContext *
1
view
1
QWidget
itsContext
ScView
-fileTools: QToolBar *
-controls: QPopupMenu *
+:
+ScView(parent:QWidget=0)
+~ScView()
+cmdAbout(): virtual void
+cmdAboutSEED(): void
+cmdAboutSEEDConfig(): void
+setMessage(message:const char *): void
#populateMenuBar(mbar:QMenuBar *): virtual void
#messengerMessage(:const char *): void
module
ScContext
-kryos: KrFacade *
+fold: FldFacade *
+ScContext()
+ ~ScContext()
+krfacade(): KrFacade *
dspace
ScDspaceModule
+dstateview: ScDstateView *
+itsWidget: QMainWindow *
+ScDspaceModule(cx:ScContext *)
+~ScDspaceModule()
+populateMenuBar(:QMenuBar *): void
+const_psat_space(): virtual void
+beign_pires(): virtual void
+()
ScDspaceView
-_instance: static ScDspaceView = 0
-fsw: FsWidget = 0
-fileTools: QToolBar *
+controls: QPopupMenu
+ScDspaceView()
+~ScDspaceView()
+populateMenuBar(:QMenuBar *): void
+const_psat_space(): virtual void
+beign_pires(): virtual void
+()
kryos
KrFacade
-_instance;: KrFacade
-_types; : ManualTypes
-_feats: ManualFeats
-_cons: ManualCons
-_psat_space: SatSpace
#KrFacade()
+construct_psat_space(): bool
+sat_space(): SatSpace
+Instance(): KrFacade

Figure D.6: The organisation of the ScDesignSpaceModule class

Figure D.7 shows the ScDesignSpaceModule, comprising views of the current state of exploration and interface constructs for supporting dialogue and task operations through mixed-initiative.



```
        HEIGHT: [3]
Printing sat node: COMPLETED
 Selected Disjunct Node ...
 Printing Sat Node to console ...
  house
  LIVING: scalar
        HEIGHT: [1] universal
  [illegible]
        HEIGHT: [1]
Printing sat node: COMPLETED
Fetching resolving types for this node ...
Type resolution
```

Figure D.7: The ScDesignSpaceModule in $\mathcal{FOLDS}$. Note that the visual notation for feature nodes is introduced in the top half of the window. The ScEntry module is shown in the lower right. The design space module includes a geometry viewer in the lower half of the window. As exploration progresses, support for exploration specific tasks are provided through type and feature operations.

Bibliography

Ait-Kaci, H. & Cosmo, R. D. [1993], Compiling order-sorted feature term unification, Technical Report PRL-TN-7, Digital Paris Research Laboratory.

Ait-Kaci, H., Podelski, A. & Smolka, G. [1992], A feature constraint system for logic programming with entailment, Technical Report PRL-RR-20, Digital Paris Research Laboratory.

Akin, O., Aygen, Z., Chang, T. W., Chien, S. F., Choi, B., Donia, M., Fenves, S. J., Flemming, U., Garrett, J. H., Gomez, N., Kiliccote, H., Rivard, H., Sen, R., Snyder, J., Tsai, W.-J., Woodbury, R. & Zhang, Y. [1997], 'SEED: A Software Environment to support the Early phases of building Design', The International Journal of Design Computing.
URL: *http://www.arch.usyd.edu.au/kcdc/journal/index.html*

Akin, O. & Sen, R. [1996], 'Navigation within a structured search space in layout problems', *Environment and Planning B: Planning and Design* **23**, 421–442.

Akin, O., Sen, R., Donia, M. & Zhang, Y. [1995], 'SEED-Pro: Computer assisted architectural programming in SEED', *ASCE Journal of Architectural Engineering* **1**(4), 153–161.

Allen, J. F. [1999], 'Mixed initiative interaction', *Proc. IEEE Intelligent Systems* **14**(6), 14–23.

Allen, J. F., Ferguson, G. & Schubert, L. K. [1996], Planning in complex worlds via mixed-initiative interaction, *in* 'Advanced Planning Technology: Technological Achievements of the ARPA/Rome Laboratory Planning Initiative', pp. 53–60.
URL: *http://www.cs.rochester.edu/research/trains/*

Amant, R. S. [1997*a*], Navigation and planning in a mixed initiative user interface, *in* 'Proceedings of the 14th National Conference on Artificial Intelligence (AAAI-97)', AAAI Press / MIT Press, Providence, Rhode Island, pp. 64–69.
URL: *citeseer.nj.nec.com/st97navigation.html*

Amant, R. S. [1997*b*], Navigation and planning in a mixed initiative user interface, *in* 'Proceedings of the 14th National Conference on Artificial Intelligence', AAAI-97, MIT Press, pp. 64–69.

Amant, R. S. & Cohen, P. R. [1997], Interaction with a mixed-initiative system for exploratory data analysis, *in* 'Proceedings of Intelligent User Interfaces', pp. 15–22.

Borgida, A., Brachman, R. J., McGuinness, D. L. & Resnick, L. A. [1989], CLASSIC: A structural data model for objects, *in* 'Proceedings of the 1989 ACM Special Interest Group on Management of Data (SIGMOD) International Conference on Management of Data', pp. 58–67.

Borning, A. [1977], Thinglab - an object-oriented system for building simulations using constraints, *in* 'Proceedings of the Fifth International Joint Conference on Artificial Intelligence', pp. 497–498.

Borning, A. [1981], 'The programming language aspects of thinglab, a constraint-oriented simulation laboratory', *ACM Transactions on Programming Languages and Systems* **3**(4), 353–387.

Burrow, A. [1999], *Design and Implementation of the* KRYOS *feature structure libraries*, University of Adelaide, Adelaide, Australia. Unpublished manual on feature structures libraries.

Burrow, A. L. [2003], Computational Design and Formal Languages, Unpublished PhD Thesis, Department of Computer Science, The University of Adelaide.

Burrow, A. L. & Woodbury, R. [1999], π-resolution and Design Space Exploration, *in* G. Augenbroe & C. Eastman, eds, 'Computers in Building: Proceedings of the CAADF'99 Conference', Eighth International Conference on Computer-Aided Design Futures, Kluwer Academic Publishers, pp. 291–308.

Burrow, A. & Woodbury, R. [2001], Design spaces–the forgotten artefact, *in* M. Burry, T. Dawson, J. Rollo & S. Datta, eds, 'Hand, Eye, Mind, Digital', Proceedings of the Third International Mathematics and Design Conference, The Mathematics and Design Society, pp. 56–62.

Burstein, M., Ferguson, G. & Allen, J. [2000], Integrating agent-based mixed-initiative control with an existing multi-agent planning system, Technical Report 729, Computer Science Dept., University of Rochester.
URL: *http://www.cs.rochester.edu/u/ferguson/papers/burstein-ferguson-allen-tr729.pdf*

Burstein, M. & McDermott, D. [1996], Issues in the development of human-computer mixed-initiative planning, *in* B. Gorayska & J. Mey, eds, 'Cognitive Technology', Elsevier Science, pp. 285–303.

Burstein, M., Mulvehill, A. & Deutsch, S. [1999], An approach to mixed-initiative management of heterogeneous software agent teams, *in* 'Thirty-second Annual Hawaii International Conference on System Sciences', Maui, Hawaii.
URL: *http://www.computer.org/proceedings/hicss/0001/00018/00018055.PDF*

Carbonell, J. R. [1970], 'AI in CAI: An artificial intelligence approach to computer-assisted instruction', *IEEE Transactions on Man-Machine Systems* **11**(4), 190–202.

Cardelli, L. & Wegner, P. [1985], 'On understanding types, data abstraction and polymorphism', *Computing Surveys* **17**(4), 471–522.

Carlson, C. [1993], Grammatical Programming: An algebraic approach to the Description of Design Spaces, PhD thesis, Carnegie Mellon University.

Carlson, C. [1994], Design space description formalisms : Discussion, *in* J. S. Gero & E. Tyugu, eds, 'Formal Design methods for CAD', TC5/WG5.2 Workshop on Formal Design methods for CAD, IFIP, Elsevier Science Publishers B.V., pp. 121–134.

Carlson, C., McKelvey, R. & Woodbury, R. [1991], 'An introduction to structure and structure grammars', *Planning and Design* **18**, 417–426.

Carlson, C. & Woodbury, R. [1994], 'Hands-on exploration of recursive patterns', *Languages of Design* **2**, 121–142.

Carpenter, B. [1992], *The Logic of Typed Feature Structures with applications to unification grammars, logic prorgams and constraint resolution*, Cambridge Tracts in Theoretical Computer Science, Cambridge University Press.

Cesta, A. & D'Aloisi, D. [1999], 'Mixed-Initiative Issues in an Agent-Based Meeting Scheduler', *User Modeling and User-Adapted Interaction* **9**(1-2), 45–78.
URL: *citeseer.nj.nec.com/199120.html*

Chang, T. [1999], Geometric Typed Feature Structures, PhD thesis, School of Architecture, Landscape Architecture and Urban Design, Adelaide University.

Chase, S. [1989], 'Shapes and shape grammars: from mathematical model to computer implementation', *Planning and Design* **16**(2), 215–241.

Chase, S. C. [1999], Grammar based design: Issues for user interaction models, *in* O. Ataman & J. Bermudez, eds, 'Media and Design Process, proceedings of ACADIA '99', pp. 198–210.

Chase, S. C. [2002], 'A model for user interaction in grammar-based design systems', *Automation in Construction* **11**(2), 161–172.

Chien, S. [1998], Supporting information navigation in generative design systems, PhD thesis, School of Architecture, Carnegie-Mellon University.

Chien, S. & Flemming, U. [1996], Design space navigation: An annotated bibliography, Technical Report No EDRC 48-37-96, Engineering Design Research Center, Carnegie-Mellon University, Pittsburgh, PA, USA.

Chien, S. & Flemming, U. [1997], Information navigation in generative design systems, *in* Y. Liu, J. Tsou & J. Hou, eds, 'CAADRIA 97', Vol. 2, Computer Aided Architectural Design Research in Asia, National Chia Tung University, Hsinchu, Taiwan, pp. 355–366.

Chu-Carroll, J. & Brown, M. K. [1997*a*], Initiative in collaborative interactions –its cues and effects, *in* 'Computational Models for Mixed Initiative Interaction', AAAI Spring Symposium Series, AAAI Press, Stanford University, pp. 16–22.
URL: *citeseer.nj.nec.com/9991.html*

Chu-Carroll, J. & Brown, M. K. [1997*b*], Tracking initiative in collaborative dialogue interactions, *in* 'Meeting of the Association for Computational Linguistics', pp. 262–270.
URL: *citeseer.nj.nec.com/82163.html*

Chu-Carroll, J. & Brown, M. K. [1998], 'An evidential model for tracking initiative in collaborative dialogue interactions', *User Modeling and User-Adapted Interaction (UMUAI)* **8**(3-4), 215–253.

Cohen, R., Allaby, C., Cumbaa, C., Fitzgerald, M., Ho, K., Hui, B., Latulipe, C., Lu, F., Moussa, N., Pooley, D., Qian, A. & Siddiqi, S. [1998], 'What is initiative?', *User Modeling and User-Adapted Interaction* **8**(3-4), 171–214.
URL: *citeseer.nj.nec.com/cohen98what.html*

Cooper, A. [1999], *About Face: The Essentials of User Interface Design*, Programmers Press.

Dalheimer, M. K. [1999], *Programming with Qt*, O'Reilly Verlag GMBH Co., KG, Köln, Germany.

Datta, S. [2002], Managing knowledge navigation in design with mixed-initiative dialogue, *in* Z. Turk & R. Scherer, eds, 'Proceedings of the fourth european conference on product and process modelling in the building and related industries: eWork and eBusiness in Architecture, Engineering and Construction', ECPPM, A.A. Balkema, The Netherlands, pp. 501–508.

Datta, S. & Woodbury, R. [1998], Reducing semantic distance in generative systems: A massing example, *in* T. Seebohm & S. V. Eyck, eds, 'Digital Design Studios : Do computers make a difference ?', Proceedings of the 1998 Association for Computer-Aided Design in Architecture Conference, Universal Printing, Albuquerque, NM, pp. 164–171.

Datta, S. & Woodbury, R. [2000], Proactivity and mixed-initiative in unfolding design spaces, *in* K. Bennett & S. Goss, eds, 'Workshop on Situated Activity Of Entities (AgentsHumans) Within Complex, Dynamic, Real-Time Environments (RealSimulated)', OZCHI2000 : Interfacing Reality in the New Millenium, pp. 15–16. submitted for publication.

Datta, S. & Woodbury, R. [2001], An approach to search and exploration through mixed-initiative, *in* S. C. J. Gero & M. Rosenman, eds, 'Proceedings of The Sixth Conference on Computer Aided Architectural Design Research in Asia', CAADRIA, Key Centre of design computing and cognition, Sydney, Australia, pp. 275–282.

Datta, S. & Woodbury, R. [2002], A Graphical Notation for Mixed-initiative dialogue with generative design systems, *in* J. S. Gero, ed., 'Artificial Intelligence in Design'02', AID, Kluwer Academic Publishers, The Netherlands, pp. 25–40.

Eggleston, R. [1999], Mixed-initiative transactions: A cognitive engineering approach to interface agent modeling, *in* 'Proceedings of the AAAI-99 Workshop on Mixed-Initiative Intelligence', AAAI Press, Orlando Fl.
URL: *http://www.cs.wright.edu/people/faculty/mcox/mii/papers/eggleston.pdf*

Erich Gamma, Richard Helm, R. J. & Vlissides, J. [1995], *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Publishing.

Feijo, B. & Lehtola, N. [1996], Reactive design agents in solid modelling, *in* J. S. Gero & F. Sudweeks, eds, 'Artificial Intelligence in Design '96', Kluwer Academic Publishers, pp. 61–75.

Fenves, S. J., Rivard, H. & Gomez, N. [1995], An information model for the preliminary design of buildings, Technical report, Engineering Design Research Center, Carnegie-Mellon University, Pittsburgh, PA.

Ferguson, G. & Allen, J. F. [1994], Arguing about plans: Plan representation and reasoning for mixed-initiative planning, *in* 'Proceedings of the 2nd International Conference on AI Planning Systems', AIPS-94, Chicago, IL, pp. 43–48.

Ferguson, G. & Allen, J. F. [1998], Trips: An integrated intelligent problem-solving assistant., *in* 'Proceedings of the Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference', AAAI/IAAI, The MIT Press, pp. 567–572.

Ferguson, G., Allen, J. & Miller, B. [1996], Trains-95: Towards a mixed-initiative planning assistant, *in* 'Proceedings of the 3rd International Conference on AI Planning Systems', AIPS-96, Edinburgh, Scotland, pp. 70–77.

Flemming, U. [1978], 'Wall representations of rectangular dissections and their use in automated space allocation', *Environment and Planning B: Planning and Design* **5**, 215–232.

Flemming, U. [1986], 'On the representation and generation of loosely-packed arrangements of rectangles', *Environment and Planning B: Planning and Design* **13**, 189–205.

Flemming, U. [1987*a*], 'More than the sum of parts: the grammar of queen anne houses', *Environment and Planning B: Planning and Design* **14**, 323–350.

Flemming, U. [1987*b*], The role of shape grammars in the analysis and creation of designs, *in* Y. Kalay, ed., 'Computability of Design', Principles of Computer-Aided Design, Wiley Interscience, New York, N.Y., chapter 12, pp. 245–272.

Flemming, U. [1990], Syntactic structures in architecture: Teaching composition with computer assistance, *in* M. McCullough & W. J. Mitchell, eds, 'The Electronic design studio : architectural knowledge and media in the computer era', MIT Press, Boston, chapter 2, pp. 31–48.

Flemming, U. & Chien, S.-F. [1995], 'Schematic layout design in the SEED environment', *ASCE Journal of Architectural Engineering* **1**(4), 162–169.

Flemming, U., Coyne, R. F., Glavin, T., Hsi, H. & Rychener, M. D. [1989], A generative expert system for the design of building layouts, Technical Report from 1989 Report Series, Engineering Design Research Center, Carnegie-Mellon University.

Flemming, U., Coyne, R., Glavin, T. & Rychener, M. [1988], A generative expert system for the design of building layouts - version 2, *in* J. Gero, ed., 'Artificial Intelligence in Engineering: Design', Elsevier, New York, N.Y., pp. 445–464.

Flemming, U., Coyne, R. & Woodbury, R. [1993], SEED: A Software Environment to support the Early phases in building Design, *in* 'Fourth International Symposium on Computer Aided Design in Architecture and Civil Engineering', Institut de Technologia de la Construccio de Catalunya, Barcelona.

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. & 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. & 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. & Woodbury, R. F. [1995], 'Software Environment to support Early phases in building Design, SEED: Overview', *ASCE Journal of Architectural Engineering* **1**(4), 147–152.

Fowler, M., Scott, K. & Jacobson, I. [1997], *UML Distilled: Applying the standard object modelling language*, Addison Wesley, Reading, MA.

Franz, B. & Jrg, S. [1994], Unification theory, *in* D. Gabbay, C. Hogger & J. Robinson, eds, 'Handbook of Logic in Artificial Intelligence and Logic Programming', Oxford University Press, Oxford, UK.

Freedman, R. [1999], Atlas: A plan manager for mixed-initiative, multi-model dialogue, *in* 'Proceedings of the AAAI-99 Workshop on Mixed-Initiative Intelligence', Orlando Fl.
URL: *http://www.cs.wright.edu/people/faculty/mcox/mii/papers/freedman.pdf*

Friedell, M. & Kochhar, S. [1991], 'Design and modeling with schema grammars', *Visual Languages and Computing* **2**, 247–273.

Fröhlich, M. & Werner, M. [1994], The graph visualization system DAVINCI - a user interface for applications, Technical Report 5/94, Universität Bremen.

Fröhlich, M. & Werner, M. [1995], 'Demonstration of the interactive graph visualization system DAVINCI', *Lecture notes in computer science* **894**.

G. Fischer, R. M. & Morch, A. [1988], 'Design environments for constructive and argumentative design', *Proceedings of CHI '89* pp. 269–275.

Galle, P. [1981], 'An algorithm for exhaustive generation of building floor plans', *Communications of the ACM* **24**, 813–825.

Gero, J. S. [1994*a*], Formal design methods for computer-aided design : closing discussion, *in* J. S. Gero & E. Tyugu, eds, 'Formal Design methods for CAD', TC5/WG5.2 Workshop on Formal Design methods for CAD, IFIP, Elsevier Science Publishers B.V., pp. 353–359.

Gero, J. S. [1994*b*], Towards a model of exploration in computer-aided design, *in* J. S. Gero & E. Tyugu, eds, 'Formal Design methods for CAD', TC5/WG5.2 Workshop on Formal Design methods for CAD, IFIP, Elsevier Science Publishers B.V., pp. 315–336.

Gero, J. S. & Kazakov, V. A. [1996], 'An exploration-based evolutionary model of a generative design process', *Microcomputers in Civil Engineering* **11**(3), 211–218.

Grice, H. [1989], *Studies in the way of words*, Harvard University Press.

Grice, H. P. [1975], Logic and conversation, *in* P. Cole & J. Morgan, eds, 'Syntax and Semantics', Vol. 3, Academic Press, pp. 41–58.

Gross, M. D., Ervin, S., Anderson, J. & Fleisher, A. [1988], 'Constraints: Knowledge representation in design', *Design Studies* **9**(3), 133–143.

Guinn, C. [1993], A computational model of dialogue initiative in collaborative discourse, *in* 'AAAI'93: Fall Symposium on Human-Computer Collaboration', Raleigh, NC, pp. 32–39.
URL: *http://citeseer.nj.nec.com/guinn93computational.html*

Guinn, C. I. [1996], Mechanisms for mixed-initiative human-computer collaborative discourse, *in* A. Joshi & M. Palmer, eds, 'Proceedings of the Thirty-Fourth Annual Meeting of the Association for Computational Linguistics', Morgan Kaufmann Publishers, San Francisco, pp. 278–285.
URL: *citeseer.nj.nec.com/56118.html*

Hakim, M. & Garrett, J. H. J. [1993], 'Using description logic for representing engineering design standards', *Journal of Engineering with Computers* **9**, 108–124.

Haller, S., McRoy, S. & Kobsa, A., eds [1999], *Computational Models of Mixed-Initiative Interaction*, Published collection of papers from the 1997 AAAI Spring Symposium, Kluwer Academic Publishers.

Harada, M. [1997], Discrete/Continuous Design Exploration by Direct Manipulation, PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, USA.

Harada, M., Witkin, A. & Baraff, D. [1995], Interactive physically-based manipulation of discrete/continuous models., *in* 'SIGGRAPH '95 Conference Proceedings', Vol. 29, ACM Siggraph, ACM, pp. 199–208.

Hartrum, T. & DeLoach, S. A. [1999], Design issues for mixed-initiative agent systems, *in* 'Proceedings of the AAAI-99 Workshop on Mixed-Initiative Intelligence', Orlando Fl.
URL: *http://www.cs.wright.edu/people/faculty/mcox/mii/papers/hartrum.pdf*

Heisserman, J. [1991], Generative Geometric Design and Boundary Solid Grammars, PhD thesis, Department of Architecture, College of Fine Arts, Carnegie-Mellon University.

Heisserman, J. [1994], 'Generative geometric design', *IEEE Computer Graphics and Applications* **14**(2), 37–45.

Heisserman, J. & Woodbury, R. [1993], Geometric design with boundary solid grammars, *in* J. Gero & F. S. (eds.), eds, 'In Formal Design Methods for CAD', North-Holland, Amsterdam, pp. 79–100.

Horvitz, E. [1999], Principles of mixed-initiative user interfaces, *in* 'Proceedings of CHI '99', ACM SIGCHI Conference on Human Factors in Computing Systems, ACM press, Pittsburgh, PA, pp. 159–166.

Hudson, S. E. & Yeatts, A. K. [1991], Smoothly integrating rule-based techniques into a direct manipulation interface builder, *in* 'Fourth Annual Symposium on User Interface Software and Technology', Vol. 4 of *UIST*, pp. 145–153.

Hybs, I. & Gero, J. S. [1992], 'An evolutionary process model of design', *Design Studies* **13**(3), 273–290.

Ishizaki, M., Crocker, M. & Mellish, C. [1999], 'Mixed-initiative dialogue using computer dialogue simulation', *User Modeling and User-Adapted Interaction* **9**(1-2), 79–91.

Jacobson, I., Booch, G. & Rumbaugh, J. [1998], *The Unified Modeling Language User Guide*, Addison Wesley, Reading, MA.

Kasper, R. T. & Rounds, W. C. [1990], 'The logic of unification in grammar', *Linguistics and Philosophy* **13**(1), 35–58.

Kiefer, B. & Fettig, T. [1995], FEGRAMED—an interactive graphics editor for feature structures, Research Report RR-95-06, German Research Center for Artificial Intelligence (DFKI), Saarbrücken, Germany.

Kitano, H. & Ess-Dykema, C. V. [1991], Toward a plan-based understanding model for mixed-initiative dialogues, *in* 'Meeting of the Association for Computational Linguistics', pp. 25–32.

Klein, E. & Pineda, L. A. [1990], Semantics and graphical information, *in* 'Proceedings of the International Conference on Human-Computer Interaction-INTERACT '90', Elsevier, New York, pp. 485–491.
URL: *www.hcrc.ed.ac.uk/Site/KLEIE9.html*

Knight, K. [1989], 'Unification : A multidisciplinary survey', *ACM Computing Surveys* **21**(1), 93–124.

Kochhar, S. [1994], 'A paradigm for human-computer cooperation in design', *Computer Graphics and Applications* **17**(16), 54–65.

Krishnamurthy, R. & Stouffs, R. [1993], Spatial grammars : Motivation, comparision and new results, *in* U.Flemming & S. VanWyk, eds, 'CAAD Futures '93: Proceedings of the Fifth International Conference on Computer-Aided Architectural Design Futures', Carnegie Mellon University, Pittsburgh, PA, Elsevier, Pittsburgh, PA, pp. 57–74.

Krishnamurti, R. [1980], 'The arithmetic of shapes', *Environment and Planning B: Planning and Design* **7**, 463–484.

Krishnamurti, R. [1992], 'The arithmetic of maximal planes', *Environment and Planning B: Planning and Design* **19**, 431–464.

Krishnamurti, R. & Earl, C. F. [1992], 'Shape recognition in three dimensions', *Environment and Planning B: Planning and Design* **19**, 585–603.

Lambrix, P. [1996], Part-Whole Reasoning in Description Logics, PhD thesis, Lingköping University.

Lester, J., Stone, B. & Stelling, G. D. [1999], 'Lifelike pedagogical agents for mixed-initiative problem solving in constructivist learning environments', *User Modeling and User-Adapted Interaction* **9**(1-2), 1–44.
URL: *citeseer.nj.nec.com/347209.html*

Lynch, K. [1960], *The Image of the City*, MIT Press, Cambridge, Mass.

McCall, R., Fischer, G. & Morch, A. [1990], Supporting reflection-in-action in the Janus Design Environment, *in* 'The Electronic Design Studio', MIT Press, Cambridge, MA, chapter 17, pp. 247–260.

Minsky, M. [1975], A framework for representing knowledge, *in* P. H. Winston, ed., 'The Psychology of Computer Vision', McGraw-Hill, New York, pp. 211–277.

Mitchell, W. J. [1977], *Computer-Aided Architectural Design*, Petrocelli/Charter, New-York.

Mitchell, W. J. [1990], *The Logic of Architecture*, MIT Press, Cambridge, MA., USA.

Mitchell, W., Steadman, J. & Liggett, R. [1976], 'Synthesis and optimization of small rectangular floor plans', *Environment and Planning B: Planning and Design* **3**, 37–70.

Neider, J., Davis, T. & Woo, M. [1993], *OpenGL Programming Guide*, second edition edn, OpenGL Architecture Review Board, Addison-Wesley, Reading, Massachusetts.

Newell, A. [1982], 'The Knowledge Level', *Artificial Intelligence* **18**(1), 87–127.

Newell, A. & Simon, H. A. [1972], *Human Problem Solving*, Prentice-Hall Inc., Englewood Cliffs, NJ.

Norman, D. [1988], *Design of Everyday Things*, Basic Books, New York.

Novick, D. [1988], Control of mixed-initiative discourse through meta-locutionary acts: A computational model., PhD thesis, Department of Computer and Information Science, University of Oregon.

Novick, D. & Sutton, S. [1994], An empirical model of acknowledgment for spoken-language systems, *in* 'Proceedings of ACL-94', pp. 96–101.

Novick, D. & Sutton, S. [1997], What is mixed-initiative interaction?, *in* 'Computational Models for Mixed Initiative Interaction', number SS-97-04 *in* '1999 AAAI Spring Symposium', AAAI Press, Stanford University, pp. 114–116.

Patel-Schneider, P. F., McGuinness, D. L., Brachman, R. J., Resnick, L. A. & Borgida, A. [1991], 'The CLASSIC knowledge representation system: Guiding principles and implementation rationale', *SIGART:ACM Special Interest Group on Artificial Intelligence* **2**(3), 108–113.

Piela, P. [1989], ASCEND, An Object-Oriented Computer Environment for Modeling and Analysis, PhD thesis, Department of Chemical Engineering, Carnegie-Mellon University.

Piela, P., McKelvey, R. & Westerberg, A. [1993], 'An introduction to the ASCEND modeling system: It's language and interactive environment', *Journal of Magagement information system* **9**(3), 91–121.

Pollard, C. J. & Moshier, M. [1990], Unifying partial descriptions of sets, *in* P. Hanson, ed., 'Information, Language and Cognition', Vol. 1 of *Vancouver Studies in Cognitive Science*, Oxford University Press, pp. 285–322.

Pollard, C. & Sag, I. [1987], Information-based Syntax and Semantics, *in* 'Volume 1: Fundamentals', Vol. 13 of *CSLI Lecture Note Series*, Center for the Study of Language and Information, Stanford University, Stanford, CA.

Poon, J. & Maher, M. L. [1996], Emergent behaviour in co-evolutionary design, *in* J. S. Gero & F. Sudweeks, eds, 'Artificial Intelligence in Design '96', Kluwer Academic Publishers, pp. 703–722.

Quadrel, R. [1991], Asynchronous Design Environments: Architecture and Behaviour, PhD thesis, Department of Architecture, Carnegie-Mellon University.

Rich, C. & Sidner, C. L. [1997], COLLAGEN: When agents collaborate with people, *in* W. L. Johnson & B. Hayes-Roth, eds, 'Proceedings of the First International Conference on Autonomous Agents (Agents'97)', ACM Press, New York, pp. 284–291.
URL: *citeseer.nj.nec.com/rich96collagen.html*

Rich, C. & Sidner, C. L. [1998], 'COLLAGEN: A collaboration manager for software interface agents', *User Modeling and User-Adapted Interaction* **8**(3-4), 315–350.
URL: *citeseer.nj.nec.com/rich98collagen.html*

Rittel, H. & Webber, M. [1984], Planning problems are wicked problems, *in* N. Cross, ed., 'Developments in Design Methodology', John Wiley Sons, pp. 135–144.

Schön, D. A. [1983], *The Reflective Practitioner : How Professionals Think in Action*, Basic Books, New York.

Schön, D. A. [1988], 'Designing: Rules, Types, and Worlds', *Design Studies* **9**(3), 181–190.

Schön, D. A. & Wiggins, G. [1992], 'Kinds of seeing and their functions in designing', *Design Studies* **13**(2), 135–156.

Schulte, C. [1997], Oz Explorer: A visual constraint programming tool, *in* L. Naish, ed., 'Proceedings of the Fourteenth International Conference on Logic Programming', The MIT Press, Leuven, Belgium, pp. 286–300.

Shieber, S. [1984], The design of a computer language for linguistic information, *in* 'Proceedings of the 10th International Conference on Computational Linguistics', Stanford, pp. 362–366.

Shieber, S. M. [1986], *An introduction to Unification-Based Approaches to Grammar*, CSLI Lecture Notes 4, Stanford, CSLI: Center for the Study of Language and Information, Stanford, CA.

Shieber, S. M., Uszkoreit, H., Pereira, F. C. N., Robinson, J. J. & Tyson, M. [1983], The formalism and implementation of patr-ii, *in* B. Grosz & M. Stickel, eds, 'Research on Interactive Acquisition and Use of Knowledge', SRI Final Report 1894, Artificial Intelligence Center, SRI International, Menlo Park, California.

Shih, S. G. & Schmitt, G. [1994], 'The use of post interpretation for grammar based generative systems', *in Formal Design Methods for CAD IFIP WG5.2 J.S. Gero and E. Tyugu(editors)* **B**(18), 101–113.

Shneiderman, B. [1982], 'The future of interactive systems and the emergence of direct manipulation', *Behaviour and Information Technology* **1**(1), 237–256.

Shneiderman, B. [1983], 'Direct manipulation: a step beyond programming languages', *IEEE Computer* **16**(8), 57–69.

Shneiderman, B. [1997], Direct manipulation for comprehensible, predictable, and controllable user interfaces, *in* 'Proceedings of IUI97', International Conference on Intelligent User Interfaces, pp. 33–39.

Simon, H. [1973], 'The structure of ill-structured problems', *Artificial Intelligence* **4**, 181–201.

Simon, H. A. [1969], *The sciences of the artificial*, Karl Taylor Compton lectures, 1968 M.I.T. Press, Cambridge, Mass.

Simon, H. A. [1975], Style in design, *in* C. M. Eastman, ed., 'Spatial Synthesis in Computer-Aided Building Design', Applied Science Publishers.

Smith, R. [1991], A Computational Model of Expectation-Driven Mixed-Initiative Dialog Processing, PhD thesis, Duke University.

Smith, R. & Hipp, R. D. [1994], *Spoken Natural Language Dialog Systems: A Practical Approach*, Oxford University Press.

Smithers, T. [1992], Design as exploration : puzzle-making and puzzle-solving, *in* J. S. Gero, ed., 'Workshop on Search-based and Exploration-based models of Design Process', Artificial Intelligence in Design '92, Carnegie-Mellon University, Pittsburgh, pp. 1–21.

Smithers, T. [1994], Exploration in design : Discussion, research issues, *in* J. S. Gero & E. Tyugu, eds, 'Formal Design methods for CAD', TC5/WG5.2 Workshop on Formal Design methods for CAD, IFIP, Elsevier Science Publishers B.V., pp. 337–350.

Smithers, T. [1996], On knowledge level theories of design process, *in* J. S. Gero & F. Sudweeks, eds, 'Artificial Intelligence in Design '96', Kluwer Academic Publishers, pp. 561–579.

Smithers, T. [1998], Towards a knowledge level theory of design process, *in* J. S. Gero & F. Sudweeks, eds, 'Artificial Intelligence in Design '98', Kluwer Academic Publishers, pp. 2–22.

Smithers, T. [2000], Designing a font to test a theory, *in* J. S. Gero, ed., 'Artificial Intelligence in Design '00', Kluwer Academic Publishers, pp. 3–22.

Smithers, T. [2002], Synthesis in Designing, *in* J. S. Gero, ed., 'Artificial Intelligence in Design '02', Kluwer Academic Publishers, pp. 3–24.

Smithers, T., Corne, D. & Ross, P. [1994], On computing exploration and solving design problems, *in* J. S. Gero & E. Tyugu, eds, 'Formal Design methods for CAD', TC5/WG5.2 Workshop on Formal Design methods for CAD, IFIP, Elsevier Science Publishers B.V., pp. 293–313.

Smolka, G. & Aït-Kaci, H. [1989], 'Inheritance hierarchies: Semantics and unification', *Journal of Symbolic Computation* **7**, 342–370.

Snyder, J., Aygen, Z., Flemming, U. & Tsai, J. [1995], 'SPROUT– a modeling language for SEED', *ASCE Journal of Architectural Engineering* **1**(4), 195–203.

Stiny, G. [1980], 'Introduction to shape and shape grammars', *Environment and Planning B: Planning and Design* **7**(3), 343–352.

Stiny, G. & March, L. [1981], 'Design machines', *Environment and Planning B: Planning and Design* **8**(3), 241–244.

Stiny, G. & Mitchell, W. [1978*a*], 'Counting palladian plans', *Environment and Planning B: Planning and Design* **5**, 189–198.

Stiny, G. & Mitchell, W. J. [1978*b*], 'The Palladian Grammar', *Environment and Planning B: Planning and Design* **5**, 5–18.

Stouffs, R. [1994], The Algebra of Shapes, PhD thesis, Carnegie Mellon University, Pittsburgh, PA 15213.

Sutherland, I. E. [1963], Sketchpad—a man-machine graphical communication system, *in* 'Proceedings of the Spring Joint Computer Conference', Vol. 23 of *IFIPS*, pp. 328–346.

Tapia, M. [1999], 'A visual implementation of a shape grammar system', *Environment and Planning B: Planning and Design* **26**(1), 59–73.

Tapia, M. A. [1996], From Shape to Style Shape Grammars: Issues in Representation and Computation, Presentation and Selection, PhD thesis, Department of Computer Science, University of Toronto.

Tecuci, G., Boicu, M., Wright, K. & Lee, S. [1999], Mixed-initiative development of knowledge bases, *in* 'Proceedings of the AAAI-99 Workshop on Mixed-Initiative Intelligence', AAAI Press, Orlando, Florida.
URL: *http://lalab.gmu.edu/publications/data/MIDKB-sent1999.pdf*

Tsai, M., Reiher, P. & Popek, J. [1999], Baby steps from GUI towards dialogue: Mixed-initiative computerese, *in* 'Proceedings of the AAAI-99 Workshop on Mixed-Initiative Intelligence', AAAI Press, Orlando Fl.
URL: *http://www.cs.wright.edu/people/faculty/mcox/mii/papers/tsai.pdf*

Veloso, M. M. [1996], Towards mixed-initiative rationale-supported planning, *in* A. Tate, ed., 'Advanced planning technology', AAAI Press, Menlo Park, CA, pp. 277–282.

Veloso, M. M., Mulvehill, A. M. & Cox, M. T. [1997], Rationale-supported mixed-initiative case-based planning, *in* 'Innovative Applications of Artificial Intelligence', Proceedings of IAAI-97, Providence, RI, pp. 1072–1077.

Walker, M. & Whittaker, S. [1990], 'Mixed initiative in dialogue: An investigation into discourse segmentation', *Proceedings of the 28th Annual Meeting of the Association for Computational Linguistics* pp. 70–78.

Weitzman, L. & Wittenburg, K. [1993], Relational grammars for interactive design, *in* 'IEEE/CS Workshop on Visual Languages', Institute of Electrical and Electronics Engineers, Loa Alamitos, California: IEEE Computer Society Press, Bergen, Norway, pp. 4–11.

Wexelblat, A. [1999], Footprints: Interaction History for Digital Objects, PhD thesis, MIT Program in Media Arts & Sciences.

Wexelblat, A. & Maes, P. [1999], Footprints: History-rich tools for information foraging, *in* M. Williams, G. Altom, M. Ehrlich & K. Newman, eds, 'Proceedings of the Conference on Human Factors in Computing Systems (CHI-99)', ACM Press, pp. 270–277.

Whittaker, S. & Stenton, P. [1988], 'Cues and control in expert-client dialogues', *Proceedings of the 26th Annual Meeting of the Association for Computational Linguistics* pp. 123–130.

Winograd, T. & Flores, F. [1987], *Understanding Computers and Cognition: A New Foundation for Design*, Addison-Wesley, Reading, MA.

Witkin, A., Fleischer, K. & Barr, A. [1987], 'Energy constraints on parameterized models', *Computer Graphics* **21**(4), 225–232.

Witkin, A., Gleicher, M. & Welch, W. [1990], 'Interactive dynamics', *Computer Graphics, Proc. 1990 Symposium on 3-D Interactive Graphics* **24**(2), 11–21.

Woodbury, R., Burrow, A., Datta, S. & Chang, T. [1999], 'Typed feature structures and design space exploration', *Artificial Intelligence in Design, Engineering and Manufacturing* **13**(4), 287–302. Special Issue on Generative Design Systems.

Woodbury, R., Carlson, C. & Heisserman, J. [1988], Geometric Search Spaces in Design, *in* 'Workshop on Intelligent CAD', IFIP WG5.2, Cambridge, England, pp. 490–492.

Woodbury, R., Datta, S. & Burrow, A. [2000], Erasure in design space exploration, *in* J. Gero, ed., 'Artificial Intelligence in Design'00', Artificial Intelligence in Design, Kluwer Academic Publishers, pp. 521–544.

Woodbury, R. F. & Chang, T.-W. [1995*a*], Building enclosures using SEED-Config, *in* M. Tan & R. Teh, eds, 'The Global Design Studio: Proceedings of the Sixth International Conference on Computer-Aided Architectural Design Futures', CAADFutures 95 Conference, Singapore, pp. 49–58.

Woodbury, R. F. & Chang, T.-W. [1995*b*], 'Massing and enclosure design with SEED-Config', *ASCE Journal of Architectural Engineering* **1**(4), 170–178.

Woodbury, R. F., Radford, A. D., Taplin, P. N. & Coppins, S. A. [1992], Tartan worlds: A generative symbol grammar system, *in* D. Noble & K. Kensek, eds, 'ACADIA 92', Charleston, SC, pp. 211–220.

Woodbury, R., Flemming, U., Coyne, R., Fenves, S. & Garrett, J. [1995], The SEED project: A Software Environment to support the Early phases in building Design, *in* G. F. Forsyth & M. Ali, eds, 'Industrial and Engineering Applications of Artificial Intelligence and Expert Systems: Proceedings of the Eighth International Conference', IEA/AIE, Gordon and Breach Publishers, Melbourne, Australia, pp. 781–786.

Woodbury, R. & Griffith, E. [1993], Layouts, solids, grammar interpreters and firestations, *in* U. Flemming & S. VanWyk, eds, 'CAAD Futures 93', Carnegie Mellon University, Pittsburgh, PA, Elsevier, Pittsburgh, PA, pp. 75–90.

Zeller, A. [1997], Configuration Management with Version Sets - A Unified Software Versioning Model and its Applications, PhD thesis, TU Braunschweig.

Zeller, A. & Snelting, G. [1995], Handling version sets through feature logic, *in* W. Schfer & P. Botella, eds, 'Proc. 5th European Software Engineering Conference (ESEC)', Vol. Vol. 989 of LNCS, pp. 191–204.