A Generalised Framework for the Design and Construction of Integrated Design Systems

Robert Wilton Amor

A thesis submitted in fulfilment of the requirements for the degree of

Doctor of Philosophy in Computer Science

University of Auckland May 1997

Abstract

The building industry employs a significant percentage of the workforce of any country, and encompasses a considerable proportion of a country's GDP. Despite that, IT tools used in the design and management of a building project are still fairly crude. Many projects have been undertaken to develop IT-based solutions to support the architecture, engineering, and construction domains (A/E/C), but little effort has gone into the tools required to support these development activities. This is the area in which this thesis concentrates.

To develop a schema representing some subsystem of a building it is necessary to have support tools which enhance the modeller's environment. The current state of the art, a replicated paper based approach, is ineffective at guaranteeing the consistency and validity of large schemas. In this thesis, a more appropriate environment is developed and demonstrated. This provides multiple overlapping views of the developing schema, with guaranteed consistency between all views, the ability for many modellers to work on the schema, and links to related aspects.

The array of schemas being developed for the A/E/C domains contain overlaps of information, though often in different representations. To enable the full use and correct transfer of information between schemas, mappings between their representations need to be defined. This thesis develops a comprehensive mapping language which describes bidirectional mappings between schemas. An automated system has been constructed which can take a mapping specification and manage the updates and consistency of data in models corresponding to the mapped schemas.

To manage the development of environments described above, as well as the finished integrated environments proposed, it is necessary to manage and control the supported processes. A notation is developed to allow this control to be defined, and an implementation is provided to demonstrate how a project can be managed.

The end result of the thesis is a set of notations and associated tools which support all aspects of the development and implementation of integrated design environments. The resultant development environment greatly raises the level of support for developers over that offered by current tools, for all aspects of specification, consistency, testing, validation, implementation, and coordination between developers.

Acknowledgments

I would like to thank my supervisor, Dr John Hosking, for his efforts in keeping me on track, his enthusiasm for the ideals that are encompassed in this research, his support in funding rounds, and for helping with yet another paper.

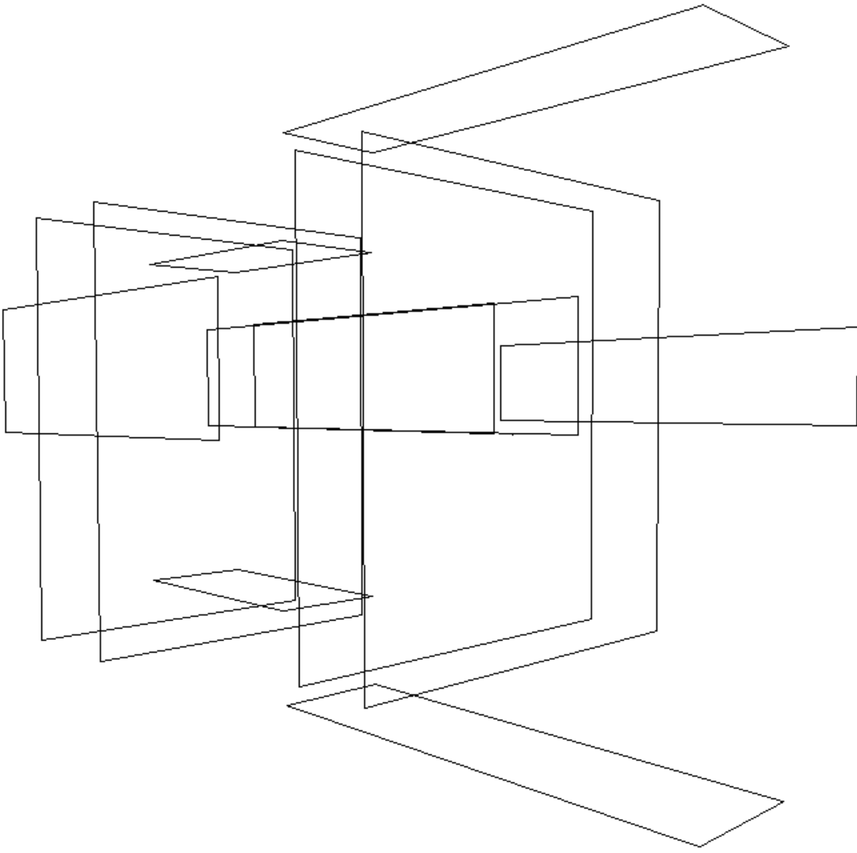
Thanks to my wife Kath for her love and support and making sure it all happened. Now it is her turn and I can get my own back! Some thanks go to our two lovely kids, the arrival of Sylvia (2/2/96) almost got me finished but then made it harder, then the arrival of Roman (21/4/97) almost got me finished again, thankfully Kath's parents have been here for the last month and enabled me to finish this thesis without abandoning Kath to all the child-minding. A word of warning, never leave university with two weeks work left on a thesis. A year and a half later, two children later, and another country later I am just thankful to get this thesis out the door. Also thanks to parents and siblings for their support and encouragement over the years, yes, I know, it is finally time to get a job.

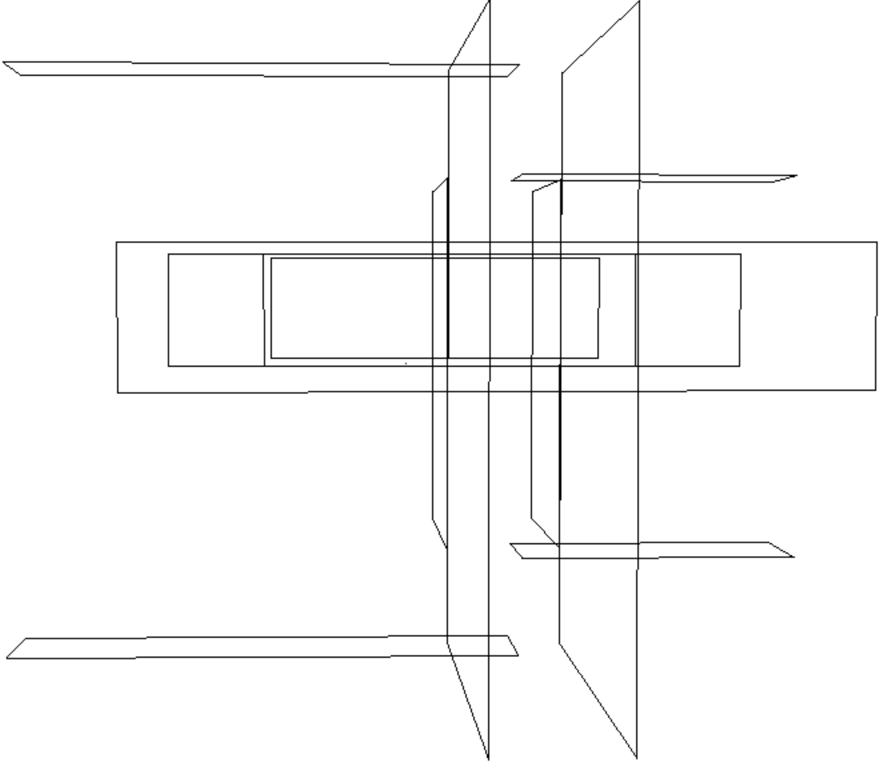
Financial support was provided by a University of Auckland PhD scholarship, a Department of Computer Science scholarship, and a working spouse, many thanks to all of them. I would also like to thank the department for their unfailing support with travel funds for conference trips. The COMBINE group partially sponsored my stay at TU Delft for 6 months and travel costs were partially covered by a grant from the University of Auckland Graduate Research Fund. I found this stay of immense value to my work and thank the groups involved for making it possible.

Thanks also to the PhD students and staff in the department for lively conversations and ears willing to have ideas bounced off. Special thanks to John Grundy, Paul Qualtrough and Achim Schneider, I will get you guys juggling yet! I found the environment at TU Delft very stimulating, you guys have a great group there. Special thanks to Marcel Verhoef, Wouter Rombouts, Godfried Augenbroe, Roel Schipper, and all the others in the group, for making us feel so welcome during our stay and introducing us to the Dutch way. We are still hooked on stroopwafels and wickedly strong coffee!

Thanks to the EU funded COMBINE group for taking the time to listen to this outsider's views. I found the project very stimulating and the flow of ideas exhilarating. This thesis started with a two month conference and university tour of the USA. Thanks to the following groups for making me feel so welcome and sparing the time to explain your projects: LBL (Greg Ward, Steve Selkowitz, Bill Carroll, Fred Winkelmann, and Kostas Papamichael), UCB (Alice Agogino), Stanford (Mark Clayton and Paul Teicholz), Cal Poly (Jens Pohl), UCLA (Chuck Eastman, Hisham Assal, and Scott Chase), MIT (Duvvru Sriram and Albert Wong), CMU (James Garrett and Skip Van Wyk), University of Michigan (James Turner), University of Wisconsin.

Incorrect Mappings as Artwork





Contents

Abstract i
Acknowledgments iii

List of Figures xiii
List of Tables xvii

1 Introduction 1
1.1 The Problem Domain 1
1.2 Related Research 3
1.3 COMBINE Project 4
1.4 Underlying Framework 6
1.5 Research Objectives 9
1.5.1 Formalism to specify mappings 10
1.5.2 Formalism to specify project and flow of control 10
1.5.3 Comprehensive tool set to support model specification 10
1.5.4 Inter-model mapping utilising mapping specifications 10
1.5.5 Control system utilising project and flow of control specification 11
1.6 Description of Example 11
1.6.1 The Snart language 12
1.6.2 PlanEntry 13
1.6.3 FaceEditor 15
1.6.4 VISION-3D 16
1.6.5 ThermalDesigner 16
1.6.6 IDM 17
1.6.7 Correspondences between models 17
1.7 Outline of Thesis 19

2 The Project Development Environment 21
2.1 Structure and Requirements 22
2.2 Schema Modelling and Development 23
2.2.1 Requirements of a schema modelling environment 24
2.2.2 IDM schema 25
2.2.3 DT schemas 25
2.2.4 Actor schemas 26
2.2.5 Related schema modelling environment research 26
2.2.6 Approach to a schema modelling environment 27
2.3 Inter-schema Relationship Modelling 28
2.3.1 Requirements of an inter-schema relationship definition language and modelling environment 29
2.3.2 Related inter-schema relationship modelling languages 30
2.3.3 Related inter-schema relationship modelling environments 31

2.3.4 Approach to an inter-schema relationship modelling language 32
2.3.5 Approach to an inter-schema relationship modelling environment 32
2.4 Design Tool Environment Modelling 33
2.4.1 Requirements for design tool environment modelling 33
2.4.2 Related design tool environment modelling work 35
2.4.3 Approach to a design tool environment modelling system 35
2.5 Project Definition 36
2.5.1 Requirements for a project definition notation and development environment 36
2.5.2 Related project definition notation work 37
2.5.3 Related project definition environment work 37
2.5.4 Approach to a project definition notation 38
2.5.5 Approach to a project definition environment 38
2.6 Project Development Environment Summary 39

3 Schema Modelling and Development 40
3.1 Introduction 40
3.1.1 Requirements for schema development 41
3.1.2 Schema specification languages 42
3.1.3 The EXPRESS and EXPRESS-G languages 43
3.2 Schema Development in the EPE Environment 44
3.2.1 Functionality offered by the EPE environment 45
3.2.2 Using the EPE environment 52
3.2.3 Implementation of EPE in the MViews framework 53
3.2.4 Internal schema representation in EPE 57
3.2.5 Summary of EPE functionality 58
3.3 Generic Schema Database Definition 58
3.4 Appraisal of Schema Modelling 59

4 Inter-Schema Relationship Modelling 63
4.1 Mapping Types 64
4.1.1 Structural mapping types 64
4.1.2 Semantic mapping types 66
4.1.3 Mapping language requirements 67
4.2 Mapping Definition Languages 69
4.2.1 EXPRESS-M 70
4.2.2 EXPRESS-V 72
4.2.3 EXPRESS-C 74
4.2.4 Transformr 75
4.2.5 EDM-2 76
4.2.6 KIF 78
4.2.7 Superviews 78
4.2.8 RDBMS views 79
4.3 Summary of Inter-Schema Relationship Modelling 80

5 The View Mapping Language (VML) 82
5.1 Mapping between Schemas 84
5.2 Mapping between Classes 86
5.2.1 Entity names and keys 86
5.2.2 Inheritance of *inter_class* definitions 88
5.2.3 Invariant specification 89
5.2.4 Initialiser specification 90
5.2.5 Equivalence specification 91
5.2.6 Mapping equations 91
5.3 A Graphical Notation for VML 98
5.3.1 Graphical icons of VML-G 99
5.4 Appraisal of VML 101

6 Mapping Modelling and Development 106
6.1 Introduction 106
6.1.1 Requirements for mapping development 107
6.2 Mapping Development in VPE 108
6.2.1 Functionality offered by the VPE environment 108
6.2.2 Using the VPE environment 116
6.2.3 Implementation of VPE in the MViews framework 118
6.2.4 Future connections between VPE and EPE 120
6.3 Management of Mapping Definitions 120
6.3.1 Name resolution 121
6.3.2 Schema modification 122
6.4 Generic Mapping Database Definition 122
6.5 Appraisal of Mapping Modelling and Development 123

7 Project Modelling 126
7.1 Introduction 126
7.1.1 Requirements 129
7.1.2 Structure 131
7.2 User and Function Modelling 132
7.3 Flow of Control Modelling 133
7.3.1 Set theoretic background for flow of control 133
7.3.2 Flow definition 137
7.4 Appraisal of Project Specification 141

8 The Project Testing and Implementation Environment 145
8.1 Structure and Requirements 146
8.2 Schema Instance Development 147
8.2.1 Requirements of a schema instance maintenance system 147
8.2.2 Related schema instance maintenance systems 148
8.2.3 Approach to a schema instance maintenance system 149
8.3 Mapping Handler and Controller 149
8.3.1 Requirements of a mapping handler and controller 149
8.3.2 Related mapping handler and controller work 150
8.3.3 Approach to a mapping handler and controller 151
8.4 Design Tool Connection 152
8.4.1 Requirements for a design tool connection system 152
8.4.2 Related design tool connection systems 152
8.5 Flow Handling 152
8.5.1 Requirements of a flow of control manager 153
8.5.2 Related flow of control manager work 153
8.5.3 Approach to a flow of control manager 154
8.6 Project Testing and Implementation Environment Summary 155

9 Schema Instance Management 156
9.1 Requirements for Instance Management 156
9.2 Instance Management Systems 157
9.2.1 EPE: an instance construction and browsing system 157
9.2.2 InSTEP: a graphical instance browser 159
9.2.3 SnartQuery and the ObjectViewer 160
9.2.4 Re*f*lex: an object-oriented CAD system 162
9.3 Appraisal of Schema Instance Management 163

10 Mapping Controller 164
10.1 Data-Store Modification Records 165
10.2 The Mapping Controller 169
10.2.1 Transaction-based mapping manager 171
10.2.2 Automatic mapping manager 172
10.3 Performing a Mapping 172
10.3.1 In preparation to map 173
10.3.2 The first mapping between two stores 173

10.3.3 Consideration of modification types . . . 174
10.3.4 Determining combinations of objects from an *inter_class* header . . . 175
10.3.5 Four pass mapping process . . . 179
10.3.6 Mapping a new combination to the other data-store . . . 180
10.3.7 Procedures followed when a new object is created . . . 181
10.3.8 Tracking objects created and referenced in mappings . . . 182
10.3.9 Mapping the deletion of an object . . . 182
10.3.10 Mapping the modification of an object . . . 183
10.3.11 Evaluating, or re-evaluating affected equations . . . 184
10.4 Appraisal of Mapping Controller . . . 186

11 Flow Handling . . . 189
11.1 Requirements for Flow Handling . . . 189
11.1.1 Project manager requirements . . . 190
11.1.2 Actor requirements . . . 190
11.2 The Exchange Executive . . . 191
11.2.1 Simulation of flow of control . . . 191
11.2.2 Representation of design tool invocation . . . 197
11.2.3 Project manager interface . . . 198
11.2.4 Actor interface . . . 201
11.3 Appraisal of Flow Handling . . . 203

12 Conclusions . . . 205
12.1 The Project Development Environment . . . 206
12.2 The Mapping System . . . 208
12.2.1 Additional applications of the mapping language . . . 209
12.3 The Flow of Control System . . . 210
12.4 Future Work . . . 212
12.4.1 Tighter system integration . . . 212
12.4.2 Distributed environment . . . 213
12.4.3 Wider incorporation of project aspects . . . 213
12.4.4 Formal definitions of the mapping domain . . . 214
12.4.5 Alternate mapping language implementations . . . 214
12.4.6 Incorporation of measure tools . . . 215
12.4.7 Dissemination and exploitation . . . 215

Appendix A. The View Mapping Language . . . 217
A.1 VML Syntax . . . 217
A.2 VML Graphical Notation . . . 219
A.3 VML Comparison to other Notations . . . 220
A.3.1 Comparison to database operators . . . 220
A.3.2 Comparison to Motro virtual integration operators . . . 222
Appendix B. Project Specification Language . . . 227
B.1 Project Model Transfer Syntax . . . 227
B.2 Project Modelling Graphical Notation . . . 229
B.2.1 User and function specification . . . 229
B.2.2 Flow of control specification . . . 230
Appendix C. Snart . . . 232
C.1 Facets . . . 232
C.2 Query Language . . . 233
C.2.1 Introduction . . . 233
C.2.2 Example schemas . . . 234
C.2.3 Old style queries in Snart . . . 235
C.2.4 New style queries in Snart . . . 235
C.2.5 Implementation of the query language in Snart . . . 239
C.3 Object Spaces . . . 239
C.3.1 Introduction . . . 240
C.3.2 Defining an object space in Snart . . . 240
C.3.3 Working with an object space model in Snart . . . 242

C.4 Persistency . . . 243
C.4.1 Introduction . . . 243
C.4.2 Persistency in the old Snart . . . 244
C.4.3 Manipulating persistent objects in the old Snart . . . 245
C.4.4 Persistency in the new Snart . . . 246
C.5 ObjectViewer . . . 248
Appendix D. Small Examples Schemas and Mappings . . . 252
Appendix E. Large Example Schemas and Mappings . . . 275
E.1 Description of Large Example . . . 275
E.2 Schemas for the Large Example . . . 278
E.2.1 IDM schema . . . 278
E.2.2 PlanEntry schema . . . 282
E.2.3 FaceEditor schema . . . 286
E.2.4 VISION-3D schema . . . 287
E.2.5 ThermalDesigner schema . . . 289
E.3 Mappings for the Large Example . . . 294
E.3.1 IDM <-> PlanEntry mapping . . . 295
E.3.2 IDM <-> FaceEditor mapping . . . 299
E.3.3 IDM <-> VISION-3D mapping . . . 302
E.3.4 IDM <-> ThermalDesigner mapping . . . 306
E.4 Project Window for the Large Example . . . 309
E.4.1 User and function specification . . . 309
E.4.2 Flow of control specification . . . 313
Appendix F. The Parsers . . . 320
F.1 ISO-10303:11 EXPRESS Parser . . . 320
F.1.1 EXPRESS to Snart translator . . . 322
F.1.2 Snart to EXPRESS translator . . . 323
F.1.3 Snart to Re*f*lex translator . . . 323
F.2 ISO-10303:21 STEP data-file Parser . . . 326
F.2.1 STEP data-file to Snart translator . . . 328
F.2.2 Snart to STEP data-file translator . . . 329
F.3 CGE Parser . . . 329
F.4 VML Parser . . . 331
Appendix G. Generalised Schema Representation Notation . . . 332
G.1 Version Tree . . . 332
G.2 Schema . . . 333
Appendix H. Generalised Mapping Representation Notation . . . 335
H.1 Schema . . . 335
H.2 Mapping . . . 336
H.3 Inverted Index . . . 337

Glossary . . . 338

References . . . 342

List of Figures

Figure 1.1 COMBINE structure 5
Figure 1.2 Structure of an integrated design system 7
Figure 1.3 An example of the Snart graphical notation 12
Figure 1.4 A building described in PlanEntry 13
Figure 1.5 The PlanEntry schema main classes 14
Figure 1.6 Wall and window materials being defined in the FaceEditor 15
Figure 1.7 The FaceEditor schema main classes 15
Figure 1.8 A wire-frame representation of Figure 1.3 in VISION-3D 16
Figure 1.9 The VISION-3D schema classes 16
Figure 1.10 Control windows in ThermalDesigner 17
Figure 1.11 The IDM schema classes 18

Figure 2.1 EXPRESS and EXPRESS-G views in the EPE modelling environment 27
Figure 2.2 Example VML textual specification 32
Figure 2.3 Graphical mapping specification in VPE 33
Figure 2.4 Project flow of control definition 38

Figure 3.1 An example of the EXPRESS-G notation 44
Figure 3.2 Use of *technical_system* in the COMBINE IDM 45
Figure 3.3 Two high-level inheritance specifications for *technical_system* 46
Figure 3.4 Design stage specification of attributes of an entity 47
Figure 3.5 Textual view derived from graphical views of an entity 47
Figure 3.6 Constraining the cardinality of an attribute at late design stage 48
Figure 3.7 The propagation of an *update_record* to a dependant textual view 49
Figure 3.8 The automatic application of an update in a textual view 49
Figure 3.9 The persistent *update_record* viewer with documentation facility 50
Figure 3.10 A textual documentation view 51
Figure 3.11 The view navigator invoked for the *technical_system* entity 51
Figure 3.12 MViews three-layer multiple view architecture as used in SPE 54
Figure 3.13 Change propagation in an MViews environment 55
Figure 3.14 Class inheritance for EPE from MViews 56

Figure 5.1 A tree of *inter_view* mappings 85
Figure 5.2 Graphical mapping specification in VPE 99
Figure 5.3 Complex VML-G specification with textual equivalent 100

Figure 6.1 Initial connections between classes in two schemas 109
Figure 6.2 Specifying multiple mappings for a single class set 110
Figure 6.3 Defining links between all features associated with an *inter_class* definition 110
Figure 6.4 A full textual specification of an *inter_class* definition 111
Figure 6.5 Modification of a graphical view 112

Figure 6.6 Receipt of an *update_record* in a textual view 113
Figure 6.7 A textual view after manual application of an update specification 114
Figure 6.8 Browsing the change log for an *inter_class* specification 115
Figure 6.9 An associated documentation view for an *inter_class* definition 115
Figure 6.10 View navigation for *inter_class* specifications 116
Figure 6.11 Class inheritance for VPE from MViews 119

Figure 7.1 Multiple project windows in a project 127
Figure 7.2 Example of user and function specification 131
Figure 7.3 Invocable design functions 135
Figure 7.4 Constrained design function 135
Figure 7.5 Constraints between two design functions 136
Figure 7.6 Design function constraints leading to apparent inconsistencies 136
Figure 7.7 Top level flow of control specification 137
Figure 7.8 Flow of control with global elements 138

Figure 8.1 Sample mapping controllers 151
Figure 8.2 An operating flow of control manager 154

Figure 9.1 Instance viewing and navigation 158
Figure 9.2 InSTEP's graphical and textual views 159
Figure 9.3 A SnartQuery dialogue and result 161
Figure 9.4 A default ObjectViewer object layout 161
Figure 9.5 Re*f*lex's multiple graphical views, along with an object's attribute dialogue 162

Figure 10.1 Structure of the traced persistent space in Snart 166
Figure 10.2 An actor's mapping controller interface 169
Figure 10.3 A transaction-based mapping manager 170
Figure 10.4 An automatic mapping manager 171

Figure 11.1 Calculating potential design functions from a CombiNet 192
Figure 11.2 Multiple actors causing select design functions to become stalled 193
Figure 11.3 Initial CombiNet in a project window 194
Figure 11.4 Multiple levels of aggregate functions 195
Figure 11.5 Exiting from a CombiNet representing an aggregate place 196
Figure 11.6 Design tool start up dialogue 197
Figure 11.7 Design tool termination dialogue 198
Figure 11.8 Project manager user interface 199
Figure 11.9 Project manager navigation through CombiNets 200
Figure 11.10 Actor's user interface 201

Figure A.1 VML graphical notation 219

Figure B.1 Icons used for user and function specification 229
Figure B.2 Icons used for flow of control specification 230

Figure C.1 Snart query language interface 234
Figure C.2 ObjectViewer interface 248

Figure E.1 Integration of tools in example 275
Figure E.2 Building design in PlanEntry with mapping controllers 276
Figure E.3 Result of mapping to VISION-3D and FaceEditor 276
Figure E.4 Building design after mapping to three tools 277
Figure E.5 Layout change to building in Figure E.4 277
Figure E.6 Result of propagating changes shown in Figure E.5 278
Figure E.7 IDM schema 279
Figure E.8 PlanEntry in use 283
Figure E.9 Planes calculated for a building 283
Figure E.10 PlanEntry schema 284
Figure E.11 FaceEditor in use 285

Figure E.12 FaceEditor schema 286
Figure E.13 VISION-3D in use 287
Figure E.14 VISION-3D schema 287
Figure E.15 ThermalDesigner in use 290
Figure E.16 Client and design roles 310
Figure E.17 Architect and design roles 311
Figure E.18 Structural consultant and design roles 311
Figure E.19 Daylighting consultant and design roles 312
Figure E.20 Thermal consultant and design roles 312
Figure E.21 Top-level CombiNet 313
Figure E.22 Design and update CombiNet 314
Figure E.23 Building design CombiNet 315
Figure E.24 Structural work CombiNet 316
Figure E.25 Daylight work CombiNet 317
Figure E.26 Thermal work CombiNet 318
Figure E.27 Acceptance CombiNet 319

List of Tables

Table 4.1 Mapping types from van Horssen et al. 1994 65
Table 4.2 Full set of structural mapping types 65
Table 4.3 Schema integration conflict types from Batini et al. 1986 66
Table 4.4 Schema integration conflict types from Kim and Seo 1991 67
Table 4.5 Schema fragments for the two schemas in the mapping example 70
Table 4.6 EXPRESS-M mapping for example problem 71
Table 4.7 EXPRESS-V mapping for example problem 73
Table 4.8 EXPRESS-C mapping for example problem 75
Table 4.9 Transformr mapping for example problem 76
Table 4.10 EDM-2 mapping for example problem 77
Table 4.11 KIF mapping for example problem 78
Table 4.12 Comparison of mapping languages 80

Table 5.1 VML mapping for example problem 83
Table 5.2 Top level definition of a VML mapping 83
Table 5.3 Definition of an *inter_view* specification 84
Table 5.4 Definition of an *inter_class* specification 86
Table 5.5 Definition of a *class_name* 88
Table 5.6 Definition of inheritance 89
Table 5.7 Definition of invariants 89
Table 5.8 Definition of initialisers 90
Table 5.9 Definition of equivalences 91

Table 7.1 Capabilities of project specification languages (after Curtis et al. 1992) 128

Table 10.1 Pseudo-code for performing a mapping 173
Table 10.2 Pseudo-code for establishing a mapping 173
Table 10.3 Pseudo-code for performing mappings 174
Table 10.4 Examples of *inter_class* headers and resultant object lists 176
Table 10.5 Pseudo-code for determining initial object groups 176
Table 10.6 Pseudo-code for generating object combinations for an *inter_class* 177
Table 10.7 Pseudo-code for the four-pass *inter_class* resolution 180
Table 10.8 Pseudo-code for mapping a new combination 181
Table 10.9 Pseudo-code for creating a new object 181
Table 10.10 Pseudo-code for mapping an object deletion 182
Table 10.11 Pseudo-code for mapping an object modification 183
Table 10.12 Pseudo-code for identifying values for an equation 185
Table 10.13 Update authorities for attributes derived from different sources 186

Table G.1 Specification of a schema version 332
Table G.2 Specification of version creation reasons 333
Table G.3 Specification of schema information 333
Table G.4 Specification of modification types 334

Table H.1 Specification of schema entities 335
Table H.2 Specification of atomic mapping components 336
Table H.3 Specification of inverted indices into mappings 337

Chapter 1

Introduction

1.1 The Problem Domain

In the fields of architecture and building engineering there is continued dissatisfaction with the state that computerised design has reached. The majority of the claims made for computerisation in these fields have never been achieved. For architects there are few, if any, tools which are useful for initial or sketch design and in many architectural firms the only use of computers during the design process is for detailing the building plans with a draughting tool (CAD). In the engineering domain there are similar problems. Even with a computerised plan of the building it is unlikely that building information can be extracted from this plan for use in other design tools. Many practitioners re-enter building information into their design tools by hand, taking measurements off the printed plans. Despite the existence of a wide range of excellent design tools, such tools are seldom used because of the difficulty and time required to enter building information in order to invoke the tool.

Examining the state of computing in these domains it is clear that the majority of the problems lie with the conceptual models of a building. In a wide range of CAD tools there is no underlying model of a building. That a plan defines a building can only be ascertained by human perception. To the computer a plan is a set of 3D graphical entities. Until computers can mimic the human ability to perceive building structure from a plan, it is impossible to extract useful building information from a traditional CAD system. There is also a problem with the wide range of design tools that model buildings. Each of these design tools has a schema of a building most suited to the type of computation it performs on the building, and in most cases this is a minimal schema to ease

the burden of data entry for the user. As can be imagined, this means that schemas of buildings for tools such as thermal simulation, structural simulation, lighting analysis, fire-code compliance, quantity surveying, etc. have very different structures. As a result, detailing a building for use with two different design tools is like trying to explain exactly the same thing in two totally different languages.

Another important problem is that of consistency and coordination. In a building project there are often several design professionals responsible for different parts of the design. A project manager has the task of coordinating the various designers, and in a manual design practice a range of procedures are used to ensure that the design is kept consistent between all designers. In a computerised design environment it has become no easier to perform this task, and in many cases much more difficult, as the number of plan portions which can be generated with computerised tools is much greater than with manual design.

One obvious solution to the problems outlined above is to find some way to integrate the design tools with a comprehensive conceptual building schema used to transfer building data between the different design tools. There have been several approaches to integrated systems which offer various benefits to the designers working with them. One approach is an integrated design assistant in which it is envisaged that the integrated system would offer a single designer the ability to design utilising any of the various design tools as assistants to check various portions or stages of the design. The integrated assistant would manage the transfer of data to and from the design tools and maintain the consistency of the global model of the designed artifact. Another approach is an integrated design system in which it is envisaged that the system would connect multiple designers working on the same artifact, maintaining the global consistency of the design as it evolves over time. This system would still allow designers to experiment with design solutions independently from other designers (as in the integrated design assistant), but maintain the global model with respect to solutions exported by the various designers. This system would also manage process control to ensure that all design functions were completed during the design and to control the time line of the design process.

The increasing demand for intelligent integrated computerised working environments in architectural and building engineering domains has spawned a multitude of research projects tackling portions of the problem. The main benefits these systems claim to offer their users are:

- Access to all the pertinent information for their portion of the design task
- Access to a range of appropriate design tools which can be utilised in the design process
- Project control management to direct the project development and ensure timely completion
- Automated maintenance of the consistency of the design amongst all designers in a project
- Ability to experiment with design modifications independently before submitting a suitable modification to the global model

- Notification of changes to the global model which affect the design tasks of a particular user
- Management and notification of conflicts between the work of various users

While the benefits noted above exist in current managed projects there are several other flow on benefits possible from an integrated system:

- Final design is likely to be more correct as all designers have been working on the same global model of the artefact
- Final design may be more innovative as designers have the ability to examine and test multiple design solutions to a particular problem through connected design tools
- Possibilities for better performance of the completed artefact as designers have many design tools available to check the properties of the artefact being designed and to evaluate design options quantitatively

There is much prior research in this area of integrated design systems, as will become clear in the next section. The next section will also detail the many other areas of computer science research which are applicable to an integrated design system. To help illustrate the place of the research encompassed in this thesis the author has chosen one of the EU funded projects (COMBINE) as an example of an evolving integrated design system. The framework of the COMBINE system is similar to most other integration projects and is used due to the author's familiarity with the project from a six month period of work in Europe.

1.2 Related Research

Whilst this thesis examines an overall framework to allow the development and testing of an integrated design system, the majority of research to date has concentrated on problems associated with the individual components of this framework. These components include developing data models, defining mapping languages, managing model development, etc., and the work in these areas is considered in Chapters 2 and 8 of this thesis. What little work that has looked at over-arching development frameworks is considered below.

In the architecture, engineering and construction (A/E/C) domains there has been intense European Union work on the development of integrated design systems since the late 1980s. This has culminated in a range of projects for various subdomains of A/E/C, for example, COMBINE (Augenbroe 1995a; Augenbroe 1995b), ATLAS (1993; Greening and Edwards 1995), COMBI (1995; Scherer 1995), CIMsteel (1995; Watson and Crowley 1995) and more recently ToCEE (Katranuschkov et al. 1996), STAR (Huovila and Seren 1995) and VEGA (1996). These projects started out examining frameworks purely for data integration, but over the years recognised the need for other project aspects, and broadened their scope to process, documents, legality, etc.

Despite their current wide scope very little work has been done in these projects to consider the framework required to ease the development of integrated design systems. Very simple tools to specify data models, or process models, all quite independent of each other are as far as any project has gone. The COMBINE project has developed the greatest number of support tools, and some of these are described in Section 1.3 below and highlighted throughout this thesis as examples of current state of the art.

Research projects undertaken by individual academic institutions have often taken a wider view than the more result-oriented European Union research, with the result that they have developed a more comprehensive framework than many European projects (Papamichael and Selkowitz 1991; Lamb 1987; Subrahmanian 1989; Pena-Mora et al. 1993). Once again the majority of these projects develop small prototype integrated systems with models crafted by hand in text-editing tools, rather than developing their own support tools. The projects which do develop support tools for framework development provide very useful and tightly coupled tools, but for very narrow domains (Poyet et al. 1995; Grundy 1993). In the majority of cases these domains are restricted to the data aspects of the integrated design system, ignoring requirements of support for mapping between data aspects, or incorporating process and project aspects into the framework.

1.3 COMBINE Project

Throughout this thesis I will draw upon several examples and use nomenclature which have originated in the COMBINE project. I will also use COMBINE to help illustrate the place of the various systems that are developed and to highlight their utility in such a project.

COMBINE (COmputer Models for the Building INdustry in Europe) is an EU-funded project which started in 1990 (Augenbroe and Laret 1989). The first phase ended in late 1992 with a seminar and workshop at which the first phase deliverables were demonstrated (Augenbroe 1993). The project, which has now completed its second phase (1992-1995), spans a total effort of 70 man-years spread over 12 partners from 7 European countries.

COMBINE worked towards the practical development of IIBDS's (Intelligent Integrated Building Design Systems) through which energy, services, functional and other performance characteristics in planned buildings can be modelled and integrated. The modelling of information has been one of the greatest concerns in this project, with the first phase of the project concentrating on the integration of data to provide the necessary information for a group of actors. COMBINE's efforts have been concentrated on establishing a data infrastructure and tools for managing the information exchange amongst design actors, i.e., members of a collaborative design team. The project thus represents a good example of international research in product data technology and of the development of an integrated design system.

COMBINE is one of many international projects with a similar aim of integrating the information requirements of a given domain to provide an enhanced working environment to users of that domain. A number of parallel projects in the European ESPRIT-CIME (a full list of acronyms used here can be found in the glossary) program are engaged in similar research and development to that of COMBINE. In time all of these projects may have input into the ISO-STEP standard which has the stated aim of covering the information requirements for all architecture, engineering and construction domains (ISO/TC184 1993). In recognition of the importance of STEP, and the practicalities of interchange of results, most projects have chosen the modelling formalisms specified for the STEP standard as the formalisms for their efforts. These include the EXPRESS language, with EXPRESS-G, NIAM, IDEF1X and IDEF0 diagramming techniques, the STEP neutral file format, for exchange of data, and SDAI, the data access specification. This is the case in COMBINE where all modelling is done with these formalisms (Augenbroe 1994).

COMBINE's first phase was concerned primarily with data integration based upon the concept of a set of actors connected to a central common data repository (Figure 1.1). Its deliverables comprised the first large conceptual integrated data model (IDM) for buildings and a number of interfaced design tools (DT). These results formed the base line technology on which the present second phase builds. The second phase was concerned with developing an operational IIBDS according to functional specifications of particular building projects in practice. In doing this, the number of interacting design tools was expanded to cover existing design applications in the area of costing, HVAC-CAD (Heating, Ventilating and Air Conditioning), architectural CAD, component databases, daylighting and building regulations.

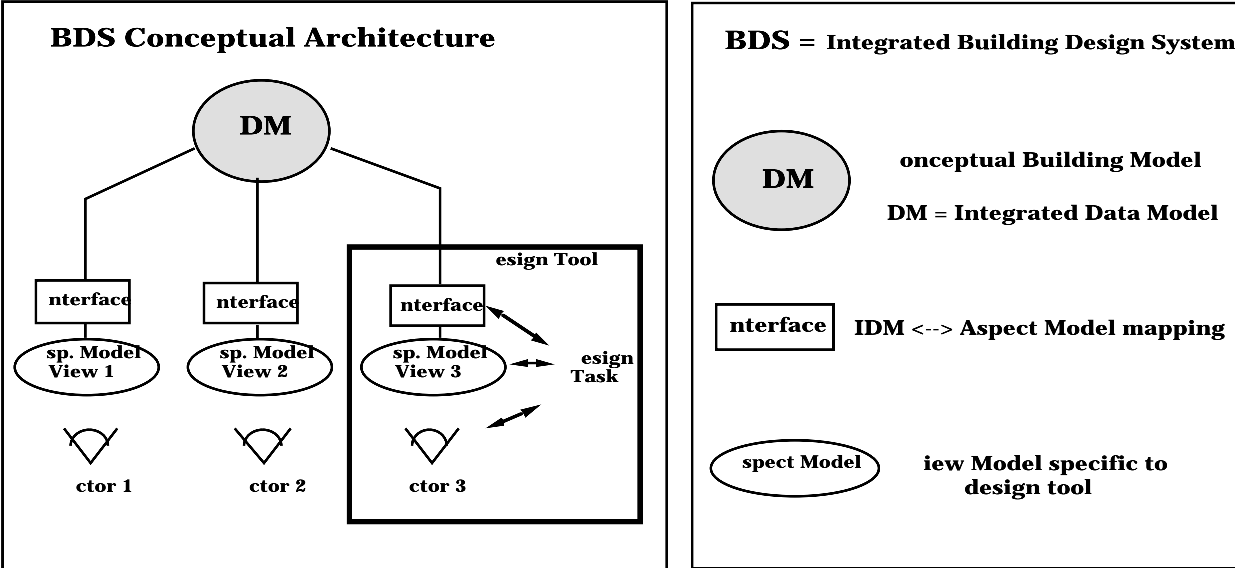


Figure 1.1 COMBINE structure

In the first phase of COMBINE the types of actors considered were limited to those in the fields of energy and HVAC performance in the early design stages of building design. However, even limiting the domain to this small set of actor types, the IDM developed for this project consisted of some 400 entities and 600 relationships between them. The development of this conceptual schema

was a complicated task. The final schema had to incorporate the data requirements of the seven design tools which were used in this first phase, which had very different data requirements to model a building for their simulation needs. The IDM was developed by first modelling the schemas of the various design tools, and then, through schema analysis and use of integration techniques, various portions of the schemas were brought together to help form the final IDM. This was a very time consuming and difficult task. In particular, the developers in the project found it impossible to verify the consistency of the IDM without the aid of computer tools (Dubois 1993).

The second phase of COMBINE involved integration of several more design tools, widening the scope of the IDM. This required a redesign of the IDM to extend the domains supported. Again, the process of development involved comparison of the IDM with the schemas of the various design tools being supported, and incorporation of portions of the design tool schemas into the IDM. The resulting iterative process of schema analysis, integration, and prototyping continued until the various design tool support teams were satisfied that the data requirements necessary for their tools could be met by the IDM.

The modelling support presently offered to partners in COMBINE comes through the use of the Configurable Graphical Editor (CGE, see Vogel 1991). CGE can be configured to support a number of diagramming techniques, among which are those mentioned at the start of this section. It also supports a customised version of NIAM called ATLIAM (Vogel 1991) which is capable of exporting an EXPRESS schema of a NIAM diagram and of exporting and importing NIAM diagrams by way of STEP files.

1.4 Underlying Framework

The creation of an integrated design system requires an enormous amount of work constructing specialised modules in a framework of great complexity. The development of a complete framework by one person is obviously unobtainable in a lifetime. However, the general structure of integrated design systems appears fairly stable and it is particular portions of the framework which require research and development to find optimal solutions. In virtually all integrated systems there are a set of common fundamental systems as illustrated in Figure 1.2.

When an integrated design system is in use its schemas and design tools are likely to vary from project to project. This is because a building design project is usually involved with a one-off design and: the group of designers involved; the aspects of the design each designer works on; the design tools required; and the flow of control for the project, are likely to change between each project. This produces a requirement that the integrated system is easily configured for a varying set of designers and design tools. Of more significance is the recognition that to properly cope

with this situation it will be necessary to modify and add schemas to the system. The designer's schemas will have to be changed to reflect the designer's roles in the current project, a new flow of control (project) model will be required for each project, and perhaps new design tool schemas will have to be introduced to cover checking of unusual aspects of the current design (which could in turn affect the scope of the IDM). This leads to the requirement that the final integrated system must contain the modelling tools used in the design of the initial system to enable the system to be configured for individual projects.

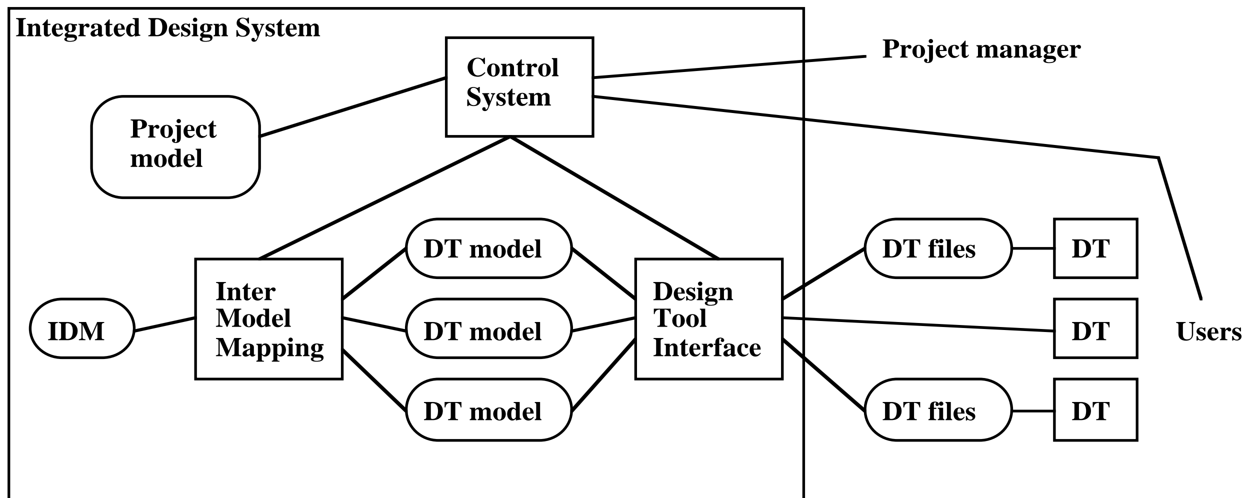


Figure 1.2 Structure of an integrated design system

The integrated design system can be implemented in a variety of ways from a monolithic system with all modules tightly coupled through to a completely modular system with links to relational or object-oriented databases and where many of the modules operate as independent systems utilising a brokering architecture to communicate data between modules. However the system is implemented the same conceptual major subsystems exist in the integrated design system. These are detailed further to give some idea of the scope of the task each must handle.

IDM: this is the very general schema of the domain which must cover the information requirements of the designers and design tools which interact with the system. The schema required for buildings is probably the largest and most complex information model developed in the world to date. The ISO 10303 standards committee (STEP) has laboured for over 10 years to define the underlying schemas for graphical, draughting and material representations. Only recently have they moved to create definitions of higher level entities for buildings, ships, etc. Numerous projects have used schemas based around those required for design tools, but expanded to cover requirements of other design tools. Recently several collaborative projects have worked on general schemas of buildings based on the STEP fundamentals for restricted aspects of buildings. Each of these projects has invested in the order of 10 person years of effort in their restricted schemas. The development of the IDM requires the input of domain experts for all the sub-domains being modelled as well as the help of modelling experts to structure the schema. The IDM remains as a fairly static schema, although the introduction of new design tools, construction techniques and building constructs may require modifications to the schema.

The development of an environment to specify the IDM draws upon work in data modelling languages, modelling environments, collaborative work systems and schema integration techniques.

Project model: this is a project specific model defining the designers who will be working on a project, the design functions they will need to perform and the design tools that will be utilised during the design process. The project model can also be used to define the flow of control in a project, specifying hand-over points between various designers and sequences of specification that must be followed to help guarantee an optimal design at the termination of the design process. As the design process is not a fully understood process it can not always be completely described at the start of a project. Therefore, it is likely that during the course of the design, new designers may need to be involved in the design or new design tools used for difficult aspects of the design. To enable this flexibility the project model must be modifiable during the project to be able to take into account changes in personnel and new flows of control. The development of an environment to specify the project model draws upon work in process modelling languages, modelling environments and collaborative work systems.

DT schemas: each of these schemas defines the structures and data requirements of a design tool. The type of design tool and its capabilities will determine the number of schemas that are required for each tool. In most cases two schemas are used, one to specify the data input to the design tool, the second to specify the data output from the design tool. The creation of these schemas is a task undertaken by modellers who have great familiarity with the particular design tool being modelled. This is because the schema must capture not only the data structures explicitly defined by the tool and the data constraints specified by the design tool (e.g., ranges on attribute values, cardinality constraints on lists, etc.), it must also contain the constraints defined implicitly by the design tool (e.g., that all walls are vertical). A fully specified DT schema allows the system to determine whether there is enough data to create the input file for the DT to execute and to determine whether a semantically correct description of the building can be created from the data available. The DT schema remains static throughout the use of the same version of the DT. The development of an environment to specify the DT schemas draws upon work in data modelling languages, modelling environments and collaborative work systems.

Control system: this directs both the inter-model mapping and design tool interface modules to perform their tasks as required by the designers or directed by the project manager. This module simulates the flow of control defined in the project model determining what design functions are able to be performed at any particular time. It also interfaces with designers to let them specify the tasks that they wish to perform next and interfaces with the project manager for decisions upon which designers should be involved in new phases of the project. This system directs the inter-model mapping module to map data between different models, or to ascertain whether it is possible to perform the mapping. It also directs the design tool interface to invoke a design tool with appropriate data and to collect results

upon termination.

Inter-model mapping: this module performs the translation of data between the IDM and the DT models. It must determine whether it is possible to create a consistent model from the IDM for any one of the DT models based upon the constraints in that DT schema, and it must ensure that the IDM remains consistent upon update from a design tool's output. It must detect conflicts between various sets of data and must be able to invoke processes to enable negotiation over conflicting data from different design tools and users. Traditionally this module is implemented by hand coding the mappings required between the IDM and each new DT schema. In future integrated design systems it is expected that this module will be realised by modelling the mappings required between the IDM and each new design tool schema. The development of the inter-model mapping module draws upon work in mapping modelling languages and modelling environments.

Design tool interface: this module provides the connection between the design tools utilised in the system and their data models. For an off-line DT this module must be able to create the input files required for the DT (from the DT schema in the system), it must then be able to invoke the DT to perform its work and finally retrieve the output from the DT into the DT model in the system. For interactive DT's it must perform the same task but driven by the demands of the design tool. Traditionally this module is implemented by hand-coding the parsing and pretty-printing for each new design tool. In future integrated design systems, and for some classes of design tools, it is expected that this module will be realised by modelling the interaction required by each design tool (for example the data-file structures) and using this model to drive the interaction with the DT. The development of the design tool interface draws upon work in parsing, pretty-printing, interaction and structure modelling languages and modelling environments.

DTs: these tools exist outside of the integrated design system. The types of design tools that can be found here include the interface a designer has to the system, off-line and interactive simulation tools, interactive knowledge-based systems, CAD systems, etc.

1.5 Research Objectives

Although the basic framework of an integrated design system as described above is fairly universally applied in integrated projects and although many hundreds of man years have been put into these projects there are many aspects of the framework which are poorly understood and not well modelled. To date the majority of the structured effort in integrated projects has been put into the development of the IDM by the domain experts in the project. Systematic modelling of other facets of the integrated system has been bypassed in favour of hand-coded modules which work for particular design tools under particular situations. I believe that it is crucial to model all aspects of the components in an integrated system to guarantee the correct behaviour of the finished system. To enable these models to be developed it is also necessary to have powerful modelling

tools which enable the modellers to describe and manage the enormous models that are being created. The exploration of these objectives forms a major component of the work described in this thesis along with an implemented framework which demonstrates how all the modelled systems can be drawn together into an implemented system. An overview of the major objectives which are examined in this thesis follows:

1.5.1 Formalism to specify mappings

To define the correspondence between two schemas it is necessary to use a formalism which allows the specification of mappings in an easy and intuitive manner. This formalism must closely match the problem domain and have a clear syntax and semantics. In current projects this module has received little attention and only recently has any effort been put into the examination of possible formalisms for this task.

1.5.2 Formalism to specify project and flow of control

As a new project specification is required for every project undertaken with an integrated design system it is essential to have a formalism that will allow the easy description of the participants and their roles in the project. In current projects this module has received little attention and only recently has any effort been put into the examination of possible formalisms for this task. This formalism must offer easy specification of the designers and their roles in the project. It must also be able to specify the flow of control during the project and the hand-over points for the various participants.

1.5.3 Comprehensive tool set to support schema specification

With the enormous schemas which are being specified in these projects more sophisticated modelling environments are required to manage the complexity of the schemas. New environments must be able to support schema specification at many levels of abstraction and with many different formalisms. Multiple views of the schema must be available in any of the formalisms with consistency between the views maintained under modification. There must also be the ability to provide annotation of the schema with documentation. Collaborative modelling must be supported to encompass the range of input from modellers who have input to the schema. Version management for the evolving schema must be supported along with version merging and conflict resolution for the versions of various modellers.

1.5.4 Inter-model mapping utilising mapping specifications

With definitions of the mapping required between two schemas it is necessary to show how they can be used to map whole models back and forth, ensuring the consistency of the models as they are worked upon. The inter-model mapping must be able to create new models from an existing model and it must be able to propagate changes between linked models. When both models have

changed it must be able to arbitrate the merging of changes between the models and when there are conflicts in the modifications being propagated it must be able to manage the resolution of those conflicts between the affected parties.

1.5.5 Control system utilising project and flow of control specification

The control system needs to maintain contact with the project manager and the designers involved with the project. The control system must be able to determine what the allowable tasks are at any time and inform the designers of the options that they have available. When a designer chooses to perform a task the control system determines whether it is possible at the given time, if it is the control system invokes the inter-model mapping to get data into the design tool's model and then invokes the design tool interface to run the design tool and extract results which will be fed back into the IDM.

1.6 Description of Example

Throughout this thesis a single example set will be referenced, as required to illustrate points pertaining to the workings of an integrated system. This is distinct from, and in contrast to, the COMBINE example which illustrates the requirements for the framework. This example set is drawn from a contract completed between the Building Research Association of New Zealand (BRANZ) and the computer science department at the University of Auckland (Hosking et al. 1995). The problem domain of the example is the development of an integrated design environment where several disparate design tools are required to work together to provide enough information for knowledge-based systems. The knowledge-based system demonstrated is ThermalDesigner (Amor et al. 1992) which checks a building using the Annual Loss Factor (ALF) method (Bassett et al. 1990, an empirical estimator of the thermal design code (SANZ 1977)). The building layout and materials information required for this knowledge-based system to operate can be supplied from two stand-alone tools which allow for basic layout design and materials definition. A third program is used to visualise a full 3D model of the building with fly-throughs and shaded images.

The use of this integrated system in a working design situation would involve several design professionals each charged with different aspects of the design. In this thesis we will envisage a design team consisting of four professionals, an architect, a thermal engineer, a daylighting engineer, and a structural engineer along with a client. The architect would be involved with the initial design of a building after liaison with the client and coordination of the three engineers leading to final design approved by the client. The thermal engineer would select and test various materials for elements of the building to ensure compliance with the thermal code. The structural

engineer would design and approve the structural elements of the design and ensure compliance with the structural code. The daylighting engineer would choose appropriate glazing materials and layout to ensure adequate daylighting whilst checking for potential overheating problems.

The three tools from the example are introduced in the following sections, along with the integrated data model (IDM) through which they will communicate. Following the tool descriptions there is a small analysis of the correspondences between the various design tool schemas and the IDM to give the reader an understanding of the type of information that needs to be mapped between the tools. The full schemas of the tools, the full mappings and the design team interaction specification can be found in Appendix E.

1.6.1 The Snart language

All of the development work presented in this thesis has been written in the Snart language (Grundy 1993) and LPA Prolog (LPA 1995). Snart is an object-oriented language, built on top of LPA Prolog, providing notions of abstract and instantiable classes, along with typed attributes and relationships, and class methods. Multiple inheritance is supported, as well as dynamic object classification. The Snart language has a graphical notation in addition to its textual form, and diagrams in this notation are used to illustrate work in this thesis both in this chapter and throughout the rest of the thesis. To allow those unfamiliar with Snart to understand these diagrams, a brief introduction to the Snart graphical notation is provided here using the diagram of Figure 1.3.

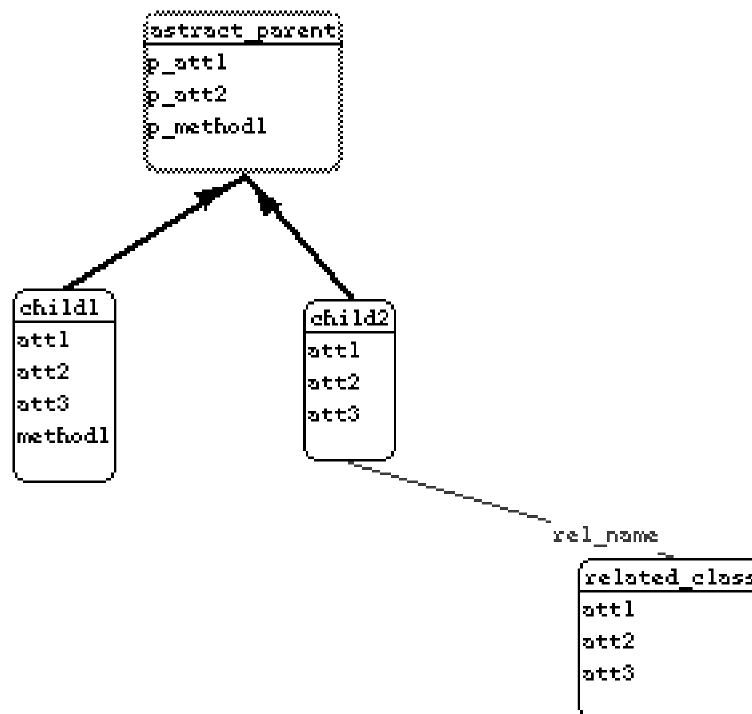


Figure 1.3 An example of the Snart graphical notation

Figure 1.3 shows four classes. A class is represented by a rounded-rectangle. The class name appears at the top of the rounded-rectangle, separated from attribute and method names by a single horizontal line. Figure 1.3 shows an abstract class (a class not able to be instantiated), through the use of a grey edged rounded-rectangle, here named *abstract_parent*. All other classes can be instantiated and are shown with a solid edged rounded-rectangle. Attribute and method names of a class are shown in the lower portion of the rounded-rectangle. No distinction is made between the representation of attributes and methods in the graphical notation. Inheritance between classes is shown through the use of a thick line with an arrow pointing to the parent class. Figure 1.3 shows *child1* and *child2* both inheriting from *abstract_parent*. Relationships between classes are shown through a thin line connecting two classes. The line is labelled with the relationship name closest to the class which is being referred to. In Figure 1.3, *child2* has a relationship named *rel_name* with the class *related_class*.

1.6.2 PlanEntry

PlanEntry (Hosking and Mugridge 1994) is a CAD-like tool to define building layouts. PlanEntry allows multiple 2D views of a base 3D model of a building (see Figure 1.4 for a snapshot of PlanEntry in use). Buildings which can be described in PlanEntry comprise: spaces; internal walls; openings; and roofs. View lines, as shown in Figure 1.4, can cut through any section of the building being described to provide a cross-section of the building. PlanEntry is a constraint-based system, allowing correspondences to be described between spaces, spaces and roofs as well as roof lines (e.g., the connection of the roof sections to form an L shaped roofline in Figure 1.4).

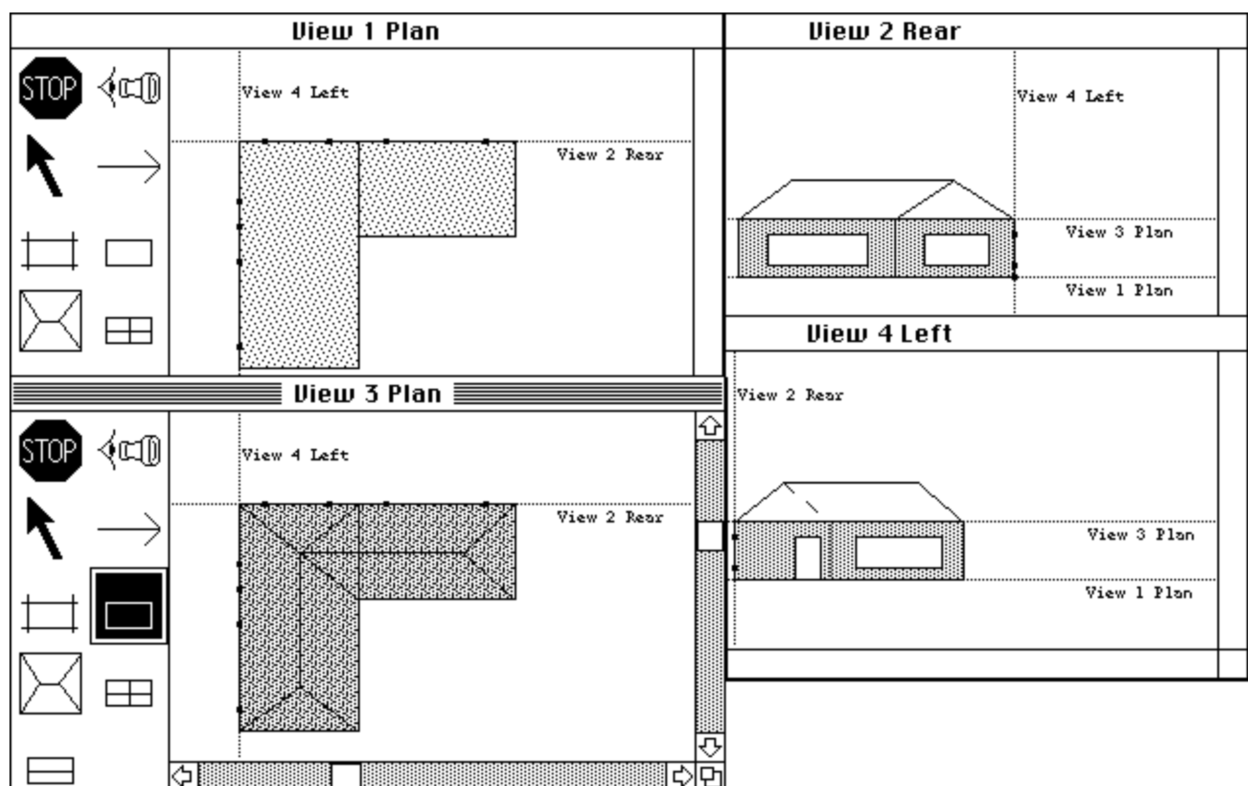


Figure 1.4 A building described in PlanEntry

The internal model utilised in PlanEntry is simple. A building is described as a set of spaces and roofs. Spaces can have openings and internal walls. PlanEntry includes a face generating mechanism which takes a building design and identifies all planes along which a wall lies. Using these planes all homogeneous faces for the spaces and roofs of the building are generated. A homogeneous face is a rectangular face which either wholly belongs to one space, or is the interface between two spaces (e.g., an internal wall). The main classes in PlanEntry are shown in Figure 1.5 (note that in this diagram the thin lines with an arrow represent components, e.g., a *pf_set* containing *pe_roof* elements, rather than inheritance as in Figure 1.3 with the thick lines).

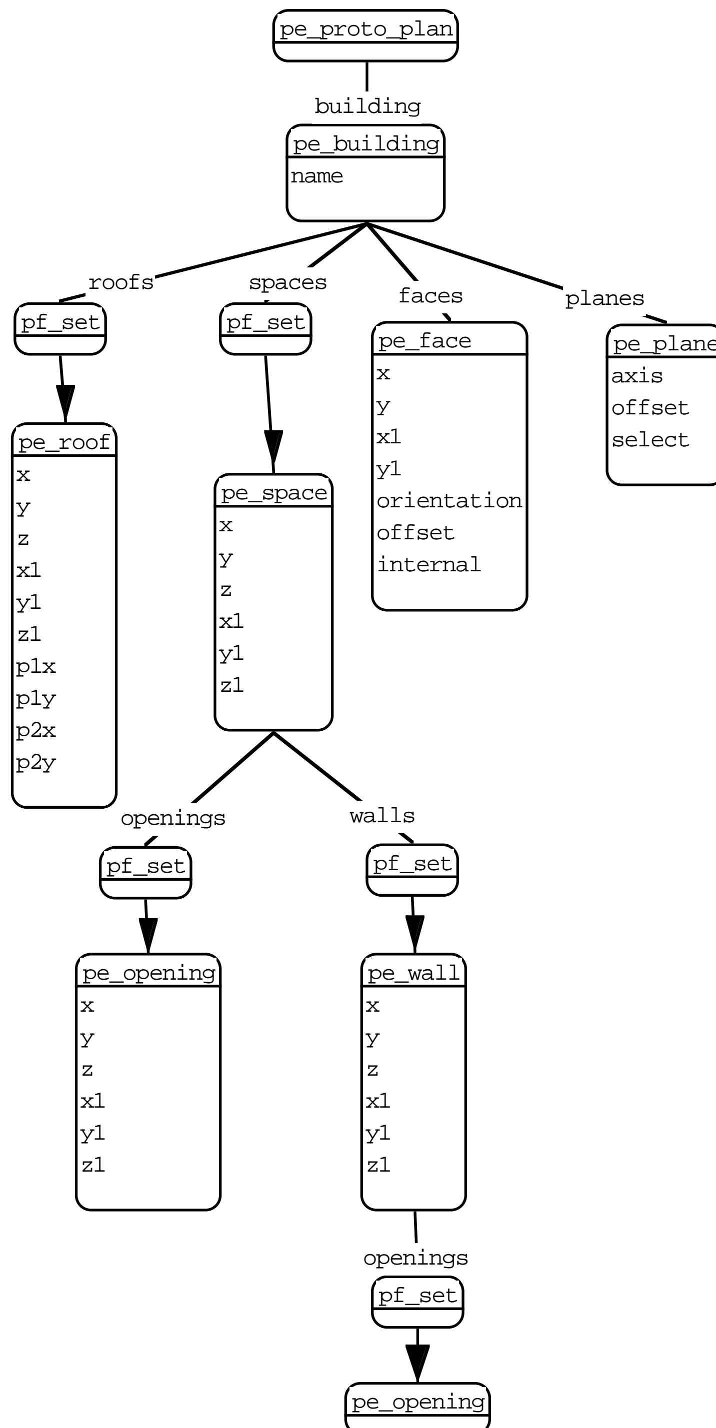


Figure 1.5 The PlanEntry schema main classes

1.6.3 FaceEditor

The FaceEditor enables material properties to be assigned to portions of walls and windows which lie along a single plane (see Figure 1.6). A plane in the FaceEditor equates exactly with a 2D plane-line which can be described in PlanEntry. The FaceEditor also allows the specification of bracing materials and bracing properties of walls in a plane. FaceEditor is a constraint-based system, allowing correspondences to be described between sections of material as well as windows and bracing lines.

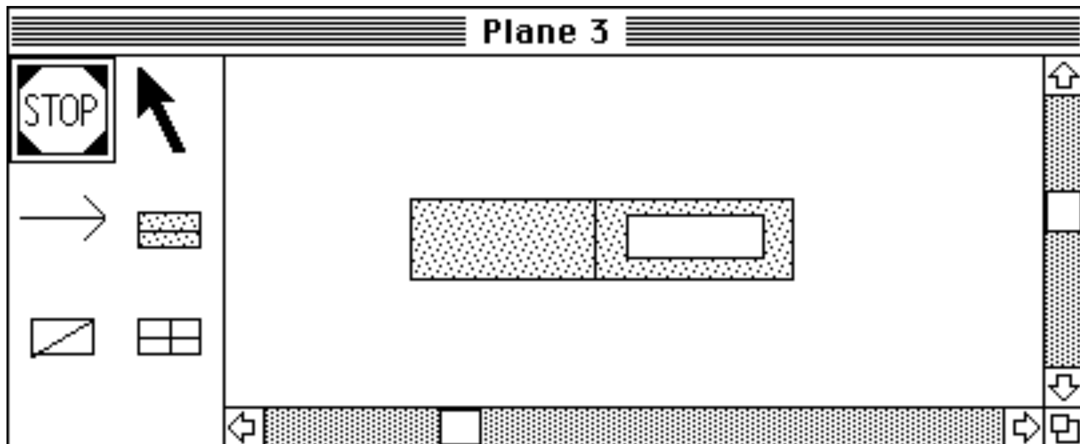


Figure 1.6 Wall and window materials being defined in the FaceEditor

The internal model utilised in FaceEditor is very simple. A plane being manipulated in a window consists of wall and window objects which are parallel to the plane (although they need not be on the plane) and rectangular sections of materials and bracing. The main classes in PlanEntry are shown in Figure 1.7.

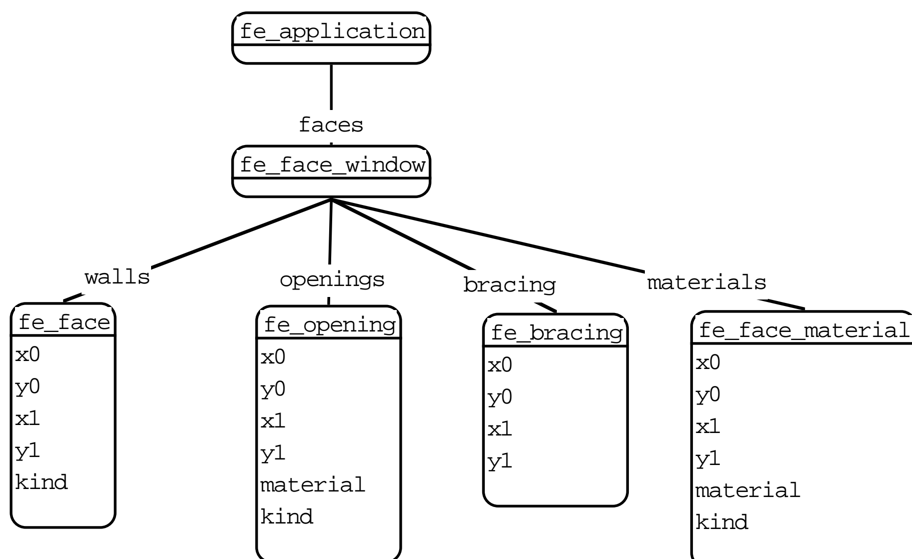


Figure 1.7 The FaceEditor schema main classes

1.6.4 VISION-3D

VISION-3D (Bourke 1989) enables a 3D representation of a model to be visualised in a wide range of formats. Models may be navigated through static camera placement, or by describing a path for a fly-through of a model. Models may be rendered in formats ranging from wire-frame through to fully shaded. Figure 1.8 shows a wire-frame representation of a model as in Figure 1.4. This tool is used in an off-line manner by importing a data-file containing a geometric description of a building to be rendered.

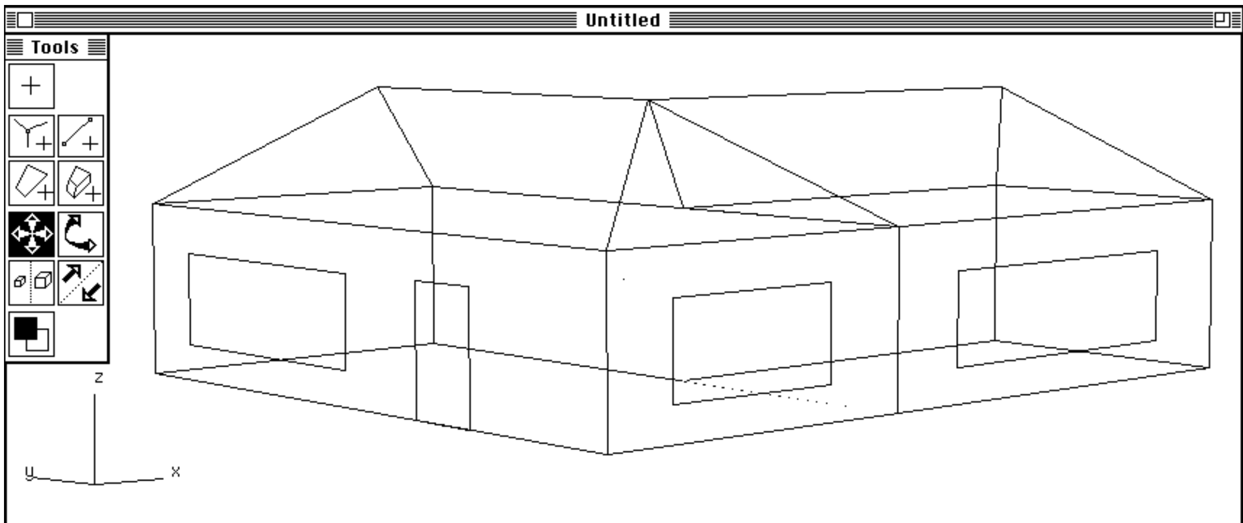


Figure 1.8 A wire-frame representation of Figure 1.3 in VISION-3D

The external model utilised by VISION-3D (see Figure 1.9) has a very simple format consisting only of polygons with some visual properties. Polygons may be grouped together to form more complex objects by specifying the same *object_id* for each polygon. This feature is used when describing roofs to VISION-3D.

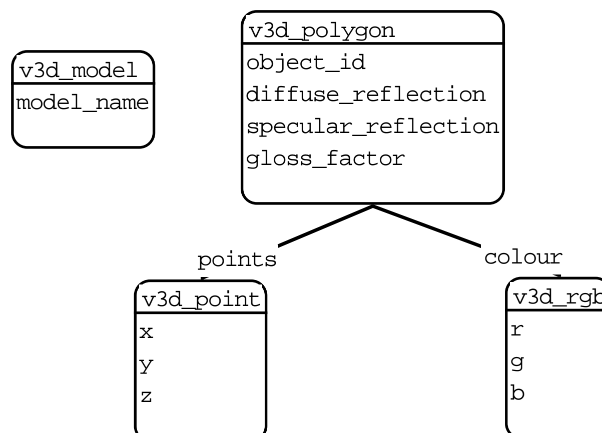


Figure 1.9 The VISION-3D schema classes

1.6.5 ThermalDesigner

ThermalDesigner (Amor et al, 1992) helps a designer to check that a building design meets the requirements of the New Zealand thermal insulation standard for residential buildings, NZS4218P (SANZ, 1977). It is based on an approach developed by the Building Research Association of

New Zealand (BRANZ) as a paper design guide (Bassett et al, 1990). The Figure 1.10 shows the forms based interface to the application. The tool has been designed to be used interactively with PlanEntry and FaceEditor providing basic information required to perform the ThermalDesigner analysis.

ThermalDesigner has a very simple model of a building being concerned primarily with the total building area, and the area of walls and windows facing in various directions. To this extent building information from external sources is usually aggregated together to provide the information required by ThermalDesigner.

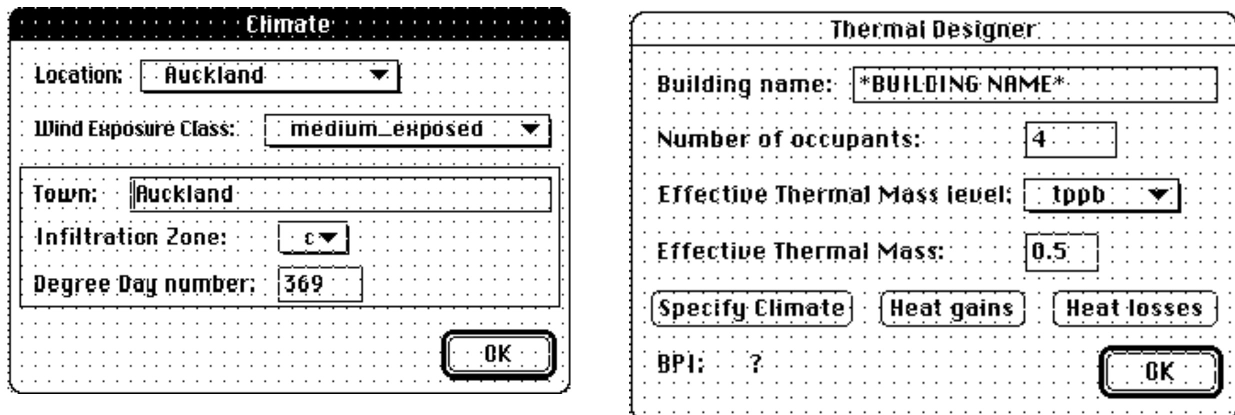


Figure 1.10 Control windows in ThermalDesigner

1.6.6 IDM

The IDM (see Figure 1.11 for a graphical schema) is adapted from a previous integration project (Mugridge and Hosking 1995). The adaptation required the expansion of the old IDM to handle the data requirements of the PlanEntry and FaceEditor packages. The model of a building in the IDM is based on a very generalised and partially redundant notion of a building. A building can be viewed as a set of spaces or as a set of faces. The space view of a building provides for hexahedrons which are placed orthogonally to the major axis. These spaces can be either interior spaces or roof spaces. The main information provided through the space view of the building is the easy determination of space abutment, though this information can be calculated from the face views. The face views break a building into rectangular faces of arbitrary size which have certain properties. In the current IDM the properties which are modelled are geometric, material and bracing properties. An IDM face can transcend space boundaries, and in many cases it is imperative that is does so (e.g., a multi-storey bracing element).

1.6.7 Correspondences between schemas

There are many obvious similarities between the schemas of the tools described in this section, along with many differences. The correspondence between each of the design tools and the IDM is further detailed in this section.

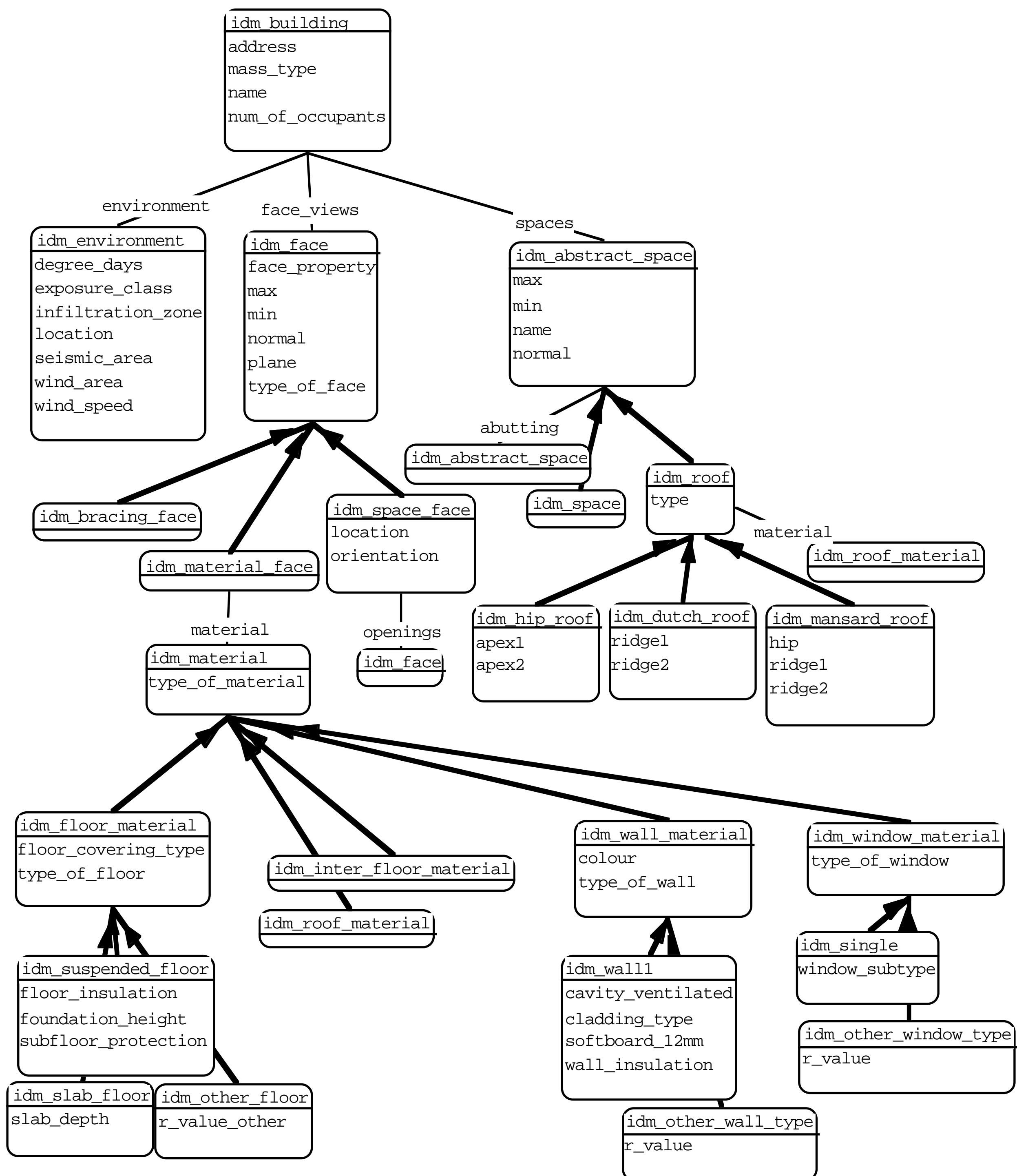


Figure 1.11 The IDM schema classes

PlanEntry's model of a building maps almost directly into the IDM. Their notions of a building are similar and a PlanEntry space is identical to an IDM space. The roofs that may be created in PlanEntry are hip roofs in the IDM schema. The faces around spaces which are generated in PlanEntry correspond to the geometric faces in the IDM (*idm_space_face*) and PlanEntry openings become openings of an *idm_space_face* in the IDM.

The FaceEditor's model of a plane also maps almost directly into the IDM. The notion of faces openings, bracing and material in a plane map directly to geometric faces, openings, bracing faces and material faces in the IDM.

VISION-3D has no model of a building, but geometric elements in the IDM such as geometric faces, openings and roofs are easily translatable into the polygons required in VISION-3D.

ThermalDesigner has a very simplified model of a building, being concerned with areas of elements facing certain directions. Therefore, the building structure in the IDM is not found in ThermalDesigner, however, a plane structure is present in ThermalDesigner to represent walls, windows, floors and roofs which maps closely to the IDM.

1.7 Outline of Thesis

Chapter 2 examines the development environment required for an integrated design system. The requirements for such an environment are specified and a possible structure put forward. The requirements of each individual component of this development environment is detailed, along with related research in the area. The individual components are discussed in Chapters 3 to 7.

Chapter 3 looks at what is necessary for a schema modelling and development environment along with the various modelling languages. An implementation of a schema modelling and development environment is presented with demonstrations of how it meets the previously specified requirements.

Chapter 4 introduces the problems of mapping between schemas as defined in environments such as that implemented in Chapter 3. The mapping problem is analysed to provide some measures of what is required and a range of mapping languages are examined to determine their strengths and weaknesses. The chapter summary shows the various language trade-offs and shows a need for a higher-level bidirectional mapping language. Such a language is described in Chapter 5.

Chapter 5 introduces the view mapping language (VML). This language provides a high-level declarative approach to modelling correspondences between schemas. The main constructs of the language are described along with a graphical notation for describing mappings. VML is shown to have a greater expressive power than previous mapping languages.

Chapter 6 details the requirements for a modelling and development environment for mapping languages. The system requirements are very similar to those detailed in Chapter 3 for schema modelling, and a similar approach is followed in demonstrating the utility of such an environment.

Chapter 7 details the requirements for a project specification notation. A developed notation, called CombiNet, is then presented. This notation allows users and their design functions to be modelled, along with flow of control for a project.

Chapter 8 examines the testing and implementation environment required for an integrated design system. The requirements for such an environment are specified and a possible structure put forward. The requirements of each individual component of this development environment is detailed, along with related research in the area. The individual components are detailed in Chapters 9 to 11.

Chapter 9 puts forward the requirements for schema instance management in a design system, looking especially at the requirements imposed by changing schema definitions. A suite of tools developed and used in this thesis are presented to highlight the benefits offered by tools meeting these requirements.

Chapter 10 describes an implementation of an interpreter able to map information between data stores and applications, through specifications in the view mapping language notation described in Chapter 5. Difficulties in implementing the high-level, declarative, and bidirectional, VML notation are identified and addressed throughout the chapter.

Chapter 11 describes an implementation of a control system able to handle a running project through the CombiNet project specification notation described in Chapter 7. This system provides functionality to a project manager as well as to the individual actors in a project by simulating the state of the running project.

Chapter 12 presents the conclusions of the work undertaken, highlighting the advances made in the understanding of the requirements of integrated design system development. A discussion of required future work and research concludes the work.

Chapter 2

The Project Development Environment

Chapter 1 provided an introduction to an area in which integrated design environments are required. In Figure 1.2 and Section 1.4, one framework for an integrated design system was discussed, with internal components and connections which are generally agreed upon. However, even with the framework in place, a large gap still exists between what must be created and how to create it. The integrated design system shown in Figure 1.2 can be implemented in a variety of system with links to relational and object-oriented databases, where many of the modules operate as independent systems utilising a brokering architecture to communicate data between modules. However the system is implemented, the same major conceptual sub-systems exist in the integrated design system. In this chapter, the environment required to develop, implement, and test the sub-systems of an integrated design environment is specified. These environment requirements form the basis for the work presented in this thesis.

The development of an integrated design system involves a number of actors playing various roles. These actors may belong to different organisations. For example, the developers of DT schemas and mappings are usually from the organisation which developed the design tool. The developer(s) of the IDM may be from the organisation implementing the integrated design system, or, as in the ISO-STEP development, from many companies and research institutes throughout the world. The project coordinator is usually from the organisation utilising the integrated design system. It is clear that whatever environment is chosen for the modelling of aspects of the integrated design system, there needs to be open communication links between all actors involved in the specification.

Considering only the modelling and specification components of the integrated design system outlined Section 1.4, the models which need to be developed are: the IDM schema; the schemas

for the DTs; the mapping specification between IDM and each DT; the interface between a DT schema and its data-files; and the project model. Though there needs to be very close links between each of the environments used for the development of each of these models, each modelling aspect and its requirements can be considered independently, as detailed in the following sub-sections. The structure and requirements section below presents informal requirements for a project development environment, as do all the sections considering the various components of such an environment. This approach is taken as the proposed system is very large, and complex, which makes the use of formal approaches infeasible in their current state.

2.1 Structure and Requirements

The development of an integrated design system is not a linear process. There is a large amount of interaction between developers of every sub-system of the environment. Examples of the types of cycles and interactions which occur are detailed below:

- The schema for a DT is developed either by the developers of the DT or by the creators of the integrated design system, in consultation with the developers of the DT. The DT schema describes the external interface of the DT as seen by the integrated design system, along with constraints imposed by the DT which must be taken into account by the integrated design system. During this process, the modeller must detail the DT schema in collaboration with those developers responsible for the implementation of different aspects of the DT. These developers are aware of the implicit constraints imposed by their implementation methods for these aspects of the DT. During this process the schema must be reviewed by the DT developers, reasons for constraints imposed on the schema documented, and fixes to the schema actioned by the modelling coordinator.
- During the IDM development, the requirements of all DTs which are going to be used in the integrated design system must be taken into account. This requires an iterative process of development by the IDM modeller(s), and checking by all the DT schema modellers, to ensure requirements are met. IDM modification requests by the DT modellers must be actioned and coordinated with possibly conflicting requirements from other DT modellers.
- At the point where the IDM has reached a fairly stable state (few major restructurings taking place), the specification of what needs to be mapped between the IDM and a particular DT may be undertaken. This specification lays the groundwork for the implementation of the mapping between the implemented IDM repository and the DT interface. This mapping specification is likely to be developed by the DT modeller in conjunction with one of the IDM modellers. The workers from these two teams need to coordinate their work and may develop the mappings at the same time as the IDM is being developed and checked against DT models. The mapping specification needs minor updates as the IDM changes. This entails coordination and communication between IDM developers and the affected DT mapping specifiers.

- The specification of the use of an integrated design system for a particular project can be undertaken at this stage. Generally, the project manager specifies the actors who will work on a project and the design roles they will play in the project. Individual actors are likely to specify the DTs they will use to complete their various design roles, and in collaboration with the project manager develop a flow of control specification for the project. This requires negotiation and collaboration between various actors and the project coordinator to identify exact roles and responsibilities, and to ensure that the resources requested by actors are available for the design roles they need to complete.
- Though the processes described above appear to flow in a continuous manner from modelling through to project implementation, this is a major simplification of the actual process. It is likely that there will be changes at all levels described above, requiring continuous restructuring of the integrated design system as new DTs appear on the market which particular actors may wish to utilise. New DTs may force changes in the IDM as well as new mapping specifications, mapping testing and changes to the project specification. It is also likely that the needs of the project require adjustment during the lifetime of the project, adding new actors as unusual problems are encountered, removing actors who may not be performing as required or become ill, adding new flows of control to force some aspect of checking that was previously not deemed important, or adding new paths to speed completion of the project. Thus changes are likely at every point of development and use of the integrated design system. This requires a close coupling between the implementation of the integrated design system and the environment used to specify the structure and requirements of the conceptual integrated design system.

The types of interaction detailed above indicate that the development environment for an integrated design system requires a diverse mix of modelling tools and implementation tools. However, to analyse the requirements of the development environment, and in the presentation of solutions to these requirements, the problems are specified in two categories: the modelling and specification tools required; and the implementation and testing tools required (see Chapter 8). Though this is an unnatural split, given the interactions specified above, it does enable the issues for each of these sections of the environment to be considered independent of competing and distracting issues from the other category of requirement.

2.2 Schema Modelling and Development

In all integrated design systems to date, the development of schemas has consumed a large proportion of the development time of the system. The amount of time expended in this phase is likely to change over the next decade as the schemas of the most commonly used design tools become available for general use. It is envisioned that these schemas will be offered, though perhaps not developed, in a common schema language (e.g., EXPRESS in the A/E/C community)

as part of the distribution of the design tool. Hopefully also in the next decade, a standard IDM will be developed for use in individual design areas (e.g., major APs from ISO 10303 for the A/E/C community). However, until this development occurs, and probably for some time after that, there will be a significant requirement for the specification of schemas.

2.2.1 Requirements of a schema modelling environment

A very general and widely applicable schema development environment must satisfy many diverse requirements. Some of the most important follow:

Modelling languages: a wide range of languages is used by schema specifiers. For example, in the A/E/C modelling domain, data specification languages such as ER (Chen 1976), EER (Gogolla 1994), NIAM (Nijssen and Malpin 1989), IDEF1X (General Electric 1985), and EXPRESS/EXPRESS-G (ISO/TC184 1992) are commonly used. As these languages take a considerable time to learn and to become proficient with, many specifiers would resist moving to a new standardised language. These languages have various strengths and weaknesses (e.g., ER has a direct mapping to relational database specifications making ER models very useful when a relational database is required to implement the system). Hence, using one language to the exclusion of all others can limit the comprehensiveness of the developed system. Schemas are not usually initially developed as pure data models. Most are developed in association with activity and process models which detail related aspects of the domain being modelled. In some A/E/C projects, models are developed detailing aspects of the domain such as activities, data definitions, process, and ontologies using interlinked modelling methods (see Mayer 1994 for the IDEF family). For example, AP 228 (the HVAC model in STEP; ISO/TC184 1995) is being written in EXPRESS in association with an IDEF0 activity model (Mayer 1990). Therefore, some link between the data specification environment and other associated specification environments needs to be supported. The schemas currently being developed consist of hundreds of object types requiring a structured approach to their development. This often means that the modelling environment needs to offer graphical and textual notations of the languages to allow views of varying levels of complexity and generality to be specified (Meyers 1991).

Consistency and negotiation: the development of a large schema can involve experts from many institutions, and, especially with ISO standards, from several countries. To enable schema development with multiple developers, a strategy needs to be adopted to ensure the consistency of the final schema. This strategy must be capable of ensuring that all modellers are aware of recent changes to the schema and can be certain that they are working on the most up-to-date version. In situations where modellers are geographically dispersed, or where the modellers are not working in close collaboration, large numbers of changes to the schema may need to be notified to modellers when they come to co-ordinate. With several experts involved, conflicts between the modellers are certain to arise. The modelling environment needs to facilitate negotiated settlements to conflicts, and to document the settlement process.

Documentation and navigation: decisions affecting the development of a schema are made throughout its specification, by individuals, institutions, and by consensus at meetings. The reason for, and reasoning behind, these decisions is important when resolving conflicts, as well as allowing for later validation of the schema. This creates a requirement to collate documentation regarding these decisions during the schema development. Though documentation at this level can be unpopular, especially where there is very little justification for a decision (e.g., the committee wanted to go home early), its collation is vital to ensure a schema which can be further developed as well as being trusted. During the development of a large schema the specifiers will need to navigate and evaluate current versions of the schema at various levels of generality. With documentation associated with the schema there are many new ways of indexing, and therefore accessing, schema information which need to be made available through the modelling environment (e.g., viewing all modifications by a particular specifier).

The different types of schemas which need to be developed in an integrated design system are described in the sub-sections below. Chapter 3 provides a more detailed description of the requirements of a modelling environment and presents a new modelling environment (EPE) which supports many of the requirements specified both here and elaborated in Chapter 3.

2.2.2 IDM schema

As described in Section 1.4 the IDM schema is the central schema through which all information is communicated. It provides structures to satisfy the requirements of all DTs and actors who will use the integrated design system. The development of the IDM is an arduous process requiring collaboration and communication between the various domain experts as well as between the integrators of the design tools. Computerised modelling systems currently in use provide little high-level modelling support to the developers of the schema. Often these systems can provide only one view of the entities being modelled using only one modelling paradigm, or very limited consistency maintenance between multiple views of the same entity.

2.2.3 DT schemas

As described in Section 1.4 the DT schemas reflect the information requirements of design tools, both their input and their output. The DT schemas contain many constraints upon their data structures to reflect the capabilities of the design tool which utilises the data from the schema. In most cases this equates to a very restricted subset of what can be described in the IDM. However, with all DT constraints described these schemas allow an integrated system to perform checks to ensure that valid models are passed through to the DT to manipulate.

2.2.4 Actor schemas

In many integrated design systems it is necessary to model the allowable roles of the actors. This task, usually undertaken by the project manager for each new project, allows the specification of an actor's frame of reference in a project. Actor schemas are defined inside the integrated design system and are defined in terms of the IDM in use. There are two schemas associated with every actor. One schema specifies the subset of the IDM which is viewable by the actor, the other specifies the subset of the IDM that the actor has the authority to create or update. The actor schemas provide a limiting boundary for an actor. Their implementation ensures that no actor can modify a portion of a design outside their area of responsibility or expertise. Actor schemas can also be used in conjunction with DT schemas to regulate the updating of a central model with results from the DT to those that are within the scope of the actor. Actor schemas are likely to change from project to project to reflect the actors' responsibilities in each project. They are also likely to be modified within a project to cope with changed responsibilities due to changing requirements during the project.

2.2.5 Related schema modelling environment research

There are a number of commercial tools used for schema development in the A/E/C world. Some of the most widely used include Design/IDEF (IDEFine 1995) and FirstSTEP (PDIT 1993). Boyle and Watson (1993) review a wider selection. There are also many tools developed at research institutes, many of which are available commercially. Some of the most common are CGE (Vogel 1991), PMShell (Luijten 1992), XP-IDEF and XP-EXPRESS-G (Poyet et al. 1990). Boyle and Watson (1993) also cover a range of the research environment tools. Almost without exception these tools provide very simple environments for defining schemas, mostly supporting only the graphical notation of a language (e.g., EXPRESS-G rather than EXPRESS). This often means only simple schemas can be developed in the tool (e.g., EXPRESS-G does not allow for constraints, functions, and procedures to be defined though they exist in EXPRESS). None of the tools allow overlapping views of a schema to be defined, and the majority only allow a single view to be used to define all attributes of an entity.

In the main, these drawbacks are due to the perceived market of the tools, which is the development of a published schema. With a paper copy of a schema, navigation amongst multiple views of a single entity is much more difficult to achieve. Though all the tools allow EXPRESS to be generated from the graphical notation (e.g., EXPRESS-G or NIAM), no connection is maintained between the graphical schema definition and the textual equivalent. This hinders a cyclic design process, as any additional specification in the textual form is lost when a new version is generated from the graphical representation. The XP-IDEF and XP-XPRESS tools from CSTB in France are the only tools in this area which maintain correspondences between textual and graphical definitions. This is achieved by generating the graphical views from the textual definition. However, this limits all graphical views to hierarchical decompositions of the textual

schema, rather than allowing predefined views of portions of the schema.

In the integrated software development research community, more sophisticated modelling environments are being developed. Integrated software development environments (ISDEs) which allow multiple overlapping graphical and textual views of a program have been available for many years (PECAN, Reiss 1985; Dora, Ratcliff et al. 1992; MViews, Grundy 1993; Meyers 1991). However, these tools are usually tied very closely to a particular programming language (e.g., Pascal for PECAN or Snart with SPE in MViews). This eases the compilation and testing requirements of such environments in comparison to abstract modelling languages (e.g., EXPRESS which has no direct mapping to any implementation language, and has constructs which can not be directly implemented in many popular languages, like C++). Recent advances in these environments (Grundy and Venable 1995) enable multiple modelling notations (e.g., EER and OOA/D) to be utilised in the same ISDE, and maintains consistency between overlapping areas modelled by the two different notations as well as between the multiple overlapping views within each notation.

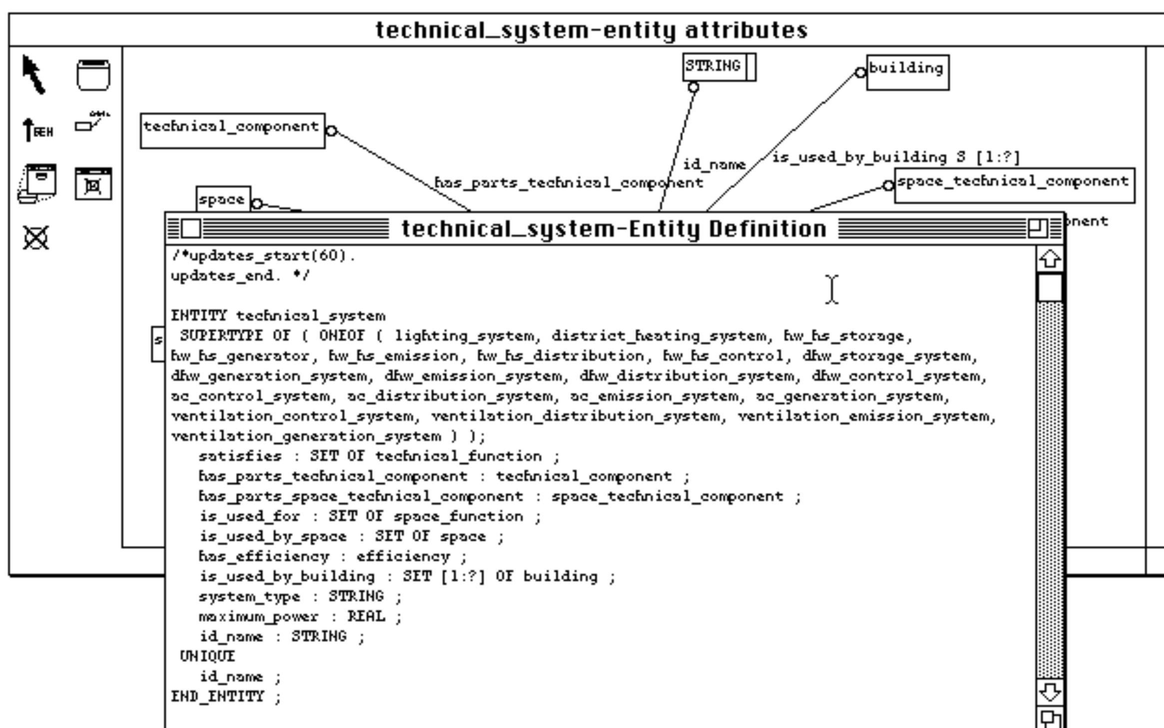


Figure 2.1 EXPRESS and EXPRESS-G views in the EPE modelling environment

2.2.6 Approach to a schema modelling environment

To be useable in the domain chosen for this thesis (i.e., A/E/C), the proposed schema modelling environment needs to support the EXPRESS and EXPRESS-G notations which are used in the development of the majority of schemas, but must provide much greater functionality than current commercial and research tools in the area, equalling that of ISDEs. This includes support for multiple overlapping graphical and textual views of the schema along with global consistency management, documentation procedures and powerful navigation features. The developed

environment (EPE) achieves all these goals, utilising the MViews system (Grundy 1993) and specialising it for EXPRESS and EXPRESS-G notations (see Figure 2.1).

Though only the EXPRESS and EXPRESS-G notations were implemented in EPE it is capable of extension to further modelling languages (e.g., Snart, NIAM, ER, OOA/D) due to the open architecture of the MViews system (EPE is currently capable of manipulating Snart models, but not modelling with the Snart formalisms). The full requirements and capabilities of the EPE system are discussed in Chapter 3.

2.3 Inter-schema Relationship Modelling

Specifying the relationships between entities in a DT schema and the IDM is currently performed either by actors from the DT development teams who wish to see their DT used in a particular integrated design system, or by integrated design system developers who want to show the utility of their system with links to as many design tools as possible. Currently, there is no formal modelling of this stage of the integrated design system development, and mappings are hand-coded for particular implementations. There are also no tools available to support the development of mappings between schemas. This is slowly changing as the benefits of modelling this stage of the development become more obvious. There are three possible scenarios envisaged for the specification of relationships between schemas.

- During the development of the IDM, the schema modellers for the IDM work in collaboration with the schema modellers of individual DT's. In this scenario the DT modellers specify a mapping from their DT schema to the IDM to help define the structures and attributes that are required in the IDM. This also guarantees that the IDM provides for all the data needs of their particular DT.
- After the development of the initial integrated design system a new DT is added, in which case the DT schema modeller specifies a mapping between the current IDM and their DT schema. If the IDM schema is comprehensive then no changes should be necessary to the IDM. If changes are required they can be passed through to the IDM modellers to handle.
- When the integrated design system is being used for a project, a new view of the data in the IDM may be requested by an actor. In this scenario either the actor, or specialised support personnel, will define a mapping from the IDM schema to the schema of the view required by the actor. This type of mapping has no possible effect on the IDM and does not involve modellers of either the IDM or DT.

2.3.1 Requirements of an inter-schema relationship definition language and modelling environment

To be useful in the development of an integrated design system, the language and modelling environment for the specification of relationships between schemas must offer the following facilities:

Mapping specification language: no mapping languages exist which allow a general high-level specification of correspondences between two schemas to be specified. However, in the same way that schema modelling notations are required which are independent of a final implementation, mapping modelling notations are required which are independent of their final implementation. To be easily understood and used by actors involved with IDM and DT schema development the mapping notation needs to be a high-level declarative language (i.e., leading to small definitions) rather than a low-level procedural language (i.e., leading to verbose definitions). Given the range of schema modelling languages used by actors the mapping notation should not be aimed at complementing a single schema modelling language. It should instead aim to encompass the generic facilities required in mapping in general.

Links to schema development environments: there is a close dependency between schemas and the definition of mappings between them. A mapping modeller needs to be informed of all changes a schema modeller makes to the schemas referenced in their mapping specification. Conversely, the schema modeller needs to be informed of all modifications needed in their schema to meet the requirements of the mapping specification from individual DT's which have information requirements not met in the current IDM structures.

Consistency and negotiation: though the definition of a mapping is only likely to involve one or two actors conversant with the schemas being mapped between, there is a strong requirement for them to remain consistent with each other as well as with the schemas the mappings reference. As new mapping notations are developed, it is likely that mapping portions will be specified in different languages and in views of varying levels of generality. These will need to be kept consistent with each other. For similar reasons to schema modelling (Section 2.2.1) it is imperative to ensure that all modellers are aware of recent changes to the mapping specification and that they can be certain that they are working on the most up-to-date version, and that conflict resolution strategies are provided.

Documentation and navigation: the mapping specification details the transformation through which data is moved between models. The reasons for using particular transformations and the perceived constraints being addressed by each mapping need to be readily accessible by those debugging or validating the mappings at a later stage. The modelling environment should help to document all reasons for the mapping being specified, as well as points where modifications are made to the schemas being mapped between. This includes tracing the modeller responsible for particular mappings, the reasons for individual or mass

changes, and offering documentation views to complement the mapping views. The modelling environment also needs to facilitate navigation through the mapping views, both to all variants which perform mappings on the same types of objects (e.g., to check for total coverage of the specified mappings) as well as to schema views for the various entities in the mapping (e.g., to check the types of attributes being mapped between).

2.3.2 Related inter-schema relationship modelling languages

Until very recently no general purpose mapping notations were available. However, with the development of the ISO-STEP schemas, the need for mapping specifications has been recognised and new notations developed. Prior to this, relevant formal work has concentrated on relational database views and schema integration (which are analogous to the mapping problem). To fit with the formal grounding of relational databases, these languages have provided very restricted capabilities, but ones which guarantee various properties of the final system (e.g., that all data can be mapped in both directions under all conditions). Relational database views (Ullman 1982; van der Lans 1988) can define arbitrarily complicated mappings as long as they are unidirectional. Bidirectional views, however, can only be described with a severely restricted set of relational operators (Banchilhon and Spyratos 1981; Dayal and Bernstein 1982; van der Lans 1988; Harrison and Dietrich 1994). Schema integration and heterogeneous database systems (Batini and Lenzerini 1984; Batini et al. 1986; Navathe et al. 1986; Motro 1987; Bright et al. 1990; Kim and Seo 1991; Qutaishat et al. 1992) implicitly allow bidirectional views between schemas or databases. However, to achieve this result, they restrict the range of operators allowed to merge the schemas. This greatly reduces the number of schemas that can be integrated, and entails that semantic mismatches in schemas have to be resolved (by modifying the original schema) before a mapping can be attempted. This is not a viable approach where existing tools with fixed schema structures exist.

Research outside of the database area comes from various sources, mapping being a general problem in all computer science domains. Garlan (1986) describes a system for integrated programming environments based on the ability to define a type conversion. Lee and Malone (1990) describe a scheme for communication among groups with different type hierarchies with specific reference to a mail/message system. These are similar to the approach taken by Eastman et al. (1995) in EDM-2, where mappings are assumed to traverse a hierarchy in a type lattice. However, many mappings can not be achieved by traversing a type lattice, yet may still be described through functional or procedural code. The following research systems allow sophisticated views to be defined in their respective domains. However, they only allow unidirectional mappings from a single specification, and provide no high-level language for the final specification of the mappings. Views for objects in OO-environments are discussed in Hailpern and Ossher (1990). The KIF (Knowledge Interchange Format) allows agents to map between different knowledge representations (Genesereth and Fikes 1992; Khedro et al. 1994). Views can also be specified in visual programming environments (Ambler and Burnett 1989, and

see MViews, Grundy and Hosking 1993b). Constraint-based systems are one of the only domains which provide a notation for bidirectional views of information with automated consistency (Bowen and Bahler 1991, 1992; Mugridge et al. 1995; Eastman et al. 1995). While these languages provide most of the structures required to map between different schemas, they were developed mindful of control structures which are not suitable for a general integrated design system. Hence they tend to be weak in some specification areas required by integrated design systems, namely the definition of conditions upon which sets of constraints can be used, and default values prior to application of constraints. Constraint systems also tend to assume all data and tools are present at all times, as well as being constructed in the same development language, in contrast to an IBDS where whole tools are connected and disconnected over time and exist and operate independently. Constraint systems therefore tend to model correspondences as though a single system is in existence.

In the A/E/C domains several integrated design systems have provided for mappings between their schemas (Gielingh 1988; Willems 1988; ESP, Clarke et al. 1989; Luiten and Tolman 1992; Wong et al. 1992; ATLAS, Greening and Edwards 1995 and ATLAS 1993; COMBINE, Augenbroe 1995a and 1995b; COMBI, Scherer 1995 and COMBI 1995; CIMsteel, Watson and Crowley 1995 and CIMsteel 1995). However, all of these systems have opted for mappings defined in their implementation languages without any formal modelling of what is required to be mapped. Only a single development in STEP has formally examined the unification of models. SUMM (Semantic Unification of Meta-Models, Fulton et al. 1992) provides a notation for describing the semantics of schemas so they can be integrated. It does not, however, lead to a methodology to map between the integrated schemas. As noted previously, the work in ISO-STEP has highlighted a requirement for mappings between schemas, and a plethora of languages have been proposed to meet this challenge (Transformr, Clark 1992; EXPRESS-M, Bailey 1994; EXPRESS-V, Hardwick et al. 1994, Hardwick 1994; EXPRESS-C, Staub et al. 1994; Operation Mapping, Bijnen 1994; XP-RULE, Zarli 1995). The ISO-STEP committees have now decided to create a standard mapping language for STEP development (EXPRESS-X, Wen et al. 1996) which pulls together the best (and worst) of the EXPRESS-V and EXPRESS-M languages. These languages range from very low-level specification (almost at the C++ level) through to partially declarative in style. They are all unidirectional in their mapping specification (though EXPRESS-V allows both mappings to be specified in the same definition), which misses the main benefit that these languages could provide in defining bidirectional mappings between overlapping schemas.

2.3.3 Related inter-schema relationship modelling environments

Incredibly, given the size and complexity of mappings defined with the range of mapping languages and environments described in Sections 2.3.2 and 2.3.3, there are almost no environments available to specify mappings. It is assumed that mapping developers will utilise standard text editors to define the mapping, managing all aspects of consistency by themselves. The only tools provided are post-processors which check the mapping specification against

schema definitions when the mapping is compiled into an internal form for the mapping implementation. The only exception is Operation Mapping (Bijnen 1994) which provides a hierarchical navigation system to select classes and attributes which are to be utilised in a textually specified mapping. This is a minimalist environment providing almost no benefit to mappers except a guarantee that correct names and paths are used in the mapping specification.

2.3.4 Approach to an inter-schema relationship modelling language

A new high-level declarative mapping language, VML (View Mapping Language), was developed for this project and is described in Chapter 5. A sample mapping can be seen in Figure 2.2. VML has the full expressive ability found in constraint languages with equational, functional and procedural specifications of mapping between attributes. However, VML also incorporates specific notions of the conditions under which a mapping can be applied and initial values for newly created objects along with the mapping specification. VML is implicitly bidirectional with explicit definitions of the schemas being mapped between to support independent tool-based mappings. A unique feature of the language is its ability to specify method-triggered mappings. This allows mappings between object-oriented systems that can lead to a more interactive integrated design system, for example, invoking functionality in a second tool based upon a user specified action in a primary tool.

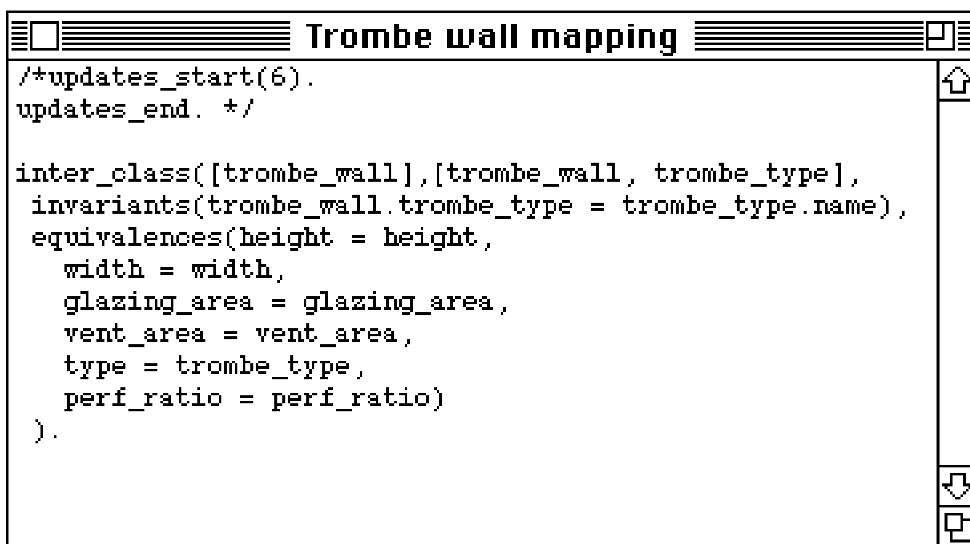


Figure 2.2 Example VML textual specification

2.3.5 Approach to an inter-schema relationship modelling environment

With a similar reasoning as for the modelling environment developed in Section 2.2.6 for schema models, the mapping modelling environment had to provide at least the same functionality as that found in ISDEs in computer science. It also needed to provide a tight coupling between schemas and the developing mapping. The VPE (VML Programming Environment), as described in Chapter 5, provides multiple graphical and textual views of mappings, similar to those described for EPE. VPE also manages the two schema definitions and verifies mappings being described

between the two schemas as they are developed. In a graphical view (see Figure 2.3) it provides a wiring mechanism to specify mapping relationships, whilst in the textual view (Figure 2.2) the full VML specification is available. Navigation facilities allow similar mappings and partial views to be identified. Visualisation functions allow the full class and mapping icons to be viewed, to identify features which have, or have not, been mapped. The developed VPE system utilised the MViews product (Grundy 1993) specialising it for the requirements of VML.

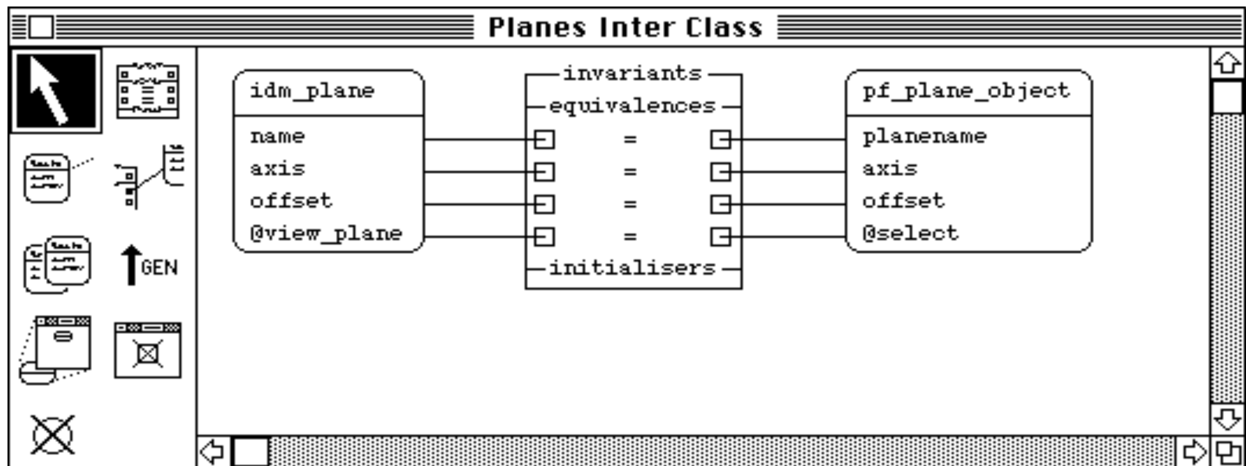


Figure 2.3 Graphical mapping specification in VPE

2.4 Design Tool Environment Modelling

Though the schemas of Section 2.2.3 define the data structures used by particular design tools and provide a definition of the constraints on data models which can be used by the design tools, they do not specify in what form the data is used by the design tool. The inter-schema relationships of Section 2.3 describe how data from models of differing semantics can be mapped, but not the syntax utilised by the design tools. It also does not describe how to invoke the design tool with the required data, or how to recognise when the design tool has completed its tasks. All that can be assumed about the data derived from the information modelling described in the previous sections is that correct data is held for the design tool, but this data is still held in some internal format foreign to the design tool (e.g., an object-oriented database).

To make the final step from internal data models to the form required by the design tools, the design tool input and output formats must be modelled as well as its invocation requirements and methods. This task has not been tackled in this thesis. However, it is clear that an ISDE of similar capacity to those referenced in Sections 2.2 and 2.3 could be developed from a general purpose development environment (e.g., MViews).

2.4.1 Requirements for design tool environment modelling

To capture the information required to manage individual design tools, the design tool environment model must be capable of specifying the following aspects of the design tool:

Input data format: the input of design tools is structured in very specific formats. This has to be accurately recreated to invoke a design tool with selected data from an IDM. The formats will include fixed format object per line data-files (from the FORTRAN punch-card era), data-files with headers before each section of objects, and data-files which allow free format data specification. The specification of the input data format must also specify the correspondence between the design tool model, as seen in the schema development environment, and the location of data in the input data-file. For design tools which require interactive input, the format must define the format of the data to be passed through, and also define how to select the requested data from the design tool model (e.g., a query based on the question being asked by the design tool). Where the design tool input data schema can change (e.g., a new version of the design tool), changes in the input data format must be passed through to the schema in the schema development environment and reflected directly in the model seen in that tool.

Invocation parameters: the majority of design tools need to be invoked with parameters detailing names of files with input data and often what type of simulation to run. This invocation data may specify input data-file names, the desired level of accuracy of the final result, standard library files used when searching for material properties, etc. The source of this data must be specified, e.g., user specified before invocation of the design tool, or default values based on the type of design function the tool is being used for.

Invocation method: in most operating systems it is possible to schedule a design tool to start operating, though the manner of starting tools is very different in various operating systems. With this specification an integrated design system can automatically operate tools which require no operator intervention. A model of invocation specifies the environment the design tool resides in (e.g., unix, VAX, Macintosh, MS-DOS, Windows) and how the parameters and design tools must be arranged to start the design tool. It must also define the commands to start the design tool, e.g., the name of the design tool and flags used.

Termination detection: for an integrated design system to schedule and operate a range of design tools it must be able to determine when they are working and when they have completed their operations. A model of termination specifies how the integrated design system detects the termination of a design tool, or the completion of its design function. This specification must cover cases from those design tools which start to perform a design function and end when it is done, through to those which are continually running and whose design function is completed without exiting the design tool.

Output data format: to be able to retrieve results from a design tool the integrated design system must be able to process the output of design tools. The range of textual output data-files is the same as those of input data-files (I assume that all graphical outputs will have a corresponding textual output). This specification must define how the output data is tied to the data supplied as input to the design tool, and it must define the correspondences to the output design tool model, as seen in the schema development environment. For design tools which are interactive the format must identify where the resultant data resides

amongst the prompts and queries of the interaction. Where the design tool output data schema can change (e.g., a new version of the design tool), changes in the output data format must be passed through to the schema in the schema development environment and reflected directly in the model seen in that tool.

2.4.2 Related design tool environment modelling work

The author has previously defined simple notations for interfacing design tools with very rigidly structured input and output data-files (Amor 1991). These methods were further augmented to provide a more flexible data format description based on DCGs (Williams 1990). In Pascoe (1994) a methodology is presented for representing GIS design tool data states including physical forms and location along with transformations to move between states. However, while this gives a method of representing the transformations that must take place to interface between various tools, it does not address representations necessary for automatic implementation of each transformation. This implementation is currently performed by hand-coded tools.

The above methods go some of the way towards what is required in the modelling of the design tool environment, but only for a limited set of design tools. Interactive design tools (e.g., knowledge-based systems) can not be handled by any of the methods described above. In Amor (1991) it was argued that defining an interface to these design tools could require as much effort as defining the design tool itself. These methods also only define the data requirements, they do not attempt to define the invocation parameters and methods required to start a design tool or to determine when the design tool terminates. Definition of these environmental parameters will be operating-system dependant and may prove to be impossible to define generically. However, the 'Tool Encapsulation Specification' project (TES 1995) does provide a standardised notation to describe tool invocation parameters and methods so that a TES system on any platform would be capable of determining how to run a particular tool. TES also provides for descriptions of methods to suspend tool execution or determine a tool's completion status.

2.4.3 Approach to a design tool environment modelling system

The modelling of the design tool environment is not attempted in this thesis as there is an emerging standard (TES) which looks as though it will provide what is required. TES has currently not been developed enough to provide a modelling system to support it. However, when a design tool environment modelling system is created as part of the project development environment, it will have to provide at least the same functionality as that found in ISDEs in computer science. It will also need to provide a tight coupling between design function specification (which defines the design tool used to perform the function) in the process model and a design tool's data requirements. As with the previous modelling systems, an environment built on top of a system such as MViews (Grundy 1993) would be able to provide the functionality and integration required to bring these models into the whole project specification.

2.5 Project Definition

As described in Section 1.4 the project model defines how an integrated design system will be used for a project. It provides a method to describe the users of the system, the roles they play, and the tools they will utilise to fulfil those roles. It also allows the definition of flow of control for project management purposes, allowing necessary process flows be defined, and enabling the determination of points at which concurrent design can take place. It also provides a tool to the actors to manage their design roles to ensure timely completion and hand-over of their work.

2.5.1 Requirements for a project definition notation and development environment

To be of utility in the development of an integrated design system the modelling environment for the project definition must offer the following facilities:

Project definition language: project managers need to specify the designers working on a project and the activities they need to carry out in their contribution to a project. Project managers also need to keep a constant overview of the activities completed in a project and those which are next scheduled for completion. To achieve this in a computerised environment project managers will require a notation to define the actors undertaking a particular project. The notation will also need to define the roles actors play in a project and the possible, or necessary, paths between the design functions that need to be completed to fulfil the design roles.

Links to schema development environments and design tool environment: flow of control specifications are tied to particular design tools which must be controlled by the implemented integrated design system. To achieve this it must know about the input and output schemas for all design tools, and actors, to be able to tie down the responsibilities of actors in a project. To be able to control the design tools associated with particular design functions it must also be able to link with the design tool environment which details the invocation procedure and termination detection for the design tool.

Documentation and navigation: the project control system is the most visible aspect of the integrated design system. To justify flows of control in the IBDS the project definition environment should help to document all reasons for the paths between design functions being specified. This includes tracing the modeller responsible for particular paths or design function specifications. The modelling environment also needs to facilitate navigation through the project definition views, to show all places that a particular design function could be performed, and by whom, and to provide some simulation of the flows to ensure that particular configurations are useable, or possible to negotiate.

2.5.2 Related project definition notation work

The majority of the project definition notations used to date have been activity modelling formalisms which are used in a specific restricted manner. For example, IDEF0 (Mayer 1990) activity models are used to model process flows by treating the relative horizontal positioning of activity icons as presupposing a temporal relationship. While this view of activity diagrams allows some forms of process flow to be defined it is very poor when attempting to handle many types of flows, e.g., recursive flows, concurrent flows, and those which are conditional on previous flow states. This is similar to the modelling capability of data flow, state diagrams, and Gantt charts which are generally linear in nature. Pert charts provide the majority of modelling behaviour required except that recursive flows are difficult to model and flows are very deterministic, which is fine for highly structured projects, but not all projects fall into this category. Many notations have been derived from the Petri Net formalism (Petri 1976, Jensen 1990). These notations range from IDEF3 (Mayer et al. 1992), which allows process flows to be described with AND and OR connections between states, through to VPL (Swenson 1993) which additionally allows specified conditions or constraints to help define paths in the work-flow. IDEF3 allows looping and recursive behaviour, but the semantics of looping with AND and OR conditions is not specified. The formalism is also developed independently of the actors involved in the flows which removes the possibility of defining the links to organisational requirements. In contrast, VPL incorporates notions of actors associated with process flows. A previous review of process modelling notations (Curtis et al. 1992) determined that a complete notation needs to specify functional, behavioral, organizational and informational requirements. This full range of requirements had not, at that stage, been incorporated into a single notation. However, during the COMBINE2 project (Augenbroe 1995a and 1995b) a two part modelling notation was developed (CombiNets, TU Delft and Amor 1993), based around Petri Nets, which models the four requirements of Curtis et al. (1992) to some extent.

2.5.3 Related project definition environment work

The specification of processes in a project has been part of the project management aspect of projects well before computers appeared on the scene. Following the paper-based process management many commercial computer tools have been developed to support the specification of flows, and in some of these tools to help analyse the practicality of the prescribed flows (PowerProject, ASTA 1996; Process Charter, Scitor 1995). These tools offer a very simple interface to define and refine what are basically linear process flows. Associated with the more sophisticated process modelling formalisms are tools of correspondingly greater sophistication. In COMBINE a CombiNet tool is offered through the CGE environment (Configurable Graphical Editor, Vogel 1991), this provides a simple hierarchical view specification of process flows and utilising design functions and actors. IDEF3 tools (ProSim, KBSI 1995; System Architect, Popkin 1996) provide simple single view specifications of process flows, but with links to IDEF0 activity and IDEF1X data models. VPL tools (Swenson 1993; Serendipity, Grundy 1996) provide

more sophisticated multiple view specification environments, but without the links to associated models.

2.5.4 Approach to a project definition notation

An extension of the CombiNet formalism is used to define process flows in this thesis (CombiNet was developed by the author whilst a guest at TU Delft for six months). This formalism takes two parts. The first part allows the specification of actors, their design roles in a project, and the design functions necessary to complete the various design roles. The second part is based around the Petri Net formalism (though the semantics are markedly different) and defines the possible flows between the design functions as specified in the first part of the formalism. The formalism incorporates aggregate process icons to represent complete sub-flows and incorporates actor overlays to allow different actors to be responsible for the same design functions in different parts of the process (see Figure 2.4 for an example of a portion of a process flow specification).

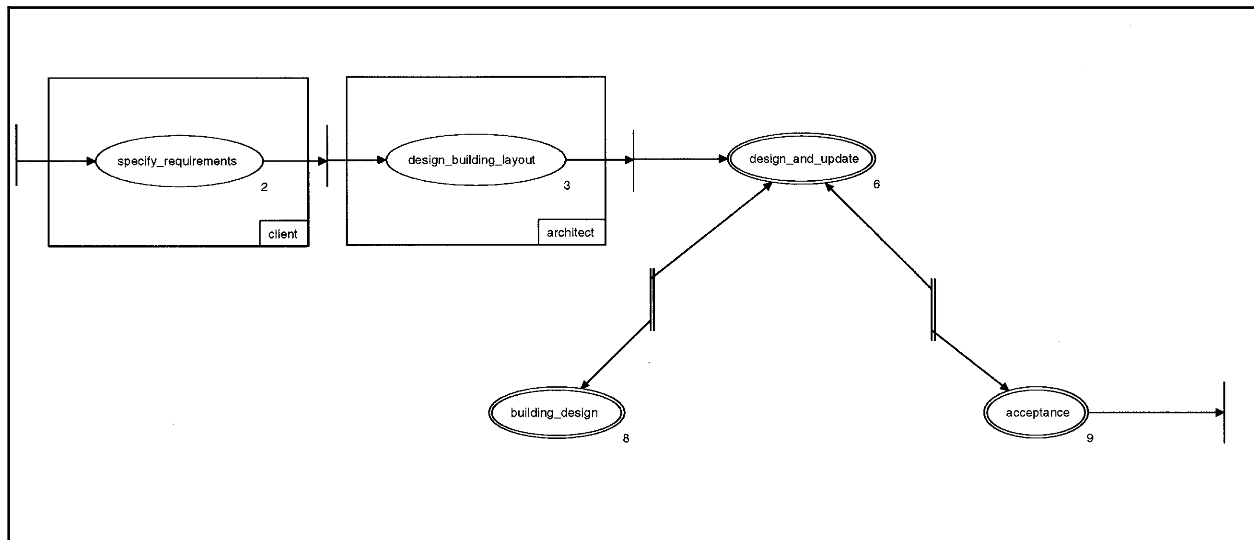


Figure 2.4 Project flow of control definition

2.5.5 Approach to a project definition environment

The CGE tool (Vogel 1991) was utilised in this thesis to implement the environment for defining process flows in a project. The choice of this tool was initially determined by the requirement in the COMBINE project for all modelling tools to be developed in CGE, which was available to all partners. Given the author's effort in defining the initial formalism in this tool, it was not felt necessary to re-implement the extended formalism in another environment. The CGE implementation of the formalism provides links between the actor and role specification, and the process flow specification. Though only single views of a process flow are supported during the specification, the tool provides hyper-linking between aggregate flows and their complete definition, allowing easy navigation around large process flow specifications.

2.6 Project Development Environment Summary

This chapter has described a range of individual modelling environments and modelling paradigms required to implement a project development environment. The individual environments have to model schemas, mappings between schemas, design tool parameters and project definitions. It is shown that these different models are all inter-related, so the total project development environment must provide for relevant communication between the different modelling environments. It is also shown that the individual modelling environments have many requirements in common. These include the ability to provide multiple views of information both textually and graphically and being able to maintain the consistency of the global model under changes in any view. The MViews development environment (Grundy 1993) is introduced and, through environment implementations, shown to provide the features required to implement the individual modelling environments as well as providing some of the interaction required between environments.

Two modelling environments, developed in this thesis for schemas and mappings between schemas, are described in detail in Chapters 3 and 6. The requirements for a schema mapping language are developed in Chapter 4, followed by the view mapping language definition in Chapter 5. The requirements for a project specification notation and an actual notation are described in Chapter 7. Taken as a whole these environments and modelling paradigms allow a full project development environment to be constructed.

Chapter 3

Schema Modelling and Development[1]

The definition of the schema for a particular domain is vital to the success of computerised projects in the A/E/C arena, whether it be the smallest application or the largest ISO standard. In all except the smallest of projects, schema development is carried out by a group of domain experts, each contributing to the part of the schema representing their areas of expertise. Development of the schema needs to be coordinated by one or more individuals who must ensure that the current schema version is passed through to all developers and that all input to the final schema is considered and acted upon in some systematic manner. This introduces a requirement for coordinated management and documentation of the schema development.

3.1 Introduction

The whole schema development process, unless closely managed, is open to sources of error and conflict. The current version of a schema must be propagated to all developers so that they are all considering the same schema. Modifications that are made from one version of a schema to the next need to be documented so that differences between versions are easy to identify (especially for very large schemas). Change requests and additions to schemas sent by developers all need to be documented and the action taken on them recorded so that developers know that their suggestions have been heard, and, if rejected, they know the reasoning behind the rejection. Multiple conflicting requirements between developers need to be negotiated to a final settlement which satisfies the majority of developers. Electronic management and design tools have the potential to solve all of these problems.

[1] Work presented in this chapter has been published in Amor et al. 1995.

However, existing software tools to help users through this process are limited in the scope of the problem that they can tackle (for example, see Vogel 1991; Luijten 1992; Poyet et al. 1990; Boyle and Watson 1993). Some tools assist in the development of a schema, while others check it for consistency, translate it to an implementation language, or allow an instance of a model to be perused. These tools do not tackle the problems introduced by multiple developers, and, where there are multiple tools to handle the process, they do not allow the use all of the tools together in an interactive and non-deterministic manner. To solve these problems, the author has designed an integrated modelling and development environment which provides many functions to users in a homogenous environment.

In this chapter the requirements for a modelling environment in a large modelling project are introduced. The EXPRESS Programming Environment (EPE), which tackles these requirements, is detailed, and its ability to meet the design requirements of a modelling environment is demonstrated.

3.1.1 Requirements for schema development

While the introduction above concentrates on requirements for the development of an IDM, many other schemas also need to be modelled in the development of an integrated design system, as discussed in Section 2.2 of this thesis. The development of DT and user schemas are tasks which can be of a similar size and complexity to the development of the IDM. They can require a similar number of domain experts to implement, and produce the same set of problems as discussed for the IDM schema development. Developing schemas for DTs and users introduces an added dimension to the management problems specified for the IDM as there is an association between these schemas and the IDM. Therefore, changes in these schemas (particularly in the user schemas) must be matched to the portions of the IDM that were influenced by the original schema specification.

In any large scale multi-partner schema development project there are a number of modelling support issues which need to be dealt with. For example, COMBINE included the following:

Communication and integration of DT schemas: the development of the building schema in COMBINE is based on a cyclic development process where DT teams supply the data view of their DT in the form of an EXPRESS schema (referred to as an aspect model). These DT schemas are taken as input by the central IDM development team for constructing a common integrated schema. This process employs a combination of top-down structuring and bottom-up expansion of the growing IDM. Resulting drafts of the IDM are subsequently distributed to the DT teams to inspect and refine. This process of iterative refinement may continue for many months and during this time all partners concerned in this process must be kept up to date with the changing schemas.

Documentation: during the schema generation process, the communication of schemas and schema constraints involves an intensive interaction between various teams in the project. All decisions and justifications for decisions must be permanently documented as they are made.

Mapping definitions: it is through the definition of the IDM to DT schema mappings that the consistency and adequacy of the IDM is totally checked by its clients (those who will use the data in the IDM). The mappings determined by the DT schema teams need to be defined formally, as these mappings describe the required interface between the DT and the IDM.

Support for multiple modelling paradigms: the modelling paradigms EXPRESS, EXPRESS-G, NIAM, IDEF1X and IDEF0 are well suited to different types of modelling. An ideal environment for COMBINE would support the specification of a schema using multiple paradigms.

Support for multiple views: the integration process is very hard to accomplish systematically, let alone automate (the required "meta modelling knowledge" is lacking), so it remains a tedious and non-deterministic process to combine multiple views into one coherent whole. To support this integration process the modelling environment should provide methods to enable views to be integrated into a central schema and to document the source of entities and structures in the integrated schema.

To support these requirements an ideal modelling support environment (MSE) needs mechanisms for: easy communication of schemas; generation and manipulation of multiple views of a schema in a variety of formalisms; annotation of the schema with documentation; and flexible update management to support iterative updates of the schema. Other important features would be support for the definition of inter-schema mappings and for integration of schemas, although it is recognised that, owing to the strong creative element in these tasks, automated tools are not, as yet, achievable. Another important issue is the ability to rapidly prototype a resulting operational system. MSEs should provide facilities to allow instantiation of schemas to be tested in a run time environment, e.g., to test mapping specifications and enhance the understanding of the underlying schema.

3.1.2 Schema specification languages

There is a plethora of languages available for modelling schemas. While there are overlaps between the expressive ability of these languages, there are also many differences in what can be expressed, in some cases due to the domain the language was developed for and in others due to the structure of the language. In the development of schemas the most commonly used languages fall into one of the following three categories: relational database schema definition, such as ER (Chen 1976), NIAM (Nijssen and Halpin 1989), IDEF1X (General Electric 1985); object-oriented schema definition, such as EER (Gogolla 1994), EXPRESS (ISO/TC184 1992); and data flow definition, such as DFD (Stevens et al. 1974), IDEF0 (Mayer 1990). All common modelling languages have graphical notations for defining schemas, and some languages also have a textual

notation.

In the ISO-STEP development there has been great pressure to use a single language in the development of schemas for the standard. Existing languages were felt not to be powerful or complete enough for the requirements of the STEP standard, leading to a decision to develop a custom modelling language. The resultant language, EXPRESS, and its graphical notation, EXPRESS-G, provide a similar coverage to that offered by other object-oriented specification languages, although without all of the standard object-oriented notions, such as class method definition. This lack is being rectified in the latest version, EXPRESS-V2.

Due to the adoption of EXPRESS as the standard language in the development of STEP many developers and integrated design system researchers have taken on EXPRESS as their modelling language to maintain easy compatibility with the results of the STEP standard. The problems they feel that they avoid through the use of EXPRESS are mainly the difficulties of moving schema information from one modelling language to another without loss of information in the transfer. In some cases these problems arise due to a mismatch between the modelling capabilities of the different languages, but in others the problem is due solely to poorly written translators. Due to this, EXPRESS is being used in a large number of projects and standardisation efforts faced with large modelling tasks in domains covered by the STEP standard (for example, ISO/TC184 1993; Gielingh and Suhm 1992; ATLAS 1993). Environments for modelling schemas, both commercial and research developments, are discussed in Section 2.2.1.5.

3.1.3 The EXPRESS and EXPRESS-G languages

A brief introduction to the EXPRESS and EXPRESS-G languages is provided here, to allow readers unfamiliar with the notation to understand the figures presented in this chapter. Many aspects of the EXPRESS-G notation are not covered in the following example as they do not appear in any of the diagrams in this chapter. A full specification of EXPRESS and EXPRESS-G can be found in the ISO standard (ISO/TC184 1992). For those interested in comparisons between graphical notations, the diagram in Figure 3.1 provides the EXPRESS-G view of the same example in Snart graphical notation presented in Figure 1.3.

Figure 3.1 shows four classes (*abstract_parent*, *child1*, *child2*, and *related_class*). A class (called an ENTITY in EXPRESS) is represented by a solid-lined rectangle with no other graphical embellishments. The name of the class appears inside the rectangle, and can be preceded by the keyword *(ABS)* to denote an abstract class (see the class *abstract_parent*). Attributes and relationships of a class are shown by single thickness lines drawn between a class and the type of the attribute or relationship. These lines are labelled with the attribute or relationship name and a small circle attaches the line to the type definition. For example, the class *child2* has attributes *att1*, *att2*, and *att3* of type string, real, and integer respectively. The class *child2* also has a relationship to the class *related_class* through the named relationship *rel_name*. EXPRESS-G also allows

aggregating types to be denoted in the diagram, as can be seen for the class *child1*. The attributes *att1*, *att2*, and *att3* are of aggregating types set, bag, and list respectively (denoted by a *S*, *B*, or *L* after the attribute name). As can be seen in Figure 3.1, these aggregated types can also show their bounds, if they are specified. The attribute *att1* has no specified bounds on the set, *att2* has a lower bound of 1 for the bag but no upper bound and the list for *att3* is constrained to be between 1 and 10 values. Inheritance between classes is shown with a thick line connecting two classes. This line terminates with a circle at the connection with the child class. In Figure 3.1 *child1* and *child2* both inherit from *abstract_parent*. Complex inheritance relationships can be defined in EXPRESS though these are not shown in any of the diagrams in this chapter. However, due to this potential complexity, the EXPRESS class descriptions always show both supertypes and subtypes for each class.

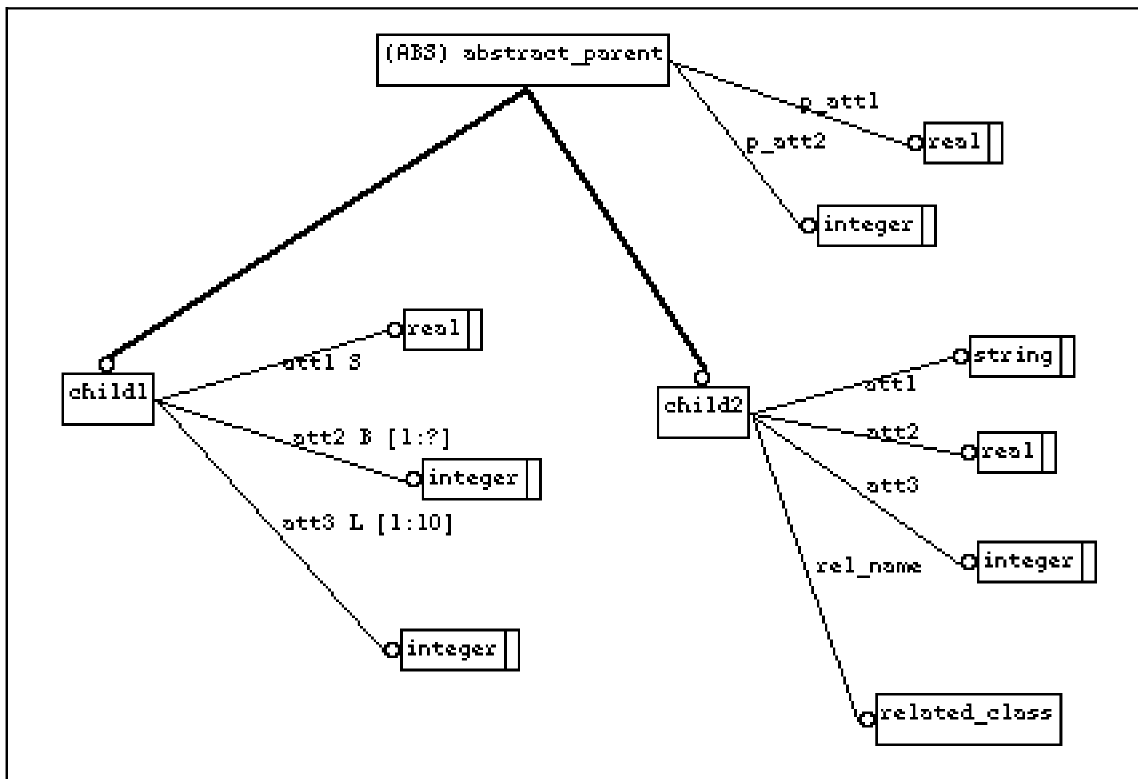


Figure 3.1 An example of the EXPRESS-G notation

3.2 Schema Development in the EPE Environment

The EXPRESS Programming Environment (EPE) has been engineered to support the modelling requirements detailed above for schema evolution and management. In EPE this equates to multiple graphical and textual views of varying degrees of complexity at different stages of the project. During analysis, simple graphical views, which embody high level concepts and relationships between them, are mapped out and manipulated. During early design, these simple graphical representations are fleshed out: constraints specified; attributes of entities added; and inheritance hierarchies fully specified. During late design, more detailed information becomes

available which is often best manipulated in free form textual representations of portions of the schema. During implementation, the developed schema is compiled, checked for syntax errors, and detailed models are loaded and checked for consistency. Throughout the iteration of these stages and during maintenance, modifications can be made at any of the levels described above and must be propagated to all dependent stages. EPE provides integrated support for each of these activities, using the MViews consistency mechanism to provide the required inter-view consistency.

3.2.1 Functionality offered by the EPE environment

In this section a description is provided of the EPE tools and view types available at each stage of development, and the methods of keeping views consistent with one another is described. These are illustrated using an example from the initial IDM of COMBINE. The COMBINE IDM comprises around 400 entities and 600 relationships; only a small portion, associated with technical systems, is shown here. The placement of technical systems in relationship to the *building* class in the IDM is shown in Figure 3.2.

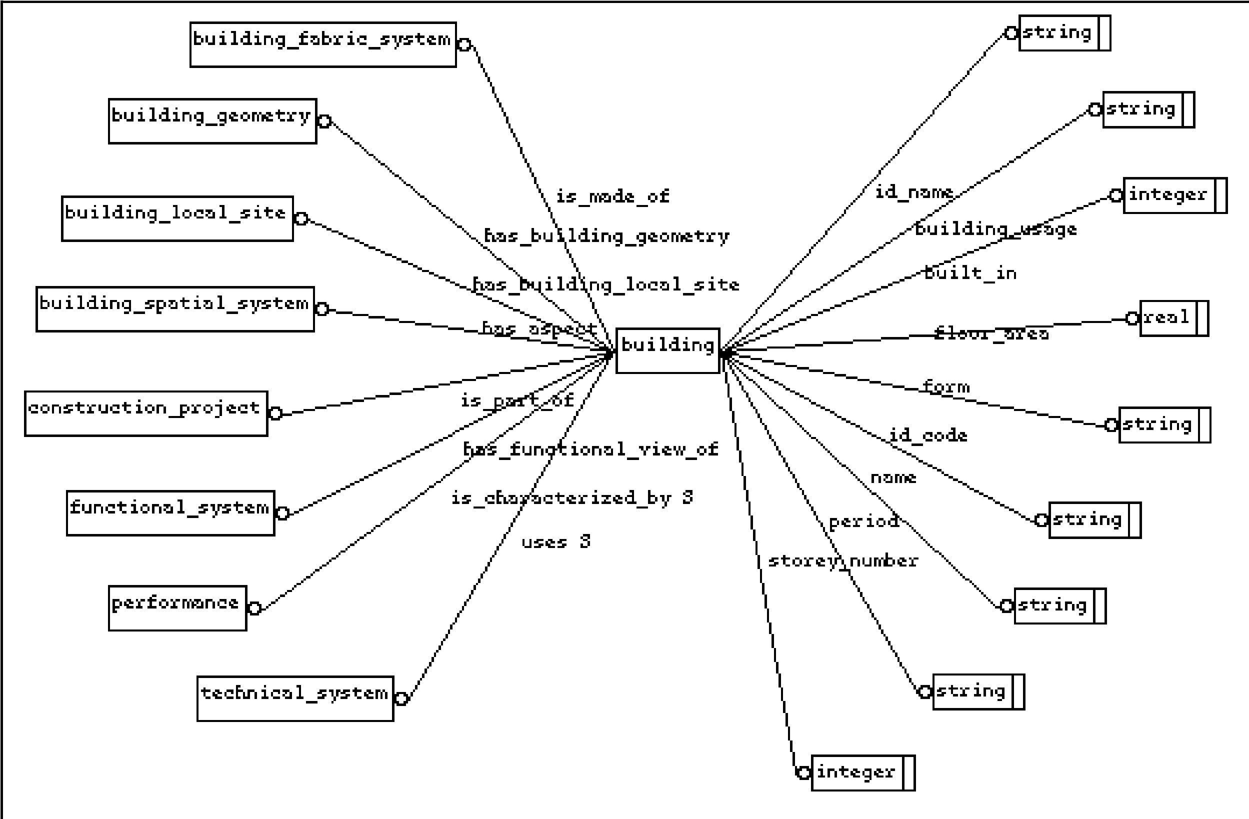


Figure 3.2 Use of *technical_system* in the COMBINE IDM

3.2.1.1 Analysis

Figure 3.3 shows the types of graphical views commonly used at the analysis stage. Two analysis views show portions of the inheritance tree for the *technical_system* entity (those entities dealing with ventilation and air conditioning) specified using EXPRESS-G (thick lines represent

inheritance links in EXPRESS-G). Each view is constructed by direct manipulation using tools selected from a tool palette, to the left of the view.

The user is free to create as many views as is desired, and may freely lay out and populate each view, either with new information or with information entered into other views. The information in each view is mapped through to the canonical representation of the schema as the view data is entered, and any similarities or conflicts with the existing data are resolved as it is created. This ability to construct multiple views permits both general purpose and specialised views to be constructed. The former may be used to obtain an overview of the system under construction, the latter to focus on more detailed parts of the system. The proliferation of views means that navigation tools are needed to quickly access desired information. EPE provides inter-view navigation using both menu-based search facilities and automatically constructed hypertext links.

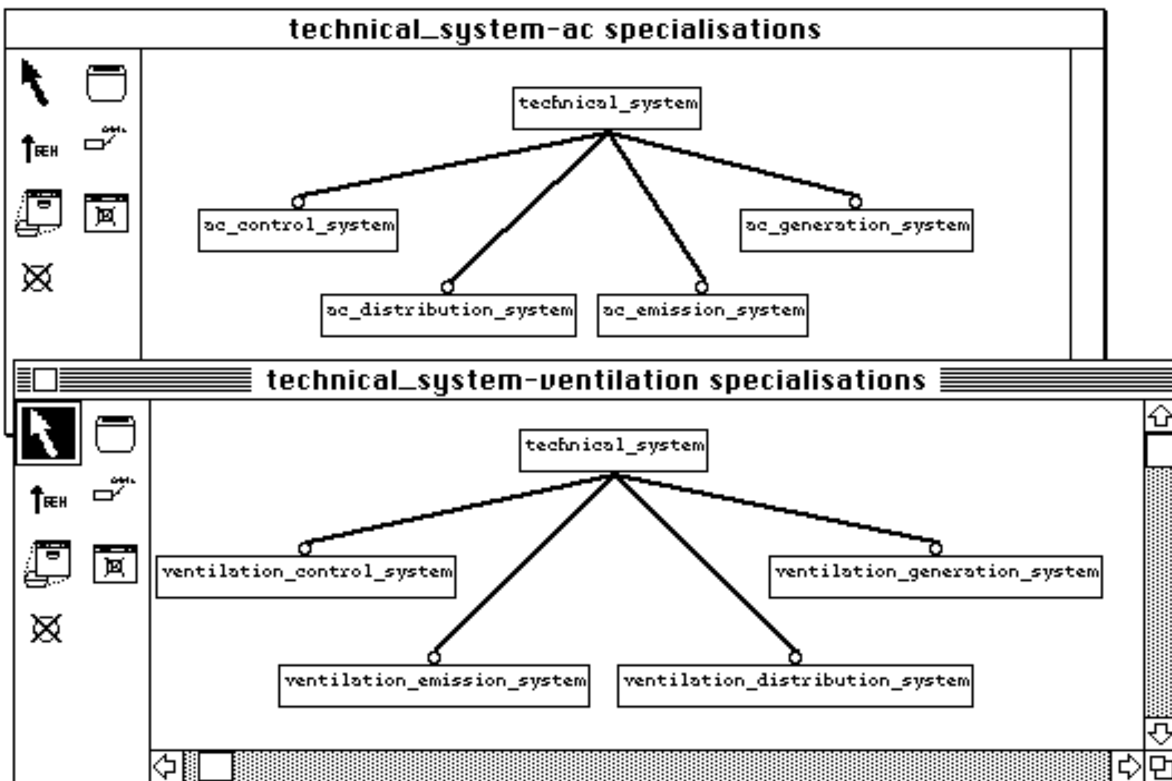


Figure 3.3 Two high-level inheritance specifications for *technical_system*

3.2.1.2 Design

Attributes are often specified at the design stage, as shown for the *technical_system* example in Figure 3.4. This can be done, as shown in this figure, with all entities in one graphical view, or by using two or more views. For example, the attributes of basic types may be presented in a separate view to those which define relationships to other entities, thus adding clarity. Again, there is no limit to the number of design views that can be constructed, and the hypertext navigation facilities are available to navigate both between design views, and between analysis and design views. As in the analysis views, all information in the design view of Figure 3.4 is mapped back to the canonical representation of the schema and all dependent views made consistent with its contents.

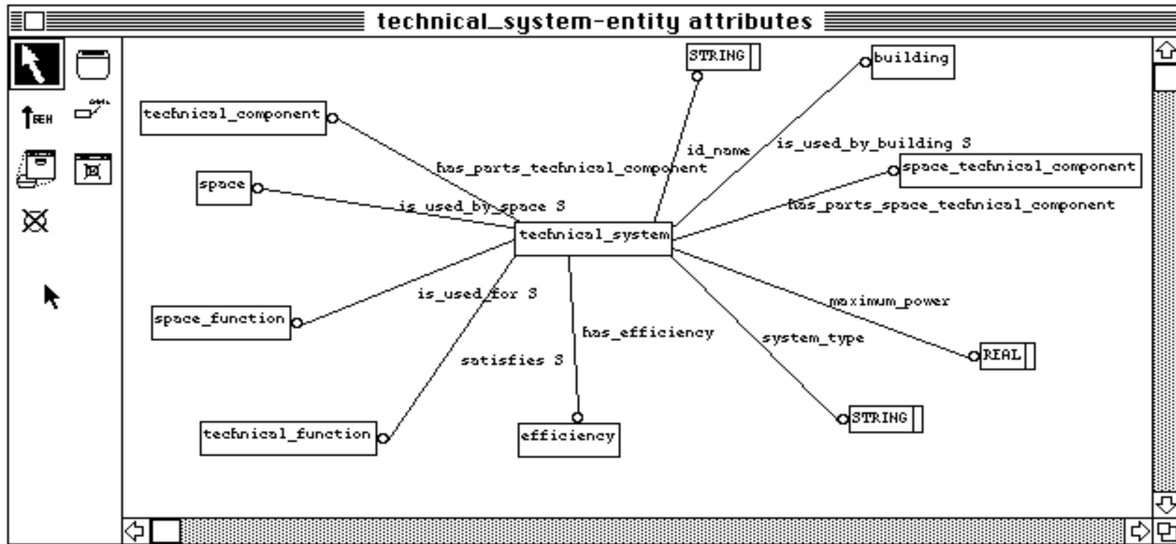


Figure 3.4 Design stage specification of attributes of an entity

3.2.1.3 Late design

At the late design stage more detailed information may need to be entered. This necessitates a textual representation of the entity in the EXPRESS language, as EXPRESS-G represents only a subset of what can be modelled in EXPRESS. For example, UNIQUE clauses, WHERE clauses, rules, and type information have no EXPRESS-G representation. Figure 3.5 shows an EXPRESS textual view which has been generated from the canonical information on the entity. This view encompasses all information found in all the graphical views which define the *technical_system* entity, such as the information in Figures 3.2 and 3.3. This textual view is editable, with the user free to make changes to any parts of the textual description in the view. Modified textual views are parsed and compiled to ensure they represent valid EXPRESS descriptions, and their information passed back to the canonical representation.

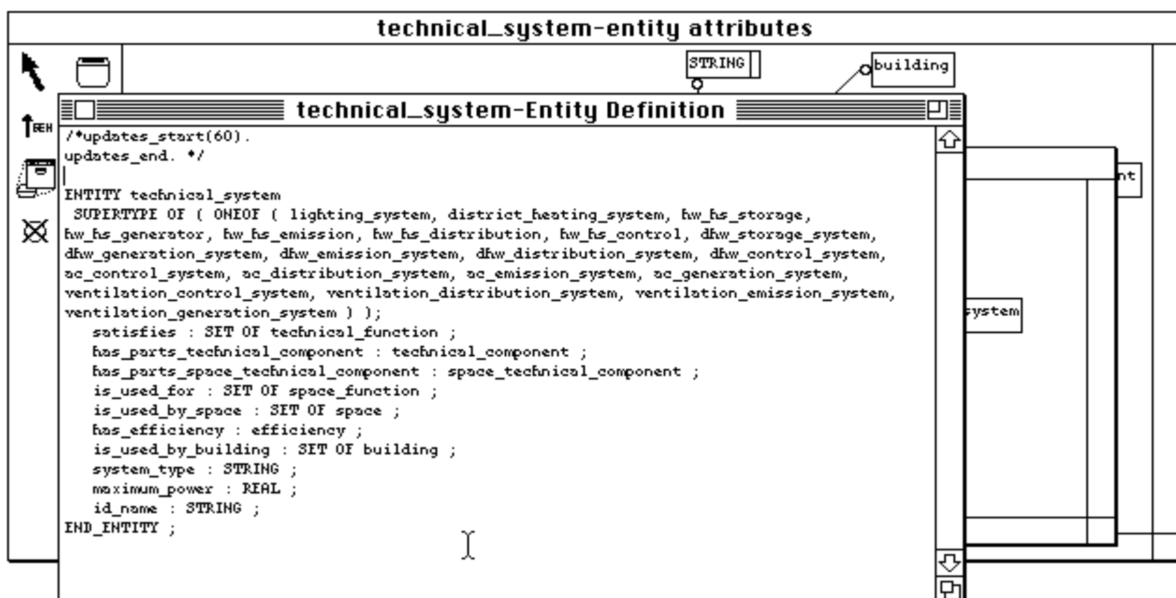


Figure 3.5 Textual view derived from graphical views of an entity

3.2.1.4 Consistency between views

When changes are made to an EPE view, other views that share information with the updated view may become inconsistent and must be updated to keep the schema consistent across all views. All views affected by a change are notified and, in many cases, are automatically modified to reflect the change. This consistency mechanism works both between views of one phase of development and between views of different development phases.

As an example, Figure 3.6 shows a modification being made to a graphical design view. The modification tightens a constraint on an entity's value; the lower bound on the number of values in the SET definition is now known to be 1 and is entered in the definition. The information entered in this manner is checked as to whether it is valid EXPRESS syntax before being accepted and allowed to modify the schema being developed. The change is propagated through to the canonical form of the schema which is updated, then all dependent views are identified and notified of the change which has been made.

EPE propagates the change to the other affected views in the form of an *update_record* (described in Section 3.2.2). This record provides a complete description of any single change. How views react upon receipt of an *update_record* depends on both the view type and the nature of the change. In the design view the modification updates the graphical representation of the design view according to the definition of EXPRESS-G syntax, as can be observed in the graphical design view at the rear of Figure 3.7. The modification is not propagated through to the analysis views, as the modified attribute is not seen in these high level views. However, the attribute does appear in the textual late design view and it must be updated to be kept consistent.

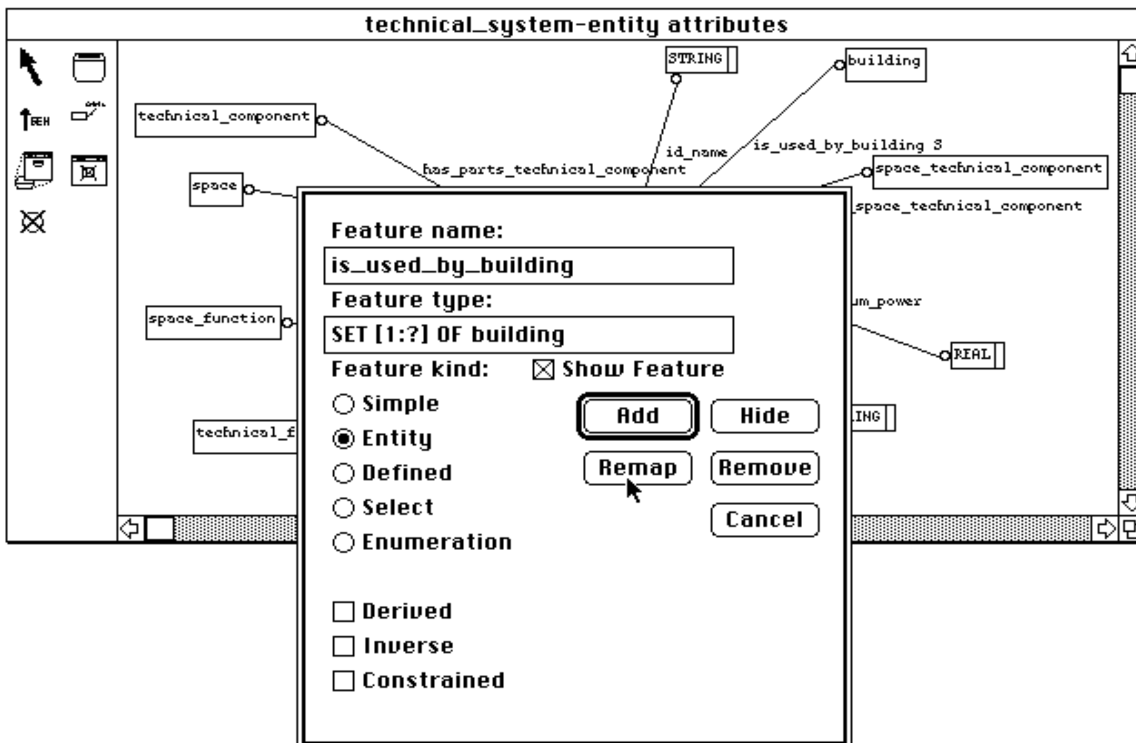


Figure 3.6 Constraining the cardinality of an attribute at late design stage

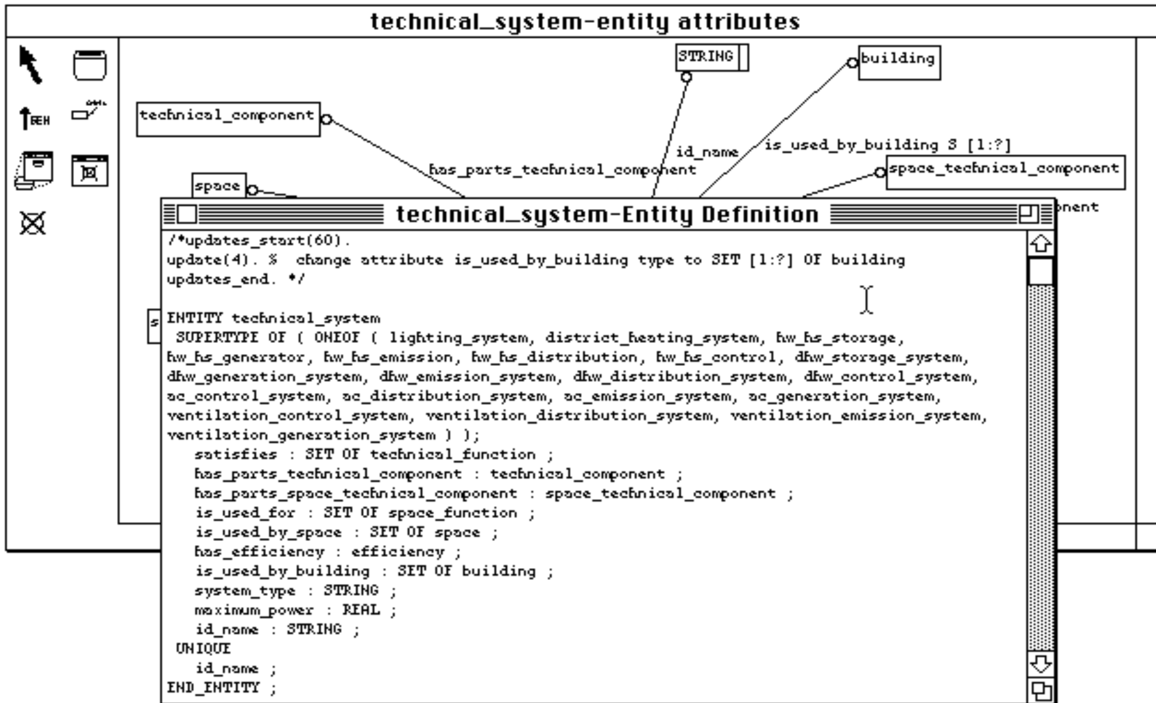


Figure 3.7 The propagation of an *update_record* to a dependant textual view

In the EPE system *update_record*s propagated to textual views are not applied automatically, although many of them could be. Instead the *update_record*s are displayed in the view and the user has control over which updates are applied at which time. As can be seen in Figure 3.7, the graphical update to the *technical_system* entity generates an *update_record* in the entity's textual view. If the user instructs EPE to apply the update, the resulting view of Figure 3.8 is generated, where the notification of outstanding updates on this view has been removed, and the attribute definition has been automatically rewritten.

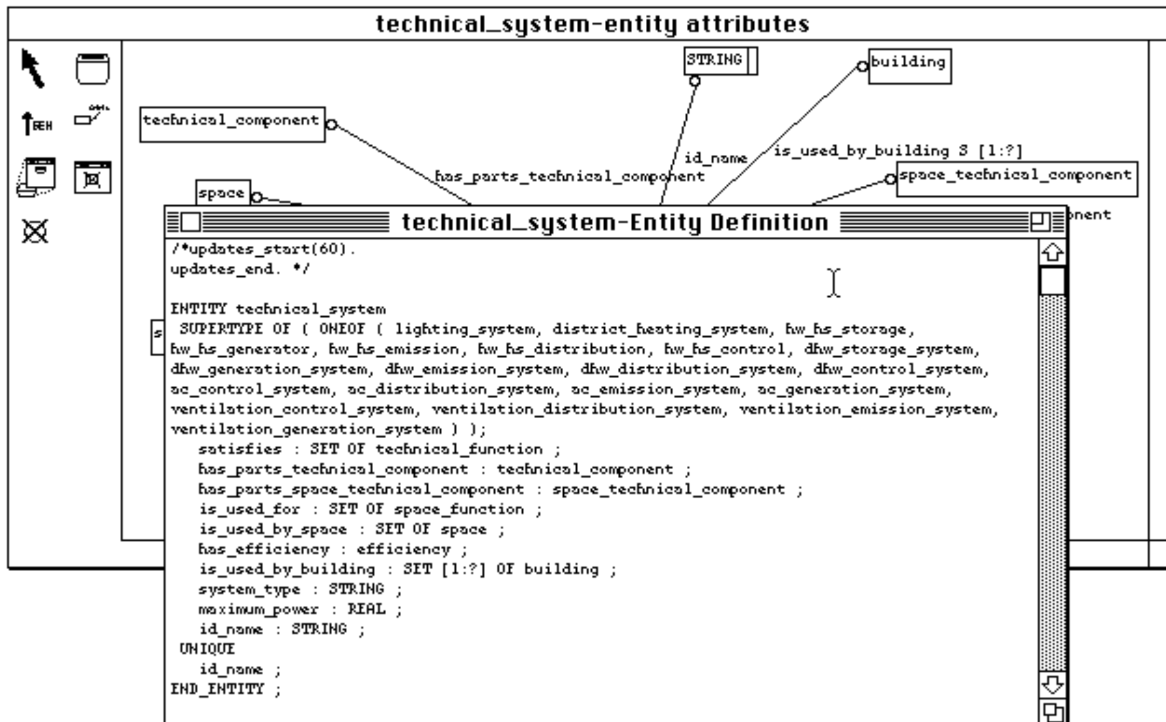


Figure 3.8 The automatic application of an update in a textual view

This strategy of allowing the user to apply updates to textual views exists to handle the problems caused by the specification of a constraint on an attribute or entity in EXPRESS-G. For example, when the user specifies that an attribute is constrained (see the attribute dialogue box in Figure 3.6) the user is denoting that the attribute takes part in either a unique clause or in a where rule. However, there is no way in EXPRESS-G to specify which one the attribute takes part in, or in what form. Therefore, an update of this form can not be automatically applied to a textual view and must be manually implemented by the user. The change description thus serves to specify an inconsistency that requires manual resolution.

3.2.1.5 Documentation

In addition to providing a consistency mechanism between views, *update_record*s are retained in a persistent form in the EPE system. An *update_record* browser and editor gives the user the ability to browse the changes that have occurred to an entity in the evolution of the schema and add further documentation to each *update_record*. In this manner a portion of the documentation of the history of development of the system is automatically built up as work progresses. Having this update history on-line also allows system developers to trace back through previous design decisions while entities are further refined.

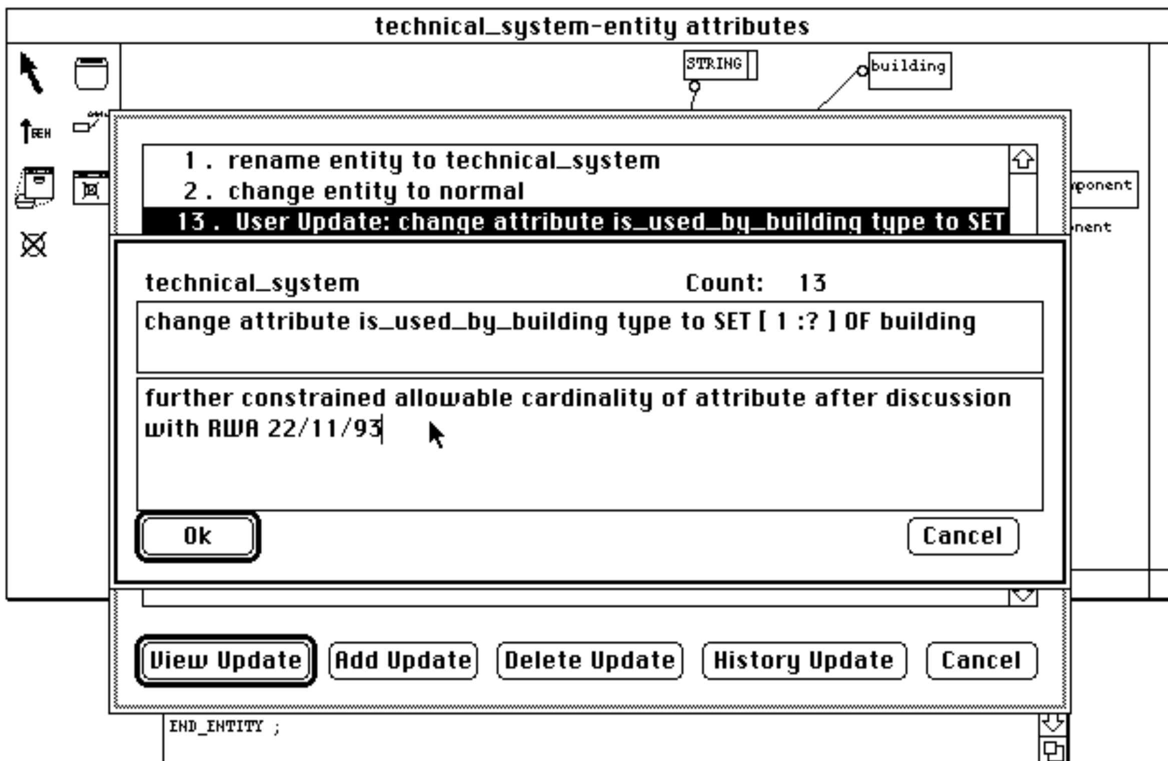


Figure 3.9 The persistent *update_record* viewer with documentation facility

Figure 3.9 shows the *update_record* browser displaying a list of changes that have been made to the *technical_system* entity based on the *update_record*s generated by the changes in various views. They include a renaming of the entity, changing the entity from an abstract supertype to a normal entity and the cardinality constraint imposed on the SET declaration of an attribute. The full details of the modification to the attribute are displayed in the top window, highlighting the

comment field that can be filled in by the system developer.

Other documentation support in EPE includes the ability to create textual documentation views (accessible via the hypertext navigation facilities) for entities. In such views, the various experts working on a schema can document the reasoning behind decisions made and other information relevant to a particular entity and its attributes in a central and managed fashion. A useful feature of documentation views is that *update_record*s relating to the entity are automatically added as textual comments to the view as the entity changes (see Figure 3.10 for a documentation view taken at an early stage of the schema specification).

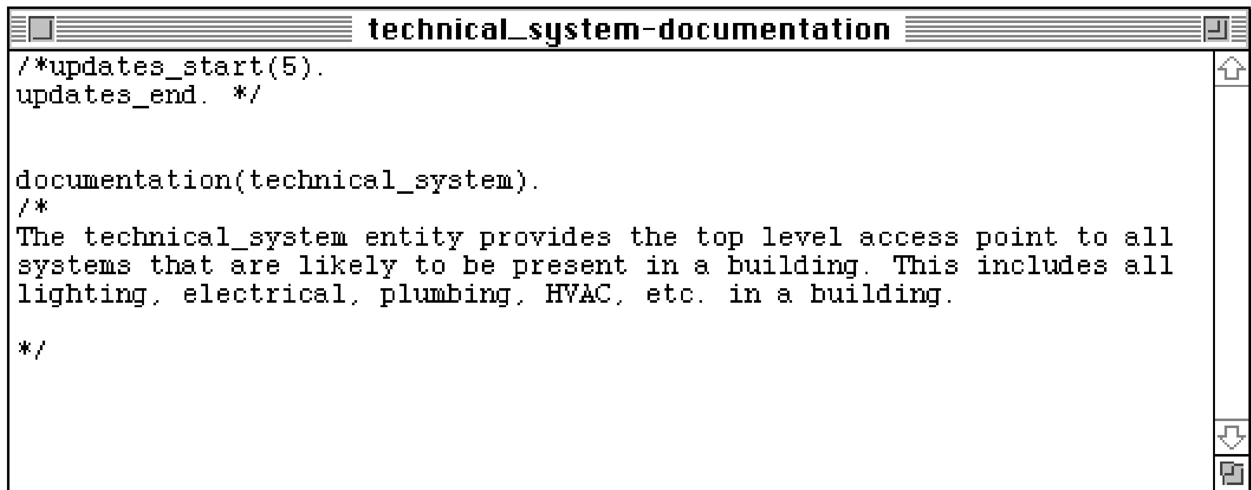


Figure 3.10 A textual documentation view

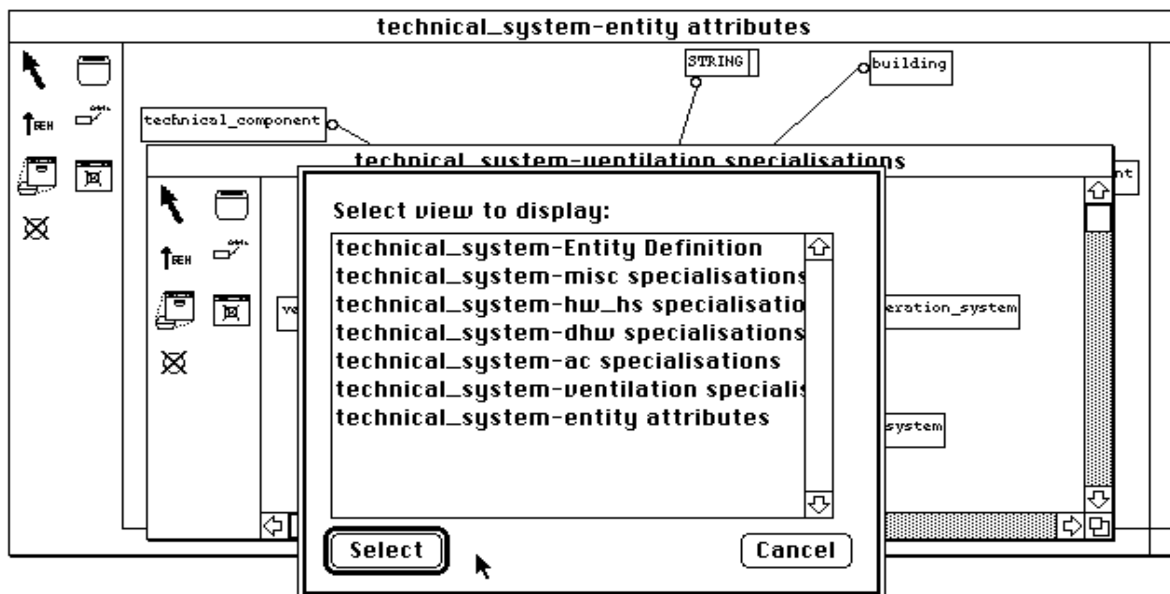


Figure 3.11 The view navigator invoked for the *technical_system* entity

3.2.1.6 View navigation

As can be seen from the preceding example, the EPE modelling environment captures a large amount of information about a schema. This includes many graphical and textual views specifying various properties of the schema, entities, and documentation of changes made to the schema. In large systems this can lead to problems in finding a particular view or detailed information that has

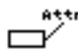
been entered. In the EPE environment the views and entities themselves act as a navigation and search facility for the project. Mouse clicks on graphical view components allow rapid access to other views containing that component. Figure 3.11 shows an example of this process for the *technical_system* entity. After clicking on the *technical_system* icon in the graphical view a list of all the graphical and textual views that index the *technical_system* is displayed and the user can navigate to these views in a hypertext-like fashion.

3.2.2 Using the EPE environment

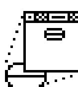
The EPE system offers two types of display views to its users. The initial type of display view is a graphic view which can contain any EXPRESS-G specification of a schema (see Figure 3.2 for an example of a graphical view). In the graphical view the user has a palette of tools available, as seen on the left hand side of the view. These tools offer the following functionality.

This tool is used to specify an entity in a graphical view. Clicking in an empty portion of the drawing window after selecting this tool will bring up an entity dialogue requesting the name of the entity and its type (abstract or normal). If a new entity name is specified it will be added to the canonical representation of the schema. If the name of an existing entity is specified then the icon will be connected through to the canonical form of that entity. If an existing entity icon is selected while this tool is current then the name dialogue is retrieved, allowing the entity information to be modified (any modification will be seen in all views which reference the entity).

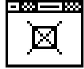
This tool is used to specify inheritance between entities in a graphical view. To specify an inheritance link the user clicks on an entity icon which is the super-class and drags to the entity icon of the entity which inherits from it.



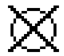
This tool allows the specification of attributes for an entity. To describe an attribute the user clicks in the entity to which the attribute should be attached, and drags out to the position the attribute icon should occupy. This invokes an attribute dialogue to specify the name and type information for the attribute (an example of this dialogue is shown in Figure 3.6). If a new attribute name is specified for the entity then it is added to the canonical representation of the entity. If the name of an existing attribute is specified then the icon is connected through to the canonical form of the attribute. If an existing attribute is selected while this tool is current then the attribute dialogue is retrieved, allowing the attribute information to be modified (any modification is seen in all views which reference the attribute).



This tool is used to create a new graphical view of the specified entity. When an entity is selected the user is asked for the name of the view to create and a new graphical window with the entity will be created. If several entities and attributes are selected when this tool is used then the user has the option of copying all selected icons to the new window.



This tool hides an attribute icon, or an inheritance link, or an entity and all its attachments (i.e., attributes and inheritance connections). The hidden item is not removed from the canonical representation of the schema, merely hidden in the current graphical view.



This tool deletes an attribute, or an inheritance link, or an entity and all its attachments (i.e., attributes and inheritance connections). The items are removed from the canonical representation of the schema as well as from all views which contain the item.

This tool allows icons in the graphical view to be selected and repositioned in the current view. Double clicking on an attribute icon brings up the dialogue for that item. Double clicking on an entity icon has two functions depending upon where in the icon the double click occurs. Double clicking on the left hand side of the icon brings up a dialogue listing all views that this entity is specified in to allow navigation to other views (see Figure 3.11). Double clicking on the right hand side of the icon makes the textual view of the entity visible (see Figure 3.8).

The second type of view offered is a textual view. Textual views allow free-form textual editing and manipulation of the canonical definition of entities using EXPRESS notation. Textual views are re-parsed at the termination of a textual editing stage and all modifications propagated through to the canonical form of the edited entity.

3.2.3 Implementation of EPE in the MViews framework

EPE is constructed by further specialising SPE, a specialisation of the MViews object-oriented framework (Grundy 1993; Grundy and Hosking 1993a). This framework provides a set of abstractions for constructing software development environments that support multiple graphical and textual views with in-built consistency between views (Grundy and Hosking 1993b). SPE integrates, in a single environment, tools to assist in systems analysis, design, implementation and maintenance of programs in Snart, an object-oriented logic language (Grundy et al. 1994). Other environments developed using MViews include an ER (Entity-Relationship) modeller for the database domain, and a graphical forms builder for specifying form layout and semantics for GUI applications (Grundy and Hosking 1994).

MViews utilises a three-layered architecture to present and maintain multiple views of an underlying entity (see Figure 3.12). The base layer contains the canonical representation of the schema being developed in the MViews environment. This canonical representation takes the form of a directed graph of components, representing entities and attributes, connected by some set of relationships, representing generalisation and containment (in an entity). Views in the view layer provide a subset of the canonical representation in the base layer. A view represents the information required for a displayed view. Elements in a view need not have a one-to-one correspondence to elements in the base layer, rather a view relationship provides the connection

between view elements and base layer elements. These view relationships handle the mapping of data from the base layer through to the view layers and vice-versa. View relationships may aggregate base layer elements to form new elements in a view providing mappings between elements of any combination of arities. The display layer contains tools which visualise the elements provided by a single view. The types of tools that can be used in the display layer are: graphical and textual editing tools which display and allow direct manipulation of elements in a view; or external tools which utilise a batch mode access to elements provided through a view.

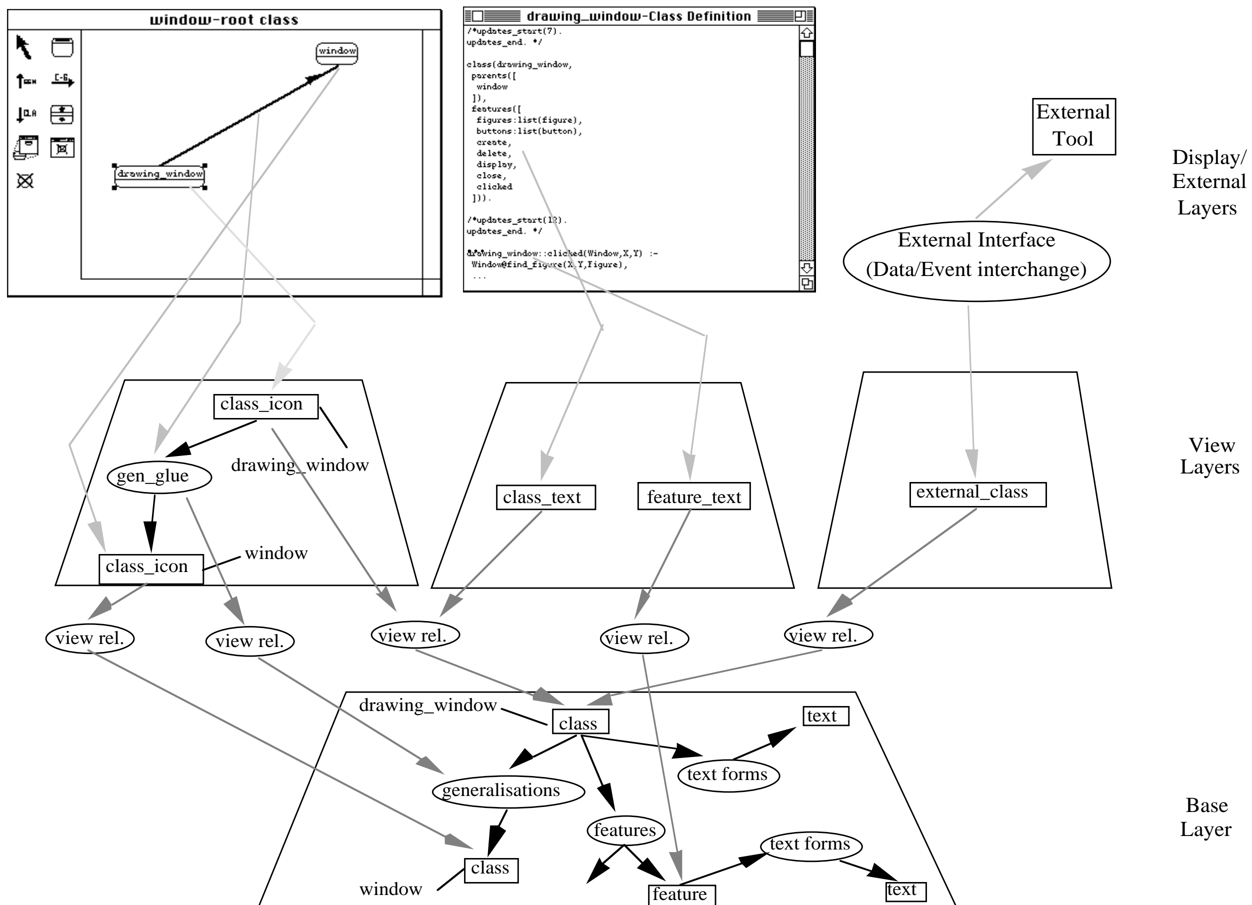


Figure 3.12 MViews three-layer multiple view architecture as used in SPE (from Grundy 1994)

In an MViews based system additions, modifications and deletions are initiated from the tools in the display layer. The actions initiated by these tools result in the creation of an *update_record* describing the particular action that was performed. The *update_record* created for an action is propagated from the display layer to its view in the view layer and down to the base layer following the links between the layers. From the base layer the *update_record* is propagated to all connected views to handle, as well as to all connected components in the base view and so onto their connected views, etc. At each stage in the propagation elements in the system can react to the *update_record* to modify their status or ignore the *update_record*. Individual elements can also propagate the *update_record* to all connected elements, or stop the propagation.

Figure 3.13 shows an example of the effect of an update in an MViews based application like EPE. In this example an action is performed by a tool in the display layer (1), an *update_record* is

propagated to all dependents of the *class_icon* (2), the view relationship translates the *update_record* into operations on the class component (3), which in turn generate *update_record*s (4). These *update_record*s are propagated through to all of the dependents of the *class* (5), the view relationships translate the *update_record*s into operations on components in the view layers (6), and modified view components re-render themselves in the tools in the display layer (7).

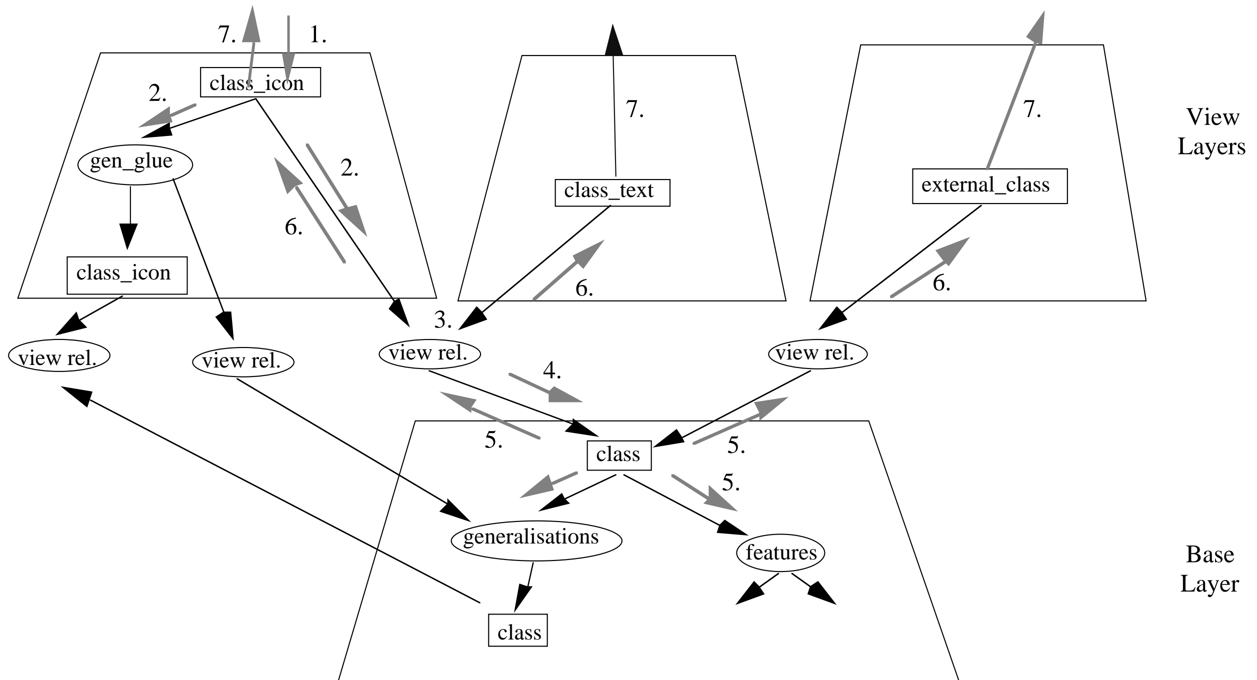


Figure 3.13 Change propagation in an MViews environment (from Grundy 1994)

As EXPRESS is a partially object-oriented language it requires almost the same graphical modelling capabilities as the object-oriented language Snart requires in its integrated software development environment SPE. It was therefore a natural step to specialise the SPE environment (Grundy 1993, Grundy and Hosking 1993a) for use with EXPRESS. This specialisation was able to use most of the same class and feature representations, along with relationships between modelled objects. Figure 3.14 shows the inheritance hierarchy for the EPE environment, with all EPE classes prefixed with an 'exp_'. This structure is very similar to that of SPE, and in fact the majority of the SPE code was able to be utilised for the EPE environment. The modifications to the SPE environment fell into the following categories.

- Rendering modifications: the visual appearance of EXPRESS-G information is quite different from that of Snart, even though their concepts are closely related. Therefore, all class and attribute representation code needed to be modified to allow the correct EXPRESS-G representation. A major part of this change was a set of additions to the SPE system to allow the EXPRESS-G attributes to be correctly handled. In Snart all attributes, except relationships, are shown inside the class icon. In EXPRESS-G all attributes are independent icons and therefore must be modelled through a relationship with the class representation.

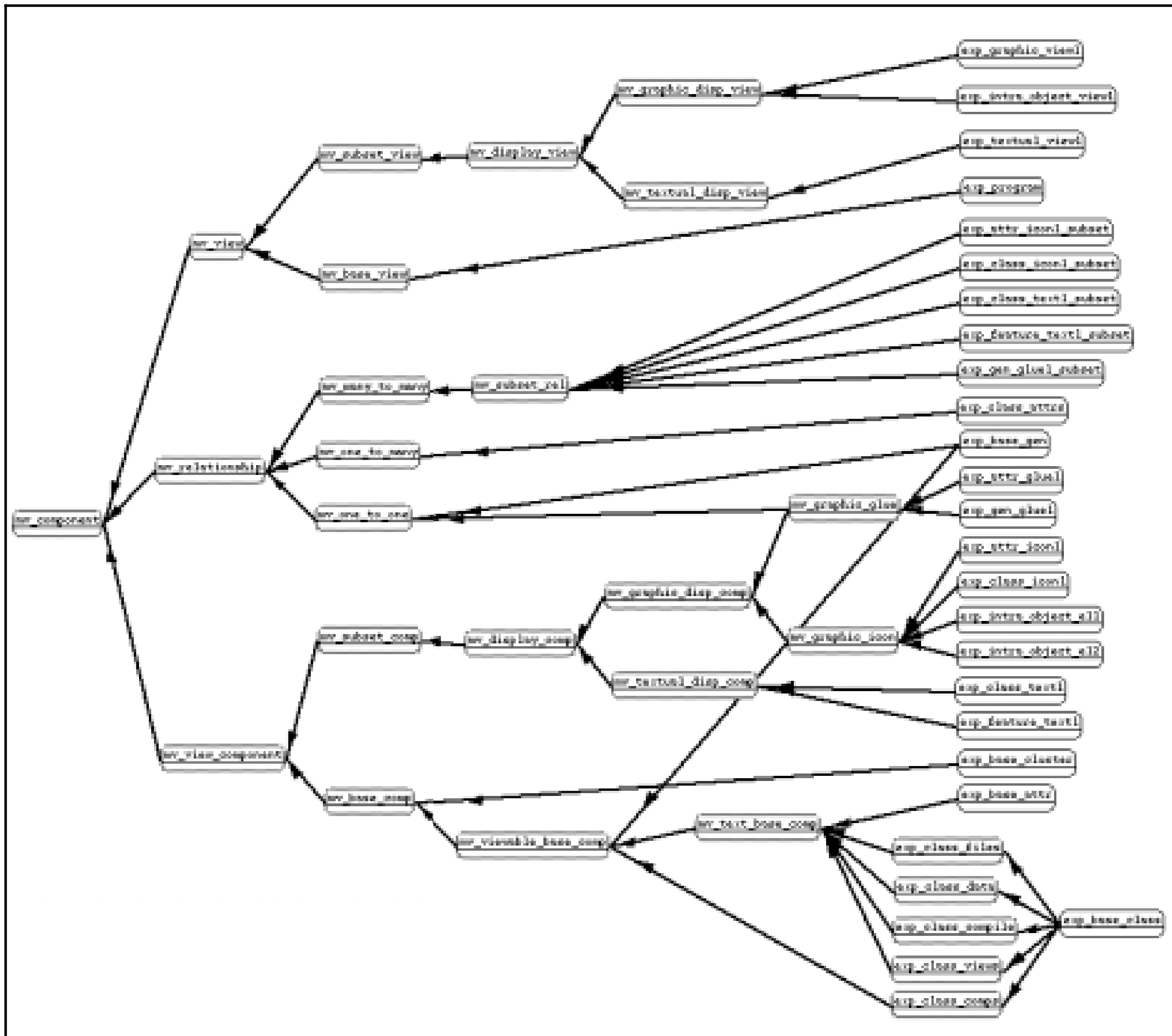


Figure 3.14 Class inheritance for EPE from MViews

- Environment modifications: though the set of operations which can be performed on an EXPRESS-G diagram are similar to those in Snart, the language used to describe them, and the information requested from the user, had to be tailored to EXPRESS terminology. This required changes to toolbox icons in the graphical views, and rewriting of all class and attribute dialogues.
- EXPRESS language parsing and generation: the underlying representation of information in the EPE system was in Snart form, as this allowed for compiling and testing of the defined schemas. This required all graphical and textual views to be translated into Snart, and the underlying canonical form to be translated from Snart to EXPRESS for textual and graphical view updates.
- *update_record* labelling: the range of *update_record*s used in SPE was examined and modified to suit the structure and terminology of EXPRESS.

- Class hierarchy management: since EXPRESS maintains both subclass and superclass links, as distinct from Snart which only specifies subtype relationships, class hierarchy management was extended to enable bidirectional tracking of updates to class hierarchies
- Removal of method handling: EXPRESS has no notion of methods associated with classes so all code for handling methods was stripped from the translated SPE system. Though EXPRESS allows function and procedure specification these are not handled in EPE.

3.2.4 Internal schema representation in EPE

In the EPE environment, an EXPRESS schema has an internal representation of its canonical form. EPE is based upon SPE, which utilises Snart as its base representation, so all EXPRESS constructs are translated into Snart definitions in the canonical form. As Snart is an implemented language this provides the ability to compile EXPRESS schemas and populate the resultant compiled schema with data to create models of the domain of the schema.

The amount of the EXPRESS language which is supported in this translation process is limited. Snart is a full object-oriented language based around Prolog, while EXPRESS is a more imperative language with a style similar to C. All entities, inheritance structures, attributes, unique attribute specifications and some forms of WHERE clauses can be translated to Snart. All procedures and functions, including WHERE clauses which use procedures and functions, are not translated to Snart. In effect, this restriction means that a translation to Snart reduces the EXPRESS definitions in EPE to basic class interfaces with uniqueness and range checking on attribute values. While this is a major reduction in the expressiveness of the original language, this type of definition forms the major part of the principal IDMs that have been developed to date, and even covers the majority of schemas that have evolved from the STEP standard.

As EXPRESS is translated into Snart syntax in the canonical form, so must it be translated back into EXPRESS for use in textual views (see Figure 3.5) or for type definition in the attribute dialogues (see Figure 3.6). To achieve this the EPE environment contains two translators, one to translate from Snart to EXPRESS, and the other from EXPRESS to Snart. The EXPRESS to Snart translator is implemented through the use of the DCG system available in LPA Prolog (LPA 1995) and further described in Appendix F.1. The Snart to EXPRESS translator is totally hand-coded.

The development of the EXPRESS parser was based on the EXPRESS grammar in the ISO standard. However, when the completed parser was being tested, many of the existing ISO and EU project schemas failed to parse correctly. This is due to the many changes made to the EXPRESS language before it became a standard. As there were many changes to the syntax, the parsers developed alongside the EXPRESS language development have allowed for all previous forms of syntax to be treated as correct. This leads to a situation where, though there is now an ISO standard for EXPRESS, many modellers write incorrect models due to their recollection of

past specifications, and these syntax errors are not picked up by current parsers. Though the EPE parser may fail to read existing schemas, it does guarantee that all schemas developed in its environment are correct to the ISO standard.

As Snart is the internal language representing EXPRESS schemas, it is clear that any specification environment which represents its model in Snart form will be able to be incorporated fully into this framework. To this extent the SPE environment which allows the specification of Snart schemas can be used to model schemas which can be used interchangeably with the EXPRESS schemas from EPE described in this section. Modelling environments which do not produce EXPRESS or Snart schemas are not precluded from use in this framework, but they will not be able to be tested interactively during development with the schema development and browsing tools highlighted in Chapter 9.

3.2.5 Summary of EPE functionality

EPE offers a wide range of functionality for a schema designer. In a single environment the schema designer has the ability to model at many levels of detail, in both graphical and textual notations and with automated consistency management between all the views. The main functionality offered by EPE is:

- provision of multiple overlapping views of portions of a schema
- views ranging from high-level design views through to low level implementation views
- support for both graphical and textual notations
- changes to a schema in one view are seen in all other views containing the structures that were modified
- automatic maintenance of change documentation which can be augmented by the users
- support for schema navigation with hypertext-like links

3.3 Generic Schema Database Definition

Although a modelling environment must produce EXPRESS or Snart schemas to work fully inside this development framework, the mapping system (described in Chapters 5 and 10) makes no assumptions about the nature of the modelling environment or language that a schema was developed in. The only assumption made by the mapping system is that a schema type is of either relational or object-oriented form. To support schemas of either of these forms a generalised schema specification language (defined in Appendix G) is used to describe schemas developed at the modelling stage to the mapping system.

The schema specification language works at the atomic level of a schema description. Single modifications to a schema are recorded in the order they occur, to be able to reconstruct a schema

in the same progression as it was developed. The atomic definitions record the previous state of the schema as well as the change so that individual modifications may be undone to reverse the effect of a modification. Atomic changes are keyed to a version number for the schema and schema version information is defined as part of the database. Versions are represented as a directed acyclic graph with a single root node. Through the in-order application of all modifications that represent the chain of versions leading to the required version number, the corresponding schema can be created.

A schema defined in any notation which can be translated into the format used in the schema database definition can be used with a View Mapping Language (VML) mapping specification. Schema databases are used during the VML parsing stage to create a checked schema mapping with all references validated and ready to be used by a mapping system. Schema databases can also be modified at the VML parsing stage to incorporate modifications (new entities and attributes) required by the mapping specification (see Chapter 5).

3.4 Appraisal of Schema Modelling

The EPE system and schema database format create an environment which supports the development, communication and integration of DT and IDM schemas as well as the documentation requirements of an integrated modelling environment, as detailed in Section 3.1.1 for COMBINE. EPE supports these requirements in the following manner:

Communication and integration of DT schemas: EPE allows the development of a schema of a particular domain with total consistency from the earliest stages of the project right through to the implementation. This consistency is maintained by communicating all changes through to all views which can possibly see the affected entities. Multiple DT schemas are supported in this environment, allowing the modeller to develop multiple graphical and textual representations of a domain during analysis and allowing these to be expanded at the detailed design stage with new or updated views. These developing schemas, and their views, are available to all who use the EPE system, allowing DT schemas to be easily shared between the integration team. Many of these developing DT schemas have overlapping information and need to be merged with the IDM schema. By developing multiple associated DT views and one canonical, consistent, representation of the IDM schema the consistency between the DT schemas and IDM can be maintained (by hand). The schemas are also utilised to provide hypertext-like navigation facilities around the various views of related information.

Documentation: EPE meets many documentation needs by tracking all updates made to the schema and recording them against the entities that were modified. These update records can be annotated by the user to record justifications and decisions, and provide on-line documentation for the developing schema. One documentation feature which would be

useful, but is not provided, is the ability to group multiple disperse update records to represent a single update or documentation record. For example, this would record a session of changes as a single documented change to the schema.

Mapping definitions: these definitions are not considered in the EPE environment, but an environment to support mapping definitions has been developed and is described in Chapter 6.

Support for multiple modelling paradigms: EPE provides multiple paradigms as it supports both the EXPRESS and EXPRESS-G paradigms. Although one is the subset of the other, they render their information in different styles. The modelling environments developed and used in this project also show that through the use of a generic model of a schema (in this case in Snart format) multiple modelling tools, in this case EPE and SPE can be used to work on a single schema. Although Snart is not particularly generic as an underlying model for schemas, a schema representation language such as that used for the database representation of schemas would provide such a model on which most schema notations could be based. Other environments developed from the MViews platform have shown support for more disparate modelling paradigms through the use of an underlying representation which subsumes the paradigms used in the different display views (Grundy and Venable 1995 for OOA/D and EER; Venable and Grundy 1995 for ER and NIAM).

Support for multiple views: although the EPE environment can manipulate multiple independent schemas, there is no direct support for the specification of correspondences between schemas as an aid to the development of an integrated schema (i.e., schema integration for IDM development). Schema integration has previously been performed to a small extent (see Mugridge and Hosking 1995 for an example of DT schema model integration using SPE), but this does not illustrate the support that could potentially be offered in such an environment to perform integration. The difficult problem in supporting schema integration is the ability to define, track and support the specification of mapping definitions between various DT schemas in a tool like EPE. Chapter 5 details a first step towards an environment for defining mappings that relate two schemas together. However, this latter environment is not yet integrated with the environment that manages the schema, leaving a semantic gap between the maintenance of information in the various environments. This problem and the ability to manage concurrent schema development in the MViews, SPE and EPE environment will be an area of continued research (Amor and Hosking 1993; Amor and Hosking 1995; Grundy and Venable 1995). A similar, though simpler, problem to that of supporting multiple views is supporting multiple versions of a schema. The schema database representation allows for the definition of versions of modelled schemas. Any version defined in the schema can be reconstituted for use by any application utilising the schema database, which is of particular use when defining a mapping between versions of a schema. However, the versioning ability of the schema representation is not supported in the EPE environment which is the point at which this would be best specified and managed. Extensions to MViews to support versions would be relatively simple, especially

if the underlying representation was modified to utilise a schema database representation as described above.

Though EPE meets the majority of the requirements for a schema modelling and development environment, there are some areas which would benefit from further work. These include:

Collaborative design support: although large modelling projects may have a single coordinator in charge of schema development, there are likely to be several modellers working on different aspects of the schema. Coordinating the work of several modellers, and maintaining the consistency of the underlying schema under change from several sources concurrently, has to be a goal of any real schema development environment. This is currently an area of intense research work in the computer-supported collaborative working (CSCW) area. Recent research results in this area indicates that when these CSCW environments mature they will be easy to incorporate into existing modelling tools. The Serendipity system (Grundy et al. 1996) has already been shown with links to MViews-based environments and commercial Microsoft products, providing collaborative design between several modellers working through defined process models. Though the mapping implementation of Chapter 10 provides a transaction-based approach to coordinating between multiple modellers, this is not supported by version merging and conflict resolution systems, as can now be found in systems such as Serendipity.

Comprehensive multi-paradigm modelling support: in exactly the same manner that an integrated building design system requires an IDM to integrate design tools, a multi-paradigm modelling environment requires an IDM covering schema, mapping and project definitions in order to manage multiple paradigms in one environment. Though some work has been done in this area (Venable 1993) there are no IDMs which cover the wide range of schema modelling paradigms currently in use (e.g., ER, DFD, EER, NIAM, EXPRESS, EXPRESS-G, OOA/D, IDEF0, IDEF1X), let alone the underlying requirements of mapping and project definition languages.

Expanded modelling support concepts: the range of support concepts required by modellers when developing large schemas is not clearly understood, and hence, not well supported. Currently documentation of schema modification and construction is made at the atomic level. Methods allowing for grouping of higher level concepts are imperative (including multiple viewpoints of change sets). The manner of notification of modifications in a collaborative environment needs attention to provide efficient methods of expressing these changes to other designers (rather than at the atomic change level), an example of developments in this area can be seen in Grundy et al. 1995. Navigation and summary features tend to be primitive. In large schemas with hundreds of entities and thousands of views, the set of views which reference a particular entity could be very large. Classification of view types, or the relationship particular entities play in a view, could well help navigation around the schema and guide novice users through the various conceptual levels of schema specification. Notions of schema versions and private

workspaces need to be considered in a collaborative environment. This allows the management of multiple design paths, and for incomplete work to be hidden until fit to be used by other participants.

In summary, this chapter introduces the requirements for a schema modelling and development environment, and through the development of EPE and a schema database format, demonstrates that an environment meeting these requirements is possible. The development of individual schemas is, however, a small part in the development of an integrated design system. Schemas for design tools need to be integrated to form an IDM, or checked that they map into an existing IDM. This process of schema integration, or checking, through the definition of mappings between schemas is described in Chapter 5. Schemas which represent design tools and actors are also required to define flows of control in a real project. The use of these schemas in project definition is described in Chapter 7.

Chapter 4

Inter-Schema Relationship Modelling

The mapping of data between models is a vital process in an integrated design system. To enable the correct specification of a mapping between schemas, a mapping language and specification support environment is required. In a similar manner to the schema specification languages, a mapping language provides an abstract specification of a portion of the problem domain. The definition of a mapping between two schemas is likely to be a very large piece of work, often involving several domain experts for both models. These experts must maintain the consistency of the mapping specification. Because of this necessary involvement of domain experts, it is clear that a high-level specification language is needed to provide a specification close in semantics to the problem domain and well removed from the implementation. The mapping language should also support multiple levels of specification in a similar manner to schema modelling languages.

As well as providing the basis for mapping of data between them, a specification of the mapping between two schemas provides modelling benefits over those that are usually obtained just from the schema definition. For example, the specification of an inter-schema mapping makes explicit those constraints that are implicit in a design tool (e.g., a design tool which assumes vertical walls needs to explicitly model this requirement when specifying a mapping with a schema that does not have this assumption).

If a formal mapping language is used to specify the mappings, mapping specification environments offering consistency management between various modellers are possible. Tools can then be created to: check statically whether the mapping references valid schema properties; check types in the conversion; check units in the conversion; identify affected mappings when components in the schema are modified; and ensure that all attributes of entities are mapped between the two schemas.

However, until recently, the mapping between the IDM and a design tool schema has not been modelled in any way in integration projects. All mappings for design tool schemas have typically been hand-coded in the programming language of the project. This approach is now changing as several projects have reached the stage of development where they recognise the utility of having mapping languages to formally model mappings between schemas (ATLAS 1993; Staub et al. 1994). To satisfy the demand for mapping languages a number of new languages have been developed.

In this chapter the types of mappings which are required to describe correspondences between schemas are examined. Languages for specifying mappings are surveyed, including formalisms recently developed for mapping in the ISO-STEP standard. An informal set of requirements for a mapping language is collated from previous work and from an analysis of mapping requirements for a range of actual schema. These requirements are then used to measure the power of the surveyed languages, and enable comparisons between the languages to be performed.

4.1 Mapping Types

Analyses of the problems posed by mappings and integration have previously been conducted in the database area. To gain an understanding of what is required to map between two schemas, the analyses performed in the field of database integration are examined and presented here in two categories. One category investigates the types of structural mapping that can be expected between two schemas, the other presents a more semantic description of the types of conflict found when integrating schemas.

4.1.1 Structural mapping types

Structural mapping types provide a key to the complexity of mappings that can be expected in a particular domain. Two evaluations of these type of mapping are those of van Horssen et al. (1994), which details a table of mapping types (see Table 4.1), and Bijnen (1994), which provides a more general specification of mapping types. Both of these evaluations are drawn from considerations of requirements of a mapping language: in van Horssen's case as a precursor to evaluating several mapping languages; and in Bijnen's case to show what his mapping language should provide. In Table 4.1 the asterisks denote the requirement for mappings of the specified type, blank cells indicate no mappings of the specified type are required. The labels on the table below are slightly misleading, as where they specify *entity* they actually mean *object*. For example an N:1 mapping for Entity->Entity denotes the mapping of N objects of one entity type to a single object of another entity type.

	Entity->Entity	Attr->Attr	Entity->Attr	Attr->Entity
1:0	*	*	*	*
0:1	*	*	*	*
1:1	*	*	*	*
1:C	*	*	*	*
C:1	*	*	*	*
1:N	*			*
N:1	*		*	
N:M				

Table 4.1 Mapping types from van Horssen et al. 1994, the left axis specifies the cardinality of mappings which may be required

In Table 4.1 the cardinalities of the various types of mapping are considered with a distinction being made between zero (0), singular (1), constant (C) and variable numbers (N, M) in the mapping. The lack of an asterisk for the many to many mapping for entities in this table is surprising as some are known to exist (for example some design tools use a set of objects to represent a schedule, with each object specifying a fixed number of hour-value pairs. Where a mapping is required between two such tools which have different numbers of hour-value pairs in each object there can be different numbers of objects in each model, hence an N:M mapping). It is also surprising that all combinations of constant values (C in Table 4.1) are not included. If the breakdown includes 1:C, etc. then a rigorous evaluation should also evaluate N:C, etc., though 0:C is not required as it can be constructed through multiple 0:1 mappings.

	Object->Object	Attr->Attr	Object->Attr	Attr->Object
1:0	*	*	*	*
0:1	*	*	*	*
1:1	*	*	*	*
1:C	*	*	*	*
C:1	*	*	*	*
C:B	*	*	*	*
1:N	*			*
N:1	*		*	
C:N	*			*
N:C	*		*	
N:M	*			

Table 4.2 Full set of structural mapping types

Following the examination of structural mapping types above, the table in Table 4.1 has been extended to provide a more rigorous definition of the possible combinations of mapping cardinalities that can occur. This is shown in Table 4.2, where all combinations of constant to multiple, and constant to constant (C:B) are provided.

4.1.2 Semantic mapping types

The semantics of individual mappings have also been examined. Tables 4.3 (Batini et al. 1986) and 4.4 (Kim and Seo 1991) are examples from their analysis of the schema integration process which examines various categorisations for integration. While these tables display conflict types found when integrating schemas, most of them hold equally well for mappings. This is because the definition of a mapping between two schemas can be seen as almost the same task as specifying the integration of one schema into another. The main difference is that in a mapping the schemas being mapped between do not have to be completely unified. The mapping need only be sufficient to create valid instances of a building in either schema.

Naming conflicts		
homonyms	same name for different concepts	
synonyms	same concept described by different names	
Structural conflicts		
type conflicts	same concept represented by different modelling constructs	
dependency conflicts	group of concepts are related with different dependencies in different schemas, e.g., 1-1 versus n-m	
key conflicts	different keys assigned to the same concept in different schemas	
behavioral conflicts	different insertion or deletion policies associated with the same class of object in different schemas	
Conflict categories		
identical	everything is the same.	
equivalent	where different but equivalent modelling constructs have been applied but the perceptions are still the same and are coherent. These are further subdivided into:	
	behavioral	if the same set of answers to any given query can be obtained from all representations.
	mapping	instances can be put on a one-to-one correspondence.
	transformational	if a representation can be obtained by applying a set of atomic transformations that by definition preserve equivalence.
compatible	not identical or equivalent, but modelling constructs, designer perception and integrity constraints are not contradictory.	
incompatible	contradictory because of the incoherence of the specification.	

Table 4.3 Schema integration conflict types from Batini et al. 1986

Tables 4.3 and 4.4 provide a checklist of problems that could be encountered and a means of categorising the difficulty of a particular problem. They do not, however, provide many clues as to what would be required in a mapping language to handle these types of problems. Some of the interesting points which come out of these two categorisations are: the need to update schemas for missing attributes and entities; the handling of unit conversion; and the handling of type conversion.

Schema conflicts
Table versus table conflicts
- one-to-one table conflicts
  - table name conflicts
    - different names for equivalent tables
    - same name for different tables
  - table structure conflicts
    - missing attributes
    - missing but implicit attributes
  - table constraint conflicts (keys and check conditions)
- many-to-many table conflicts (as in one-to-one)

Attribute versus attribute conflicts
- one-to-one attribute conflicts
  - attribute name conflicts
    - different names for equivalent attributes
    - same name for different attributes
  - default value conflicts
  - attribute constraint conflicts
    - data type conflicts
    - attribute integrity-constraint conflicts
- many-to-many attribute conflicts (as in one-to-one)

Table versus attribute conflicts

Data conflicts
Wrong data
- incorrect-entry data
- obsolete data

Different representations for the same data or same representation for different data
- different expressions
- different units
- different precisions

Table 4.4 Schema integration conflict types from Kim and Seo 1991

4.1.3 Mapping language requirements

An examination of the mappings required between the design tool schemas used in this thesis, based on the classifications in Sections 4.1.1 and 4.1.2, leads to a range of pragmatic requirements for an ideal mapping language:

Language level: To enable rapid and concise specification, a language's notation should closely mirror the domain in which it is being used. A language to describe a mapping needs to be able to represent relationships between entities, attributes, references, and, in an object-oriented environment, methods, using a high-level notation. Using low-level notations will require the mapping specifier to concentrate on the practicalities of how to implement the mapping rather than on the specification of the actual mapping.

Language notation and modelling environment: The bulk of a mapping specification is concerned with relationships between attributes. Experience shows that many of these relationships need to be described in an equational form which is basically textual in nature.This tends to indicate that mapping languages will be based around a textual notation. However, given the potentially large size of the schemas involved in a mapping, a graphical notation allowing high-level views of the mapping will prove of benefit in the early definition of the mapping. A graphical notation for high-level views is likely to provide faster cognition of

relationships between entities in the schemas, in contrast to reading a purely textual specification, as relationships will be explicitly depicted rather than having to be determined by the user parsing a textual expression. A graphical notation supported by a modelling environment is also likely to prove a useful checking tool in ensuring that mappings are completely specified for particular entities. This is because a graphical notation will show the entities and their attributes in the same view as the mapping, allowing easy determination of what has or has not been mapped. As some of the class definitions are large and contain various types of data (attributes, relationships and methods), it will be useful to allow several views of a mapping specification, with different views concentrating on different aspects of the mapping. The specification of a mapping is also likely to require a modelling tool of a similar magnitude to that used for managing schema definition to maintain consistency between various mapping specifiers as well as between various views of the mappings.

Language style: A mapping language needs to support many levels of correspondence specification, from simple equivalences between attributes through to complicated programs to extract and manipulate data into the required form. The language should not presuppose a single style of implementation (e.g., batch mode; full model translation; or automatic incremental mapping). Rather, any implementation style should be able to be implemented from the specified mapping. This requirement would tend to favour a declarative style of mapping specification over a procedural style, as a procedural style is less amenable to many implementation styles (e.g., to an incremental update model). A declarative notation is likely to provide the highest level of specification and most closely satisfy the modelling level requirement defined above. This is because declarative languages do not specify a particular method for performing a certain function, but specify more what is required. Although a declarative approach may provide the highest level of definition, it is recognised that not all mappings will be able to be specified in a declarative manner. A procedural form may also need to be supported.

Bidirectionality: Many connections between tools require the same structures to be mapped in both directions. Where bidirectional mappings are required, the mapping language should support their specification without the need to duplicate information on the correspondences. Where only unidirectional mappings are required this should also be able to be specified.

Conditional mapping: Dependent upon the state of a model, or portions of a model, the data in the model may need to be mapped in different ways. To enable this in a mapping, it must be possible to specify conditions which must be satisfied prior to the application of a particular type of mapping. Such conditions provide one way of making explicit assumptions that are only implicitly specified in a schema.

Aggregation: The level of detail used to represent entities in a schema will vary enormously. When mapping between schemas with very different levels of detail it is necessary to aggregate information in very detailed schemas to fill higher level (more abstract or less detailed)

schemas. The reverse process will also be required, but it is often impossible to do this unambiguously (e.g., given a total glazing area it may be impossible to work out the area of each individual window). In this case it may be possible to constrain the constituents in the detailed model to match the values of the aggregated model.

Relationship handling: Similar to the problem of various aggregation levels is the problem of relationship structures. Different schemas of a domain are likely to choose different structures to represent the relationship between entities in their schemas. A mapping language needs to be able to restructure relationships in a schema, compressing long pointer chains, telescoping down into deep structures, and moving relationships between entities in different schemas.

Initialisers: During the mapping process new objects must be created and, in some cases, initial values set for attributes. Having a method to specify initial values independent of mappings, which may calculate values for these attributes as data becomes available, is important. This provides another way of making explicit assumptions that are only implicitly specified in a schema. It also provides a mechanism to ensure that a minimum set of data is created in a mapping, for example, to ensure that a design tool could run after any amount of data is transferred through in a mapping.

Unit handling: Attributes in different schemas often use different units to represent their quantities. Whether due to the country in which the schema was developed (i.e., imperial or metric units), the equations used, or just the magnitude of the result presented to the user, the ability to convert attributes between different units is required of a mapping language.

Type handling: Different schemas may use different precisions of types to represent their values, depending upon the accuracy required or the time and space limitations in the calculations. Different schemas may also use different structures to represent the same information (e.g., tree versus list) and a mapping language needs to map between the various types to ensure the model is in a valid state after a mapping has occurred.

4.2 Mapping Definition Languages

A wide range of languages are being developed (or have been developed over the last few years) to specify the mapping between schemas, or to enable the integration of schemas, or to provide a view of a schema. In this section, a representative set of these languages is examined to highlight the styles available and their expressive power.

To help illustrate the style and form of these languages, an example from a survey of mapping languages (Verhoef, Liebich and Amor 1995) will be used. This is one of five example mappings used in the paper. The mapping is between two schemas drawn from existing applications requiring data transfer in an integrated environment. The schema fragments for this example are shown in Table 4.5.

The schema fragment shown on the left-hand side describes building components which are either structural (i.e., columns or beams) or a connector between structural components. The relationship component has two forms, defined by the *quality* attribute, which are either support or element connections. The right-hand side schema fragment describes structural components, one specialisation of which is a structural connector. This structural connector has two specialisations of either a support or element connector. The mapping problem is to map between the component relationship in the left-hand side schema and either the support or element connector in the right-hand side schema. Performing the mapping in this example illustrates: a conditional mapping between entities; a type conversion (rather contrived) between two attributes; and attribute mappings which require the results of other mappings between entities.

```
TYPE   connection_type =
  ENUMERATION OF
       ( support_connection,
         element_connection ) ;
END_TYPE ;

ENTITY building_component
  ABSTRACT SUPERTYPE OF ( ONEOF
       ( structural_component,
         component_relationship ) ) ;
  id           : REAL ;
  UNIQUE u1    : id ;
END_ENTITY ;

ENTITY structural_component
  ABSTRACT SUPERTYPE OF ( ONEOF
       ( column, beam ) )
  SUBTYPE OF ( building_component ) ;
  specified_by : SET [1:?] OF
       product_characteristic ;
  represented_by :
       geometric_representation_item ;
END_ENTITY ;

ENTITY component_relationship
  SUBTYPE OF ( building_component ) ;
  related      : structural_component ;
  relating     : structural_component ;
  quality      : connection_type ;
END_ENTITY ;
```

```
TYPE   support_connection = ENUMERATION OF
  ( free_support, restrainted_support,
    un_known ) ;
END_TYPE ;

TYPE   element_connection = ENUMERATION OF
  ( joint_connection, rigid_connection,
    un_known ) ;
END_TYPE ;

ENTITY structural_component
  ABSTRACT SUPERTYPE OF ( ONEOF
       ( structural_assembly,
         structural_element,
         structural_connector ) ) ;
  identified_by      : INTEGER ;
  UNIQUE       u1     : identified_by ;
END_ENTITY ;

ENTITY structural_connector
  ABSTRACT SUPERTYPE OF ( ONEOF
       ( support_connector,
         element_connector ) )
  SUBTYPE OF ( structural_component ) ;
  related, relating : structural_element ;
END_ENTITY ;

ENTITY support_connector
  SUBTYPE OF ( structural_connector ) ;
  type_of      : support_connection ;
END_ENTITY;

ENTITY element_connector
  SUBTYPE OF ( structural_connector ) ;
  type_of      : element_connection ;
END_ENTITY
```

Table 4.5 Schema fragments for the two schemas in the mapping example

4.2.1 EXPRESS-M

EXPRESS-M (Bailey 1994) is an evolving language being developed to solve the problem of application protocol inter-operability in the STEP standard. As the language is intended for use in STEP it has been designed to look very similar to the EXPRESS language. EXPRESS-M mappings are unidirectional and map a whole model at a time (no partial updates of models). The

EXPRESS-M language has the following major components:

SCHEMA_MAP: specifies the EXPRESS schemas which are the source for the mapping and the schemas which are targets for the mapping. In most cases there is a one-to-one mapping from a single source schema to a single target schema.

MAP: specifies the mapping between entities in the source schema(s) and entities in the target schema(s). There can be only one MAP for a particular target entity so all conditional clauses for a mapping must be handled inside one MAP. The actual mapping of data between attributes is done through assignment statements with a large range of functions available to calculate the value to assign. Iteration constructs may be used to specify values for aggregate attributes, and user defined functions are accessible in the mapping. Type conversion is specified in the mapping by means of casting from the type of the source value to that required in the target. All conditional mapping, including determination of which type of entity to create as well as what equations to apply to calculate a value for an attribute, must be handled in the single MAP definition.

TYPE_MAP: specifies the mapping required to instantiate an attribute of one type from an attribute of a second type. This is for non-simple types (simple types have a default casting regime) and can handle enumerations, lists, sets, bags, and other structures. The TYPE_MAP is also used to map between attributes of differing units. In EXPRESS there is an overlap between the unit and type specification of an attribute which can lead to the case of two attributes in two schemas being of type 'litre' with one being a real number while the other is an integer. TYPE_MAP allows these problems to be tackled, as well as handling more usual type mapping.

```
MAP ONEOF(support_connector, element_connector) <- component_relationship;
       IF quality = support_connection THEN
              MAP support_connector <- component_relationship;
                     identified_by := {INTEGER}id;
                     related := {structural_element}related;
                     relating := {structural_element}relating;
                     type_of := un_known;
              END_MAP;
       ELSE
              MAP element_connector <- component_relationship;
                     identified_by := {INTEGER}id;
                     related := {structural_element}related;
                     relating := {structural_element}relating;
                     type_of := un_known;
              END_MAP;
       END_IF;
END_MAP;
```

Table 4.6 EXPRESS-M mapping for example problem

PRUNE: specifies entities which might be created twice in a SCHEMA_MAP and which should be culled to one instance. Multiple instances of an entity occur when there is a MAP for an entity and a reference to the entity attributes from an associated entity (i.e., the entity is also created by reference). EXPRESS-M is able to determine what to prune by monitoring separately those instances created in a MAP and those created by reference.

Manual entity instantiation: instances of entities can be created without a mapping from source entities through a simple manual instantiation specification which follows that used in STEP exchange files.

Table 4.6 shows the EXPRESS-M mapping for the example in Table 4.5, but only from the left-hand side schema to the right-hand side schema as EXPRESS-M is a unidirectional mapping language. The top level mapping specification specifies that a *component_relationship* is mapped to either a *support_connector* or an *element_connector*. The conditions which decide which type of mapping is performed are specified in the if-then-else statement. Type conversion must be specified explicitly when required as seen with the casting of *id*, which is a real, to an integer, as well as the *related* and *relating* object references.

Some of the drawbacks of EXPRESS-M are:

- mapping specifications are unidirectional, therefore it requires two mapping specifications to map data in both directions between models. For schemas where a bidirectional mapping is required, having two separate mappings which are inverses of each other is problematic. Apart from the extra time required to create the two mappings, it provides more chance for inconsistencies between the mapping specifications as schemas change and mappings are refined.
- only one MAP can exist for a particular entity combination. This can make the conditional mapping specification very convoluted and difficult to decipher. Table 4.6, where there are two mapping types to choose between, shows this in a small way.
- although parts of the language are declarative there are many procedural components (all handling of aggregate components) which interfere with the readability of mapping specifications.
- no graphical formalism is offered with the textual notation, allowing only the very low level implementation view of any mapping to be manipulated.
- the ordering imposed on evaluation (i.e., as it appears in the mapping) means very careful consideration of ordering is required when specifying mappings.

4.2.2 EXPRESS-V

EXPRESS-V (Hardwick et al. 1994, Hardwick 1994) is similar to EXPRESS-M in approach but with a limited ability to support bidirectional mapping between models. EXPRESS-V is intended to be an extension to EXPRESS to support database views in an ISO-STEP environment. EXPRESS-V definitions are specified along with EXPRESS definitions for a schema. The major components of EXPRESS-V are:

VIEW: specifies that an entity or entities are to be viewed as another entity. This can be a conditional specification using a WHEN clause to specify the conditions under which this VIEW may hold true. The actual mapping of data between attributes is done either in a VIEW_ASSIGN section, to map from source to target, or in an UPDATE section, to map

from target to source. Mappings are performed through an assignment statement with a large range of functions available to calculate the value to assign.

VIEW_ASSIGN: is defined in the VIEW clause and specifies what attributes in the target schema need to be updated when source entities are modified. The VIEW_ASSIGN section uses equations to perform the mapping of attributes in the source entities as described in the VIEW section. The VIEW_ASSIGN can be conditional through the use of a WHEN clause. There can be multiple VIEW_ASSIGN clauses in a VIEW specification.

UPDATE: is defined in the VIEW clause and specifies what attributes in the source schema need to be updated when target entities are modified. The UPDATE section uses equations to perform the mapping of attributes in the target entities as described in the VIEW section. The UPDATE can be conditional through the use of a WHEN clause. There can be multiple UPDATE clauses in a VIEW specification.

CREATE: is defined in the VIEW clause and specifies attribute values for source entities which have to be created by the creation of target entities. The CREATE can be conditional through the use of a WHEN clause. There can be multiple CREATE clauses in a VIEW specification.

DELETE: is defined in the VIEW clause and specifies which attributes and objects need to be deleted in the source entities upon the deletion of a target object. The DELETE can be conditional through the use of a WHEN clause. There can be multiple DELETE clauses in a VIEW specification.

```
VIEW support_connector
FROM (component_relationship)
WHEN (component_relationship.quality = 'support_connection');
VIEW_ASSIGN
       identified_by := component_relationship.id;
       type_of := 'un_known';
       related := component_relationship.related;
       relating := component_relationship.relating;
UPDATE
       id := support_connector.identified_by;
       related := support_connector.related;
       relating := support_connector.relating;
       quality := 'support_connection';
END_VIEW;

VIEW element_connector
FROM (component_relationship)
WHEN (component_relationship.quality = 'element_connection');
VIEW_ASSIGN
       identified_by := component_relationship.id;
       type_of := 'un_known';
       related := component_relationship.related;
       relating := component_relationship.relating;
UPDATE
       id := element_connector.identified_by;
       related := element_connector.related;
       relating := element_connector.relating;
       quality := 'element_connector';
END_VIEW;
```

Table 4.7 EXPRESS-V mapping for example problem

Table 4.7 shows the EXPRESS-V mapping for the example in Table 4.5. This example highlights the relational database approach underlying the language definition. New views must be defined for each class to be mapped, in this case views of a *component_relationship* as either a *support_connector* or an *element_connector*. The WHEN statement provides the conditions under which the view can be supported. The VIEW_ASSIGN statements describe the mappings required, and include implicit type conversion. The UPDATE section describes what can be modified in the original model when a view element is modified, this highlights a limitation of the language as it assumes one of the schemas to be a master schema from which views are created. In effect this limits views to what can be derived from one schema and does not allow creation of new objects from the view schema.

Some of the drawbacks of EXPRESS-V are:

- the mapping specification is associated with a particular schema (as in standard RDBMS views) and does not directly specify the other schema being accessed (although it is specified in a USES clause). This could make a schema specification for an IDM very convoluted as it could contain all mappings to design tools along with the schema definition. This approach also requires a flat name-space, as all entity definitions in all views are visible at the same time. This is an unreal expectation in a situation where existing applications (with predefined entity definitions) are to be integrated with an IDM.
- although the language allows for bidirectional mappings (though this is through separate VIEW_ASSIGN and UPDATE sections in the VIEW definition), the mapping in each direction needs to be specified independently, due to the procedural specification of a mapping. Although this allows the mapping specifications to be described at one point, having a section for each direction leads to the duplication of information in the mapping specification.
- the language assumes the source schema is always much more sophisticated than the target schema as CREATE and DELETE blocks are only available for mappings in one direction. Although in a generalised integrated design system it is likely that some target schemas would require CREATE and DELETE blocks as well.
- no graphical formalism is offered with the textual notation, allowing only the very low level implementation view of any mapping to be manipulated.

4.2.3 EXPRESS-C

EXPRESS-C (Staub et al. 1994) was developed to extend and enhance the capabilities of EXPRESS by enabling the modelling of both static and dynamic properties of a domain. It is considered as a first step towards a fully object-oriented version of EXPRESS as was suggested within the EXPRESS v2.0 development targets. The major mapping component of EXPRESS-C is:

TRANSACTION: a named transaction can be used to specify a unidirectional mapping between sets of objects accessed from the current model. As transactions describe a procedural

mapping between their referenced objects, two transactions would be required to describe a bidirectional mapping.

```
TRANSACTION t_map_component_relationship;
LOCAL
        socr : SET OF component_relationship;
        sosc : SET OF structural_connector := [];
END_LOCAL;
        socr := POPULATION('BSSC.COMPONENT_RELATIONSHIP');
        REPEAT i := 1 TO HIINDEX(socr);
                sosc := sosc + map_component_relationship(socr[i]);
        END_REPEAT;
END_TRANSACTION;

FUNCTION map_component_relationship
        (cr : component_relationship) : structural_connector;
LOCAL
        sc : structural_connector;
END_LOCAL;
        IF (cr.quality = support_connection) THEN
                sc := compare (support_connector(support_connection.un_known) ||
                structural_connector (map_structural_component(cr.related),
                map_structural_component(cr.relating)) || structural_component(cr.id));
        ELSE
                sc := compare (element_connector(element_connection.un_known) ||
                structural_connector (map_structural_component(cr.related),
                map_structural_component(cr.relating)) || structural_component(cr.id));
        END_IF;
        make_instances_persistent([sc]);
        RETURN(sc);
END_FUNCTION;
```

Table 4.8 EXPRESS-C mapping for example problem

Table 4.8 shows the EXPRESS-C mapping for the example in Table 4.5, but only from the left-hand side schema to the right-hand side schema as EXPRESS-C is a unidirectional mapping language. The transaction specification shows the very low level implementational approach taken with this language. The transaction definition identifies all *component_relationship* objects in a model and creates the corresponding set of mapped objects using the function *map_component_relationship()*. This function utilises standard EXPRESS statements to describe a conditional creation of objects and calls other functions to map the two object references.

EXPRESS-C has the same set of drawbacks as described in EXPRESS-V along with the lack of explicit support for bidirectional mappings.

4.2.4 Transformr

Transformr (Clark 1992) was designed to be able to propagate instances between different versions of a model. As with EXPRESS-M and EXPRESS-V, Transformr was developed for use in the ISO-STEP environment. The major components of Transformr are:

COPY: specifies the copying of an entity to another entity. In its simplest form it simply names an entity and all attributes of this entity are copied across. COPY can move all objects between entities with different names, or, through the use of derived attributes, add, or drop attributes from the new entity.

BUILD: specifies the creation of a new entity from a set of entities in the source. This is a conditional creation, where all attributes that need to appear in the target entities must be described, unlike COPY which maps everything unless the user specifies otherwise.

```
BUILD support_connector FROM component_relationship
       WHERE
              component_relationship.quality = support_connection;
       DERIVE
              identified_by := REAL_TO_INT(component_relationship.id);
              related := component_relationship.related;
              relating := component_relationship.relating;
              type_of := un_known;

BUILD element_connector FROM component_relationship
       WHERE
              component_relationship.quality = element_connection;
       DERIVE
              identified_by := REAL_TO_INT(component_relationship.id);
              related := component_relationship.related;
              relating := component_relationship.relating;
              type_of := un_known;
```

Table 4.9 Transformr mapping for example problem

Table 4.9 shows the Transformr mapping for the example in Table 4.5, but only from the left-hand side schema to the right-hand side schema as Transformr is a unidirectional mapping language. Separate BUILD statements are used for each type of object to be created with the WHERE statement defining the condition under which the BUILD can be used. Mappings in the DERIVE section are simple statements, though explicit type conversion through user defined functions is required for simple types, as shown with the mapping of *id*.

Transformr is a simple language which has an elegant style and appears well suited to the process of propagating data between versions of the same schema. It is, however, very much a unidirectional mapping language, with limited functionality in terms of the combinations of entities which can be clustered and created. Transformr is also limited in the types of equivalences that can be specified between attributes.

4.2.5 EDM-2

EDM-2 is a novel language, database system and environment developed at UCLA for use in the A/E/C domains (Eastman et al. 1995). EDM-2 incorporates three major features not found in traditional database systems, but which are of key importance in the development of an integrated design environment. These are:

Dynamic schema specification and evolution: a central schema in an integrated environment can be modified at any time to take account of new applications to be utilised in the system, or to incorporate new views of the existing schema for specific user needs.

In-built integrity management: through the use of constraints specified in the schema, or defined "on the fly" as a model is developed, the integrity of a model can be determined at any stage. Constraints with parameters which have been modified can be rechecked at any

time, and in this manner the global consistency of a model can be monitored as a design progresses.

Explicit translation definition: high-level support for translation between applications (bidirectionally) is incorporated into EDM-2.

```
CREATE DE bssc_component_relation  KEYNAME
        DESC "BSSC component_relation class";

CREATE DE pss_structural_connector KEYNAME
        DESC "PSS structural_connector class";

CREATE DE gen_part KEYNAME
        ATTR(bssc: bssc_component_relation, pss: pss_structural_connector)
        DESC "Generalized beam class";

CREATE MAP components
        (bssc_component_relation)
        RETURN (pss_structural_connector)
        IMPL $MAP_METHODS/components.so
        DESC "Mapping  from BSSC model to PSS model";

CREATE MAPCALL component_mapping
        MAP components
        (bssc)
        RETURN (pss)
        REF gen_part
        DESC "Map call with reference to generalized object";
```

Table 4.10 EDM-2 mapping for example problem

The definition of mapping is supported by two constructs in the EDM-2 language:

MAP: defines a process through which entities of a particular type can be translated from one type to the other.

MAPCALL: describes the use of a MAP for particular instances in the model.

Table 4.10 shows the EDM-2 mapping for the example in Table 4.5, but only from the left-hand side schema to the right-hand side schema. The MAP definition describes the inputs and outputs of the mapping whose implementation is hidden in the file referenced by the IMPL statement (the C code of this function is not shown here for brevity). The MAPCALL definition lists the MAP statements which are to be used when mapping from one schema model to another, in this case from *bssc* to *pss*.

Mapping constructs have only recently been added into EDM-2, so details of their incorporation are still under review. Currently the maps provide unidirectional mapping between entities, with the maps associated with the central schema (though separate schemas tend to be merged in the EDM-2 implementation). Mapping definitions are not seen explicitly in the mapping definition, as currently the implementation is as C programs which are invisible to the user browsing the schema.

4.2.6 KIF

KIF (The Knowledge Interchange Format: Genesereth and Fikes 1992; Khedro et al. 1994) was originally developed in the ARPA knowledge sharing initiative as a means to exchange information between applications. KIF provides a mechanism for agents to communicate messages to inter-operating applications. In KIF each agent has the responsibility of translating messages received from other agents from their native format to the format required locally. To achieve this, each agent must define a translation for messages they wish to handle. To use KIF in a standard IDM-type system would require an IDM where translations for every attached schema are written into the IDM. Each application would require translations from the IDM structures into their own internal format. This places the onus of translation on every application working in the integrated system. However, it also allows for a system where individual modifications can be propagated to all interested applications as they occur and where incremental consistency of the whole integrated system can be maintained. The main drawback of KIF is the requirement that each application be aware of the IDM and be able to translate information from the IDM whilst performing its own application tasks.

```
(<= (pss!support_connector ?ent)
       (bssc!component_relationship ?ent)
       (= (bssc!component_relationship.quality ?ent) support_connection))
(<= (pss!element_connector ?ent)
       (bssc!component_relationship ?ent)
       (= (bssc!component_relationship.quality ?ent) element_connection))
(<= (= (pss!support_connector.identified_by ?ent) ?id)
       (= (bssc!component_relationship.id ?ent) ?id)
       (= (bssc!component_relationship.quality ?ent) support_connection))
(<= (= (pss!element_connector.identified_by ?ent) ?id)
       (= (bssc!component_relationship.id ?ent) ?id)
       (= (bssc!component_relationship.quality ?ent) element_connection))
(<= (= (pss!support_connector.type_of ?ent) ?type)
       (= (bssc!component_relationship.quality ?ent) ?type)
       (= ?type support_connection))
(<= (= (pss!element_connector.type_of ?ent) ?type)
       (= (bssc!component_relationship.quality ?ent) ?type)
       (= ?type element_connection))
```

Table 4.11 KIF mapping for example problem

Table 4.11 shows the KIF mapping for the example in Table 4.5, but only from the left-hand side schema to the right-hand side schema as KIF is a unidirectional mapping language. The mapping reflects the blackboard architecture that KIF is implemented upon, with every item mapped individually. The first two definitions create either a *support_connector* or an *element_connector* depending upon the value of the *quality* attribute. The second two definitions create the *identified_by* attribute for the objects and the last two create the *type_of* attribute for the objects. The mapping of object references (i.e., *related* and *relating*) are not attempted in this mapping.

4.2.7 Superviews

Superviews (Motro 1987) describes a method for the virtual integration of multiple databases. The method is founded on a set of ten operators with which the integrator defines the requisite transformations to the base schemas to produce the superview. The product of this method is a

superview of the base schemas and the set of mappings (reversible) required to move information back and forth between the superview and base schemas. The ten operations available are:

Meet: produces a common generalisation of two entities. This is only possible when two entities have a common key. A *meet* also introduces a consistency constraint, that values in both entities with the same key must agree on shared attributes.

Join: the resultant type is the union of the types of two entities.

Fold: allows a generalisation to absorb a more specific entity. The system must use a null value for attribute values of the specific entity that were not in the original generalised entity.

Rename: renames a entity.

Combine: joins two entities which have identical types, and creates a new entity with a new entity name.

Connect: joins two entities which have identical types, and the new entity has the name of one of the original entities.

Aggregate: creates an intermediate entity between an existing entity and a subset of its attributes. *Aggregate* can be used to normalise the schema to a form where non-key attributes of each entity are fully dependant on the key. *Aggregate* allows the relocation of an attribute on the schema.

Telescope: removes an entity by assigning its attributes directly to its ancestor entity. *Telescope* allows the relocation of an attribute on the schema.

Add: allows the addition of implicit attributes to an entity, along with a constant function to assert the value.

Delete: removes a portion of the database not relevant to the application.

Superviews appears well suited to a specific class of problems, those that can be described with these ten operators. For this class of mapping, data can be moved between any schema and the superview with guaranteed consistency. For any schemas which fall outside that which can be described with these operators it provides no help. Such schemas include those which do not have simple one-to-one correspondences between attributes in a schema and those in the superview. Therefore, if attributes are derived from an equation, or aggregated values, Superviews can not be used. This limits the utility of Superviews in large model mapping systems, as most schemas in this domain require sophisticated manipulation of attributes as well as manipulation of object references, which are also not handled in this RDBMS-based integration scheme.

4.2.8 RDBMS views

In relational database systems, views of the conceptual database are a well established concept. A view provides an abstract schema of a portion of the conceptual database. This notion differs from the notion of mappings between schemas, in that the mapping may not be to a schema which is a portion of a conceptual database, the mapping may be to a schema which is a superset of the conceptual schema. RDBMS views may, under certain conditions, be updateable, thus providing the effect of a bidirectional mapping as discussed in this thesis. However, views that are

updateable have severe restrictions on their definition, which limit them to what amounts to a one-to-one mapping between tables with direct equivalences between attributes (i.e., no equations or functions in the view definition).

Language / **Requirement**	EXPRESS-M	EXPRESS-V	EXPRESS-C	Transformr	EDM-2	KIF	Superviews	RDBMS
Language level	M	M	M	M	M	M	L	L
Declarative / Procedural specification	P	P	P	P	D	D	P	P
Object-oriented language support	L	L	L	-	M	M	-	-
Object-oriented method support	-	-	-	-	-	L	-	-
Bidirectional mapping support	-	L	-	-	L	L	-	-
Conditional mapping invocation	M	M	M	L	H	H	L	M
Initial values for attributes and object creation	M	M	M	L	M	L	L	L
Aggregate detail over objects and attributes	M	M	M	L	H	M	M	H
Relationship handling (expanding and truncating pointer chains)	M	M	M	M	M	M	M	M
Class graph-based model (Yes / No)	Y	Y	Y	Y	Y	Y	Y	Y
Class and attribute graph-based model (Yes / No)	N	N	N	N	Y	Y	Y	Y
Unit handling is Implicit / Explicit	E	E	E	E	E	I	E	E
Type handling is Implicit / Explicit	E	E	E	E	E	I	E	E
Temporary structures (objects and attributes) available (Yes / No)	N	N	N	N	Y	Y	Y	Y
Graphical notation available (Yes / No)	N	N	N	N	Y	N	N	N

Table 4.12 Comparison of mapping languages (H=High, M=Medium, L=Low, -=None)

4.3 Summary of Inter-Schema Relationship Modelling

Table 4.12 provides a summary relating the surveyed mapping languages to the requirements detailed in Section 4.1.3. As shown in the table, none of the surveyed languages provides for the full range of requirements necessary. Of particular concern is the low level of support for bidirectional mapping, given that this type of mapping is the norm in the domains of these languages. There is also a need for object-oriented support, especially as the modelling notations that many of the mapping languages support are themselves object-oriented. Missing from many of the EXPRESS-based languages (which are promoted for this domain) are notions of classes and attributes forming the graph representing a mapping, as well as the ability to define temporary structures for partial mappings or reusable states. Almost all of the languages have no graphical

notation, and even EDM-2's notation does not provide a good overview of what takes place in an individual mapping.

To address the deficiencies in the existing languages, a new view mapping language (VML) is proposed in Chapter 5. It has both a textual and graphical notation, and a modelling environment for VML is described in Chapter 6.

Chapter 5

The View Mapping Language (VML)

All the work described in the previous section tackles the problem of mapping between schemas to some extent, but none of the approaches described provide the full range of abilities described in Section 4.1.3 as being required for a general inter-schema mapping language. In this section the View Mapping Language (VML) is presented. VML overcomes many of the problems identified in the languages canvassed in Chapter 4. VML is a high-level, declarative, and bidirectional language suitable for the description of correspondences between two arbitrary schemas of a domain. VML dispenses with all notions of target and source schemas in the mapping definition. As far as practicable, a VML definition treats both schemas as equal partners in a mapping. VML also removes many distinctions between entities and attributes, to allow mappings between entities and attributes to be specified in the same way that attribute to attribute mappings are specified.

Throughout this section examples are used to illustrate each construct in the VML language. These examples are drawn from the large mapping example described in Section 1.6 and are fully specified in Appendix E. However, to complement the examples in Section 4.2, the VML specification for the example shown in Table 4.5 can be seen in Table 5.1. All of the components of this formalism are fully described later in this section, but, a brief description of the mapping in Table 5.1 is as follows. The example shows separate *inter_class* definitions for each type of class to be mapped between, along with *invariants* specifying the conditions under which the mapping holds. This combination of classes and invariants must be unique for every *inter_class* definition, and will be checked by any mapping implementation. *Equivalences* specify the mappings to be undertaken, all with implicit type conversion and implicit object type mapping (i.e., for *related* and *relating*). All *inter_class* definitions can be used in both directions depending upon where data resides that requires mapping. *Initialisers* specify initial values of attributes when they are created by the application of a mapping.

```
inter_class([component_relationship], [support_connector],
	invariants(
		quality = 'support_connection'
	),
	equivalences(
		id = identified_by,
		related = related,
		relating = relating
	),
	initialisers(
		type_of = 'un_known'
	)
).

inter_class([component_relationship], [element_connector],
	invariants(
		quality = 'element_connection'
	),
	equivalences(
		id = identified_by,
		related = related,
		relating = relating
	),
	initialisers(
		type_of = 'un_known'
	)
).
```

Table 5.1 VML mapping for example problem

In a VML environment it is assumed that all mappings are between two schemas. When it is necessary to map information between several schemas and a single schema each mapping is specified independently so that the mapping implementation can manage updates to and from each model independently.

At the top level, considering the complete mapping between schemas, the work from database views gives two possibilities for the types of mappings which can exist. These are read-only views and read-write views. This carries over to schema mappings as well: a mapping can operate in one direction, giving a read-only view; or in both directions giving read-write views.

A VML mapping consists of an introductory specification of the schemas to be mapped between, and then a set of correspondences between entities and attributes to describe how the mapping is to be achieved (see Table 5.2). The syntax of VML is similar in style to that of Prolog and Snart, the implementation languages of this thesis, but it could easily be rewritten to resemble the syntactic style of EXPRESS or other modelling languages without affecting the semantics of the language.

```
mapping = inter_view_def { inter_class_def } .
```

Table 5.2 Top level definition of a VML mapping

5.1 Mapping between schemas

A VML mapping definition commences with an *inter_view* definition, which specifies the two schemas between which a mapping is to be described, the type of view represented by the schemas in this mapping, and the completeness of mapping that is required (see Table 5.3).

```
inter_view_def = 'inter_view(' model_id ',' model_type ',' model_id ',' model_type ','
       map_type ')' '.' .
model_id = simple_id [ '{' version '}' ] .
model_type = 'integrated ' |
       'read_only ' |
       'read_write ' .
version = integer_literal |
       real_literal |
       atom_literal |
       string_literal .
map_type = 'complete' |
       'partial' .

For example:
inter_view(idm{0.09}, integrated, planentry, read_write, complete).
```

Table 5.3 Definition of an *inter_view* specification

The *model_id*s specify the names and optional version numbers of the schemas which are being mapped between in this mapping. The first *model_id* specified is treated as the left-hand side schema and the second as the right-hand side schema. The side of the schema is used to determine which entities and attributes are being referenced in a mapping. If an entity appears on the left-hand side of a specification then, by default, it belongs to the first schema specified in the *inter_view* (though this can be explicitly over-written, as described later).

The optional version number (e.g., *idm{0.09}* in Table 5.3) associated with a schema enables a mapping to be specified to a specific version of a schema. The use of version numbers also allows a mapping to be specified between different versions of the same schema, supporting schema evolution.

The *model_type* indicates the role that a schema instance plays in this mapping. As the reason for a mapping between models varies depending upon the model being connected, the *model_type* allows the specification of different roles for a model dependent upon the connection being made. There are three allowable values for *model_type*:

read_only: specifies that data can be mapped from that model to the other model, but not vice-versa. Specifying a *read_only* role does not restrict the data in either model from being modified, it just affects a model's ability to pass changes through to its connected model. The *model_type* for both models can not be set to *read_only*.

read_write: specifies that data can be mapped to this model as well as mapped out of it. Where both models are depicted as *read_write* there is no synchronisation when propagating changes between the models.

integrated: has almost the same meaning as a *read_write* model. The exception is that all changes waiting to be passed on from the *integrated* model must be accepted before a change is mapped to this model. The need for this level of synchronisation is discussed in Chapter 2. Only one of the two model types can be *integrated* to avoid deadlock situations.

The three *model_types* cover all the connections that may be required between models. A *read_only* model is for the situation where it is necessary to provide information to a participant but not necessary for them to modify the model it is being passed from. The *read_write* and *integrated* model types are for mappings where changes from the connected model can be accepted. With the *integrated* model type it is possible to ensure the consistency of data in a connected model before allowing modifications to be made by it. Individual models can play different roles in different mapping specifications which can lead to interesting chains of connections. For example, a model which has a *read_only* connection to one model could have an *integrated* connection to another set of models. This type of connection could be used to provide a barrier between a central model and a set of users who wish to experiment with aspects of the data as well as share their modifications with each other (see Figure 5.1).

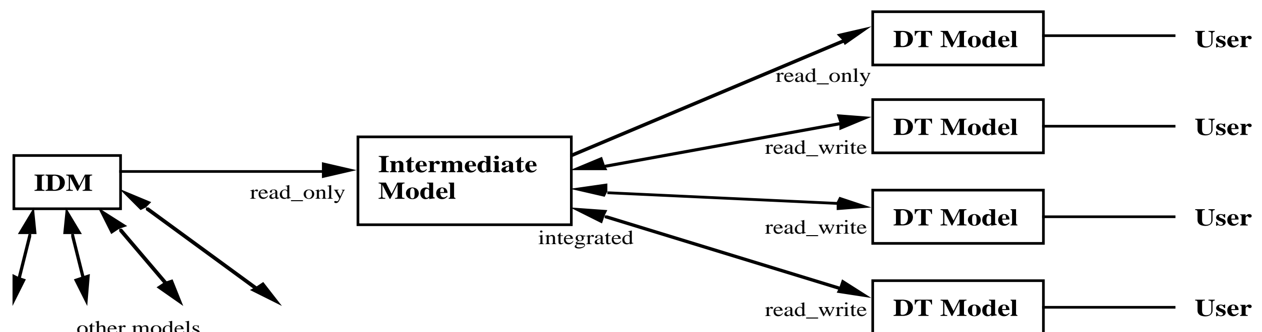


Figure 5.1 A tree of *inter_view* mappings

The *map_type* indicates how complete the movement of data needs to be between the two models. The two allowable values for *map_type* are:

complete: all objects of the entities described in the mapping must be mapped to the datastore for the other model. This is used for models representing the same type of objects (e.g., a whole building) and where a mapping only makes sense if everything can be moved across. That is, the derived model is inconsistent if it is not possible to map all objects of types mentioned in the mapping specification.

partial: is used where one schema describes a much smaller domain than the other (e.g., a room schema versus a building schema) and where it is not necessary to map all the information across to the other model (e.g., only a single room is required to be mapped rather than all the rooms in the building).

5.2 Mapping between classes

Following the *inter_view* definition is a set of *inter_class* definitions which describe the relationships between entities in the two schemas. An *inter_class* definition details: the entities from both schemas that take part in the mapping; an optional set of conditions which must hold to use the mapping; the actual relationship between data in the two entities; and, optionally, initial values for attributes when an object is created (see Table 5.4). Table 5.4 also shows an example *inter_class* specification. This example describes the mapping between *idm_space_face* objects in one schema and *v3d_polygon* objects in another schema under the condition that the *type_of_face* of the *idm_space_face* is equivalent to the value *opening*. Where this is the case there are two functional mappings specified in the equivalences section to map the object identifier and shape between representations. There are also initial values specified for attributes of the *v3d_polygon* object which provide default reflection and colour information.

```
inter_class_def = 'inter_class(' class_list ',' class_list [ ',' inherits ]
        [ ',' invariants_def ] [ ',' equivalences_def ] [ ',' initialisers_def ] ').' .
class_list = '[' [ class_key_name { ',' class_key_name } ] ']' .
class_key_name = 'group(' class_name ')' | class_name .
inherits = 'inherits(' inherit_list ')' .
invariants_def = 'invariants(' invariant_expr { or_op invariant_expr } ')' .
equivalences_def = 'equivalences(' equivalent { ',' equivalent } ')' .
initialisers_def = 'initialisers(' initialiser { ',' initialiser } ')' .

For example:
inter_class([idm_space_face],[v3d_polygon],
        invariants(
                type_of_face = 'opening'
        ),
        equivalences(
                map_id_to_num(idm_space_face, object_id),
                map_polar_rect_to_polygon(min=>x, min=>y, max=>x, max=>y, plane=>axis,
                        plane=>offset, points[1], points[2], points[3], points[4])
        ),
        initialisers(
                diffuse_reflection = 0.1,
                colour=>r = 0,
                colour=>g = 0,
                colour=>b = 0
        )
).
```

Table 5.4 Definition of an *inter_class* specification

5.2.1 Entity names and keys

The two class lists in an *inter_class* definition specify the entities which are involved in the mapping being specified. As detailed in the *inter_view* definition, the first *class_list* refers, by default, to entities from the first schema and the second to the second schema. The union of the two lists of entity names provides the key for this mapping. There can be any number of mappings with the same key. However, each specification must be distinguishable from the others by the conditions specified in the *invariants* specification. In the example shown in Table 5.4 the key is [idm_space_face, v3d_polygon] and though there are several *inter_class* definitions between these

two classes (see Appendix E) they are all uniquely identified by their invariants, in this case *type_of_face = 'opening '*.

In some mappings it is necessary, and in many other mappings very convenient, to be able to describe a mapping to temporary entities, e.g., where two mappings can reuse a partial mapping in their transformations. To be able to distinguish mappings to temporary entities from those which create real objects in a particular view we denote temporary entities by names prefixed either with an underscore symbol, e.g., _temp1, _temp2, or with a capital letter, e.g., Temp1, Temp2 (see Table 5.5 for the syntax of allowable class names). Temporary entities provide a mechanism to specify entities which do not exist in the schemas of the two systems that are being mapped between.

In Table 5.4 the *class_list* is defined as consisting of one or more *class_key_name*s. Having more than one class specified in a *class_list* allows the mapper to associate objects from a model when the objects previously had no association. A *class_list* with multiple *class_key_name*s denotes that when constructing an instance of this mapping, an object from each class in the *class_list* is required. This provides a way of associating objects of classes which may not be directly accessible from objects of the first class defined in the list, creating an effect similar to a join in a relational database system. The manner in which objects are associated is dependent upon the invariants specified in the mapping. Where no invariants are specified, the number of mappings that would be performed is equal to the cross-product of all objects for each of the named classes. Where invariants are specified, the number of mappings is restricted by application of the invariants to the cross-product of all objects of all named classes. The following small example helps illustrate how this works. The example shows five objects, two of class *a* and three of class *b*. Both classes have an attribute called *type,* and the value of *type* is shown for all five objects. Two *inter_class* definitions are shown. The first has no invariants specified, and as can be seen from the set of object pairs, displayed after the *inter_class,* this forces a complete cross-product of objects from both classes. The second *inter_class* has an invariant requiring the *type* attribute of objects of class *a* and *b* to be equivalent. This reduces the object grouping down to three sets of object pairs, rather than the six for the full cross-product.

Object ID	Class	Object.type
o1	a	1
o2	a	2
o3	b	1
o4	b	1
o5	b	2

```
inter_class([a, b], [c], .....).
        [[o1, o3], [o1, o4], [o1, o5], [o2, o3], [o2, o4], [o2, o5]]
inter_class([a, b], [c], invariants(a.type = b.type), .....).
        [[o1, o3], [o1, o4], [o2, o5]]
```

The *group()* specifier also suppresses creation of the full cross-product. It allows a collection of objects of the named class to be grouped together for the purposes of the mapping. This is commonly used to group multiple objects of a single class into a collection that is not explicitly supported in the original schema, and then to map the collection to a schema which requires this grouping. Without invariants, *group()* selects all objects of the named class; with invariants, the objects of the class are restricted by application of the invariants to each individual object being grouped. To illustrate how *group()* works consider the small example above with modified *inter_class* definitions as below. The first *inter_class* shows that all objects of class *b* are grouped with objects of class *a* in a set rather than a cross-product. The second *inter_class* shows that the objects in the grouped set can be restricted through the use of invariants.

inter_class([a, group(b)], [c], ...).

[[o1, [o3, o4, o5]], [o2, [o3, o4, o5]]]

inter_class([a, group(b)], [c], invariants(a.type = b.type), ...).

[[o1, [o3, o4]], [o2, [o5]]]

In an *inter_class* specification, one of the *class_list*s can be left empty. This allows the specification of initial conditions that must be established when a mapping between models is initiated. In most systems this would be to allow the creation of an object with initial values when a model is created (e.g., the controller of a design tool, or entities which have no representation in the schema being mapped from).

Table 5.5 shows the definition of a class name. While the default reading of a mapping is that the order of schemas in the *inter_view* definition is the order of classes in an *inter_class* definition the ordering can be overridden through the specification of the *model_id* in the class name. This allows classes from a schema to be specified in either side of an *inter_class*, and also allows for mappings between classes of a schema and temporary entities to be defined (or between temporary entities and temporary entities) in any order the mapping specifier desires.

```
class_name = [ model_id ':' ] class_id |
        variable_id .
model_id = simple_id [ '{' version '}' ] .
class_id = simple_id .
variable_id = upper_case { simple_id_char } |
        '_' ( letter | digit ) { simple_id_char } .

For example:
idm_space_face
idm{0.09}:building
_temp
```

Table 5.5 Definition of a *class_name*

5.2.2 Inheritance of *inter_class* definitions

The specification of inherited *inter_class* definitions allows *inter_class* specifications to closely model the structures that are found in object-oriented schemas. Where there are correspondences between the parent classes of a set of child classes it is more efficient and more maintainable to

specify inherited correspondences between the parent classes than to re-specify the mappings for each child class. The type of mapping where this feature will be most commonly utilised is in version mapping for object-oriented schemas. The specification of an inherited mapping, as shown in Table 5.6, utilises the key of a mapping to determine which mappings to inherit. Where the mapping specified has multiple definitions for that key (i.e., with different invariant specifications), then the combined mapping will be expanded into a set of mappings encompassing all combinations of invariants from the inherited *inter_class* definitions.

```
inherits = 'inherits(' inherit_list ')' .
inherit_list = inherit_map { ',' inherit_map } .
inherit_map = 'inter_class(' class_list ',' class_list ')' .

For example:
inherits(inter_class([person],[person]))
```

Table 5.6 Definition of inheritance

5.2.3 Invariant specification

Invariants are an optional part of an *inter_class* definition, and describe the conditions under which it is possible to use a particular *inter_class* definition. For example, defining an invariant *invariants(building.type = 'commercial')* in an *inter_class* definition would denote that it was only possible to use this *inter_class* definition for buildings which are commercial buildings, and presumably there would be other *inter_class* definitions which would specify what to do with other types of buildings. Thus, invariants are selection criteria to use when deciding which *inter_class* definition to apply to any given object. Each individual invariant expression can only reference objects and attributes from a single schema in the mapping. Invariants are therefore broken into two sets, those which apply to classes in one schema and those which apply to classes in the other. When deciding if an *inter_class* definition is to be used, all invariants which apply to the schema being mapped from must evaluate to *true* on the objects being tested.

```
invariants_def = 'invariants(' invariant_expr { or_op invariant_expr } ')' .
invariant_expr = invariant_simple_expr { and_op invariant_simple_expr } .
invariant_simple_expr = '(' invariant_expr { or_op invariant_expr } ')' |
        expression rel_op expression |
        predicate |
        function |
        method |
        'group(' attribute_name { ',' attribute_name } ')' .

For example:
invariants(
        type_of_face \= 'opening',
        pe_face.offset = pf_plane_object.offset,
        map_orientation_axis(pe_face.orientation, pf_plane_object.axis),
        contained_in_face(pe_face, pe_opening)
)
```

Table 5.7 Definition of invariants

The invariants also create constraints on the objects that are mapped, which in some cases can be used to fill in values in an object, or in others to create constraints on an object. In the example above it can be seen that when an object is mapped back onto the building with invariant

building.type = 'commercial' then the value of *type* can be automatically set to *'commercial'* without having to use an equivalence definition to specify this.

Invariants can be thought of as boolean expressions which must equate to *true* to allow the mapping to proceed. Invariants (the syntax of which is shown in Table 5.7) can be composed of functions, predicates or object method calls which succeed or fail, or expressions containing relational operators (e.g., =, >=, <). Sets of invariant conditions can be joined together through the use of *and* operators ',' or *or* operators ';'. VML offers one special function to be used with grouped classes in an *inter_class* definition. The *group()* function can only be used with a grouped class (see Section 5.2.1). It can only be used on an attribute of a grouped class which has a finite domain (usually an enumerated type) and is used to collate all objects of the named class whose value for the named attribute is identical. For example, specifying *inter_class([group(building)], [aggregation_of_type], invariants(group(building.type)), ...)* would ensure that a separate mapping would be performed for each type of building in the first model, pooling values into a single object in the model of the second schema for every different value of *type* found in the model of the first schema.

The example in Table 5.7 provides a complex set of invariants which must be satisfied to allow a particular *inter_class* specification to be used. In this specification the *type_of_face* attribute must contain the value *'opening'*, and the *offset* object referenced by *pe_face* and *pf_plane_object* must be the same, also the functions *map_orientation_axis()* and *contained_in_face()* must evaluate to true with the supplied parameters.

```
initialisers_def = 'initialisers(' initialiser { ',' initialiser } ')' .
initialiser = expression '=' expression |
        predicate |
        method .

For example:
initialisers(
        idm_space_face.face_property = 'idm_space_face',
        idm_material_face.face_property = 'idm_material_face',
        idm_material_face.material=>type_of_material = 'idm_window_material',
        idm_material_face.material=>type_of_window = 'idm_single',
        idm_material_face.material=>window_subtype = 'clear',
        fe_opening@create(idm_space_face.plane, idm_space_face.plane, 'space', 0, 0,
                idm_space_face.min=>x, 0 - idm_space_face.min=>y,
                idm_space_face.max=>x, 0 - idm_space_face.max=>y,
                idm_material_face.material=>window_subtype)
)
```

Table 5.8 Definition of initialisers

5.2.4 Initialiser specification

The optional initialiser section allows the definition of initial values for attributes of objects created in an *inter_class* specification. For models that are object-oriented, the initialiser section also provides a location to specify the *create* method parameters for objects that may be created during

the mapping. The initialiser section (the syntax of which is defined in Table 5.8) usually comprises mainly assignment statements specifying values for attributes, though predicates and procedures can also be specified.

Any entity attribute, or referenced attribute, of any class specified in the class lists of the *inter_class* specification can be initialised in the initialiser section. Initialisers will only be applied to newly created objects (i.e., not to an existing object which is associated through an invariant specification) and may cause the creation of other objects (e.g., attribute assignment through pointer chains).

5.2.5 Equivalence specification

The equivalence section comprises the bulk of most *inter_class* definitions as it specifies the correspondences between the attributes of entities defined in the class lists of the *inter_class*. It is this section which contains all the equations, functions, and procedures which will need to be solved or executed to map between models of the schemas in the mapping. The declarative nature of VML is most evident in this section as equivalences are used to define mappings between attributes. The ordering of expressions is unimportant as their solution is dependent only on the state of the model being mapped from. The syntax of the equivalence section is shown in Table 5.9.

```
equivalences_def = 'equivalences(' equivalent { ',' equivalent } ')' .
equivalent = expression '=' expression |
        'map_to_from(' predicate ',' predicate ')' |
        'bijection(' bijection_expr ',' bijection_expr ')' |
        predicate .

For example:
equivalences(
        bijection(idm_space_face[].type_of_face \= 'opening', walls[]),
        bijection(idm_space_face[].type_of_face = 'opening', openings[]),
        idm_material_face = materials,
        idm_bracing_face = bracing,
        idm_plane.name = name,
        idm_plane@view_plane = fe_application@create_view(_, idm_plane.name)
)
```

Table 5.9 Definition of equivalences

5.2.6 Mapping equations

The style of equations that can be used to define the mapping between classes was developed to meet the mapping types identified in Section 4.1.3. The major abilities of the VML language are described below. For the full syntax of the language refer to Appendix A.

5.2.6.1 Attribute initialisation or constant value specification

A constant value can be assigned to an attribute by equating the value with the named attribute as shown in the examples below.

type_of_face = 'opening'

diffuse_reflection = 0.1
gloss_factor = 90.0

This type of equation can appear in invariants, equivalences and initialisers and has slightly different semantics in each. When specified in an initialiser these equations provide the initial value for an attribute which may be modified by other equations in the equivalences section. When specified in an invariant or equivalence they specify a constant value. If the value is specified in the invariant it is used either to determine which mapping to apply to the class the attribute belongs to, or to initialise an attribute for a newly created object. If the value is specified in an equivalence, it specifies a value for the attribute which may not be modified (i.e., it will always be reset to the specified value). This specification corresponds to the Attr->Attr mapping of cardinality 0:1.

5.2.6.2 Equality

Direct equality between two attributes of a simple type (e.g., REAL, INTEGER, BOOLEAN), or named types, can be specified by equating one attribute with the other attribute as shown in the examples below.

name = planename
axis = axis
offset = offset

This type of equation can appear in invariants, equivalences and initialisers and has slightly different semantics in each. When specified in an initialiser these equations provide the initial value for an attribute which may be modified by other equations in the equivalences section. If the equality is specified in the invariant it specifies attributes of entities on the same side of a schema, the values of which must match for the *inter_class* to be used to perform the mapping. If the equality is specified in an equivalence then it denotes that the attributes of the respective entities must hold the same value. This specification corresponds to the Attr->Attr mapping of cardinality 1:1.

5.2.6.3 Pointer equality

Equality between pointers to objects of entities in the schemas is specified in the same manner as for equality between attributes of simple types, e.g.:

plane = fe_face_window

This type of equation can appear in invariants, equivalences and initialisers and has slightly different semantics in each. When specified in an initialiser these equations provide the initial object reference for an attribute which may be modified by other equations in the equivalences section. If the equality is specified in the invariant, then it will be specifying reference attributes of entities on the same side of a schema whose object references must match for the *inter_class* to be used to perform the mapping. If the equality is specified in an equivalence, then it denotes that each attribute has the object identifier of the object that was created from the *inter_class* definition between the entities of the two objects. This specification corresponds to the Attr->Attr mapping of cardinality 1:1.

5.2.6.4 Simple equations

Equations can also be used to define relationships between various attributes, as shown in the examples below. The range of algebraic and transcendental functions available is the set of functions supported by LPA Prolog.

```
min=>y = 0 - y0
r * sin(theta) = y_coord
r = sqrt(x_coord * x_coord + y_coord * y_coord)
```

This type of equation can appear in invariants, equivalences and initialisers and has slightly different semantics in each. When specified in an initialiser, these equations provide the initial calculated value for an attribute, which may be modified by other equations in the equivalences section. If the equation is specified in the invariant, then it specifies attributes of entities on the same side of a schema, the calculated value of which must match for the *inter_class* to be used to perform the mapping. If the equality is specified in an equivalence, then it denotes that the attributes of the respective entities must hold the value determined by the re-arrangement of the equation to solve for each particular attribute. This specification corresponds to the Attr->Attr mapping of cardinalities 0:1, 1:1, 1:C, C:1, C:B. Where there are more than two attributes in an equation (i.e., cardinalities 1:C, C:1, C:B) it is assumed the implemented mapping system will re-arrange equations to solve for unknown values, or arrange sets of equations to ensure that values calculable in other equations can be used to solve more complex equations, the method for achieving this in an implementation of VML is described in Chapter 9.

5.2.6.5 Pointer references

Following pointer chains to reference attributes of the referenced object is possible with the => operator in VML, as shown below. Pointer chains can be followed to any depth and provide a method for collapsing, or expanding, reference structures in a schema.

```
apex1=>x = apex2=>x
```

Pointer references have the same semantics wherever they are used. When solving equations which contain pointer references, it is sometimes necessary to create objects of the referenced type to have a full reference for solving the equation. For example, in the equation above, if *apex1* had no value it would be necessary to create an object of type *point*, the reference to which would become the value of *apex1*, so the attribute *x* could be set to the value of *apex2=>x*.

5.2.6.6 Functions

Functions in VML are the built-in functions of LPA Prolog, which are predicates which either fail or succeed along with the special functions *exists/1* and *var/1* which are used to determine whether an attribute has a value specified or if it has not been assigned to (see the examples below).

```
member(fe_face_material, fe_face_window.materials)
exists(end_point=>z)
var(end_point=>z)
```

Functions are used in the invariants section of an *inter_class* definition to determine whether to use the particular *inter_class* with the objects currently under consideration.

5.2.6.7 Aggregation functions

In contrast to the functions described above, aggregation functions are not invocable in both directions, or re-writable as most equations are. The set of aggregation functions (sum, maximum, minimum, average, count) provide summary details of lists of values or objects, as shown in the examples below.

sum(wall.windows=>(height * width)) = glazing_area

maximum(panes=>(offset=>y + height)) - minimum(panes=>offset=>y) = height

Aggregation functions can not be solved in both directions (e.g., for the first example, knowing the *glazing_area* does not allow the calculation of *height* and *width* of all the *windows* in a *wall*). Instead, when performing a mapping to an equation with an aggregation function, the calculated value from one side of the equation is used to specify a constraint on the values of the schema the mapping is being applied to. For instance, in the first example above, if the *glazing_area* is known, then the sum of *height * width* for all *windows* in the *wall* will be constrained to the value of *glazing_area*. Aggregation functions can be used in any part of an *inter_class* definition. The implementation of these constraints is assumed to be handled by the implemented mapping system, Chapter 10 details how this was achieved for one VML implementation.

5.2.6.8 List and array references

Individual elements of a list or an array may be referenced through the indexing operator. The examples below show equations which reference elements of a list and an element of a 3-dimensional array.

axes[2] = v_ref

exists(axes[2]=>direction_ratios[3])

vector[2, 2, 3] = iz

List and array references can be used in any part of an *inter_class* definition.

5.2.6.9 List and array iteration

A mapping between lists or arrays of the same dimensions and bounds can be specified through the iterator operator. This operator provides a short-cut notation for the specification of for-loops over the contents of lists and arrays, as long as the relationship can be specified in an equation. Mappings between lists and arrays which do not meet these conditions must be specified with predicates, procedures or methods. The example below shows a mapping between a list of strings (*classified_by*) and a list of object references (*material*) which has an attribute, *name*, of type *string*. The result of a mapping between these two attributes will be that the first item in the *classified_by* list is identical to the value of *name* of the first object reference in *material*, and so on for the number of items in *classified_by*, or in *material*, depending upon which direction the mapping is being run.

classified_by[] = material[].name

Iterators can appear in invariants, equivalences and initialisers and have slightly different semantics in each. When specified in an initialiser, an iterator provides the initial calculated values for an attribute of *list*, *set*, *bag*, or *array* type (within the specified bounds of the aggregate type), which may be modified by other equations in the equivalences section. If the equation is specified in the invariant then it specifies a condition which must hold for every value in the aggregate attribute referenced for the *inter_class* to be used to perform the mapping. If the equality is specified in an equivalence, it denotes that each element of the iterated attributes of the respective entities must hold the value determined by the re-arrangement of the equation to solve for each particular attribute. This specification corresponds to Attr->Attr mapping of cardinalities 1:1, 1:C, C:1, C:B. It can also denote Attr->Entity mapping of cardinalities 1:C, C:B, 1:N, or Entity-Attr mapping of cardinalities C:1, C:B, N:1, or Entity->Entity mapping of cardinality 1:1.

5.2.6.10 Conditional list and array iteration

VML provides bijections to extend the flexibility of the iterator operator. Bijections allow the specification of conditions to be checked on each element of an aggregate attribute, or grouped set of objects (formed by a *group()* specification in a class list), before a mapping can take place. The examples below show the bijection between lists of object references where the mapping is conditional on the class type of each object in the list. In the example below the right hand schema entity has attributes: *spaces*, which is a collection object containing a list of references to space objects; *roofs*, which is a collection object containing a list of references to roof objects; and *faces,* which is a list of references to face objects, which specify face geometry. Entity *idm_building* from the left hand schema has attributes: *spaces*, which is a list of references to abstract spaces including roofs and spaces; and *face_views*, which is a list of references to different views associated with faces and which can include geometry-oriented views, materials-oriented views, and bracing-oriented views. There is thus a partial overlap between the sets of objects referenced by the attributes involved in each of the two classes. The first bijection specifies that only *idm_building* spaces which are really living spaces (i.e., their type is *idm_space*) should be mapped to the *spaces=>list*, though all spaces in *spaces=>list* can be mapped to the *idm_building* spaces aggregate attribute. The second bijection specifies the same conditions, except this time for roof objects, and the third bijection performs a similar mapping for geometric faces. The @ operator specifies a method call in the example bijections, in this case to the meta-method *class()* which returns the class of the referenced object. The first bijection therefore iterates through all objects in the *spaces* list, extracting those whose class is of type *idm_space*.

bijection(idm_building.spaces[]@class('idm_space'), spaces=>list[]),

bijection(idm_building.spaces[]@class('idm_roof'), roofs=>list[]),

bijection(idm_building.face_views[]@class('idm_space_face'), faces[])

Bijections may only appear in the equivalences section of an *inter_class* definition. Bijections can be run in both directions and are taken to mean: iterate over all elements in the specified attribute or group of objects, but perform a mapping only for those elements which match the conditions

which are specified. Conditions can be in the form of method calls (as in the example above), predicate calls, or conditional equations. As multiple bijections can be specified over a single attribute (e.g., *idm_building.spaces[]* in the examples above), the result of the conditional evaluation and the equation solving are unioned with values from the other bijections on the list or array being mapped to. In the example above this means that in mapping from the right-hand side to the left-hand side the *idm_building.spaces[]* will contain the union of *spaces* and *roofs* from the right-hand side schema list attributes being mapped from. This specification corresponds to Attr->Attr mapping of cardinalities 1:1, 1:C, C:1, C:B. It can also denote Attr->Entity mapping of cardinalities 1:C, C:B, 1:N, or Entity-Attr mapping of cardinalities C:1, C:B, N:1, or Entity->Entity mapping of cardinalities 1:1, 1:C, C:1, C:B, N:M.

5.2.6.11 Functions

User defined functions extend the set of built-in functions of VML. User defined functions are broken into two categories: those which reference attributes from a single schema; and those which reference attributes from both schemas (examples of the latter are shown below, as detailed in Appendix E).

```
list_splitter(vals, _temp_schedule.splitvals)
map_polar_rect_to_polygon(min=>x, min=>y, max=>x, max=>y, plane=>axis,
                          plane=>offset, points[1], points[2], points[3], points[4])
```

User defined functions which reference attributes from one schema are used in the invariants and initialisers sections of the *inter_class* definition. User defined functions which reference attributes from both schemas must be invocable with the attributes from either side instantiated (i.e., able to run in either direction). When used, these functions are called with the values from the schema the mapping is coming from, and the results are used to instantiate the attributes of the schema the mapping is being applied to. A function is not invoked unless all function attributes of the schema the mapping is coming from have values. In general, functions may not manipulate objects (e.g., create new objects, delete objects, reference object attributes, call object methods), but allow structures to be manipulated and values to be computed. Functions correspond to the Attr->Attr mapping of cardinality 0:1, 1:1, 1:C, C:1, C:B.

5.2.6.12 Procedures

Where a mapping can not be specified bidirectionally using equations or functions, it is necessary to describe the mapping procedurally. In VML the *map_to_from* predicate is used to denote two procedures which perform a mapping. In a *map_to_from* definition there is a procedure for mapping in each direction (if the mapping is one-way, i.e., *read_only*, then one of the procedures is not necessary and may be replaced with the function *true*). Dependent upon the direction in which a mapping is applied, the appropriate procedure will be invoked to perform its mapping. Procedures assume full control over how to perform a mapping, and as such may create objects, reference attributes, delete objects, etc. An example of the use of procedures is shown below.

map_to_from(map_3D_rect_to_polar(x, y, z, x1, y1, z1, min, max, plane),
map_polar_to_3D_rect(min, max, plane, pe_wall))

Where *map_3D_rect_to_polar()* and *map_polar_to_3D_rect()* are Prolog predicates that map from a 3D rectangular representation (given by two points in 3D space) to a polar representation, or vice versa. As procedures may need to manipulate objects in the stores they may require information about the current status of the mappings in the system and the store managers handling the objects in the system. To allow this information to be accessed in a procedure, a special parameter may be passed to the procedure. This parameter *$mapping_system$* is replaced with the object ID of the mapping system controller when a procedure is invoked. The three most utilised services offered by the mapping system are: return of the type of mapping being handled (currently just transaction-based or interactive); return of which pass through the mappings is being executed (for implemented systems where multiple passes are made during the mapping); and return of the object ID of an object that was created in a mapping based on another object ID (see describing pointer equality above). As procedures have full control over what they do in the system, they can perform mappings between any combination of Attr and Entity in the system, and can describe mappings between any cardinality of these Attr and Entity maps.

5.2.6.13 Method invocation

As VML is designed to work with OO environments as well as in interactive environments, it is necessary to handle OO method invocation. Methods are specified as an optional class or attribute definition, followed by the @ symbol and the name of the method, with parameters if they exist. Method handling in VML covers two cases: for classes which have creation methods with parameters, the initial parameters of the *create* call can be specified in the initialisers section of an *inter_class*; in a mapping where method invocation in one model can trigger a method invocation in the model being mapped to, the parameters of the methods to call can be specified. The examples below illustrate both these cases. The first example shows a case where whenever the *view_plane* method of an *idm_plane* object is called then the *create_view* method of the corresponding *fe_application* object in the mapped model should be called, or vice versa. The second example defines the parameters required in a create call for objects of type *fe_face_window*.

idm_plane@view_plane = fe_application@create_view(_, idm_plane.name)

fe_face_window@create(idm_building, idm_plane.name, idm_plane.axis, 0, '+', [])

Method parameters which return values not required for the mapping can be specified with a _ to indicate that the parameter should be ignored (as in *create_view* above). Methods are called if all parameters from the side being mapped from are known. This needs to be kept in mind when *create* methods are specified, as all the parameters from the side being mapped from must be guaranteed to be bound at the time that the object creation takes place. This also means that object references in a *create* method must be resolvable through pointer equality at the time of the object creation. For example, in the *create* method above *idm_building* will be replaced with the object ID created from the *inter_class* mapping for *idm_building*. The level of support for mapping of methods will be dependant upon the approach taken for the implementation, as methods can have

side-effects which are only valid if methods and data are mapped in a strict sequence.

5.2.6.14 Type conversion

Simple type conversions are implicit in a mapping definition. Casting of results of equations is not required in VML, as this is implicitly defined by the type of the attributes specified in the equation. Pointer equivalence handles the type conversion of object references, as well as lists and arrays of object references. Procedures are expected to correctly cast results calculated in the procedure when setting the value of an attribute. Mappings of attributes which have complex types are expected to explicitly define the type conversion as part of the mapping specification.

5.2.6.15 Unit conversion

Unit conversion is not supported implicitly in the VML language. Any unit conversion which must be performed between attributes must be modelled explicitly. While it would have been possible to assume an implicit unit conversion in a mapping there would have been some practical difficulties in the implementation. One difficulty is that implementing a system to determine the final unit of a complex equation is no simple task. Another problem is that predicate, procedure and method definitions provide no meta-information on the units of their outputs, so it would be impossible to check that the result of a predicate, procedure or method was in the correct units.

5.2.6.16 Temporary attributes

Temporary, or local, attributes can be defined in a VML mapping. These attributes have the same syntactic definition as a temporary entity, i.e., names prefixed either with an underscore symbol, e.g., _temp1, _temp2, or with a capital letter, e.g., Temp1, Temp2. Temporary attributes allow the specification of partial computations which may be used in further equations. To this extent temporary attributes can reduce the complexity of an equation definition and can be used to improve calculation performance in an implemented mapping system (as demonstrated in the example below).

```
WallTheta = tan_1(y_offset / x_offset),
WallDist = sqrt(x_offset * x_offset + y_offset * y_offset),
wall_x = WallDist * cos(WallTheta + azimuth),
wall_y = WallDist * sin(WallTheta + azimuth)
```

5.3 A Graphical Notation for VML

To satisfy the modelling notation requirements of Section 4.1.3, and to complement the textual notation of VML described in Sections 5.1 and 5.2, a graphical notation (VML-G) was developed. The graphical notation describes a subset of the full VML language and is aimed at high-level views of the mapping specification.

As a graphical language provides fast reading and comprehension of the textual equivalent, VML-G is likely to be of use in large integration projects where the schemas of the IDM and the design tools can be very large requiring hundreds of *inter_class* specifications to detail a full mapping. To manage a mapping specification of this size, the developers of integrated design systems will require many diagrams showing parts of a mapping, from high-level design views during the initial specification phase (which can identify entities which must be mapped between), through to more detailed descriptions of the mapping between attributes heading towards the implementation stage. Graphical specifications will also provide some benefit where single entities have a large number of attributes. In these cases the modeller may wish to consider subsets of the entities attributes when specifying a mapping, requiring multiple views of the mapping between entities.

5.3.1 Graphical icons of VML-G

In VML-G there is a single graphical icon type representing an *inter_class* definition (see middle icon in Figure 5.2). This icon has three sections corresponding to the three sections in an *inter_class* definition. These three sections allow invariants, equivalences and initialisers to be grouped into localised areas in the icon and provide a visual separation of these distinct functions. The other icon type defined in VML-G denotes an entity taking part in the mapping with the *inter_class* (see the left and right hand icons in Figure 5.2).

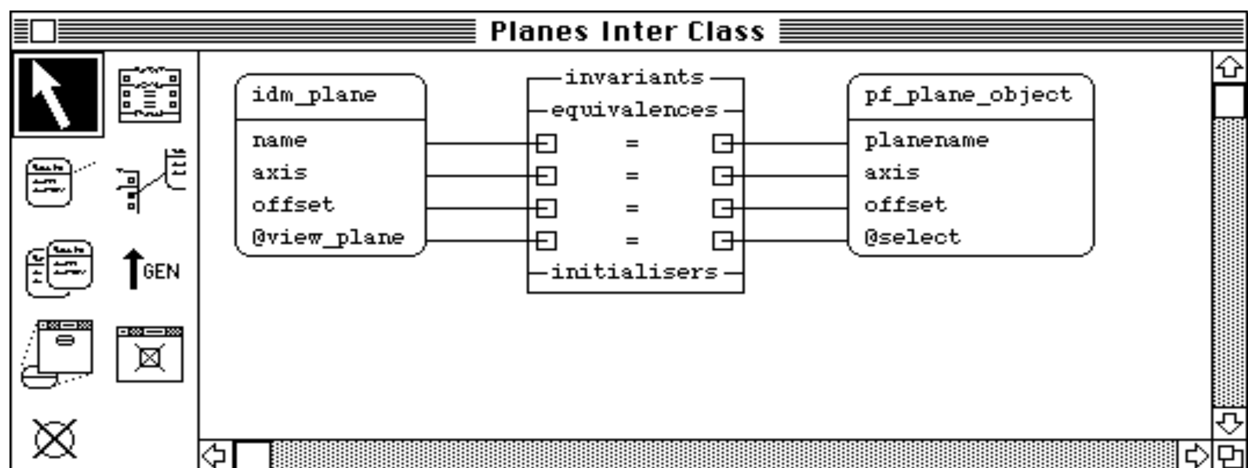


Figure 5.2 Graphical mapping specification in VPE

A graphical view of a mapping between entities from two schemas is specified by drawing an *inter_class* icon with icons for the entities on either side of it. By convention we put entity icons from the left hand schema on the left of the *inter_class* icon and the others on the right. This is not strictly necessary as the schema an entity belongs to can usually be ascertained quite simply. Entity icons specify the name of the entity represented (with optional schema and version information) and can list any attribute and method (prefixed with a '@') names defined in the entity (see Figure 5.2 for an example of direct attribute and method mappings). When there are multiple entities from a single schema participating in a mapping, the vertical order of the icons defines the ordering in the class list of the *inter_class* definition (Figure 5.3 shows the VML-G for such a mapping along with the textual equivalent). An entity which is to be grouped in the class list of the mapping

specification is drawn with a double line around the outside (like a stack of entities), also shown in Figure 5.3.

In Figure 5.2 we see that the icon for an *inter_class* definition consists of three parts. At the top is the invariants section which specifies conditions which must hold for this mapping to take place. Below that is the equivalences section which contains all mappings between attributes and entities. At the bottom is the initialisers section which holds the definition of initial values for attributes.

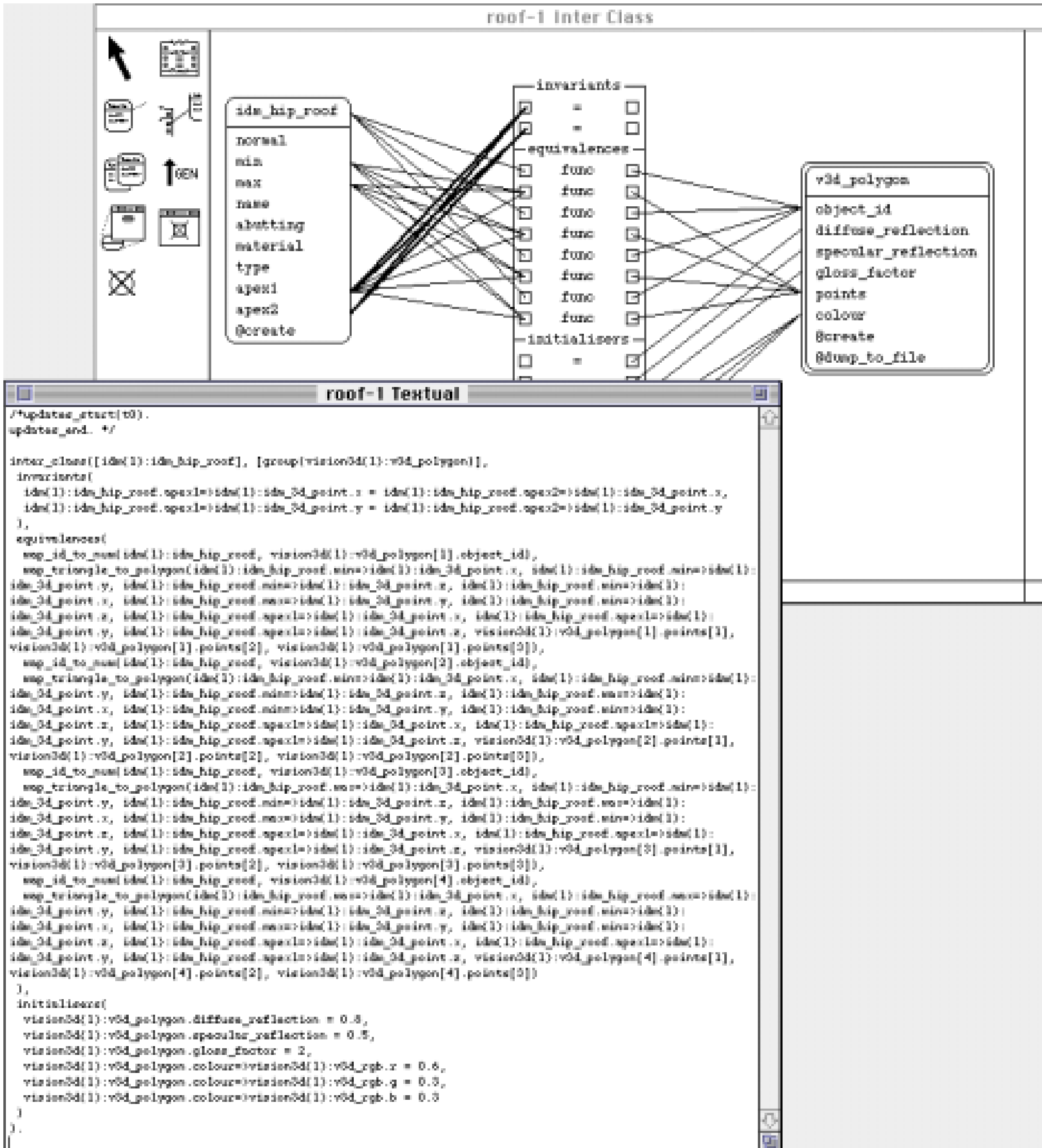


Figure 5.3 Complex VML-G specification with textual equivalent

Each individual equation, function, or procedure is given a single row in the *inter_class* icon. At each end of the row is a box to which the attributes and entities involved in the particular equation,

function, or procedure are connected. In the middle of the row there can be one of four symbols which are used to define the type of mapping being defined between attributes and entities. The four symbols are:

= denotes that there is a direct equivalence between the attribute (or entity) in one schema and the attribute (or entity) in the other. These one-to-one mappings are distinguished as a special case as they occur frequently in mapping specifications.

eqn denotes that the attributes or entities in one schema are related to attributes or entities in the other schema through an equation that is not a one-to-one equivalence.

func denotes that the attributes or entities wired together in this row are mapped through the use of a functional mapping specification.

proc denotes that procedural code is required to map between the specified attributes or entities. In the textual notation this has to be specified with different procedures to map in each direction.

5.4 Appraisal of VML

VML and VML-G define notations which are capable of modelling all known types of mappings between two schemas. They provide the ability to describe high-level views between entities in schemas, through to low level detailed views of mapping equations. They are supported by an environment (described in Chapter 6) which provides for multi-view graphical and textual specification, along with a global consistency mechanism. Mapping checkers and a generic mapping definition (also described in Chapter 6) provide a means to describe the required mappings between schemas for any implementation paradigm. In the remainder of this section, we review how these new notations meet the requirements laid out in Section 4.1, and what further work remains.

VML satisfies the requirements of Sections 4.1.1 and 4.1.2 in the following manner:

Structural mapping types: the class lists of an *inter_class* definition, along with invariant specifications, provide the means of describing any of the cardinalities of object to object mappings that can occur. Equivalence specifications provide a means of describing attribute to attribute mappings. In most cases, relationships between attributes and objects can also be defined in the equivalences section. The creation of objects during mapping initialisation (corresponding to 0:1 mappings) can be handled through an *inter_class* with one class defined in one class list and none in the other. Other types of new object or attribute creations are able to be specified through the initialisers section of the *inter_class* definition. VML has no explicit notion of object deletion or attribute value removal, therefore 1:0 mappings are not possible to encode explicitly. However, instead of explicit deletion, a notion of constrained creation is utilised. For example, if there is a condition under which an attribute should have no value then this would be specified in an invariant condition and

the following mapping would not map a value across.

Semantic mapping types: the majority of these conflicts can be resolved through explicit definition of mappings between attributes in schemas. For example, homonym and synonym conflicts from Table 4.3 can be resolved by specifying a mapping between the correct attributes. However, the recognition of these conflicts is the responsibility of the mapper and is not detected in the VPE support environment. The same is true for most of the schema and data conflicts of Table 4.4. Equivalences and initialisers provide the mechanism for resolving the conflicts; their recognition is the responsibility of the mapper. Structural conflicts of Table 4.3 are resolved through specification of classes in the class lists and the invariants which tie together objects of these classes only under specified conditions.

An appraisal of VML with regard to the more mapping-specific requirements of Section 4.1.3, yields the following:

Language level: the use of VML-G allows rapid and concise specification of which entities and attributes are related, without having to provide exact details of the connections. The declarative style of the VML notation allows for simple specification of exact relationships between entities and attributes, without having to detail specific methods to implement the relationships, and without having to rewrite the relationships to solve for all referenced attributes. In VML there is an underlying assumption that the mapping system will determine how to perform the mapping, and rearrange equations as required. However, it is recognised that a declarative approach can only be used for a certain set of correspondences, and in some cases a procedural specification will be necessary. VML provides for this as well. Some of the surveyed languages provide a more concise notation than VML for special cases (e.g., a Transformr 'COPY' to copy between identical objects during a version update). VML, however, provides a consistent level of specification for any type of mapping.

Language notation and modelling environment: VML is supported by a graphical notation (VML-G) to allow high-level views of mappings, as well as VML's lower level implementation views. The VPE environment demonstrates some of the power that can be gained from a multi-view graphical and textual specification environment with consistency maintained between all views. For example, select portions of a mapping can be described in different views, and entity icons can be expanded and contracted to show as much or as little of the entity interface as required.

Language style: the declarative style of VML ensures that it is easily translated into implementation environments of many paradigms. For example, translating VML into a procedural, batch operated mapping system would be possible by transforming VML mappings (re-arranged to solve for the required attributes) to procedural statements. This approach has been explored at BRANZ (Price 1995) to tie together several in-house design and analysis tools. As an example of the other implementation paradigm extreme, an interpreted on-line

implementation of VML mappings is illustrated in Chapter 10.

Bidirectional: all VML mappings are implicitly bidirectional, though this can be overridden in the *inter_view* definition. Any VML mapping is specified with the understanding that the implementation system will use the mapping specification, re-arranged as necessary to solve mappings, dependent on the direction in which data is mapped.

Conditional mapping: the invariants section of a VML mapping makes explicit the conditions under which a particular mapping can be utilised. These conditions are specified independently of the mapping definition.

Aggregation: functions to perform aggregation are included in the VML language. It is assumed that the implemented mapping system will determine which aggregate equations can be solved in both directions, and handle the consistency problems of those that can not.

Relationship handling: VML provides an operator to navigate the various structures found in different schemas. Through this operator, the compression of complex structures and the telescoping of simple structures can be defined.

Initialisers: the initialisers section of a VML mapping allows for the specification of initial values for entities and attributes, as well as specifying parameters for the creation of objects in an object-oriented environment. VML also allows the specification of an *inter_class* mapping with only one entity in one of the class lists as a way of defining objects which must be created at the start of a mapping process.

Unit handling: VML provides no implicit mapping for attributes of different units. Any unit conversions required between attributes must be identified by the mapper, and the conversion detailed as part of the mapping. This decision was made as a result of the difficulty of determining the resultant unit of the re-arrangement of any arbitrary equation, and the lack of unit information in the specification of functions, procedures and methods which can have serious flow-on effects (e.g., if a method returns a result used in a later calculation).

Type handling: simple type conversion is implicit in VML, so an implementation mapping system is required to implement type conversion for all simple types (e.g., float to integer). For simple types, the required conversion can be ascertained from the schema definition. Conversion of complex types must be detailed as part of the mapping.

VML provides the fundamental mapping representational requirement for schemas that need to share information with an IDM in an integrated design system. However, there are some aspects of the VML language which require further work:

Mapping language requirements: the list of mapping types around which the mapping language analysis is based is not known to be complete. The semantic mapping types are drawn from a slightly different domain and the mapping language requirements are drawn mainly from experience with mappings over the course of this project. For example, the requirement for bijections did not become clear until the large mapping problem illustrated in Appendix E was attempted. Early work on this thesis attempted to determine mapping

types from analysis and classification of known mapping problems based around the types specified in Section 4.1. However, further analysis of combinations of these types to determine high-level mapping types foundered in the combinatorial explosion of low-level combinations to be analysed. There appears to be no definitive classification of requirements in a mapping and further work looking at requirements would provide a sounder basis to compare mapping languages and evaluate their descriptive ability.

Greater control specification in mappings: the level of control specification in VML is too general. The *inter_view* specification is the only place where the level of mapping between schemas is defined. In the case where there is a *partial* mapping between the schemas there is no methodology available to determine how to calculate which part of which schema should be mapped. The underlying assumption is that a selection tool will be invoked at the start of the mapping, in order to select the subset of the model which will be mapped. It should be possible to specify the constraints on partial mappings in a more formal manner. There is also no control on individual *inter_class* specifications, so one-way mappings can not be explicitly defined on an *inter_class* by *inter_class* basis. However, this is not a problem for design tools (e.g., an IDM mapping to the output values of a design tool), as the project specification of Chapter 7 defines the input and output subschemas for all design tools in terms of the IDM's schema.

Greater micro-level control in an *inter_class* definition: Though the invariants section provides the conditions under which a mapping may take place, there are mappings which are almost identical, but which require separate *inter_class* definitions because a single equation or method call needs to be different. The main example of this problem is with create methods for objects, in these calls, where all parameters must be instantiated, it is necessary to supply a default value if an attribute is uninstantiated. However, with VML this requires two separate *inter_class* definitions, one which checks that the attribute is instantiated, and the other which checks that it isn't. Simple conditional wrappers around equations would provide a solution to most of these problems.

Generic function and procedure definition: though the majority of a VML specification is generic, and able to be mapped to almost any language paradigm, function and procedure definitions are not. Currently, functions and procedures are defined in Prolog and Snart, as these are the underlying languages in which the existing mapping implementation is written. It is unlikely that a totally generic procedure and function definition language could be defined, but work on the Neutral Model Format (Sahlin et al. 1995) and the Java language (Gosling and McGilton 1995) would suggest that some degree of generality could be supported.

Unit converter: VML makes the assumption that there is no equation re-arranger available that can calculate the final unit of any arbitrarily re-arranged equation. This is mainly due to the lack of unit information available from function, method and procedure definitions, which may calculate values for attributes used in later equations. Providing a notation for the specification of units on all parameters of functions, methods and procedures, and hence to

the design of a unit calculator for arbitrary equations, would prove an interesting challenge.

Explicit specification of type conversion: it would be useful to be able to define a mapping between complex types. This type mapping could be assumed to be used automatically, in the same way that simple types are assumed to be automatically converted. This functionality is available in mapping languages such as EXPRESS-M, and syntactic changes to VML to support the definition of type mappings would be trivial.

Method mapping: the VML specification of method invocation is limited to one-to-one correspondences. There is currently no way of specifying combinations of method invocations due to their temporal nature. For example, the set of method calls required to invoke a method call in the mapped to schema could be spread over several transactions. While data values are constant over multiple transactions, methods are only seen at the time that they occur. A more sophisticated conditional language specification would be required to fully model methods allowing combinations of several method calls to be required to invoke a mapping and also to enable mappings based on the parameters used when invoking the method. This would also impact on the type of system required to handle mappings between schemas as the whole history of the use of the particular schema would have to be available to be scrutinised.

In summary, this chapter introduces VML, a language that can be used to describe bidirectional mappings between two schemas. The language is shown to provide the functionality to meet the requirements for a mapping language. An environment which supports the specification of VML mappings is described in Chapter 6. The specification of mappings is the penultimate step in the specification of an integrated design system for a particular project. The final step is to model the project specification, which utilises the schemas for design functions from Chapter 3, and assumes that there is a mapping between the schemas for each design function and the IDM for the integrated design system. Chapter 7 introduces a formalism for project specification which details the flow of control between design functions to build upon the mappings to the IDM specified in this chapter.

Chapter 6

Mapping Modelling and Development

Chapter 5 specifies a mapping language and the type of mapping problems in which it will be used. It is clear that these problems will involve large schemas (each with several hundred class definitions), which are not static, and where several modellers will be working on the problem at any one time. To minimise the amount of work required to develop and maintain these mappings it is clear that coordinated creation, management and documentation of mapping specifications needs to be supported. A mapping specification environment supporting the textual and graphical notations of VML is described in this chapter. This environment provides the user with multiple, consistently maintained, graphical and textual views of a mapping.

6.1 Introduction

The problems associated with the development of mapping definitions are of a similar nature to those of schema development, as described in Section 3.1. The current version of a mapping must be propagated to all developers so that they are all considering the same mapping. Modifications that are made from one version of a mapping to the next need to be documented so that differences between versions are easy to identify (especially for very large mappings). Changes required to a mapping, resulting from modifications to a referenced schema, need to be notified to the developers and tracked to ensure that their effect on the mapping is properly taken into account. Although VML-G provides a notation to describe mappings at a high-level, a paper-based definition of a mapping using VML-G will prove difficult to maintain. Although a mapping may be easily drawn, a paper-based system cannot detect errors in a mapping specification, nor can it guarantee that entities and attributes drawn on the paper exist in the schemas they are meant to be from, and when a large number of mappings are described, navigating through the mapping views

(especially if there are multiple, partial, and overlapping views) will become cumbersome. Maintaining the consistency of the drawn views under changes to the schemas will also prove a difficult task. A computerised mapping modelling and development environment, similar to the environment described in Chapter 3, can provide many of the solutions to these problems.

Currently, almost no tools exist to help in the definition of mappings using the languages surveyed in Chapter 4. Developers use a text editor to develop the specification and it is then parsed and checked against the referenced schemas. Errors in the mapping specification terminate the parsing process and require modifications to the textual representation and a reparsing. No tool allows a consistently maintained mapping to be specified by multiple developers, or to be notified of modifications to referenced schemas. To solve these problems, the author has designed an integrated modelling and development environment for VML which provides many functions to users in a homogenous environment.

In this chapter the requirements for a modelling environment in a large modelling project are introduced. The VML Programming Environment (VPE), which tackles these requirements, is detailed, and its ability to meet the design requirements of a modelling environment is demonstrated.

6.1.1 Requirements for mapping development

In any large scale multi-partner mapping development a number of modelling support issues need to be dealt with, including the following:

Connection with the schema development process: as the mapping is defined wholly between two schemas it is imperative that any changes to referenced schemas be notified to the mapping developers. If schema development environments, such as EPE in Chapter 3, are utilised then the tracked updates to the schema must also be accessible from the mapping environment. In many cases changes made to a schema can be automatically applied to a defined mapping (e.g., renaming of an attribute in a schema) and the mapping environment should offer this time saving service to its users.

Documentation: during the mapping specification process many decisions are made about the meaning of schema constructs and implied constraints existing in a schema. These decisions will determine the mappings which are specified as well as compromises which may be made in the mapping specification. All such decisions, and their justifications, must be permanently documented as they are made.

Support for multiple views: the mapping specifiers may well be experts on particular aspects of the schemas being mapped between, and hence will only want to define mappings for these aspects. When dealing with large schemas with large inheritance hierarchies the lower level class definitions can hold a wide range of attributes and references spanning many aspects of the object being defined. Mapping specifiers may well wish to consider specific aspects

of a class definition when specifying mappings, requiring several partial mapping specifications for a single class mapping. These multiple partial views must be synchronised and coordinated by the mapping development environment.

To support these requirements an ideal modelling support environment (MSE) needs mechanisms for: easy communication of mapping definitions; generation and manipulation of multiple views of a mapping; annotation of the mapping with documentation; links to external schema MSEs for change notification; and flexible update management to support iterative updates of the mapping definition. An important issue is the ability to rapidly prototype a resulting operational system. MSEs should provide facilities to allow instantiation of mapping definitions to be tested in a run time environment, e.g., to test against models developed for the schemas being mapped between.

6.2 Mapping Development in VPE

The VML Programming Environment (VPE) has been engineered to support the modelling requirements detailed above for mapping specification and management. VPE provides multiple graphical and textual views of varying degrees of complexity useful at different stages of the mapping definition. In the initial phases of defining a mapping users typically specify just the classes which must be mapped between in a graphical manner. Further specification identifies conditional characteristics of mappings for these classes, which are expressed graphically as multiple inter-class mappings with invariants. Specifying the full mapping requires all equivalences and initialisers to be entered in textual form, though likely utilising multiple views of the single mapping specification. Throughout this development modifications may be made to the referenced schemas which will need to be propagated to the developing mappings and actioned appropriately. VPE provides integrated support for each of these activities, using the MViews consistency mechanism (Grundy 1993) to provide the required inter-view consistency. Because of the common implementation framework VPE (deliberately) has much of the “look and feel” of EPE.

6.2.1 Functionality offered by the VPE environment

In the following, a description is provided of VPE’s functionality, illustrated by the VML specification of part of the large mapping example described in Section 1.6, and fully specified in Appendix E.

6.2.1.1 Initial connection

Figure 6.1 shows the type of graphical view commonly used when first connecting together two schema definitions. Here the different sets of classes which have relationships to each other are identified and clustered together. Though it is not shown in this example it is also likely that inheritance between mappings would be specified in this level of view. This view shows all the different sets of classes considered in mapping between the IDM schema and the VISION-3D

design tool. This view is constructed by direct manipulation, as are all the graphical views, using tools selected from a tool palette, shown to the left of the view.

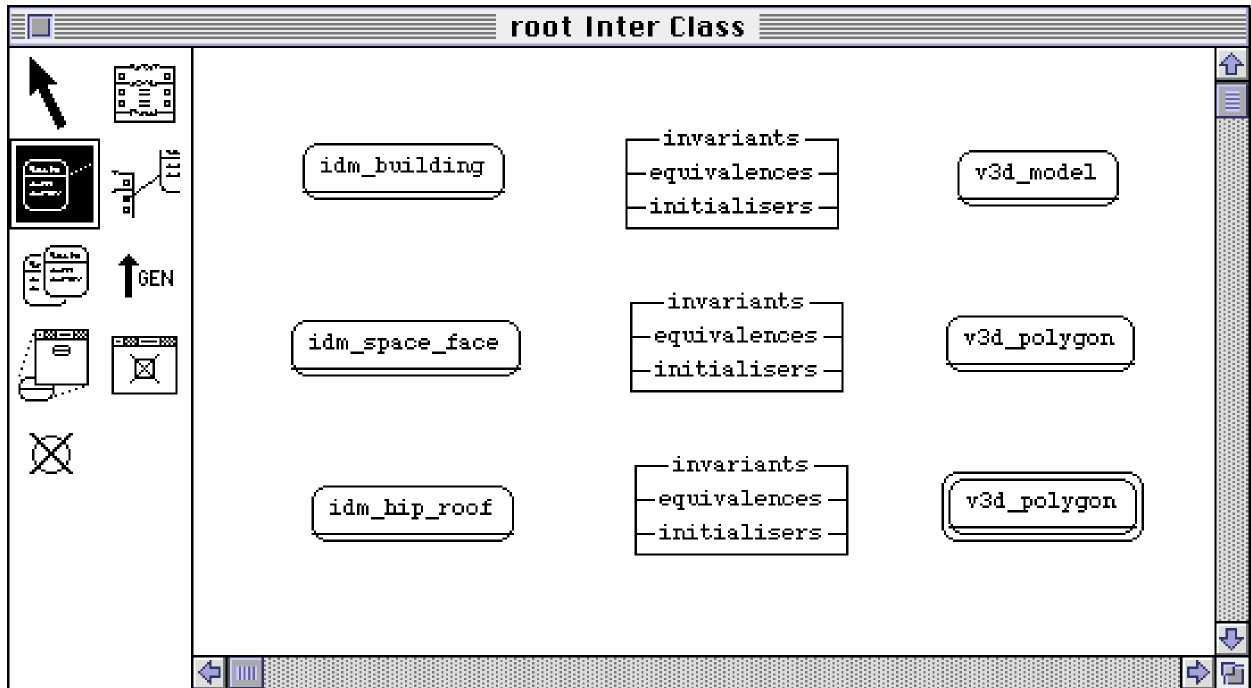


Figure 6.1 Initial connections between classes in two schemas

Though only a single view is shown here the user is free to create as many views as is desired, and may freely lay out and populate each view, either with new information or with information entered into other views. The information in each view is mapped through to the canonical representation of the mapping as the view data is entered, and any similarities or conflicts with the existing data are resolved as it is created. This ability to construct multiple views permits both general purpose and specialised views to be constructed. The former may be used to obtain an overview of the mapping under construction, the latter to focus on more detailed parts of the mapping specification. The proliferation of views means that navigation tools are needed to quickly access desired information. VPE provides inter-view navigation using both menu-based search facilities and automatically constructed hypertext links.

6.2.1.2 Multiple mappings for class sets

A second type of view, shown in Figure 6.2, is typically used to commence the specification of individual mappings between the same set of classes. The individual mappings are distinguished by the different invariants required. In Figure 6.2 we see three *inter_class* definitions between the *idm_hip_roof* class and the *v3d_polygon* class, with the distinguishing feature being the line of the roof (represented by two apex points for each roof segment). Either the roof segment is a pyramid, or the roof line runs in the x direction or the y direction, with different mappings required for each case. As in the initial connection view, all information is kept consistent with the canonical representation of the mapping and hence with other dependent views.

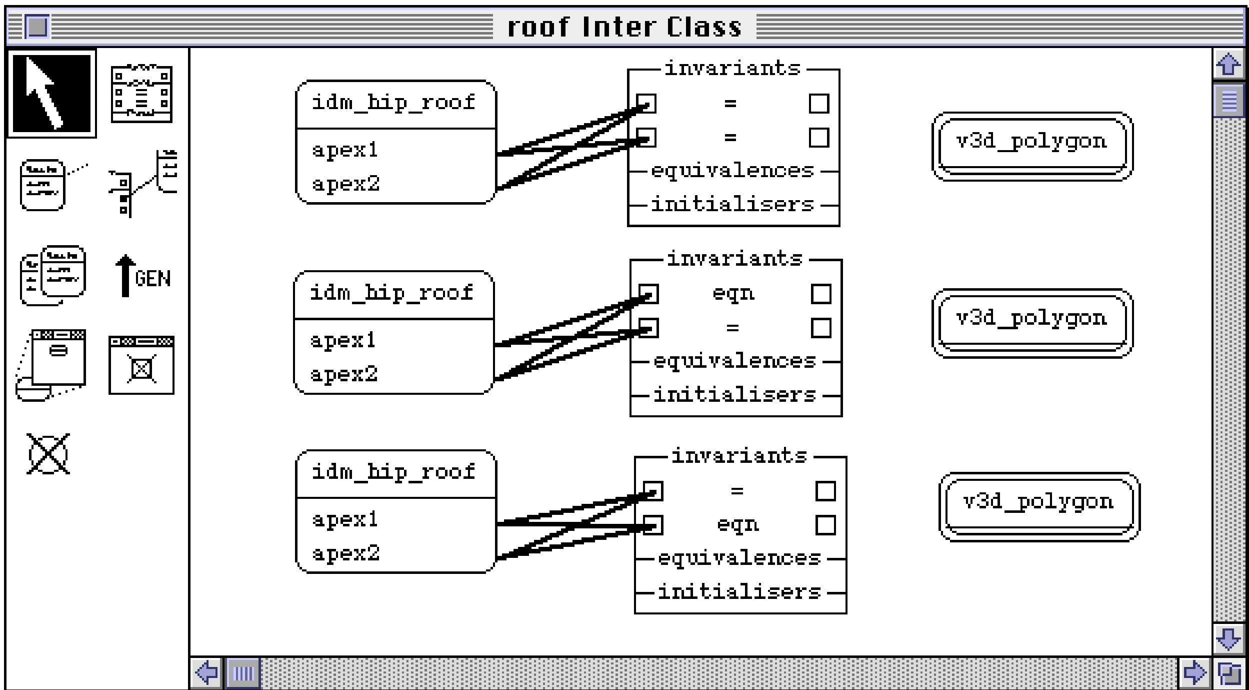


Figure 6.2 Specifying multiple mappings for a single class set

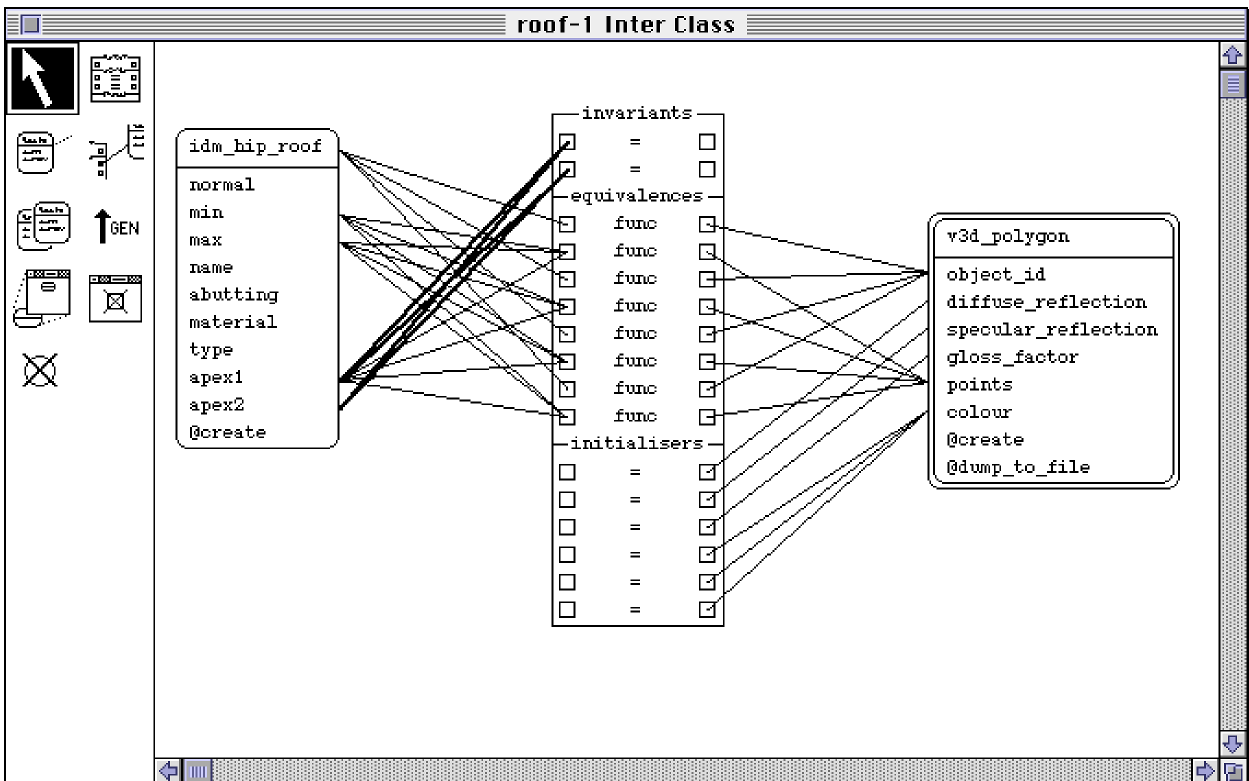


Figure 6.3 Defining links between all features associated with an *inter_class* definition

6.2.1.3 Linking of features in a class set

Based on the views of Section 6.2.1.2 the mapping specifier may proceed to define the features involved in a single mapping. Figure 6.3 shows how features of classes involved in a single *inter_class* are "wired together". This type of graphical view permits a rapid check that everything is connected and a simple overview of the features involved in each mapping without a full

specification of the equations and functions involved. This view type is also useful for providing partial representations of an *inter_class* definition, focussing on a particular aspect of the mapping (e.g., all simple attribute mappings, or the initialisers). As in the previous two views, all the information in this type of view is kept consistent with the canonical representation of the mapping and hence with other dependent views.

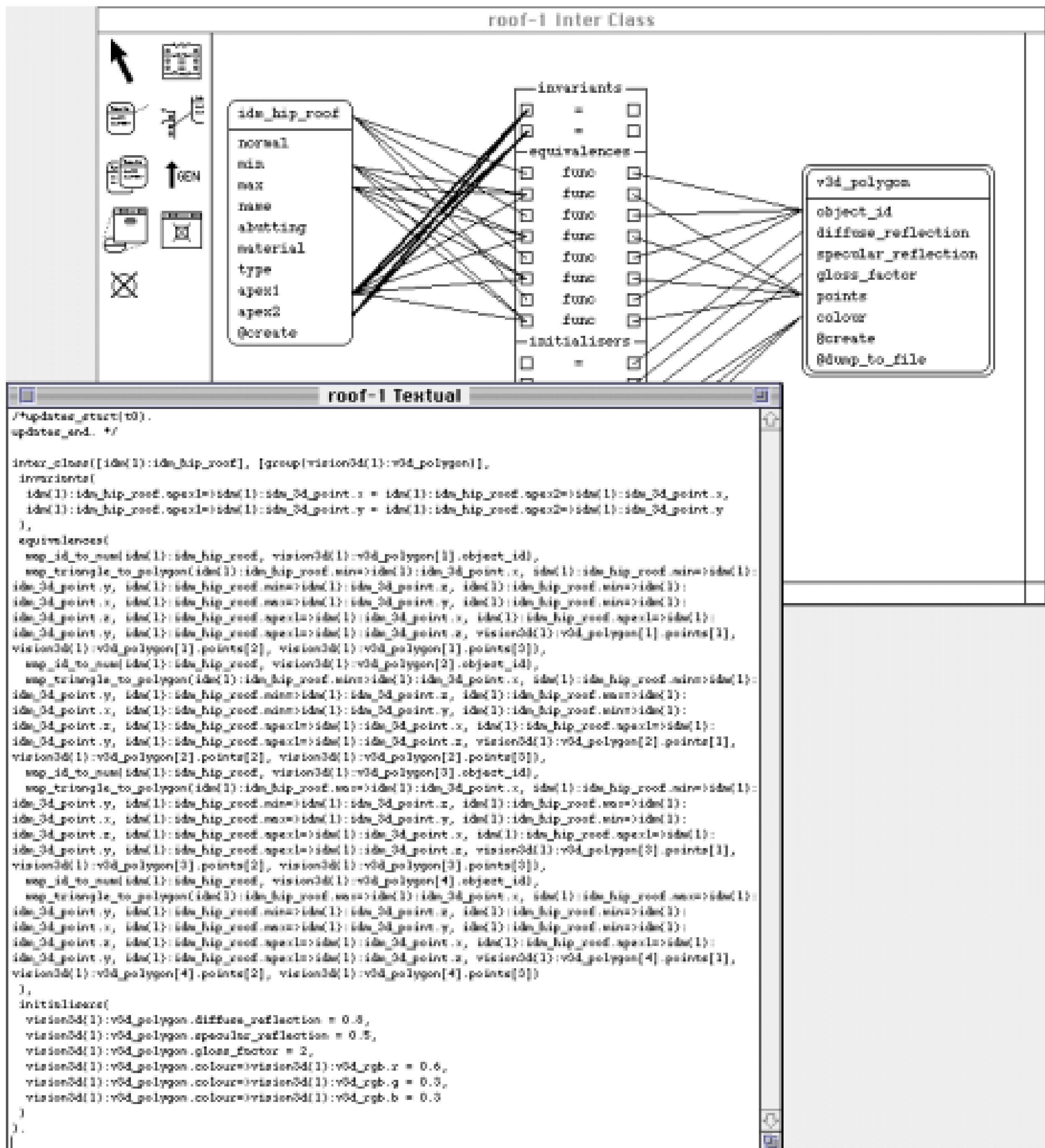


Figure 6.4 A full textual specification of an *inter_class* definition

6.2.1.4 Full mapping specification

The full specification of a mapping necessitates a textual representation of the mapping using VML, as VML-G lacks the full representational power of VML. Figure 6.4 shows a fully detailed VML specification for the graphical representation shown in Figure 6.3. This completely defines

the mapping between these sets of classes, but takes much more effort to read and to determine which features are involved in the various functions and equations than does the graphical form. The textual views are freely editable. Modified textual views are parsed and compiled to ensure they represent valid VML descriptions, and their information kept consistent with the canonical representation.

6.2.1.5 Consistency between views

When changes are made to a VPE view, other views that share information with the updated view may become inconsistent and must be updated to keep the mapping consistent across all views. All views affected by a change are notified and, in many cases, are automatically modified to reflect the change. This consistency mechanism works both between views of one phase of development and between views of different development phases.

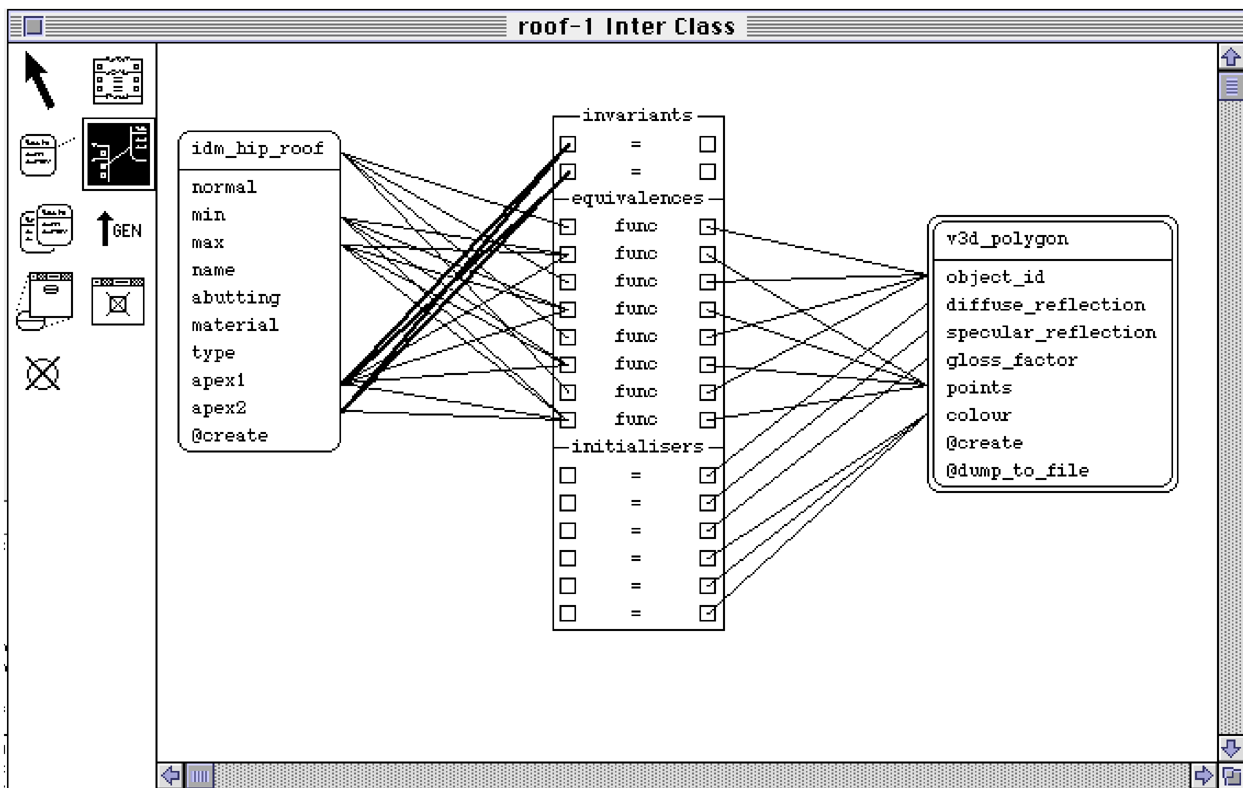


Figure 6.5 Modification of a graphical view

As an example, Figure 6.5 shows a modification being made to the graphical view shown in Figure 6.3, where the number of features associated with a single equivalence is being increased (adding *apex2*, from *idm_hip_roof*, to the last equivalence), allowing a new function to be used to calculate the equivalence between the two forms of roof specification. This change is propagated through to the canonical form of the schema which is updated, then all dependent views are identified and notified of the change which has been made. VPE propagates changes to other affected views in the form of an MViews *update_record* (described in detail in Section 3.2.2). This record provides a complete description of any single change. How views react upon receipt of an *update_record* depends on both the view type and the nature of the change.

roof-1 Textual

```
/*updates_start(t0).
update(23). % add feature idm(1):idm_hip_roof.apex2 to map_triangle_to_polygon(idm(1):idm_hip_roof.max=>
idm(1):idm_3d_point.x, idm(1):idm_hip_roof.max=>idm(1):idm_3d_point.y, idm(1):idm_hip_roof.min=>idm(1):
idm_3d_point.z, idm(1):idm_hip_roof.min=>idm(1):idm_3d_point.x, idm(1):idm_hip_roof.max=>idm(1):
idm_3d_point.y, idm(1):idm_hip_roof.min=>idm(1):idm_3d_point.z, idm(1):idm_hip_roof.apex1=>idm(1):
idm_3d_point.x, idm(1):idm_hip_roof.apex1=>idm(1):idm_3d_point.y, idm(1):idm_hip_roof.apex1=>idm(1):
idm_3d_point.z, vision3d(1):v3d_polygon[4].points[1], vision3d(1):v3d_polygon[4].points[2], vision3d(1):
v3d_polygon[4].points[3])
updates_end. */

inter_class([idm(1):idm_hip_roof], [group(vision3d(1):v3d_polygon)],
 invariants(
  idm(1):idm_hip_roof.apex1=>idm(1):idm_3d_point.x = idm(1):idm_hip_roof.apex2=>idm(1):idm_3d_point.x,
  idm(1):idm_hip_roof.apex1=>idm(1):idm_3d_point.y = idm(1):idm_hip_roof.apex2=>idm(1):idm_3d_point.y
 ),
 equivalences(
  map_id_to_num(idm(1):idm_hip_roof, vision3d(1):v3d_polygon[1].object_id),
  map_triangle_to_polygon(idm(1):idm_hip_roof.min=>idm(1):idm_3d_point.x, idm(1):idm_hip_roof.min=>idm(1):
idm_3d_point.y, idm(1):idm_hip_roof.min=>idm(1):idm_3d_point.z, idm(1):idm_hip_roof.min=>idm(1):
idm_3d_point.x, idm(1):idm_hip_roof.max=>idm(1):idm_3d_point.y, idm(1):idm_hip_roof.min=>idm(1):
idm_3d_point.z, idm(1):idm_hip_roof.apex1=>idm(1):idm_3d_point.x, idm(1):idm_hip_roof.apex1=>idm(1):
idm_3d_point.y, idm(1):idm_hip_roof.apex1=>idm(1):idm_3d_point.z, vision3d(1):v3d_polygon[1].points[1],
vision3d(1):v3d_polygon[1].points[2], vision3d(1):v3d_polygon[1].points[3]),
  map_id_to_num(idm(1):idm_hip_roof, vision3d(1):v3d_polygon[2].object_id),
  map_triangle_to_polygon(idm(1):idm_hip_roof.min=>idm(1):idm_3d_point.x, idm(1):idm_hip_roof.min=>idm(1):
idm_3d_point.y, idm(1):idm_hip_roof.min=>idm(1):idm_3d_point.z, idm(1):idm_hip_roof.max=>idm(1):
idm_3d_point.x, idm(1):idm_hip_roof.min=>idm(1):idm_3d_point.y, idm(1):idm_hip_roof.min=>idm(1):
idm_3d_point.z, idm(1):idm_hip_roof.apex1=>idm(1):idm_3d_point.x, idm(1):idm_hip_roof.apex1=>idm(1):
idm_3d_point.y, idm(1):idm_hip_roof.apex1=>idm(1):idm_3d_point.z, vision3d(1):v3d_polygon[2].points[1],
vision3d(1):v3d_polygon[2].points[2], vision3d(1):v3d_polygon[2].points[3]),
  map_id_to_num(idm(1):idm_hip_roof, vision3d(1):v3d_polygon[3].object_id),
  map_triangle_to_polygon(idm(1):idm_hip_roof.max=>idm(1):idm_3d_point.x, idm(1):idm_hip_roof.min=>idm(1):
idm_3d_point.y, idm(1):idm_hip_roof.min=>idm(1):idm_3d_point.z, idm(1):idm_hip_roof.max=>idm(1):
idm_3d_point.x, idm(1):idm_hip_roof.max=>idm(1):idm_3d_point.y, idm(1):idm_hip_roof.min=>idm(1):
idm_3d_point.z, idm(1):idm_hip_roof.apex1=>idm(1):idm_3d_point.x, idm(1):idm_hip_roof.apex1=>idm(1):
idm_3d_point.y, idm(1):idm_hip_roof.apex1=>idm(1):idm_3d_point.z, vision3d(1):v3d_polygon[3].points[1],
vision3d(1):v3d_polygon[3].points[2], vision3d(1):v3d_polygon[3].points[3]),
  map_id_to_num(idm(1):idm_hip_roof, vision3d(1):v3d_polygon[4].object_id),
  map_triangle_to_polygon(idm(1):idm_hip_roof.max=>idm(1):idm_3d_point.x, idm(1):idm_hip_roof.max=>idm(1):
idm_3d_point.y, idm(1):idm_hip_roof.min=>idm(1):idm_3d_point.z, idm(1):idm_hip_roof.min=>idm(1):
idm_3d_point.x, idm(1):idm_hip_roof.max=>idm(1):idm_3d_point.y, idm(1):idm_hip_roof.min=>idm(1):
idm_3d_point.z, idm(1):idm_hip_roof.apex1=>idm(1):idm_3d_point.x, idm(1):idm_hip_roof.apex1=>idm(1):
idm_3d_point.y, idm(1):idm_hip_roof.apex1=>idm(1):idm_3d_point.z, vision3d(1):v3d_polygon[4].points[1],
vision3d(1):v3d_polygon[4].points[2], vision3d(1):v3d_polygon[4].points[3])
 ),
```

Figure 6.6 Receipt of an *update_record* in a textual view

In the VPE system, like EPE, *update_record*s propagated to textual views are not applied automatically, although many of them could be. Instead they are displayed in the view and the user has control over which updates are applied at which time. As can be seen in Figure 6.6, the graphical update to the *inter_class* definition generates an *update_record* in the equivalent textual view. Many *update_records* can be directly applied by the system on user request, although this particular update cannot as the system does not know where the new features should appear in the function's parameter list. In this case, the user is tasked with applying the required modification and informing VPE that it has been applied. Following this, the notification of the outstanding update is removed from the view leaving the modified textual view as shown in Figure 6.7 (where the last equivalence now references *apex2* for some parts of the function).

roof-1 Textual

```
/*updates_start(t0).
updates_end. */

inter_class([idm(1):idm_hip_roof], [group(vision3d(1):v3d_polygon)],
 invariants(
  idm(1):idm_hip_roof.apex1=>idm(1):idm_3d_point.x = idm(1):idm_hip_roof.apex2=>idm(1):idm_3d_point.x,
  idm(1):idm_hip_roof.apex1=>idm(1):idm_3d_point.y = idm(1):idm_hip_roof.apex2=>idm(1):idm_3d_point.y
 ),
 equivalences(
  map_id_to_num(idm(1):idm_hip_roof, vision3d(1):v3d_polygon[1].object_id),
  map_triangle_to_polygon(idm(1):idm_hip_roof.min=>idm(1):idm_3d_point.x, idm(1):idm_hip_roof.min=>idm(1):
idm_3d_point.y, idm(1):idm_hip_roof.min=>idm(1):idm_3d_point.z, idm(1):idm_hip_roof.min=>idm(1):
idm_3d_point.x, idm(1):idm_hip_roof.max=>idm(1):idm_3d_point.y, idm(1):idm_hip_roof.min=>idm(1):
idm_3d_point.z, idm(1):idm_hip_roof.apex1=>idm(1):idm_3d_point.x, idm(1):idm_hip_roof.apex1=>idm(1):
idm_3d_point.y, idm(1):idm_hip_roof.apex1=>idm(1):idm_3d_point.z, vision3d(1):v3d_polygon[1].points[1],
vision3d(1):v3d_polygon[1].points[2], vision3d(1):v3d_polygon[1].points[3]),
  map_id_to_num(idm(1):idm_hip_roof, vision3d(1):v3d_polygon[2].object_id),
  map_triangle_to_polygon(idm(1):idm_hip_roof.min=>idm(1):idm_3d_point.x, idm(1):idm_hip_roof.min=>idm(1):
idm_3d_point.y, idm(1):idm_hip_roof.min=>idm(1):idm_3d_point.z, idm(1):idm_hip_roof.max=>idm(1):
idm_3d_point.x, idm(1):idm_hip_roof.min=>idm(1):idm_3d_point.y, idm(1):idm_hip_roof.min=>idm(1):
idm_3d_point.z, idm(1):idm_hip_roof.apex1=>idm(1):idm_3d_point.x, idm(1):idm_hip_roof.apex1=>idm(1):
idm_3d_point.y, idm(1):idm_hip_roof.apex1=>idm(1):idm_3d_point.z, vision3d(1):v3d_polygon[2].points[1],
vision3d(1):v3d_polygon[2].points[2], vision3d(1):v3d_polygon[2].points[3]),
  map_id_to_num(idm(1):idm_hip_roof, vision3d(1):v3d_polygon[3].object_id),
  map_triangle_to_polygon(idm(1):idm_hip_roof.max=>idm(1):idm_3d_point.x, idm(1):idm_hip_roof.min=>idm(1):
idm_3d_point.y, idm(1):idm_hip_roof.min=>idm(1):idm_3d_point.z, idm(1):idm_hip_roof.max=>idm(1):
idm_3d_point.x, idm(1):idm_hip_roof.max=>idm(1):idm_3d_point.y, idm(1):idm_hip_roof.min=>idm(1):
idm_3d_point.z, idm(1):idm_hip_roof.apex1=>idm(1):idm_3d_point.x, idm(1):idm_hip_roof.apex1=>idm(1):
idm_3d_point.y, idm(1):idm_hip_roof.apex1=>idm(1):idm_3d_point.z, vision3d(1):v3d_polygon[3].points[1],
vision3d(1):v3d_polygon[3].points[2], vision3d(1):v3d_polygon[3].points[3]),
  map_id_to_num(idm(1):idm_hip_roof, vision3d(1):v3d_polygon[4].object_id),
  map_triangle_to_polygon(idm(1):idm_hip_roof.max=>idm(1):idm_3d_point.x, idm(1):idm_hip_roof.max=>idm(1):
idm_3d_point.y, idm(1):idm_hip_roof.min=>idm(1):idm_3d_point.z, idm(1):idm_hip_roof.min=>idm(1):
idm_3d_point.x, idm(1):idm_hip_roof.max=>idm(1):idm_3d_point.y, idm(1):idm_hip_roof.min=>idm(1):
idm_3d_point.z, idm(1):idm_hip_roof.apex2=>idm(1):idm_3d_point.x, idm(1):idm_hip_roof.apex2=>idm(1):
idm_3d_point.y, idm(1):idm_hip_roof.apex1=>idm(1):idm_3d_point.z, vision3d(1):v3d_polygon[4].points[1],
vision3d(1):v3d_polygon[4].points[2], vision3d(1):v3d_polygon[4].points[3])
 ),
 initialisers(
  vision3d(1):v3d_polygon.diffuse_reflection = 0.8,
  vision3d(1):v3d_polygon.specular_reflection = 0.5,
  vision3d(1):v3d_polygon.gloss_factor = 2,
  vision3d(1):v3d_polygon.colour=>vision3d(1):v3d_rgb.r = 0.6,
  vision3d(1):v3d_polygon.colour=>vision3d(1):v3d_rgb.g = 0.3,
  vision3d(1):v3d_polygon.colour=>vision3d(1):v3d_rgb.b = 0.3
 )
).
```

Figure 6.7 A textual view after manual application of an update specification

6.2.1.6 Documentation

In addition to providing a consistency mechanism between views, *update_record*s are retained in a persistent form in the VPE system, as they are in EPE. An *update_record* browser and editor gives the user the ability to browse the changes that have occurred to an *inter_class* definition in the evolution of the mapping specification and to add further documentation to each *update_record*. In this manner a portion of the documentation of the history of development of the system is automatically built up as work progresses. Having this update history on-line also allows system developers to trace back through previous design decisions while mappings are further refined.

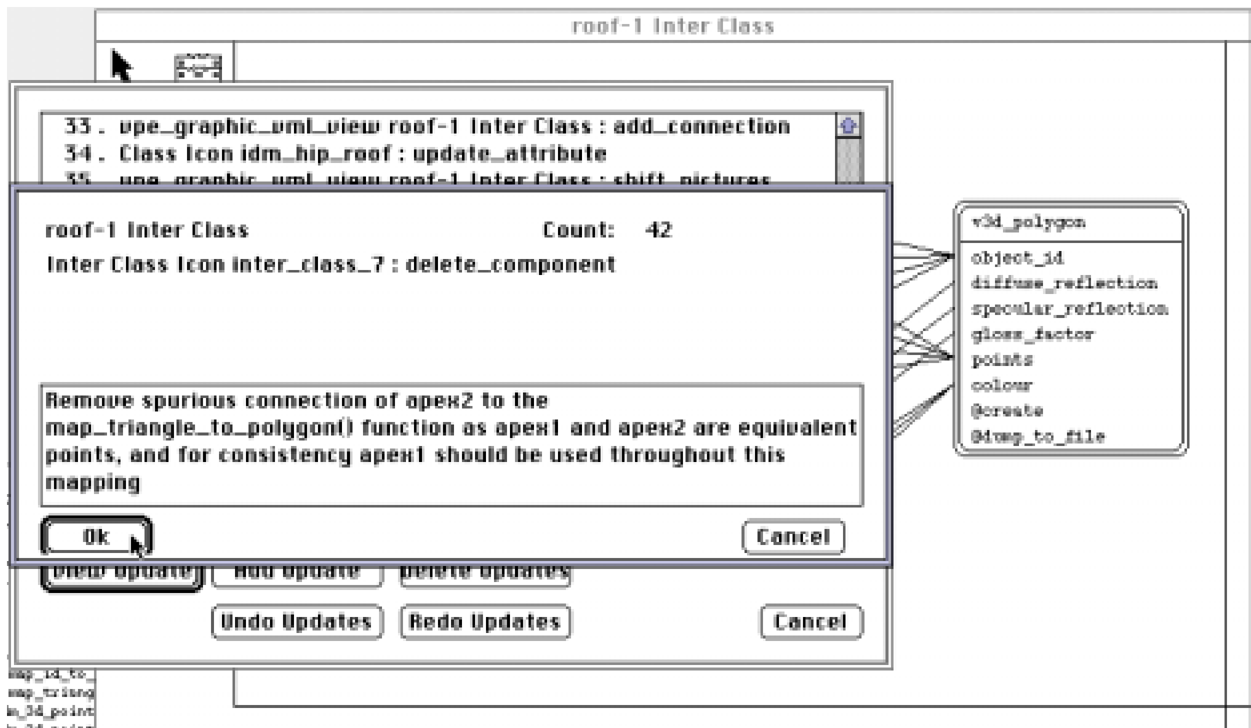


Figure 6.8 Browsing the change log for an *inter_class* specification

Figure 6.8 shows the *update_record* browser displaying a list of changes that have been made to an *inter_class* definition based on the *update_record*s generated by changes in various views. These include a change in the function used to map between features, and changes resulting from the renaming of a feature in one of the referenced schemas. The full details of the modification to the *inter_class* are displayed in the top window, highlighting the comment field that can be filled in by the system developer.

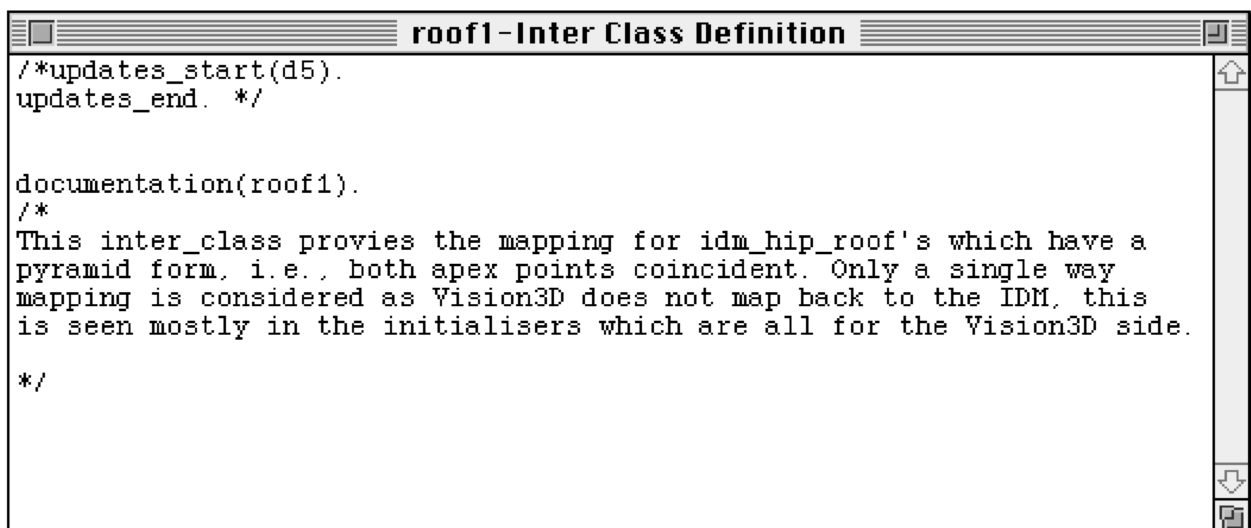


Figure 6.9 An associated documentation view for an *inter_class* definition

VPE also provides textual documentation views (accessible via the hypertext navigation facilities) for *inter_class* definitions, used to document the reasoning behind decisions made and other information relevant to a particular *inter_class* specification in a central and managed fashion. A useful feature of documentation views is that *update_record*s relating to the *inter_class* definition

are automatically added as textual comments to the view as the *inter_class* changes. Figure 6.9 shows a documentation view taken at an early stage of the mapping specification.

6.2.1.7 View navigation

As can be seen from the preceding example, the VPE modelling environment captures a large amount of information about a mapping specification, and hence navigation facilities are needed. In VPE the views and *inter_class* icons themselves act as a navigation and search facility for the project, in a similar way to the views and icons of EPE. Mouse clicks on graphical view components allow rapid access to other views containing that component as shown in Figure 6.10.

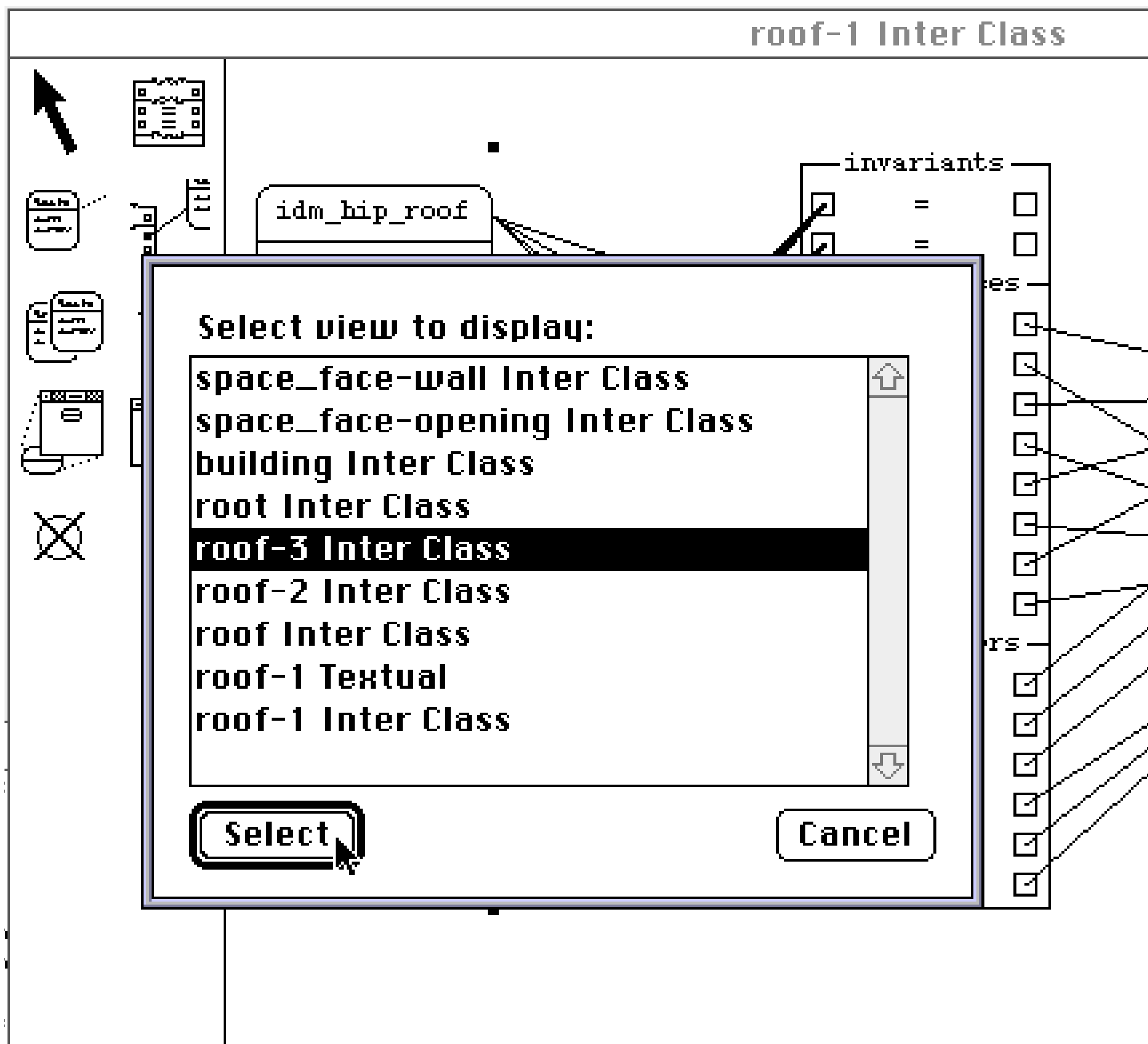
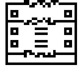


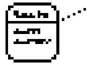
Figure 6.10 View navigation for *inter_class* specifications

6.2.2 Using the VPE environment

The VPE system offers two types of display views to its users. The initial type is a graphic view which can contain VML-G icons, representing *inter_class* definitions, tied to icons representing entities from the two schemas being mapped between (see Figure 6.1 for an example of a graphical view). In the graphical view the user has a palette of tools available, as seen on the left hand side of the view. These tools offer the following functionality:

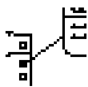


Used to specify an *inter_class* definition in a graphical view. Clicking on an empty portion of the drawing window after selecting this tool will create an empty *inter_class* icon in the window.

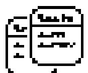


Allows the attachment of an entity definition to an *inter_class*. To attach an entity, the user clicks inside the *inter_class*, and drags the pointer to an empty portion of the drawing window, which is where the icon will be placed. This will invoke an entity dialogue to determine which entity to place in the window, followed by a features dialogue to determine which attributes and methods will be visible in the icon (the default is all attributes and methods). If an existing entity icon is selected when this tool is current then the entity dialogue and features dialogue will be retrieved. During the entity dialogue the user may specify a new entity, rather than select one from the proffered list. A new entity can either be a normal entity to be added to one of the schemas, or a temporary entity to be managed by the mapping system. During the entity dialogue, the user may also specify the position of the named entity in the class list for the *inter_class,* or check a box which defines that the entity is grouped in the class list specification.

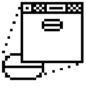
Used to specify the inheritance hierarchy between *inter_class* definitions drawn in a graphical view. To specify an inheritance link, the user clicks on the parent *inter_class* icon and drags to the child *inter_class* icon.



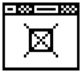
Used to specify the inclusion of an entity's attribute or method in an *inter_class* equation. The user selects the attribute, method, or entity name shown in the entity icon, and drags to the *inter_class* icon. The effect of the drag is dependent on where in the icon the drag terminates. If the drag terminates in: the connection box of an existing equation, the attribute or method is attached to the existing equation; if it terminates in one of the three section headers (invariants, equivalences, initialisers), a new equation is created and a connection made to that equation. If the user clicks on the line separating the entity name from the attributes and methods, then a new attribute or method can be described through an attribute dialogue when the connection has been made to the *inter_class* icon. A new attribute or method can either be a normal one which extends the schema definition of the entity, or a temporary one to be managed by the mapping system.



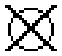
This function is not implemented in the current modelling environment. It is intended to allow an *inter_class* mapping between two versions of an entity to be automatically specified, with as many one-to-one links between the two entity versions as possible based on attribute and method names. However, the current Snart language has a single name space which means that two entity definitions having the same name can not be specified.



Used to create a new graphical view of the specified *inter_class*. If several *inter_class*es and entities are selected when this tool is used then the user will have the option of copying all selected icons to the new window.



Hides an entity icon, or an inheritance link, or an *inter_class* and all its attachments (i.e., entities and inheritance connections). The hidden item is not removed from the canonical representation of the schema, merely hidden in the current graphical view.



Deletes an inheritance link, or an *inter_class* and all its attachments (i.e., inheritance connections). The items are removed from the canonical representation of the schema as well as from all views which contain the item.

Allows icons in the graphical view to be selected and repositioned in the current view. Double clicking on an entity icon has two effects, depending upon the area in which the double click occurs. Double clicking on the entity name brings up the entity dialogue which allows the position of the entity in the class list to be specified, and whether the entity is to be grouped or not. Double clicking in the attributes and methods area brings up the attribute dialogue allowing the set of attributes and methods which are visible in the entity icon to be modified, or to add new attributes and methods (or delete attributes and methods). Double clicking on an *inter_class* icon has three functions depending upon where in the icon the double click occurs. If the double click occurs in an equation, then the full description of the equation is displayed in the equation dialogue. In this dialogue the user can manipulate a textual description of the equation, function or procedure call. After modification the textual specification is re-parsed to determine what symbol should be shown in the *inter_class* icon, and what connections should be made to entities currently in view. Double clicking on the left hand side of the icon headings (invariants, equivalences or initialisers) brings up a dialogue listing all views that this *inter_class* is specified in to allow navigation to other views. Double clicking on the right hand side of the icon headings (invariants, equivalences or initialisers) makes the textual view of the *inter_class* visible.

The second type of view offered is a textual view. Textual views allow free-form textual editing and manipulation of the canonical definition of VML *inter_class* definitions (see Figure 6.4 for an example of a textual view). Textual views are re-parsed at the termination of a textual editing stage (as signified by the user), and all modifications propagated through to the canonical form of the edited *inter_class* and to any graphical views which may be affected.

6.2.3 Implementation of VPE in the MViews framework

VPE is constructed by tailoring MViewsER (Grundy 1993), a specialisation of the MViews object-oriented framework which is fully described in Section 3.2.3. Though VML and VML-G have very little relationship to the ER diagramming notation there was quite a similarity between the

relationships defined between objects in the MViewsER system and those needed in the VPE system. Therefore MViewsER was used as the starting point for the VPE system and, through some careful modification, specialised to construct the VPE environment. Figure 6.11 shows the inheritance hierarchy for the VPE environment, with all VPE classes prefixed with a 'vpe_'. This structure is very similar to that of MViewsER, and the majority of the MViewsER code was able to be utilised for the VPE environment. The modifications to the MViewsER environment fell into the following categories:

Incorporating class icons: unlike other MViews-based environments, the VPE system requires an associated modelling notation to be displayed in a graphical view. The associated notation is for the schema classes that are wired to denote the mapping in a graphical view. The Snart notation was chosen as all attributes appear in the same icon, making it easy to render and manage (unlike EXPRESS-G). Extra coding was required to manage the two schemas being mapped between, and to present the modeller with relevant schema information in the graphical views.

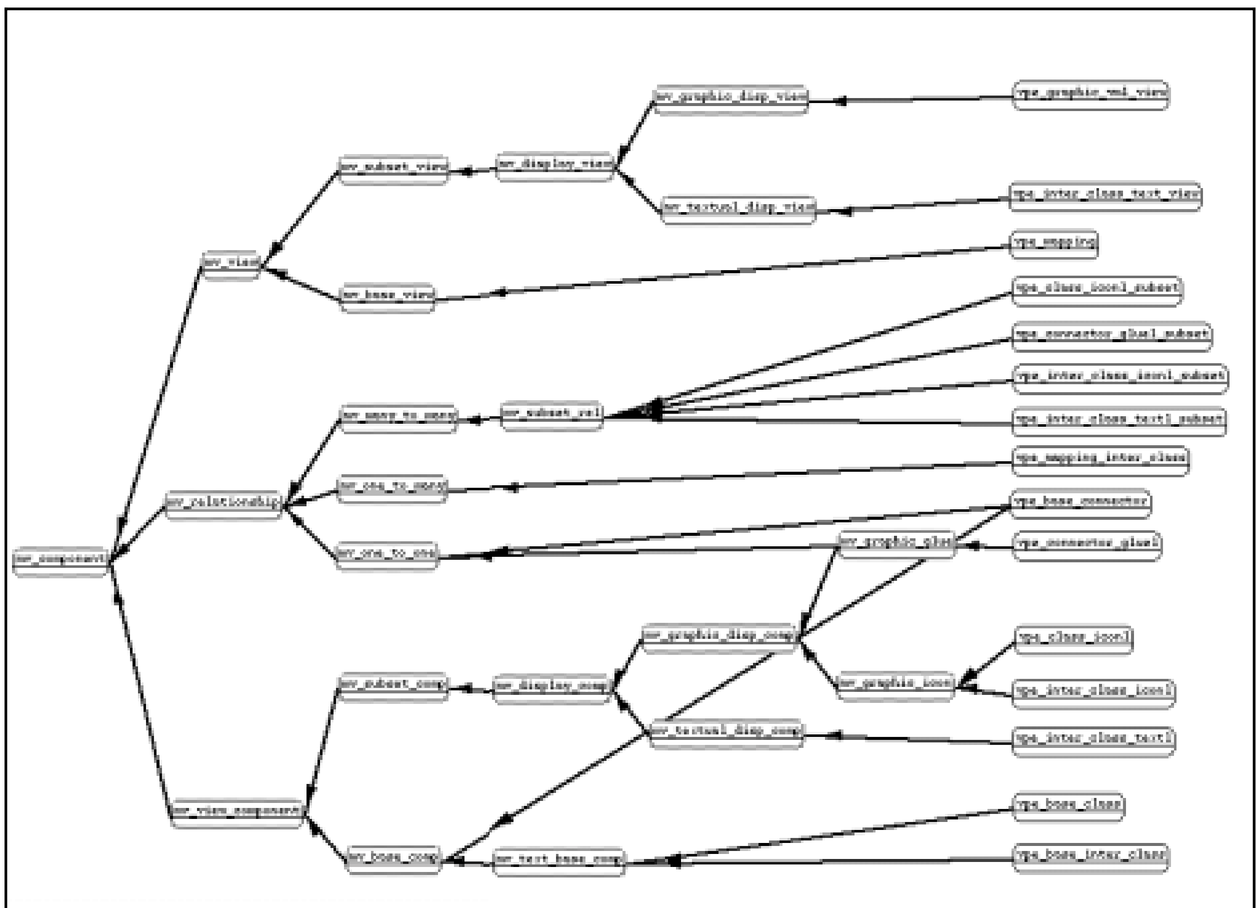


Figure 6.11 Class inheritance for VPE from MViews

Rendering modifications: VML and ER have a very different appearance and meaning on screen. All MViewsER display code was changed to reflect the VML appearance, and class names in MViewsER modified to reflect the functionality of VML icons.

Environment modifications: The set of functions which can be performed on a VML diagram are very different from those performed on an ER diagram. Therefore, new function icons had to be created, and the functionality associated with them fully coded. All dialogues had to be rewritten to reflect the requirements of a VML environment and to use the correct terminology for a mapping support environment.

VML language parsing and generation: the underlying representation of information in the VPE system is VML, which provides for an easy mapping between the canonical representation and the display views. However, a parser and VML code generator were still required to parse textual views and determine their syntactic correctness, and to generate VML code snippets from graphical representations.

update_record labelling: the range of *update_record*s used in MViewsER was examined and modified to suit the structure and terminology of VML.

6.2.4 Future connections between VPE and EPE

Currently there are no connections between the schema definition environment EPE and the mapping language environment VPE. However, in the same manner that MViews has been used to integrate multiple modelling paradigms (Grundy and Venable 1995; Venable and Grundy 1995), it is possible to connect the EPE and VPE environments. The benefits that this would produce are:

- mapping specifications which add to, or modify, the schemas being mapped between could be propagated to all affected views in the EPE environment. In this way, the mapping environment could be used as a schema integration system, allowing an IDM to be constructed from the requirements of the various design tools being mapped to it.
- schema changes made using EPE could be propagated to all affected mappings definitions in the VPE environment. Many of these changes could be automatically applied to the VML specifications (e.g., renaming of entities or attributes, changes to basic attribute types). Changes which could not be applied automatically (e.g., deletion of attributes, changes to a method's parameters) could be tracked and notified in all the mapping views which reference the affected item.

6.3 Management of Mapping Definitions

VML mappings specified through the graphical views of the VPE system are easily checked against the schemas being mapped between, as the wiring to entity icons makes explicit the exact entity, attribute or method involved in the mapping. However, when parsing a textual mapping specification in VPE, and, to a much greater extent, when utilising a complete VML mapping from some other source, there is scope for ambiguity in the mappings and a necessity to cope with schema modifications. Ambiguity in VML specifications occurs when shorthand forms of attributes and methods are used in a mapping. In many cases it is desirable to allow this shorthand form, as reading a specification with full references (i.e., schema name and version, then entity

name, then attribute or method) is fairly cumbersome. In most cases, the side of the attribute references makes explicit the entity to which an attribute or method belongs. In some, however, the same attribute or method name can appear in multiple entities from the same schema and clarification is required. Another problem is that when parsing a mapping it is impossible to distinguish between a mistyped entity, attribute, or method name and a reference to a new entity, attribute, or method for a schema.

To cope with these problems, a VML parser and verifier (VML-Check) has been constructed which checks initially whether a VML mapping is syntactically correct, and then checks it against the schemas it references, expanding the VML mapping to full references and updating the schemas where necessary. VML-Check requires three inputs: a VML mapping definition; and two schema definitions. Schemas that can be loaded are either in the generic schema definition format, as described in Section 3.3, in EXPRESS format (with the same restrictions as defined in Section 3.2.4), or Snart format. Schemas in either EXPRESS or Snart format are transformed into the generic schema definition format during the checking process. Schemas in Snart format can be multi-file schema definitions or even whole applications (in which case only the class interface information is extracted). VML-Check produces both a fully expanded textual definition of the original VML mapping, and a generic database representation of the mapping definition (as described in Section 6.4).

6.3.1 Name resolution

If all schemas had unique entity, attribute and method names, and no typing mistakes were made during mapping specifications, the process of resolving references in a VML mapping would be a simple task. However, this is an ideal situation and unlikely to occur, so the following problems needed to be tackled during the resolution of references:

- determining the entity an attribute or method reference belongs to when there are multiple entities which have the referenced name in their definition. Handling this problem is broken into two cases. If the duplicate references are from different schemas and the mapping is from an equation where a default side can be determined, then the reference defaults to the entity from the schema associated with that side of the equation. If there are multiple entities with the name reference for a particular side of an equation, or for a function where there is no notion of side, the user is presented with a list of possible resolutions for the mapping and they choose the correct reference.
- determining whether a reference is to an entity or an attribute when both an entity and an attribute have the same name in a schema. This is treated in a similar fashion to above. If the side of the equation the reference comes from has only the entity, or the attribute associated with it then the checker defaults to that reference. Otherwise, the user is presented with a dialogue to determine the correct reference.
- determining whether an unresolvable reference should be an addition to a schema definition or if it is a typing mistake by the user. This cannot be handled automatically, so the user is

presented with a dialogue to direct the checker as to what to do with the reference.

6.3.2 Schema modification

Mapping definitions can include references to entities, attributes and methods of a schema which do not currently exist. A mapping definition can also reference temporary entities and temporary attributes of existing entities. All of these constructs need to be recorded against the schemas utilised for the current mapping.

Where additions are specified for an existing schema, the generic schema definition is modified to contain a new version, inheriting from the currently specified version, which has a creation reason of type *mapping_dependant* (see Appendix G). All modifications to the schema are then specified in this version, the mapping being checked is updated to be a mapping between the new schema version and the other schema version. This new version captures not only new entities, attributes and methods, but also all temporary attributes and methods which are specified in a mapping. The assumption is made that all temporary attributes and methods belong to a particular entity, that is, there is an occurrence of a temporary attribute for every occurrence of a particular entity in a mapping.

Temporary entities, on the other hand, belong to neither schema, and so are recorded separately from the generic schema definitions. Temporary entities are only recorded in the generic mapping definition (see Section 6.4) with the assumption that any implementation of VML mapping specifications will be able to manage their creation.

6.4 Generic Mapping Database Definition

Although VPE provides a multi-view, automated consistency, modelling environment for VML, it is not assumed that this environment will always be used to specify mappings. Mappings could well be created in other environments (some of the simple mappings shown in this thesis were defined utilising a text editor), or even translated from other mapping languages to be utilised in a particular implementation of a mapping system. To support the range of possible mapping sources a generic mapping notation (detailed in Appendix H) has been specified. Mappings defined using this notation act as the input to any implementation environment for VML mappings.

A generic mapping definition contains all the information required for a VML implementation to perform mappings between instances of two schemas. This includes definitions of all the referenced entities from the defined versions of the two schemas. These entity definitions define the complete entity interface, including all attributes and methods added to the schemas by mappings, as well as temporary attributes and methods, and definitions for temporary entities. The actual VML *inter_class* definitions are split into component parts in the generic mapping definition,

and are all uniquely identified. The component parts of a mapping (i.e., each individual invariant, equivalence and initialiser) contain a parse tree representation of their definition and are classified as to the type of equation they represent, as well as listing the set of schema references which are contained in the equation. Inverted indices (which can be statically determined) are also stored for every entity, attribute and method referenced in all mappings. This provides quick access to *inter_class*es and individual invariants, equivalences and initialisers which reference any particular entity, attribute or method.

6.5 Appraisal of Mapping Modelling and Development

The VPE system and mapping database format create an environment which supports the development and communication of inter-schema mappings as well as the documentation requirements of an integrated modelling environment, as detailed in Section 6.1.1. VPE supports these requirements in the following manner:

Connection with the schema development process: this is only supported to the extent that schemas can be loaded into the VPE environment to be attached to *inter_class* definitions. Though a user can hand edit a class definition (e.g., rename an attribute) there is no automatic flow of consistency checks throughout the VPE environment. The VPE environment provides hooks which will allow the automatic handling of schema modifications at a later date (by propagating schema update records to and from the VPE environment).

Documentation: VPE meets many documentation needs by tracking all updates made to the mapping and recording them against the *inter_class* definitions that were modified. These update records can be annotated by the user to record justifications and decisions, and provide on-line documentation for the developing mapping. One documentation feature which would be useful, but is not provided, is the ability to group multiple disperse update records to represent a single update or documentation record. For example, this would record a session of changes as a single documented change to the mapping.

Support for multiple views: multiple views are supported for all *inter_class* definitions which have a full invariants specification (providing the key for the *inter_class*). This allows any combination of graphical inter_class definitions to be specified and maintained with full consistency at all times. However, due to the difference in specification level between the textual and graphical views, there are many graphical updates which can not be automatically applied to textual views, though all textual updates can be automatically applied to graphical views.

Though VPE meets the majority of the requirements for a schema modelling and development environment, there are some areas which would benefit from further work. These areas include:

Tight coupling between specification environments: though the VPE environment accesses schema definitions when constructing a mapping view and when checking the correctness of a mapping specification, there is no formal coupling between the schema modelling environment EPE and VPE. A tight coupling between these two environments would allow schema modifications in EPE to be displayed in appropriate VPE views as updates. These updates could either be automatically applied to the current mapping, or the mapper could apply them by hand. In a similar fashion, the specification of a mapping which modifies an existing schema could be propagated through all affected views in EPE, and either be automatically or manually applied. This would increase the level of consistency between schema definitions and mappings which are related to them. The experience of Grundy and Venable (1995) suggests that this integration can be made with little or no modification to EPE or VPE, by using the inter-repository relationships and multiple-base layers used in their ISDE integration.

Collaborative design support: although large modelling projects may have a single coordinator in charge of mapping specification development, there are likely to be several modellers working on different aspects of the mapping. Coordinating the work of several modellers, and maintaining the consistency of the underlying mapping under change from several sources concurrently, has to be a goal of any real mapping development environment (see Grundy et al. 1995 for an example of developments in this area).

Expanded modelling support concepts: the range of support concepts required by modellers when developing large mappings is not clearly understood, and hence, not well supported. Currently, documentation of mapping modification and construction is made at the atomic level. Methods allowing for grouping of higher level concepts are imperative (including multiple viewpoints of change sets). The manner of notification of modifications in a collaborative environment needs attention to provide efficient methods of expressing these changes to other designers (rather than at the atomic change level), an example of developments in this area can be seen in Grundy et al. (1995). Navigation and summary features tend to be primitive. In large mapping definitions with hundreds of *inter_class* specifications and thousands of views, the set of views which reference a particular *inter_class* could be very large. Classification of view types, or the relationship particular *inter_class* definitions play in a view, could well help navigation around the mapping and guide novice users through the various conceptual levels of mapping specification. Notions of mapping versions and private workspaces need to be considered in a collaborative environment. This allows the management of multiple design paths, and for incomplete work to be hidden until fit to be used by other participants.

In summary, this chapter introduces the requirements for a mapping modelling and development environment, and through the development of VPE and a mapping database format, demonstrates that an environment meeting these requirements is possible. This now provides a tie-in between the schema for an IDM and the schema models for DTs and users. However, to provide the full

description of a project requires specifying information flows and where particular schemas are required. This is the problem tackled in Chapter 7, the last of the modelling chapters, before detailing the implementation of a system which utilises all of the developed models.

Chapter 7

Project Modelling

To utilise the types of integrated design system that can be described with the modelling and mapping specifications described in Chapters 3, 5 and 6, it is also necessary to manage the tasks and people involved in the projects in which the integrated system is used. This level of modelling enables an integrated design system to be customised for use in a specific project. Project modelling is a task that must be undertaken at the start of every project as participants and their tasks will vary for almost every project. This task is usually assigned to a project manager, though project models must be checked and accepted by all participants.

The work on project modelling presented here includes and extends work undertaken by the author during a six-month visit to the COMBINE group at TU Delft in the Netherlands, the results of which are reported in TU Delft and Amor 1993, and Amor and TU Delft 1993.

7.1 Introduction

A project encompasses all stages of work from inception through to demolition and possible reuse of a particular artifact. To manage and understand what happens in a project, a model is required of the various actors involved and the responsibilities and tasks they play in the project. The development of a project model also provides an understanding of which data models are required in a project and what data transfer is required between these models. This influences the work done on schema development and on mappings between schemas. A project model provides a formal description of the various users who will be involved in the project, and formalises the roles they will be fulfilling in the design task. The various design roles can be further refined to individual design functions which can, in turn, be associated with the design tools available to

perform them. The set of design functions must be able to be scheduled, to allow a project manager to ensure that design policies are followed, quality assurance conventions are maintained, handover points between contractors are formalised, and design progress can be monitored. Without the ability to model and manage these aspects, an integrated design system offers a meaningless data transfer mechanism unrelated to a real-world project.

A large design and construction project involves many development phases, some of which may be independent of each other, and, in many cases, need not be specified (and may not even be known) at the start of the project. To understand and manage the large scale of projects, and their initial indeterminacy, they are often conceptualised as happening in stages, though these stages often overlap. Common stages in a project may include inception, management, design, construction, maintenance and demolition. Stages may be very broad, or quite specific; for example, the structural design stage in a particular design office. In COMBINE, the term 'project window' was introduced to describe these coherent portions of a project (see Figure 7.1). This term is used throughout this thesis. The project windows represent a given time-slice of the project, or some sub-process. Therefore, a project is considered as being divided into multiple, conceivably overlapping, project windows which are specified prior to the execution of a new phase of a project.

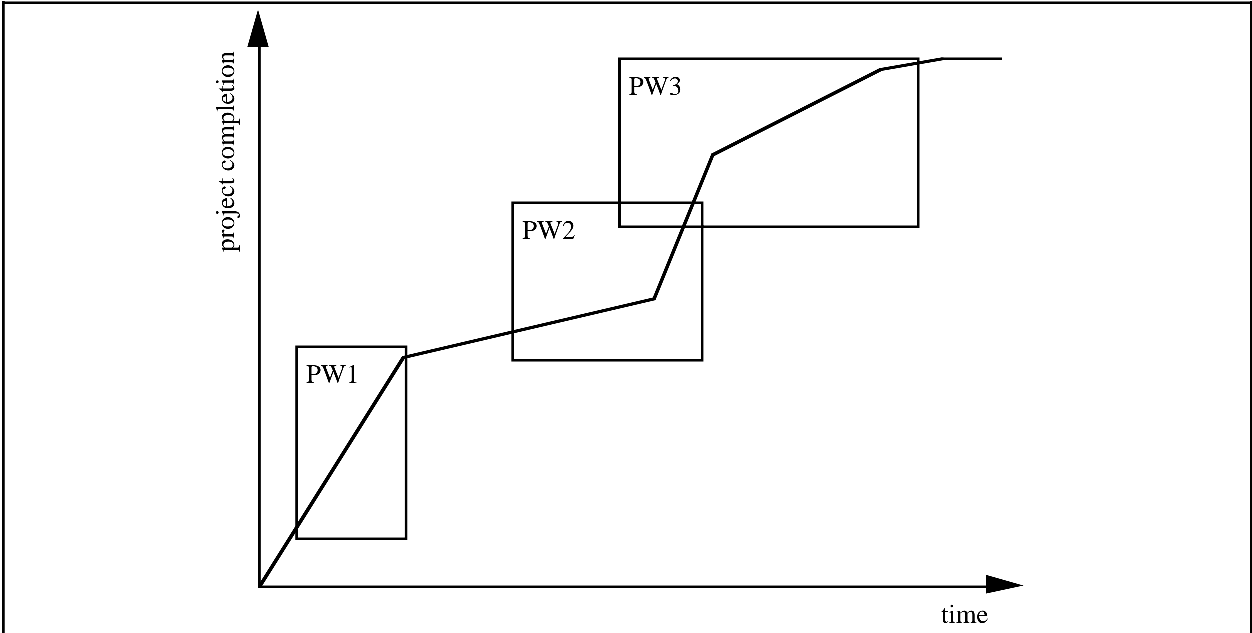


Figure 7.1 Multiple project windows in a project

Even splitting a project into multiple project windows is unlikely to provide enough flexibility for real projects. A single project window may model weeks of work, over which period the participants in the design team could change (e.g., bringing in an expert to help with unforeseen problems) and various flows between design functions may need to be modified (e.g., to ensure new aspects of the design are checked). To cope with this variability, a previously specified

project window model must be able to be modified and updated as the design progresses, with immediate flow-on effects to the running control system.

The modelling of projects and project windows requires many aspects of a project to be captured. Other research in this area has been examined to provide an understanding of the requirements for project modelling. The notion of project models outlined above has great similarity to the concepts of project and process in software engineering. Curtis et al. (1992) provide a review of process modelling which categorises requirements and various approaches to process modelling. Their four most commonly utilised perspectives in process representation (functional, behavioral, organisational and informational) are used to rate previous process models, and are also used to evaluate the CombiNet model developed here. The meaning of these four perspectives (from Curtis et al. 1992, page 77) are:

Functional: represents what process elements are being performed, and what flows of informational entities (e.g., data, artifacts, products) are relevant to these process elements.

Behavioral: represents when process elements are performed (e.g., sequencing), as well as aspects of how they are performed through feedback loops, iteration, complex decision-making conditions, entry and exit criteria, and so forth.

Organisational: represents where and by whom (which agents) in the organisation process elements are performed, the physical communication mechanisms used for transfer of entities, and the physical media and locations used for stored entities.

Informational: represents the informational entities produced or manipulated by a process; these entities include data, artifacts, products (intermediate and end), and objects; this perspective includes both the structure of informational entities and the relationships between them.

	Functional	Behavioral	Organisational	Informational
Procedural programming languages	*	*		*
Systems analysis and design	*		*	*
AI languages and approaches	*	*		
Events and triggers		*		
State transition and petri-nets	*	*	*	
Control flow		*		
Functional languages	*			
Formal languages	*			
Data modeling				*
Object modeling			*	*
Precedence networks		*		
CombiNet	*	*	*	*

Table 7.1 Capabilities of project specification languages (after Curtis et al. 1992)

Curtis et al. (1992) evaluate different styles of project specification against these perspectives, summarising their results in tabular form. Table 7.1 replicates this summary (where asterisks represent the ability of a language to support a perspective) and extends it to include the work presented here (CombiNet entry). The styles of project specification presented below cover all the different process notations and tools presented in Sections 2.5.2 and 2.5.3.

7.1.1 Requirements

Considering the four most common perspectives offered by Curtis et al. (1992) led to the definition of a set of views that must be able to be supported. These views include users, tasks, data and workflow. The requirements of these views are as follows:

User View: all the actors involved in a project window and the tasks they must undertake to ensure completion of the project window role. This view must capture the responsibilities of the various actors and their rights in terms of viewing and modifying information in a project.

Task View: all design functions that can be performed by the design tools within the project window. The granularity of a design function can vary from atomic changes to a design, through to a complete design. The level of granularity is determined by the requirements of individual projects. The only fixed requirement is that each design function must happen between a starting and ending exchange event of information with respect to the integrated design system. Everything that goes on between these exchange events is invisible in the project window model, being the domain of the design function. Thus, design functions can be both on-line atomic CAD design tool operations and off-line batch mode design tool runs.

Data View: all schemas concerned with design functions and users. In the project windows described here this includes:

- The complete IDM schema.
- The IDM subschemas corresponding to the input of design functions.
- The IDM subschemas corresponding to the output of design functions.
- The IDM subschemas corresponding to the input view of actors.
- The IDM subschemas corresponding to the change access of actors.

This offers a purely static view of the project window definition, not addressing aspects of control over instances (unless it can be modelled in the schema). The subschemas do, however, define the state that the project must reach before a particular design function may be utilised, or before an actor can be called into the project.

Workflow View: defines all possible flows from a particular point in a project. It ties the design functions together and provides the way of describing the handover between various actors in a project. The workflow defines how tightly managed and controlled a project is going to be, from highly specified to very open. To enable the workflow view to be used, a management system will have to be able to determine all states of a design function, the state of the integrated design system, and control over exchange events according to control flow constraints. This will include the following aspects:

- whether a design function is being performed.
- whether a design function is a candidate to be invoked (in view of design functions that it depends on).
- whether a design function is a candidate to be re-run because the running of another design function changed its original input.
- the design function state and integrated design system state resulting from previous exchange events.

The views specified above describe aspects of a project which must be modelled to ensure that required project perspectives can be supported. However, they do not provide a measure of the level of control that will be exerted on a project through the views. The level of control could vary widely, from very rigid and autocratic management through to very free and autonomous management. Current perspectives on this include:

- No control as typified by current integrated design systems. These systems are merely able to transfer data around without any in-built process context and purpose of data exchange events. For such a system to be operational in a particular design project, a project manager needs to establish work procedures, task scheduling, etc. among the team of users. The system is an insensible data router which provides no guidance on what should be done next.
- Shallow control is seen in several of the more recent integrated design system efforts, e.g., COMBINE, and ToCEE. This approach uses a scheduler which deals only with the control over who, or what task, is next, and which monitors the states resulting from exchange events. At this level of control, a project manager can enforce much rigour in the use of the system. Although this might suit project windows for parametric (routine) design, it is to be expected that most project windows will require a more flexible approach, where the human users may interact with the control layer as well as with each other (outside the system).
- Deep control is the level that integrated design system developers aspire to and AI researchers still ruminate over. This type of control deals with much greater complexity than just scheduling. Deep control must deal with the pre- and post-conditions which are related to exchange events. It should have the ability to look inside exchange events and DT interfaces, and propose remedial actions for any failure of a DT invocation or analysis, or propose and control how the system can recuperate from deadlocks in design scheduling. Deep control should also deal with declarative knowledge on the meaning and purpose of design functions and thus support goal-driven design strategies.

A deep control system, while perhaps being the type of control system that should be aimed for in the future, is out of the scope of this thesis. Deep control, as defined above, requires the solution of many hard research problems in the artificial intelligence field. As a first step towards introducing process control into integrated design systems, shallow control has been selected for

use in all the large EU funded projects and this is the approach implemented by the author and shown in this chapter.

7.1.2 Structure

To cover the four most commonly utilised perspectives specified in Section 7.1, and meet the requirements of Section 7.1.1, the CombiNet formalism developed here defines three main types of information to be modelled for a project window: project window requirements, defining part of the informational perspective; user requirements, defining most of the functional, organisational and informational perspectives; and flow of control requirements, defining the behavioral perspective. Project window requirements allow the specification of starting conditions for entry into a project window, and exit conditions (e.g., data that is required by the end of the project window). User requirements allow the specification of the participants and their design functions in a particular project window. These participants (actors) perform certain design roles in a project and these design roles can be completed through the application of various design functions. A design function can be represented as a particular design tool used in a certain manner, e.g., to perform one type of analysis from the range a design tool offers. Flow of control requirements allow the specification of the paths that may be followed between various design functions to complete the design phase encapsulated by the project window.

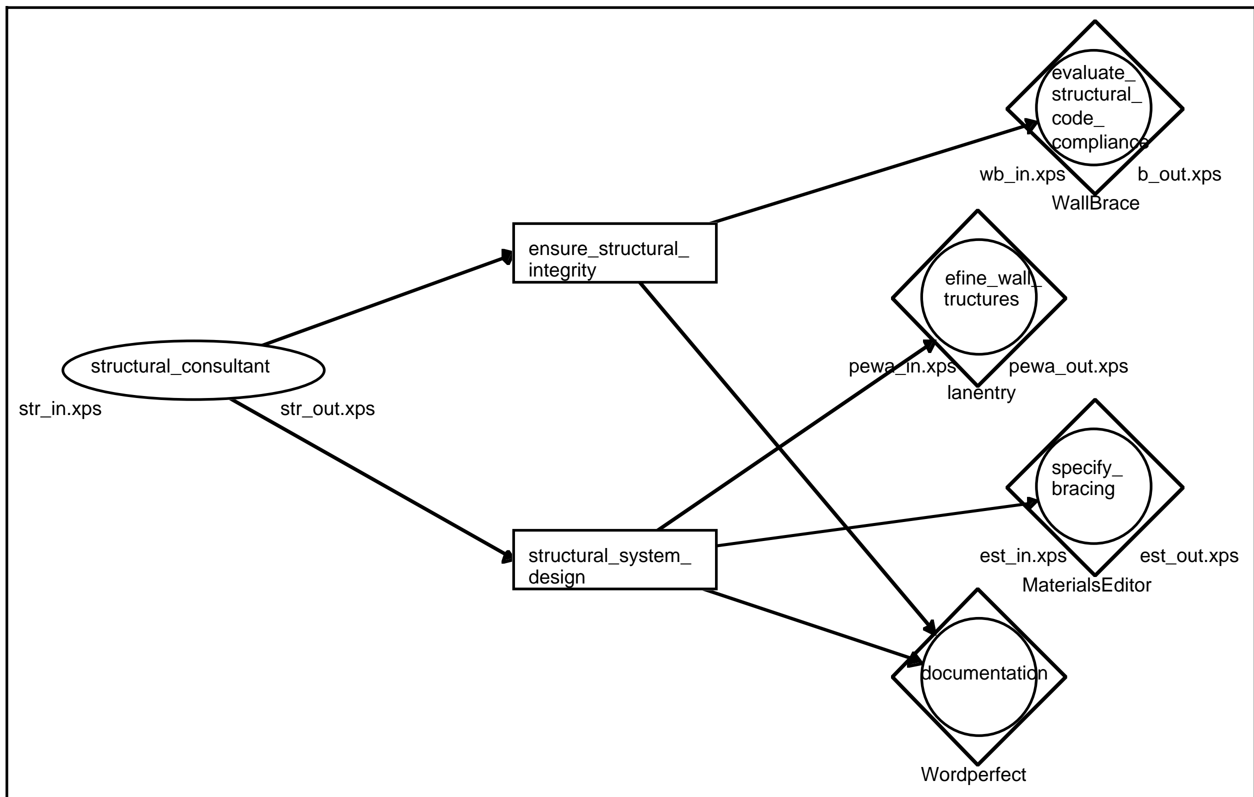


Figure 7.2 Example of user and function specification

The CombiNet formalism utilises two specification notations to model user requirements and flow of control requirements. Though two notations are described, they are used in an integrated manner in the specification environment, with explicit links between diagrams in both notations.

7.2 User and Function Modelling

A majority of the functional, organisational and informational perspectives, as introduced in Section 7.1, can be grouped together and defined in a single formalism. Functional and informational perspectives can be defined for design functions, actors and the project window through the definition of input and output models. These schemas specify structures and constraints of the models used by, and produced through, the use of a particular design function, actor or project window. Organisational perspectives can be partially defined in the same model through connections between actors, their design roles, and the design functions required to complete the design roles.

In Figure 7.2, a graphical specification of user and function, detailed in a CGE (Configurable Graphical Editor, Vogel 1991) project window formalism (developed by the author), is presented. There are three main types of icon in this diagram, connected by arrows representing *performs* and *requires* relationships for actors and design roles respectively.

Actor: the oval icons in the left column of the diagram represent actors (users) participating in the project window being specified. An actor has a name defining either the actor or, as in Figure 7.2, the type of actor which will be performing particular roles in the project window. Each actor is associated with a pair of schema defining both the subset of the IDM that they are able to view, and the subset of the IDM that they are allowed to modify. These two schemas define the area of responsibility of a particular actor, and are used to ensure the actor acts within a specified domain. These schemas can also be used to check that the roles an actor plays are not outside the actor's area of responsibility. Actor responsibility can be checked against design role responsibility, as the area of responsibility of a design role can be determined by the union of the schemas associated with each design function it utilises. An actor is associated with one or more design roles. In Figure 7.2 each actor has unique design roles, but this is not mandatory, multiple actors are allowed to perform the same design role.

Design Role: the rectangular icons in the centre column of the diagram represent the various design roles which are performed by actors in this project window. Each named design role can be fulfilled through the application of various design functions (the scheduling of which is specified at a later stage). Several design roles can utilise the same design function in the completion of their role. For example, the documentation design function is used by most of the design roles.

Design Function: the icons in the right column of the diagram represent the various atomic design functions which can be carried out in the completion of design roles in the project window. A design function has a name defining the type of function it performs and directly under the icon the name of the design tool which will be used to perform the named function. Each design function is associated with two schemas defining both the subset of the IDM which will be the input to the design tool, and the portion of the IDM which can be updated at the completion of the design function. These two schemas define the responsibilities of the design function. Though not shown in Figure 7.2, several design functions can be accomplished with the same design tool, but often with different input and output schemas defining responsibilities for the tool. The definition of the different functions performed by the same design tool is indicated by the two schemas specified along with the design function.

In the CGE environment the modeller can navigate from this user and function view to the top level flow of control view through a menu item in the environment. The CGE environment is not as sophisticated as the MViews environment utilised in Chapters 3 and 6, limiting the type of environment that can be offered to the modeller. The main restrictions are that it can only provide a single view of a model (i.e., one user and function view) and sophisticated navigation facilities are not available (e.g., it is not possible to navigate from a design function to all flow of control views which reference that design function).

7.3 Flow of Control Modelling

A Petri net formalism provides much of the behavioral perspective required for project specification, and, along with the DT schemas for actors and design functions defined in the user specification detailed in Section 7.2, many properties of a project's state can be calculated. In this section the calculable properties of a flow of control specification are defined, along with a formalism for describing them. This formalism is based around the design functions defined in the user specification. However, it also overlays actor's areas of responsibility to provide the link to organisational perspectives for the project specification.

7.3.1 Set theoretic background for flow of control

The control view states defined in Section 7.1.1 are calculable from analysis of data flows in the integrated design system based on the schemas associated with each design function (DF). The basis of the analysis is to assume that the input and output schemas for a design function together describe a subset of the IDM schema. This is not strictly true, as the schema for a design function is likely to have cardinality constraints, keys and value constraints which differ from the IDM. However, these added constraints can be ignored when checking subset relationships and when

performing intersections of various schemas. What will be used for the set operations will be the definition of entity names and their inherited entities, and attribute names and types which appear in the design function schemas and the IDM. In the following conditions and constraints $_{In}$ is used to denote the input schema to a design function or actor, and $_{Out}$ to denote the output schema of a design function or actor.

There are two conditions which must hold on the design function and actor schema definitions:

Condition 1: $\forall_{i\ 0\ \ (DF\ c\ Actor)}\ (IDM \supseteq i_{In},\ IDM \supseteq i_{Out})$

Condition 2: $\forall_{i\ 0\ \ Actor}\ ,\forall_{j\ 0\ \ DF\ of\ Actor\ i}\ (i_{\ In} \supseteq j_{\ In},\ i_{\ Out} \supseteq j_{\ Out})$

This just states formally that the input and output schemas of all design functions and actors must be a subset of the IDM, and that the input and output of any design function used by an actor is a subset of that actor's schemas. In practice, this means that the input and output schemas must be defined in terms of the IDM (i.e., using the same entity, relationship, method and attribute names), as well as being defined by the model structure used in the actual design tool or actor view. Using these definitions enables a static check of all schemas in the integrated system. The schemas can be checked against the IDM to determine whether they are valid subsets of the IDM (i.e., whether the schema has been defined properly. This will mainly pick up typing errors). The allowable differences in schemas of a design function or actor over that of the IDM are that the former may define: different uniqueness constraints (keys); different constraint clauses or ones which are additional to those defined in the IDM; different cardinalities on attributes; and different optional specifications for attributes.

Given a system which contains the IDM schema and the input and output schemas of the various design functions used in a particular project window, and the definition of a flow of control for a given project window, there are various properties which can be calculated. To derive these properties from the design function schemas two constraints are defined:

Constraint 1: $\forall_{i\ 0\ \ running\ DF},\ \exists_{c\ 0\ DF}\ (i_{\ Out} \cap c_{\ In} = \emptyset)$

This constraint is a concurrency check, or an invocation check, stating that a design function (c) is a candidate to be invoked if its input schema has no intersection with the output schema of any of the running design functions (written as $_{running\ DF}$ in the constraint).

Constraint 2: $_{j\ =\ completed\ DF}\ ,\ \exists_{i\ 0\ \ previously\ run\ DF}\ (i_{\ In} \cap j_{\ Out} \neq \emptyset)$

This constraint is a re-invocation check, stating that if the output schema of a design function which has just terminated ($DF_{j\ Out}$) intersects with the input schema of any other design function which was previously run ($DF_{i\ In}$), then the previously invoked design function (DF_i) is a candidate to be rerun.

It must be noted at this point, that intersections between design functions are described only at the schema level (i.e., static determination). Where the design functions require a model of a full building this will be sufficient to determine the properties detailed above. However, if a design function models only a small portion of a building (e.g., calculates properties for a single space) then the properties calculated above could present a design function as a candidate to be rerun in more cases than necessary. In the implementation of the flow of control system, this is handled by also tracking the objects which are used by each design function. The intersections between design function schemas gives a static determination of whether a design function needs to be further examined at run time to determine the properties defined above.

The working of the two constraints described above is illustrated in the figures below:

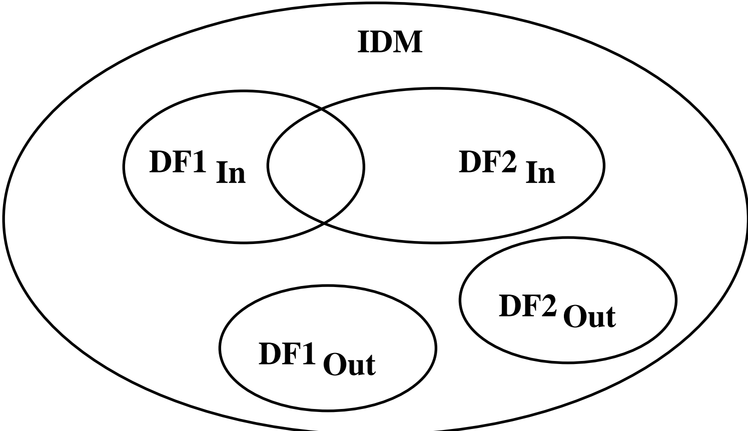


Figure 7.3 Invocable design functions

Figure 7.3 presents the situation where, even though the two design functions share data in their input schemas, neither of the design function input schemas have an intersection with a design function output schema. In this case, both design functions may run concurrently, and the result of either tool will not cause the re-invocation of the other design function.

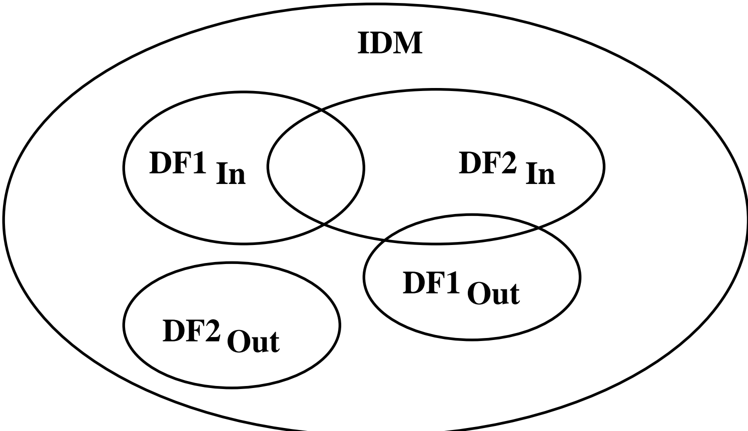


Figure 7.4 Constrained design function

Figure 7.4 presents an example where there is an intersection between an input and output schema of two design functions. In this case, if DF1 is running then DF2 may not start. This figure also illustrates what may happen with Constraint 2. If DF2 was run at some time in the past, and then DF1 is run, then the output of DF1 may overwrite what was previously supplied to DF2, so DF2 becomes a candidate to be rerun.

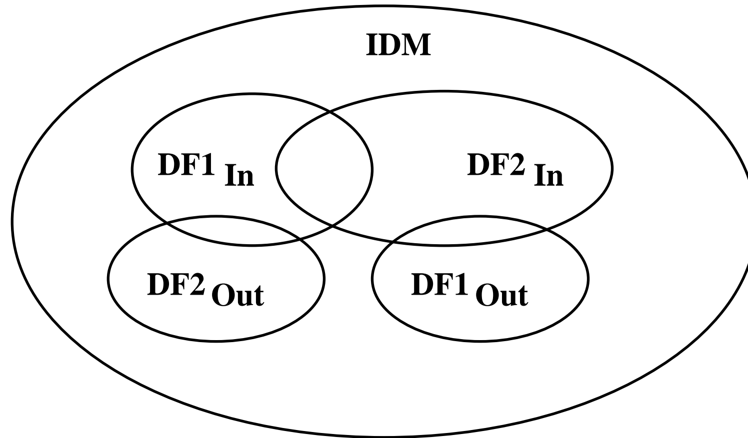


Figure 7.5 Constraints between two design functions

Figure 7.5 presents a case where the two design functions may not run concurrently, as the input schema of each design function intersects with the output schema of the opposite design function. This figure illustrates a situation where the invocation of either of these design functions can cause the other design function to be a candidate for a rerun, apparently causing a cycle. However, this is not problematical as the design functions are only candidates to be run. The final decision of what can be run at any particular time is made via the project window control flow information.

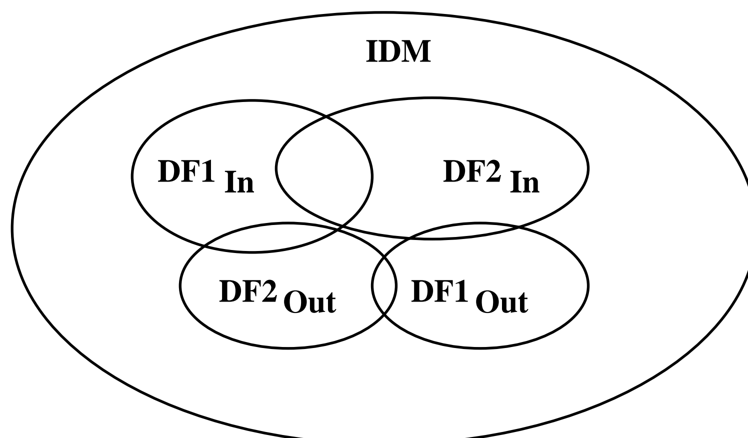


Figure 7.6 Design function constraints leading to apparent inconsistencies

Figure 7.6 presents a definition which could apparently lead to inconsistencies. The two design functions have the same properties as those in Figure 7.5, but with the additional complication that they overwrite portions of each others output. Again, this is acceptable, as the running of each of these design functions is determined by the project window definition which by its own definition gives rights to a design function to place its output in a portion of the resultant data store. The

termination of either design function will mark the opposing design function as a candidate to be rerun, making explicit the fact that the output of the design function has been overwritten.

While these examples demonstrate the interaction between just two design functions, the constraints described earlier are general and the results can be applied over any number of design functions.

7.3.2 Flow definition

The graphical formalism developed to specify flow of control in a project window is based very loosely on the Petri net formalism (Jensen 1990). Icons representing places, transitions and tokens are used in the modified definition, along with new icons to allow further functionality to be described. Although the new formalism (called a CombiNet) looks very similar to a Petri net, the semantics are very different. The icons in this formalism and their functionality are described below, with reference to Figures 7.7 and 7.8.

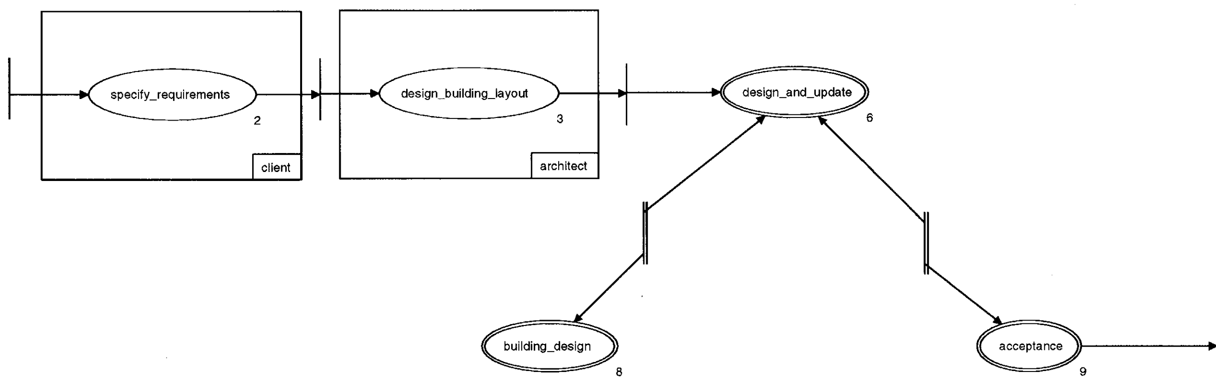


Figure 7.7 Top level flow of control specification

Place: A place in the CombiNet (the single lined oval icon in Figure 7.7) represents a single design function. It can be reached from some set of transitions and it will exit to some set of transitions. Each place represents exactly one design function, but the same design function can appear many times in a CombiNet.

Aggregate Place: An aggregate place (the double lined oval icon in Figure 7.7), as its name suggests, represents a set of icons defining an identifiable subset of the process being modelled, defined here to be all icons drawn in a window. Thus, allowing the modeller to group complex interactions into logical subnets and reference them through a higher level mechanism. Aggregate places representing the same CombiNet may be referenced from any number of CombiNets or many times from a single CombiNet. This allows a common sequence of events to be represented by a single icon in a CombiNet. It is also used to distinguish the flow of control of different actors, or different design roles, into separate definitions which can be linked together appropriately at a higher level CombiNet.

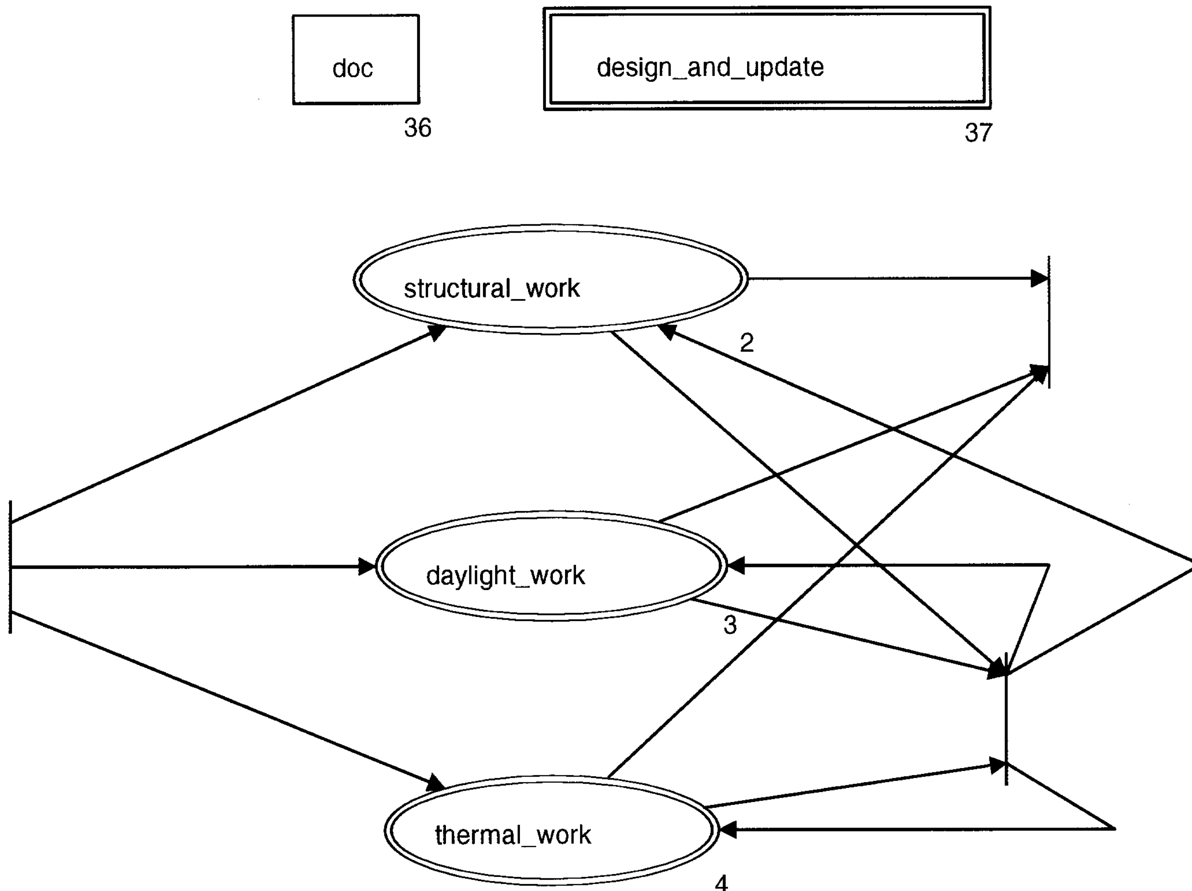


Figure 7.8 Flow of control with global elements

Global Place: The global place (the single lined rectangular icon in Figure 7.8) represents a single design function exactly like a place, but it has no connection to the transitions. The global place denotes a design function that can always be activated when in the current CombiNet (or in any descendant CombiNets, i.e., those represented by an aggregate place or global net). Global places allow design functions which can always be invoked (e.g., documentation functions) to be specified in the CombiNet without having to draw links from every single place and aggregate place in the diagram. It also allows a specific flow semantics to be associated with the global place. This semantics is that after invoking the design function in a global place the flow of control passes back to the place or aggregate place which invoked the global place (rather than any place or global place in the diagram, as would occur if everything was wired back and forth to a single place to achieve this effect).

Global Net: This is a combination of an aggregate place and a global place. The global net icon (the double lined rectangular icon in Figure 7.8) represents a complete CombiNet and is invocable from any place in the current net, and all descendants of the current net. When inside the global net all other global elements, other than those that are already entered (i.e., a global net can not call itself recursively), can be seen. The set of visible global nets reduces as each one is stepped into, and grows as each one is completed. The CombiNet representation of a global net is the same as a normal net, so it can have its own global

places and global nets as well as normal place-transition flows and aggregate places. A global net is used for many of the same reasons as a global place (mainly for functions which can occur at any time). Its main benefit is that a whole flow of control can be defined in the global net rather than a single design function. For example, after completing some set of design functions in a global net it would be possible to force the flow of control through an evaluation function before returning control back to the original CombiNet. The semantics of entering and exiting a global net are the same as for a global place.

Transition: A transition in the CombiNet (the single vertical line icon in Figure 7.7) works as the choice point between one design function invocation and the next. It can be used to work out the set of candidate design functions that can be invoked after a running design function has terminated. There are two special kinds of transition that can be created. A transition which has no entering arrows (i.e., is not exited to by any place or aggregate place) is called a start transition and denotes an entry point into a CombiNet. A transition which has no arrows exiting to other places is called an end transition, and denotes an exit from a CombiNet. In the current version of the project window definition, a CombiNet is allowed to have only one start transition, though it may have any number of end transitions. Having a single start transition in a CombiNet is purely a reading convenience for a user, as this guarantees that having found one start transition the reader has found the only entry into the CombiNet (a single CombiNet can be very large and complicated and it could be easy for the modeller to overlook some starting transitions if more than one was allowed). A single start transition also eases the computation required when attaching to aggregate nets. All transitions which invoke an aggregate place are connected to the start transition of the CombiNet represented by the aggregate place. In a similar manner, the exits from an aggregate place to a transition are all tied to the set of end transitions in the CombiNet represented by the aggregate place.

Double Transition: The double transition (the double vertical line icon in Figure 7.7) is a shorthand notation for a loop between two places, which is represented by two normal transitions, one in each direction, in a running system. In Figure 7.7 the two double transitions provide two design function choices when exiting the *design_and_update* aggregate net, either *building_design* or *acceptance*. If the *building_design* aggregate place is entered, there is only one exit, back to *design_and_update*. If *acceptance* is entered then there are two exits, either back to *design_and_update* or to terminate the CombiNet (and in this case the whole project window).

Actor: The actor overlay (the large box icon with the actor name in the lower right corner, see Figure 7.7) defines the actor responsible for a set of places, aggregate places, global places and global nets in a CombiNet. This allows a clear distinction to be made between areas of responsibility of various actors, and indicates the point of transfer of control from one actor to the next. The actor overlay is needed in cases where multiple actors and design roles utilise the same design function to perform their tasks. In this case, without the

overlay, there is no way of determining the actor responsible for a design function in the CombiNet. When implementing the flow of control, moving from the design function of one actor to that of another is treated as indicating an acceptance and sign off of all responsibilities for the initial actor. This includes accepting and terminating all outstanding re-run requests for the actor handing-over control. However, the new actor will see re-run requests generated by modifications that affected run states of their previous design function work (if any) since the last time they were involved in the project. New re-run requests will be added for the new actor as they work through their design functions, as in the previous definition. Again, at the hand-over to another actor all current re-run requests for the completing actor are removed. This helps control the proliferation of re-run requests caused by changes which may impact on every design function which has previously run. This management of re-run requests is local to a single project window, and will not affect the same actor's work in another project window of the same project.

Although tokens are never shown in the CombiNet diagrams, they are used in the implementation to illustrate the flow of control in the CombiNet. A token represents the actions of one actor performing their design roles with various design functions. A token in a place represents the running of a particular design function. When the design function has terminated, the transitions are used to determine the candidate design functions from the current place. When the next design function is decided upon, the token crosses the choice point created by the transitions and invokes that design function. If a design function is unable to start, or terminates unexpectedly, the token is sent back to the place it previously came from, and transitions available from that place must be re-evaluated. If a token moves to a global place or global net then after completing that function it will return to the place from which the global function was invoked. A token enters a CombiNet from a start transition and leaves on an exit transition. In every flow of control specification there is a top level CombiNet from which a single token is started. When a token moves across a boundary between areas of responsibility of different actors then the user the token represents changes, and this represents a formal hand-over between the two actors.

The differences between the working of a Petri net and a CombiNet are seen most clearly in the flow of tokens. In a Petri net there must be a token in each place entering a transition before a token can cross the transition. When a token crosses a transition it places a token in every place pointed to by the transition. In the CombiNet it is completely different; a token, which represents an actor, is only constrained from traversing transitions by the state of concurrency constraints as defined in Section 7.3.1. Transitions provide a choice point between functions in the design. Also, as a token represents the workflow of an actor, the same number of tokens are passed over a transition as approach it. Therefore the number of tokens (workflows of actors) in the project window remains static in the flow of control system unless a new workflow (which would require a new token) is activated by the project manager (e.g., concurrent design in the project window). As the project manager is shown the analysed state of a running project window, it always clear

when multiple workflows can be scheduled concurrently. This is shown by the set of design functions calculated able to be run at the same time as currently running design functions.

7.4 Appraisal of Project Specification

The two part project specification formalism described above has been used (see Augenbroe 1995a and the example in Appendix E) to describe, and then implement, a shallow level of control in integrated design systems. The formalism, although simple, allows the definition of complicated project models and actor interactions. In relation to the four perspectives of project representation it meets them in the following manner:

Functional: this perspective is addressed in both notations. The definition of what process elements are performed can be ascertained from the set of design functions specified in the flow of control diagrams. The informational entities relevant to each process element are specified through the input aspect model of each design function in the user and function diagram.

Behavioral: this perspective is addressed in the flow of control diagrams. The CombiNet allows specification of all sequencing requirements of design functions as related to the design roles of individual actors in a project.

Organisational: this perspective is partially addressed in the user and function diagram. This hierarchical diagram associates actors with specific design roles, and through those design roles to individual design functions. CombiNet's actor overlays are used to distinguish actor responsibilities for particular design functions where multiple actors need to utilise the same design function. Actor responsibilities are rigidly defined through the use of aspect models defining an actor's view of a schema and the schema components they are allowed to modify. Physical communication mechanisms and physical media are not addressed in this formalism as this is currently managed independently in integrated design systems.

Informational: this perspective is addressed in the user and function diagram. The definition of an output aspect model specifies what is produced by a particular design function, while the input aspect model defines what is manipulated by a particular design function.

The CombiNet formalism for flow of control specification provides a simple yet powerful notation to define control flow in a project. The main benefits of the formalism are:

Explicit flow of control: The use of places and transitions allows for the definition of explicit paths through the design process. There can be iterations in the design process and choice points in the direction that the design progresses. The flow of control can be tightly constrained to follow set paths and invoke certain functions in a specified sequence.

Aggregation: The introduction of aggregate places reduces the complexity of individual nets by allowing subnets to be identified and packaged as individual components. The aggregate

places allow common tasks or processes to be described once and referenced many times throughout the design process. Aggregation also allows processes pertaining to individual actors to be specified separately from all other actors, providing a very visible separation between actor tasks.

Global icons: These icons provide a simple notation to encode flows of control which can occur at any time in the design process. Multiple global nets in a CombiNet represent a wiring structure which would be very intricate and cumbersome to define with other process models.

Wide range of control: The combination of global components and connected components provides for a wide range of control strategies to be implemented. These range from totally unconstrained, through to totally constrained. The various states are characterised by:

Totally unconstrained: all states are defined by global places, so that anything can start at any time. Flow between various actors in the system is still monitored, but any actor can do anything at any time.

Partially constrained: some states are defined by global nets, therefore invoking certain states requires passage through other states in a controlled manner.

Partially unconstrained: most states are described through the usual place-transition flows, with some global places or global nets to represent states that can occur at any time.

Totally constrained: there are no global places or global nets defined, so all states must be invoked by following the flows defined in the place-transition flows.

Well defined hand-over points: the hand-over of control between different actors in a project window is explicitly modelled in the flow of control specification. This provides well defined and easily distinguishable areas of responsibility in a design project.

There are also certain weaknesses to the formalism as developed, the most important being a poor handling of complicated boolean conditions between design functions in a CombiNet. For example, it may be required to inhibit the execution of a particular design function until a set of other design functions have executed, without consideration for the order in which this other set of functions is executed. This type of specification will prove hard to code in this formalism due to the ability to have loops in the flow of control. For example, does satisfying all the conditions for the first time make these conditions satisfied every other time they are encountered? Although these types of conditions can not be made explicit in the formalism, a very weak form of these conditions can be implicitly defined in the input schema definition of a design function. An input schema can specify a requirement for input which could only be met through the execution of a given set of design functions (again only useful for the first iteration in the flow of control). An approach to alleviating this problem would be a formalism for explicitly specifying the set of conditions that must hold before a design function is invoked. This would allow for both the semantics of repeat invocation to be specified and for full system-state, and data-state, pre-conditions to be defined.

The project specification provides a way of describing shallow control in a project with a degree of

flexibility sufficient for small through to large projects. However, there are aspects of the specification which could be extended:

Deep control: the shallow control offered in the current project specification should be extended towards the requirements of a deep control system. Initially this would cover greater flexibility for specifying constraints on the project model. These could be constraints to be satisfied before the invocation of particular design functions or constraints which guide the flow of control based on the results of a design function. A large amount of research would be required to try to implement a more comprehensive control system based on design intent and it would have to be carefully crafted so as not to intimidate the users of such a system.

Greater specification of environment in model: an assumption in the project specification was that physical communication mechanisms and physical media need not be addressed in the formalism, as it is currently managed independently in the implemented integrated design systems. This assumption could be lifted to provide a more general project specification which can model design tool parameters. For example, the environment in which they operate, their invocation parameters, the format of input and output data-files (if they use data-files), etc. Previous research has considered this problem and the papers on TES (1995), and to some extent by Pascoe (1994), describe notations which allow the majority of these parameters to be defined.

Flexible specification environment: the CGE environment was used by the author to implement this formalism as it was the modelling tool used in the COMBINE project. Re-implementation in an MViews-like environment would provide a more sophisticated modelling environment, allowing multiple overlapping views of the flow of control specifications, as well as all the coordination and documentation benefits offered in an MViews environment. An MViews environment would also offer the possibility of tying the project specification through to the schema and mapping specification environments described in Chapters 3 and 6. This type of specification environment has subsequently been demonstrated in the Serendipity system (Grundy 1996). Serendipity allows the specification and coordination of process models alongside software development for multiple developers, with consistency management and developer conflict resolution and management.

In summary, this chapter presents the requirements for project specification in an integrated design system as well as a formalism to allow its specification. This formalism allows a shallow level of control to be specified between actors in a project, the design roles they play, and the flow of control between design functions used to satisfy particular design roles. The project specification, along with schemas (from Chapter 3) and mappings (from Chapter 5 and 6), is sufficient to implement an integrated design system which can be used in a specific project. Chapter 11 describes such a control system and gives examples of its use with the project definition used as

examples in this chapter, and fully specified in Appendix E.

Chapter 8

The Project Testing and Implementation Environment

Although the development of an integrated design system requires the development of schemas for the IDM, design tools, and actors, as well as mappings, design tool environment models, and project flow definitions, they can not be developed in isolation. After defining the various schemas, the developer(s) must be able to test what is being created to verify that it fulfils its required purpose. To achieve this, the schemas must be instantiated with test data from the domain they will be used in. Mappings between schemas must be tested to ensure that the result of mapping from a model is equivalent to the original. Design tool environment models must be tested to ensure that the design tools start and finish as expected and that they find no problem with the data presented to them.

To make this possible requires testing environments that allow models to be quickly instantiated and validated to ensure that relationships and dependencies are as expected. To test mappings requires a system which simulates the functionality of a full mapping system, coupled with model visualisers to enable the correctness of the mapping to be ascertained. Similar testing is required for project models and design tool environment models. Though the environment developed for this testing could be the final integrated design system there are many reasons why it should not be. The major reason is that integrated design systems can take many forms and assuming that several are to be developed it is likely that each would vary in implementation to some extent. For example, the model implementation may be very different in different systems, from a relational database through to an object-oriented or frame-based system. To test models in each of these variations would require very different testing environments. However, if a single model visualisation tool was created for testing all integrated design systems during development, the final schema could be specified in any paradigm with the developer being sure of the correctness of the specification. Having a single suite of testing tools also allows closer integration between the testing tools, with flow-through benefits from the different stages of testing. For example, the

model visualisation tool can be used to inspect a derived model after performing a mapping between two models.

In this section, the various testing environments required to test the range of models detailed in Chapter 2 are described. A testing environment is described for each class of model described in Chapter 2 and where they can benefit from close links between each other and the modelling environments this is specified. Some of the environments are major pieces of work completed as part of this thesis, and described in more detail in subsequent chapters. Others are tools developed in other projects or are yet to be implemented, and are only described below to provide an idea of the functionality they can provide in this integrated framework.

8.1 Structure and Requirements

Complementing the non-linear modelling process described in Section 2.1, there is a similar non-linear implementation and testing process. Examples of the types of cycles and interactions which occur at this stage are detailed below:

- During any schema development stage a test suite needs to be set up to validate the schema development. This test suite is likely to be developed incrementally as the schema progresses, with tests representing normal, as well as unusual, design situations which the schema should be capable of representing. This evolving suite of tests must be able to be run against each new schema version to ensure that restructurings have not reduced the schema's expressive power. Later in the project whole building models must be able to be loaded in, not just as a test of the whole schema, but to enable testing of mappings to and from other design tools.
- During the IDM development, and particularly towards the final stages of its development, the IDM needs to be tested with an instantiated model. This is a paradoxical problem, as the IDM is developed to integrate DTs which can create an instance model, but the DTs can not be integrated until the IDM is developed. Bootstrap tools are thus required to create, visualise, and manipulate instances of the schema. These tools are used by the IDM developers to test their model, by the DT modellers to ensure the actual data in the IDM matches their conceptions of what should be there from the IDM development process, and by the DT interface developers to test the interfaces as they are developed. The IDM is still likely to change at this stage, so the instance model needs to be constantly restructured to take into account all modifications to the IDM. Changes in the instance model must then be passed through to all DT developers who rely on the instance model for their development.
- When the mapping specifications are being developed they need to be tested with the instance models of the IDM, to ensure that the mapping is correctly specified (the 'artwork' on page *v* shows the 'small' problems that are easily picked up from this type of testing, this problem originated from an incorrect specification of axes in a building model

mapping). These tests are run by the mapping developers. Where problems are detected they require assistance from the schema modellers of the IDM and the DT. Problems in test mappings may require changes to the mapping specification, or to the schemas of the IDM and DT, where the problem is one of missing information or constraints in the schemas. All of these changes have flow-on effects to anyone using the IDM as specified above.

- The specification of a project's utilisation of an integrated design system is a negotiated process and, especially in large projects, may need to be simulated to ensure that all required design roles can be completed in the available time-frame. Results from simulations may highlight areas of concern in the specified project flow, and lead to further modifications and re-negotiations between actors in the project.

Modelling tool and integrated environment requirements have already been described in Chapter 2. Similar requirements exist for the implementation tools and these are described throughout the rest of this chapter. Again, this chapter looks at implementation environments as if they are independent from the modelling stage. This is not the case, but this unnatural split does enable the issues for each of these sections of the implementation to be considered independent of competing and distracting issues from the other category of requirement.

8.2 Schema instance development

To help reduce the schema development time mentioned in Section 2.2, it is necessary to provide methods to instantiate sample buildings for the evolving schema and to navigate the resultant structures to ascertain that they meet their required purpose. There are many tools which can provide this level of utility. Four which have been used throughout this project are described in Section 9.2, however, requirements for this type of tool are detailed below.

8.2.1 Requirements of a schema instance maintenance system

To provide maximum utility to designers these environments must satisfy the following diverse requirements:

Instance manipulation: developers instantiating or browsing a model need to inspect tailored views of an instance, to specify values for attributes, and to specify references to other objects. The views of the data should be able to be presented in forms which are of a similar semantic level to that in which the developer envisages it (e.g., a graphical view of an object's geometry). Attribute values should also be able to be specified at a similar semantic level (e.g., clicking on an iconic representation of objects to collate identifiers to be referenced, rather than typing in all the object identifiers).

Correspondence with an evolving schema: as schemas evolve the previous model instances need to be brought forward into the new representation. While a mapping specification and mapping system can be used to achieve this syntactic transformation of the data, the model

maintenance system must be made aware of new schema versions. The system can then ensure that new objects are created with the correct parameters and attributes.

Multiple views: for model developers, multiple graphical and textual views are useful beyond the schema development stage. Similar methods for navigation, perusal and checking of the instances in a model are essential when dealing with a model as complicated as a building. Thus, the ability to view graphically the physical components of a building and to navigate through the building structure at either a textual or a graphical level and with the ability to switch between different representations of a selected instance, is particularly useful. Tools offering support for viewing and manipulating instances of the model are used by both the IDM development team and the aspect model teams to check instances of schemas, determining whether the structure is as they imagined when detailing their schemas and whether it accurately represents the information requirements of their DT.

Consistency: the development of an instance model can involve many of the schema developers, especially where a schema covers many domains in a field (e.g., steel structures, concrete structures, HVAC systems). Where several developers view instance model portions in varying levels of detail and representation, the various views need to be kept consistent with each other. This ensures that all modellers are aware of recent changes to the instance model and can be certain that they are working on the most up to date version.

Navigation: even a relatively simple building model (e.g., a domestic dwelling) from a schema with a hundred or more classes can contain thousands of objects. Navigating such a model can be a major task in itself. Methods of overcoming this problem are to provide high-level navigation mechanisms (e.g., walking down *part-of* structures utilising a GUI) and the incorporation of query languages which enable the identification of elements of interest. For elements which have a graphical representation, graphical navigation methods should also be provided.

8.2.2 Related schema instance maintenance systems

The majority of the schema modelling tools surveyed in Section 2.2.5 have no associated instance creation and perusal system. Though this is mainly due to the vendors' perception of market requirements, many of the modelling languages lack an ability to translate models into implementation languages (e.g., multiple conditional inheritance in EXPRESS has no direct translation into languages such as C++). The majority of development systems assume that the developed schema can be correctly specified without testing against actual data and can be transferred to the final implementation of the developed system which will contain the methods for creation of instances of the model. One modelling system (for software development) which allows instances to be created against the developing schema is SPE (Grundy and Hosking 1993a) and the associated development of CernoII (Fenwick et al. 1994) provides for many of the requirements described above.

The lack of associated instance manipulation environments in large schema development projects has proven to be a problem. In COMBINE, COMBI, ATLAS, etc. example buildings were created for final proof-of-concept demonstrations. These model instances were invariably developed late in the project and, with no complete tool to create the model, relied upon hand instantiation and laborious checking to ensure validity. This meant that the models developed were small (10-20 spaces and not always utilising all classes defined in the schema) and required hand-patches as schemas continued to evolve at the late stages of the project or as missing information was discovered (Augenbroe 1994b).

8.2.3 Approach to a schema instance maintenance system

To work with the modelling environments described in Section 2.2 the schema instance maintenance system has to comprehend EXPRESS schema definitions and data models sent in the associated STEP Part 21 data-file format (ISO/TC184 1994). The tools built for the maintenance system all operate in the same environment, which provides parsers for these formats and allows persistent databases to be created and manipulated according to the resultant specifications and data. The environment in which the maintenance system is built is the same as used to build the modelling environment, which eases the burden of providing connections between the two systems. The maintenance system tools provide the majority of requirements in Section 8.2.1, including instance manipulation and modification in a persistent store, viewing and navigation in both textual and graphical formats, and guaranteed consistency of the persistent store for data being manipulated in multiple views. The tools created and used in this project for instance creation and navigation are described in Chapter 9.

8.3 Mapping handler and controller

This module performs the translation of data between the IDM and the DT models. It must determine whether it is possible to create a consistent model from the IDM for any one of the DT models based upon the constraints in that DT model, and it must ensure that the IDM remains consistent upon update from a design tool's output. It must detect conflicts between various sets of data and must be able to invoke processes to enable negotiation over conflicting data from different design tools and users. An implementation of a mapping controller meeting these requirements is described in Chapter 10.

8.3.1 Requirements of a mapping handler and controller

A generalised mapping controller must offer the following facilities:

Consistency of data models: as developers or actors complete design functions with a particular DT they must be able to move relevant data through to the IDM whilst maintaining global consistency of the developing model. They must also be able to extract the current IDM

model for use in a particular design function. Through all of this the data in various data models must be kept consistent with each other when mapped through different models. In this way every user of the integrated system knows that they are utilising the same data as the other users of the system, or if they are out of date they will know of any possible problems and be provided with methods to become consistent. If consistency of the data models can not be maintained automatically then the integrated system must provide negotiation methods to allow the possible conflicts to be resolved.

Traceability of modifications: in any developed model a number of developers and actors will have been involved in the specification of various parts of the model. Modifications made to a particular model must be able to be traced to a particular mapping and ultimately to the user or DT which asserted the values. Tracking the modifications allows complete mappings to be applied and undone if necessary. It can also provide leverage in determining what mappings can be applied dependent upon the source of the data, or it allows a negotiation to be initiated with the specifier of the data to resolve conflicts.

Utilisation with different DT regimes: different developers and actors require different types of DTs to perform their design functions in a project. The mapping controller must be able to handle the requirements of various types of DTs. Some of these require all their data at the start of a simulation or design task, and others have interactive requirements for data based upon previous data (e.g., knowledge-based systems).

Actor and project manager links: the mapping of data between tools is a small part of the overall development of a coordinated design in a single project. To allow coordinated and managed design tasks amongst all participants in a project the mapping controller must be tied closely to the project management system. With close links to a flow of control system mappings can be activated to initiate a new design task, or used at the termination of a design task to capture the output of the task. To this extent the mapping component of most integrated design systems should be almost invisible to its users, providing many services in the system but unseen. The only point where it need be visible is to notify its user of model conflicts and allow a negotiated settlement.

8.3.2 Related mapping handler and controller work

The range of mapping handlers and controllers is as wide as the range of mapping notations described in Section 2.3.2. However, as none of the notations satisfy the requirements for a mapping language in that section, their environments can not provide for the requirements in this section. RDBMS provide the most sophisticated environments of any of the previously referenced notations, allowing tight management of modifications and their flow through to affected users. Though they can not perform the types of mappings required in this problem domain many of their facilities for management of transactions (including roll-back and roll-forward), view updates, traceability, and view interfaces are utilised in this thesis.

All of the mapping languages appearing in this domain (e.g., EXPRESS-M, EXPRESS-V, EXPRESS-X, Transformr, View mapping, etc) are associated with implementations. These implementations are very limited in their scope as they implement unidirectional mappings from one data-file of a model to the transformed data-file. Therefore they provide no management of global consistency of data in a project. Though all mappings in the COMBINE project were hand-coded their integrated building design system at least managed the mapping of data between tools (in a very conservative manner) to ensure that the IDM model was globally consistent.

8.3.3 Approach to a mapping handler and controller

The user interface of the mapping controller developed for this thesis is shown in Figure 8.1. This system implements mappings specified with the VML notation, utilising either a transaction-based, or individual, update mechanism to move data between related models whilst maintaining the consistency of the models. The mapping handler allows a VML mapping specification to be used in either direction to map data bidirectionally between models. Dependent upon the type of mapping specified, it will ensure that only coordinated updates are performed on an IDM (i.e., ensuring consistency of the IDM). It also automatically manages the notification of changes to all actors in a project who are affected by a particular modification to the IDM. The system allows both on-line and off-line DTs to be utilised with a mapping and incorporates full tracking of modifications, down to changes to individual attributes and permissions to change values at this level. Chapter 10 provides a full description of the implemented mapping handler for VML.

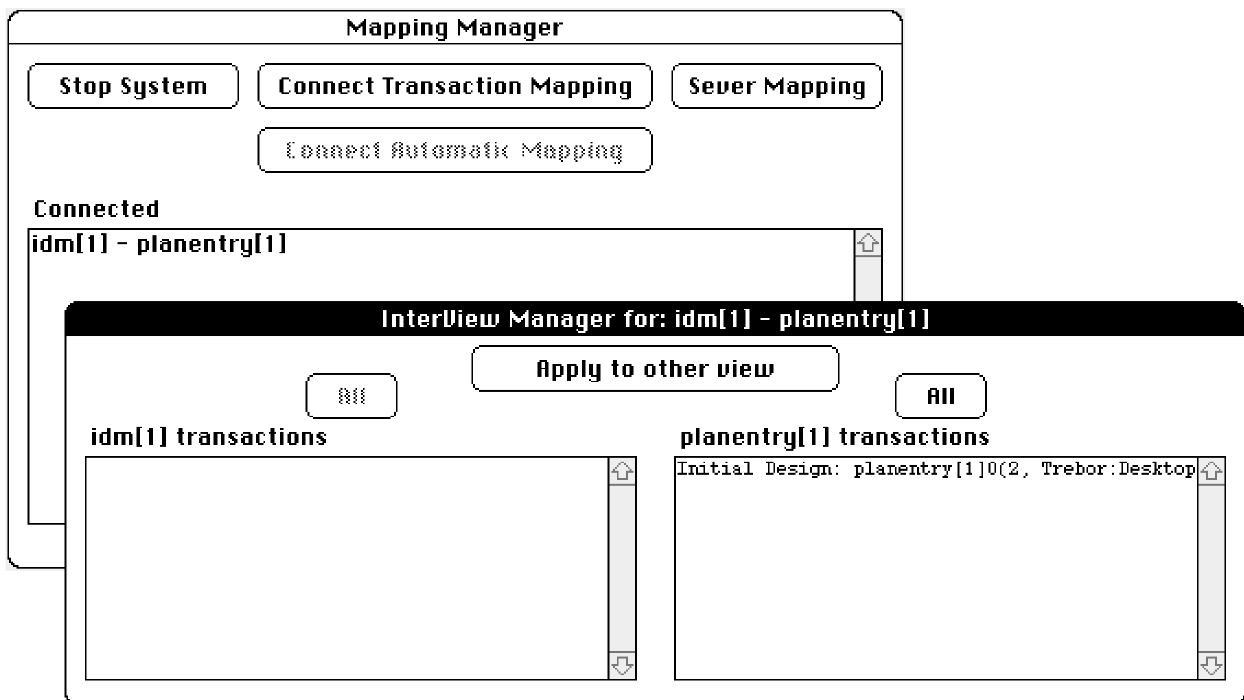


Figure 8.1 Sample mapping controllers

8.4 Design tool connection

This module provides the connection between design tools and the integrated design system. It should also be capable of invoking design tools when required, and determining when they have terminated. In a generic integrated design system it should be capable of performing these tasks over a range of hardware and software platforms, providing the interface to the integrated design system for all actors.

8.4.1 Requirements for a design tool connection system

A design tool connection system must offer the following facility:

Interface with IBDS: to lessen the workload on the actors all of the modelled requirements in Section 8.3.3.1 must be automated. For an off-line DT this module must be able to create the input files required for the DT (from the DT model in the system). It must then be able to invoke the DT to perform its work and finally retrieve the output from the DT into the DT model in the system. For interactive DTs it must perform the same task but driven by the demands of the design tool.

8.4.2 Related design tool connection systems

Traditionally this module is implemented by hand-coding the parsing and pretty-printing for each new design tool, and leaving it to the actors to run their particular design tools with the generated input. This is the approach seen in all of the EU funded projects (COMBINE, ATLAS, COMBI, CIMsteel, etc.). The systems of Amor (1991) and Pascoe (1994) provide the parsing and pretty-printing of data-files without the need for hand-coded tools (i.e., created based on the design tool data-file definitions). However, the design tool invocation still needs to be performed by the actor. No known system can detect design tool invocation and termination in a general environment (though TES environments (TES 1995) should, when implemented, provide this ability). However, it is possible for an IBDS to provide user activated procedures which allow it to be notified of a design tool's invocation and termination. Such a system is used in the implemented project flow of control module described in Chapter 11.

8.5 Flow handling

This module directs both the inter-model mapping and design tool interface modules to perform their tasks as required by the designers, or as directed by the project manager. This module simulates the flow of control defined in the project model, determining what design functions are able to be performed at any particular time. It also interfaces with designers to let them specify the tasks they wish to perform next, and interfaces with the project manager for decisions about which designers should be involved in new workflows in the project. The system directs the inter-model

mapping module to map data between different models, or to ascertain whether it is possible to perform the mapping. It also directs the design tool interface to invoke a design tool with appropriate data and to collect results upon termination.

8.5.1 Requirements of a flow of control manager

A flow handling system must offer the following facilities:

Calculation of possible design functions at any time: in a large project, with many design functions and where actors are working concurrently, it becomes very difficult to determine which design functions can operate without affecting current work. The flow handler must be able to simulate the flow of control specified with the tools described in Section 2.5. This initial simulation indicates design functions which are candidates to be performed at any instant in time. In a concurrent design environment many of these candidate functions will not be available due to possible conflicts with the data used by design functions already underway. Problems in the design are also registered by the flow handler which must backtrack through the specified flow paths to determine previous design states that can be worked from.

Visualisation of project state: the project manager must always be able to determine the current state of the project, and actors need to see what tasks they must still complete. Through coordinating a project the flow handler holds much information about the state of the project, both current and past. This information can be utilised to compare against expected or required deadlines as well as to estimate completion time for the project. Time-lines of expected versus actual time to completion of design functions can be derived from information held by the flow handler.

Control interface for actors and the project manager: all users of an IBDS need an interface with the system. The flow handler is likely to be the only interface the user has with the integrated design system. It will provide the actor with an interface for initiating design functions and for signifying the termination of design functions assigned to them. It will also allow the project manager an overview of the state of the project and the design functions being undertaken by different actors. This interface will allow the project manager to move and manage resources as required in the project to ensure its timely completion.

8.5.2 Related flow of control manager work

While Sections 2.5.2 and 2.5.3 describe several notations for describing flow of control in a project, as well as environments to define project flows in, only one of these systems follows through to a flow of control manager (TU Delft and Amor 1993). The majority of tools associated with the standard flow definition notations (IDEF0, IDEF3, Pert charts, state diagrams, etc.) either provide simulation capabilities or can output the model in a form understandable by stand-alone simulation tools. These simulations provide as output a likely schedule for a particular project, often presented in the form of a Gantt chart. Such charts are used by project managers to direct and

manage design functions on a project. As a paper system, however, they are not useful in managing design tools and the flow of information to and from them. They also become difficult to use when a very large number of design functions are specified on the chart(s), as the project manager needs to be mindful of all running and upcoming tasks and their relationships at the same time.

The CombiNet (TU Delft and Amor 1993) does have an associated management tool. This tool can load in a CombiNet definition and uses it to inform actors of their upcoming design functions, as well as to determine allowable design functions at particular times in the project. An initial prototype of the management tool was developed by the author for the COMBINE project. This prototype was later used as the specification for a more robust tool written in ‘C’ which was demonstrated at the end of the COMBINE project as the top level controller of their whole integrated building design system.

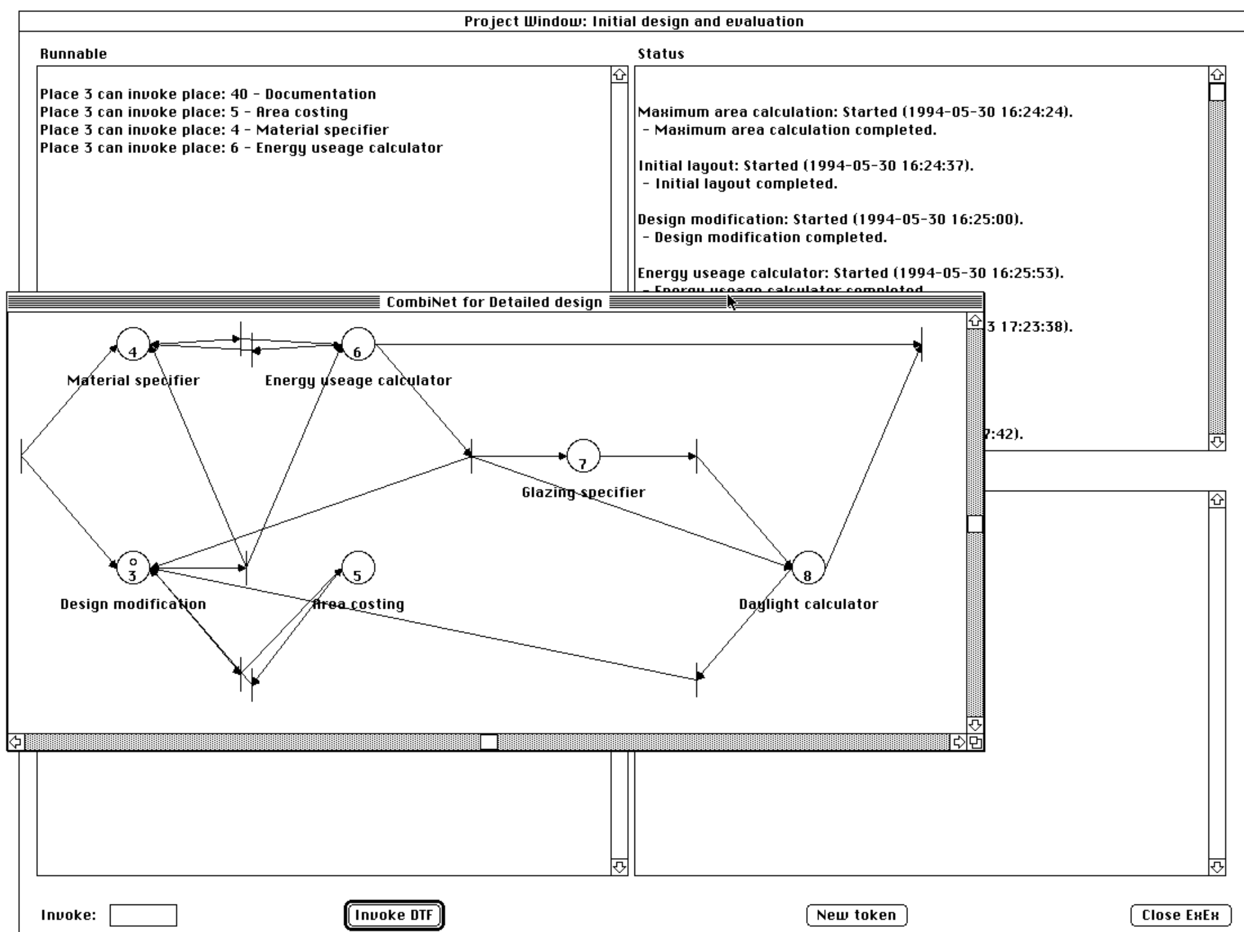


Figure 8.2 An operating flow of control manager

8.5.3 Approach to a flow of control manager

The flow of control manager, described in Chapter 11, is an extension of the system developed by the author during his work on the COMBINE project and provides for the majority of requirements as specified above. As seen in Figure 8.2, the interface provides a visual reference to an actor’s location in a project flow, as well as the calculation of invocable design functions, those which are

stalled, and those which may be required to be reinvoked. Though the current project state is always ascertainable there is no global project state assessment or simulation. There is, however, enough information held in the flow of control manager to make it possible to output data which could be used with standard scheduling simulation software to calculate estimated time to completion. Actors and the project manager are provided with control interfaces of slightly different functionality. Project managers may investigate and control work over the whole project flow, whilst actors are restricted to the flows related to their design roles. As design tool invocation and termination can not be automatically determined by this flow of control manager, each actor is provided with an interface to specify when a particular design tool is working and when it has terminated.

8.6 Project Testing and Implementation Environment Summary

This chapter described the tools required to provide for the testing and implementation of the models developed in the types of environments specified in Chapter 2. The provision of tools and an environment as described in this chapter compliments the development of models in their modelling environment by allowing testing and development to be totally inter-connected. The tools described in this chapter taken as a whole provide an environment which has the majority of the functionality of an actual implementation of an integrated design system. Thus the validity of such an environment can be tested with all design tools and users of a final system prior to a commitment to a large implementation process. The major tools required for this testing and implementation environment are further described in the rest of this thesis. Chapter 9 describes schema instance maintenance in the Snart environment along with connections to EXPRESS and STEP Part 21 data-files, which have instance maintenance systems as well. Chapter 10 describes an implemented mapping system which can maintain data correspondences between many participating users, design tools and integrated data models. Chapter 11 describes a project management tool which controls flows of control between the various users and a project manager, helping to ensuring that the final designed artifact has been created with the input and attention desired by the client and project manager.

Chapter 9

Schema Instance Management

In an integrated design system instances of evolving schemas must be able to be quickly and correctly created and maintained. This aids in the testing and validation of both the schemas and the inter-schema mappings, and helps to reduce schema development time. Tools are needed to instantiate sample building models for evolving schemas, and to navigate the resultant structures to ascertain that they meet their required purpose. To a large extent this requires tools which mimic the functionality of the tools (DTs) that will be part of the final integrated system. This leads to a conflict between the need for tools which can quickly create and maintain complete models for the developing schemas, versus avoidance of effort spent developing tools specialised for a particular schema, which is likely to change, and will be discarded at the end of the development phase. Generic tools are required which are either independent of the developing schemas, or quickly tailorable for specific schemas where independence is not achievable. A set of requirements for this type of tool is detailed below. Four tools of this type which have been developed or used in this project are described in this Chapter.

9.1 Requirements for Instance Management

To enable instance creation and maintenance in the development environment the tools used must satisfy a diverse range of requirements. The main requirements are covered in Section 8.2.1, two smaller requirements are described below:

Genericity: instances for a wide range of schemas need to be created, maintained and navigated.
Tools must be independent of the developing schemas, or easily adaptable. Developers will not wish to spend large amounts of time customising tools for the different schemas, so tools which adapt to specified schema definitions will be of great benefit.

Persistence: as the schema instance test set may become very large (e.g., for a complete building), it is preferable that instance maintenance and navigation tools maintain a persistent representation of the data independently of the ASCII data transfer format (ISO/TC184 1994) used to communicate between design tools in the final system. This will save a significant amount of loading, validation, and saving time when working with instances of a model.

9.2 Instance Management Systems

To work with the modelling environments described in Section 2.2, the schema instance management system has to comprehend EXPRESS schema definitions and data models sent in the associated STEP Part 21 data-file format (ISO/TC184 1994). The tools built for the instance management system in this project all operate in environments which provide parsers for these formats, and allow persistent databases to be created and manipulated according to the resultant specifications and data. These instance management tools meet the majority of requirements in Section 9.1, including instance manipulation and modification in a persistent store, viewing and navigation in both textual and graphical formats, and guaranteed consistency of the persistent store for data being manipulated in multiple views. The four tools created or modified for use in this project for instance creation and navigation are described below.

9.2.1 EPE: an instance construction and browsing system

The EPE system (Amor et al 1995; and as described in Section 3.2), developed to satisfy the requirements of Section 2.2.1, also supports implementation and maintenance. A schema developed in EPE can, at any time, be compiled to an underlying object-oriented form (Snart), and instances of the entities can be created and manipulated. Whole instance models, in the form of STEP data transfer format files, may be loaded into and transferred out of the system. Figure 9.1 shows an instance view of the *ventilation_generation_system* entity modelled in a previous COMBINE IDM.

Instance views consist of visual renditions of entity instances listing their attributes, each attribute's value (where one exists), and the links between entity instances. Navigation through the instance model is via the entity instance links, which are displayed as small boxes in an instance view. These links can be expanded to view the data contained in the linked instances. The amount of data seen for each entity instance can be controlled by the user to present views which show, for example, all instantiated attributes, or all attributes which are references, or all attributes with a basic type, or some combination of types. In Figure 9.1, all attributes of an instance of a *ventilation_generation_system* are shown, including links which are not instantiated (shown as greyed boxes). This figure also shows the single reference to a building contained in the *is_used_by_building* attribute, and all the information in that instance of a building.

The user also has control over the way in which the information in entity views is laid out. This is defined through the use of a special purpose display language. This display language allows for displays which present sets of information in a variety of useful forms, such as bar graphs and tables, and allows templates of a given layout to be created.

In a similar fashion to the requirements for analysis and design views in the schema modelling section, multiple instance views with overlapping information can be created. The elements (e.g., entity attributes) in instance views are editable, allowing changes to be made to the underlying model instance. EPE's instance editor is a specialisation of CernoII, a run-time debugger and visualisation system developed as a companion for SPE (Fenwick et al. 1994; Grundy et al. 1993).

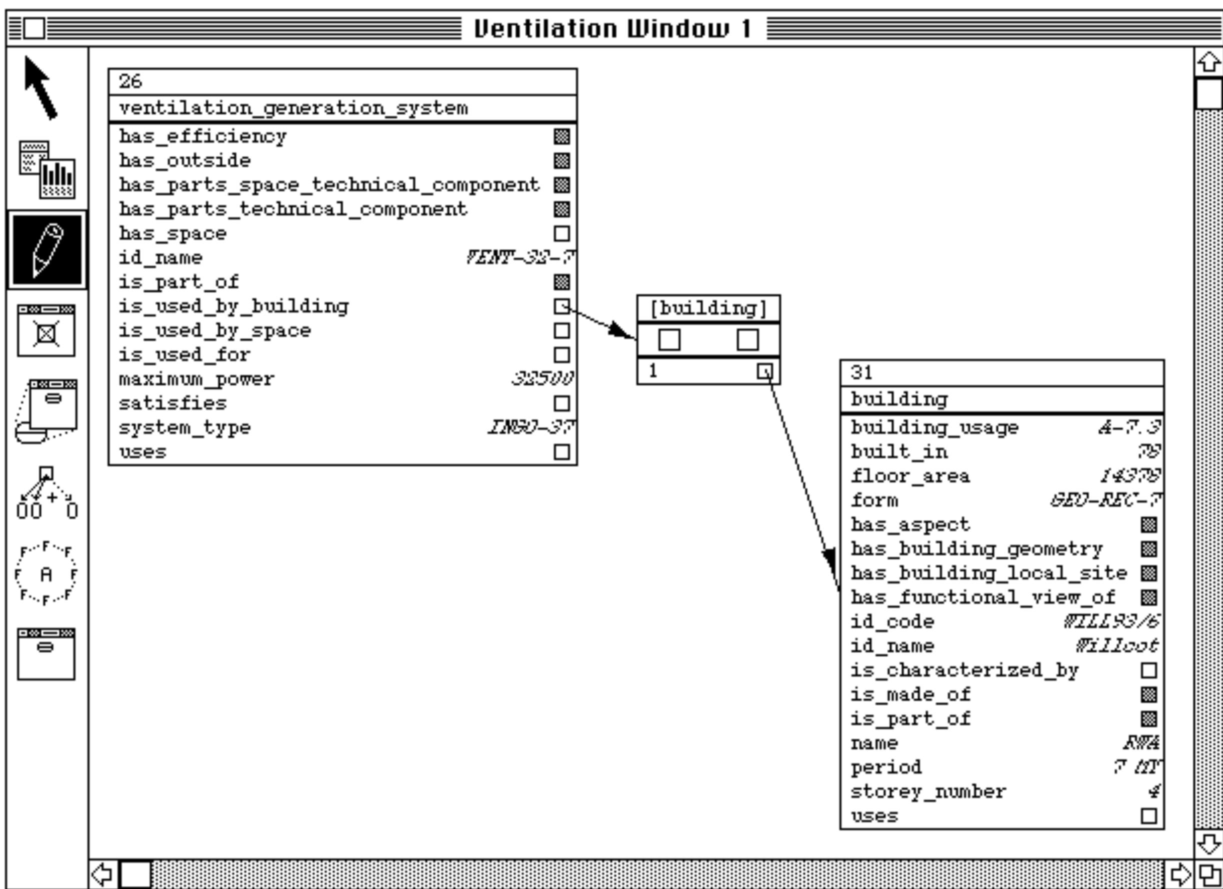


Figure 9.1 Instance viewing and navigation

The original CernoII environment provided most of the functionality initially required for this project. However, extensions were made to specialise the system for the types of model and domain the system had to be tested in. The main extension was to allow instances of a model to be loaded, or saved, in the standard form used in model development in the building domain (i.e., STEP Part 21 data-files, ISO/TC184 1994). Another extension was to add the ability to view and modify facet information for any attribute of an object. This facet information includes the unit of an attribute’s value, along with where the value was derived from, and constraints on its value. A direct manipulation tool was also incorporated which allows objects to be created and automatically linked to an attribute’s value, or into a set, bag or list of object references for an attribute.

While EPE provides direct manipulation of objects and references, along with multiple views and consistency, it does not allow for graphical representation of objects which have an underlying graphical state (e.g., most physical building objects).

9.2.2 InSTEP: a graphical instance browser

The author was not directly involved in the development of InSTEP. It is described here to provide an understanding of the type of tool used to navigate and modify schemas both in this thesis project and in the COMBINE project.

InSTEP is one of several tools developed during the first phase of the COMBINE project to enable participants in the project to manage the developed schemas and instance models. InSTEP enables textual and graphical browsing, navigating and editing of a STEP file according to a specific EXPRESS schema. While this is not by itself remarkable, InSTEP is unique in that it provides a graphical view of any model which is based on a schema which uses graphical entities of known types. Figure 9.2 shows both the graphical and textual view of a portion of a building. In this case InSTEP recognises entities that have a predefined geometrical representation, and will render them in a graphical view which gives the user the ability to navigate through the instance model using either textual links or graphical links.

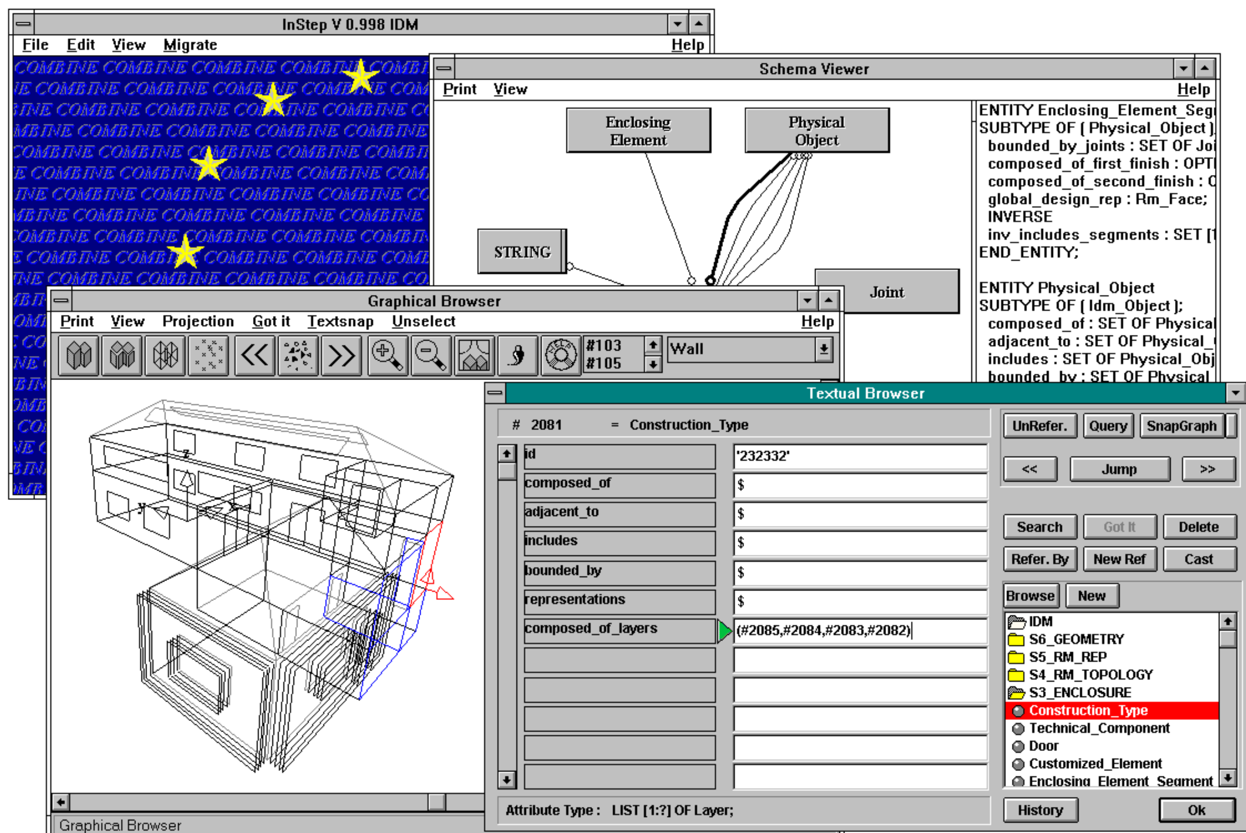


Figure 9.2 InSTEP's graphical and textual views

The textual browsing and editing of the model is performed on an object by object basis, though the user may follow relationships between objects in a hypertext like fashion. The graphical view displays a selected portion of the model, and the user can set the level of navigation and selection at which the tool will operate. In Figure 9.2 the user is navigating through the graphical model at a wall level, though other object types are displayed for reference.

In COMBINE this tool provides feedback to the conceptual task. Once a conceptual schema has reached a more or less stable version, DT teams and anyone who has a direct interest in the IDM are encouraged to browse instances of the schema and suggest modifications. In this thesis InSTEP was mainly used to ensure the validity of graphical representations in STEP data-files being developed for demonstration purposes.

9.2.3 SnartQuery and the ObjectViewer

The Snart language and development environment have been extended by the author to provide query and visualisation tools, as an adjunct to the environments described above. The query language allows a data model to be searched for objects matching a general query term. The ObjectViewer allows a dynamically updated view of an object to be created, and includes buttons for invoking object methods or allowing object modification.

SnartQuery allows a named data store to be searched, and for sets of objects matching several classes to be returned (i.e., response to a query is not limited to a single class type). The query can consist of attribute references, object ID references, referenced attributes (i.e., following pointer chains) and method calls which can be treated in a functional manner (i.e., either true or false or returning a single value). Compound query terms can be constructed with full arithmetic calculation, including Prolog predicates and aggregating functions (i.e., sum, count, minimum, maximum, average). List and array elements can also be accessed individually in the query. The returned list of object tuples matching the classes specified in the query can be utilised to invoke the ObjectViewer or the EPE view navigation environment. Invoking SnartQuery can be performed by accessing a menu item in the LPA Prolog environment or through an equivalent predicate call. Figure 9.3 shows both a query dialogue and the query result in the working window. A complete description of the query language can be found in Appendix C.2.

The ObjectViewer provides a general purpose debugging tool for the Snart language as well as a data model viewing and navigation tool. The basic premise of the ObjectViewer is that it spies upon an object, and always reflects the current state of the object. It is more powerful than that, as it also allows attributes to be modified, and methods of the class to be invoked, from the object view window. The ObjectViewer system allows a layout template for attributes and methods of a class to be specified when the class is defined, or by default it will create a view with all attributes and a selected set of methods (see Figure 9.4 for a default view for an *idm_hip_roof* class). Buttons and pull-down menus can be specified in the class template, allowing class methods to be

invoked from the object view independently from the application that created and is manipulating the object. An object view can be created by accessing an extended menu item in the LPA Prolog environment, or through a hot-key. By default, the view command searches the currently highlighted text and creates an object view for every object identifier found in the highlighted text. A complete description of the ObjectViewer can be found in Appendix C.5.

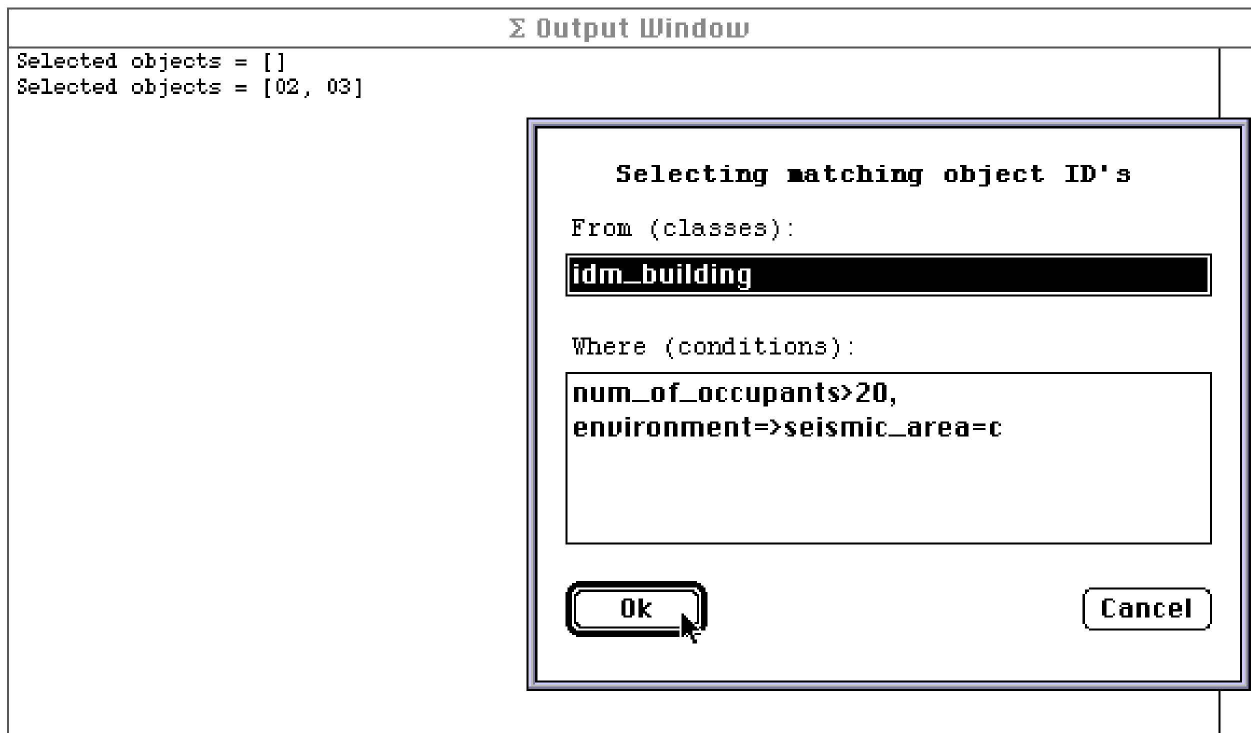


Figure 9.3 A SnartQuery dialogue and result

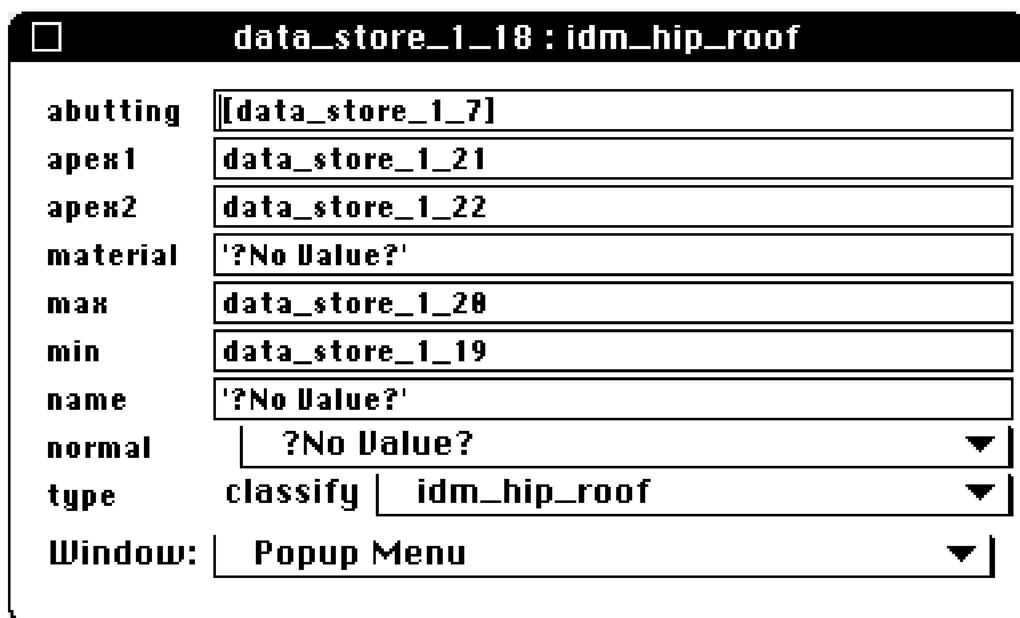


Figure 9.4 A default ObjectViewer object layout

While the ObjectViewer provides direct manipulation of objects and references, along with design tool independent method invocation, it is exactly like EPE in that it does not allow for graphical representation of objects which have an underlying graphical state. The advantage of this tool over

EPE is that it is built directly into the Snart language as a very small and efficient sub-system, unlike the EPE system which is a large system built using Snart, creating a large overhead to have loaded and operating for many applications. The SnartQuery language is also built into Snart and the result of a query is in the form required by the ObjectViewer to enable the identified objects to be viewed.

9.2.4 Re*f*lex: an object-oriented CAD system

The commercial OO-CAD system Re*f*lex (Re*f*lex 1996) only became available at a very late stage of the project, but was incorporated as it offers many features which make it highly suitable as an instance creation and manipulation system. Re*f*lex provides the majority of features found in normal CAD systems, with the benefit of a central object-based model. This means that objects with a graphical representation can be quickly created and assembled with other objects. The Re*f*lex system also provides an object and dialogue specification language. This means that new libraries of object types can be created easily. Appendix F.1.3 describes the translator built on top of the EXPRESS parser which allows any schema to be transformed into the required form for a Re*f*lex library. Because Re*f*lex's main domain is buildings, libraries of intelligent object definitions are a standard part. With a small modification of the translated EXPRESS schema it is possible for the library of object definitions to inherit the full specification of the existing Re*f*lex objects. This provides objects which have the complete graphical representation and checking methods as in Re*f*lex, but with the attributes defined in the EXPRESS schema. Figure 9.5 shows a model being developed in Re*f*lex with object definitions translated from an EXPRESS schema. Object definitions can also include default values, which means that complete model definitions can be defined very quickly simply by placing objects in a 2D or 3D Re*f*lex view.

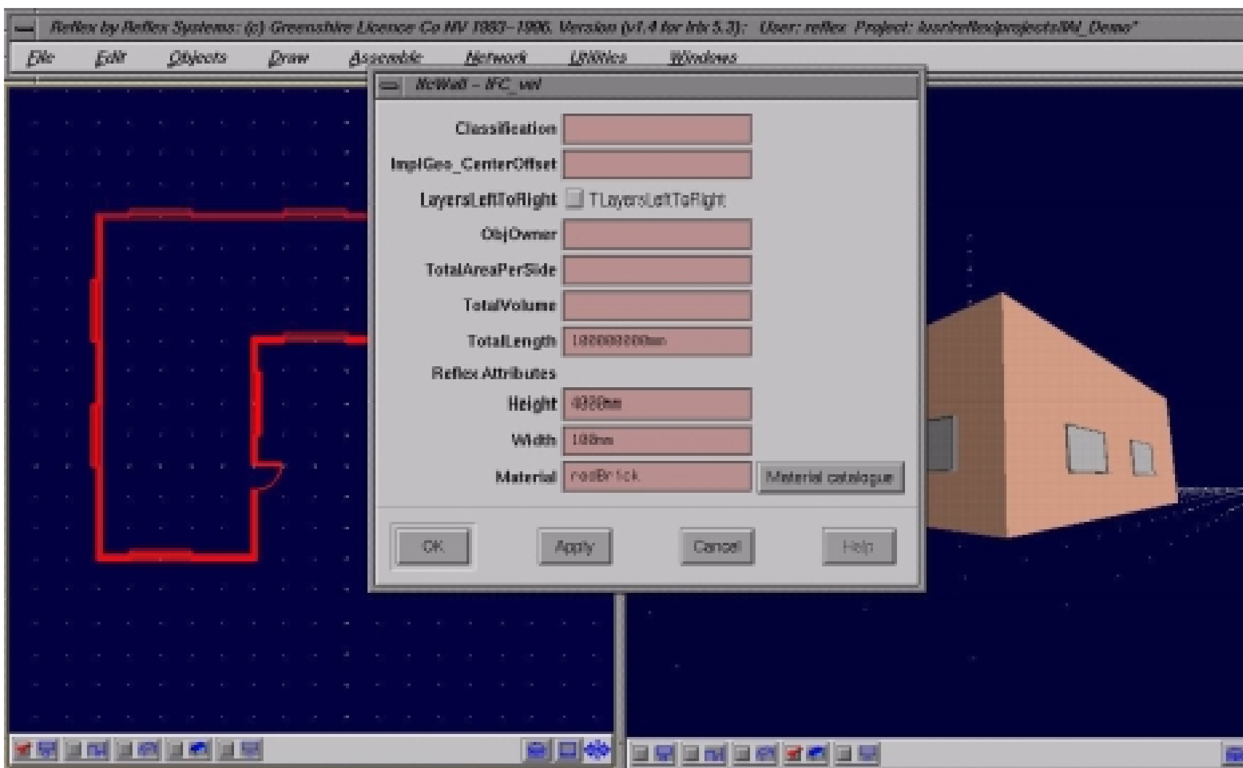


Figure 9.5 Re*f*lex's multiple graphical views, along with an object's attribute dialogue

STEP data-file models can be imported into, or exported from, Re*f*lex. Importing models with geometric specifications requires extra coding, as the internal representation of an object's geometry in Re*f*lex is different from that specified by all other schemas. In the context of this thesis, Re*f*lex is best suited to the initial creation of a model for a particular schema, which can then be exported for use with the other tools described above. Another difficulty is that the relationships between objects that Re*f*lex maintains are usually different from those required in the loaded schemas. To handle this, either the Re*f*lex object library definitions need to be augmented with code which maintains relationships as defined in the schema, or these relationships must be created and maintained in another tool. The latter method was the one used in this project.

9.3 Appraisal of Schema Instance Management

The set of tools described in Section 9.2 meet the majority of requirements defined in Section 9.1. All of the tools in their standard form can work with any schema definition, requiring no tailoring. However, most can be enhanced by providing further information about the schema definitions. In the case of Re*f*lex this provides objects with their most suitable graphical representation and behaviour. With InSTEP it allows a model's objects to be rendered graphically if the geometric representation is defined. Multiple views of models are supported, offering geometrical representations, textual representations and graphical layouts of text using EPE. These views provide many ways of navigating around instances or through following references, traversing geometric connections between objects, or direct queries to identify required objects. All tools reflect the state of the central model in their views, and EPE allows multiple overlapping views of data to be displayed and manipulated.

The only aspect which is not closely managed by any of these tools is the correspondence with an evolving schema. All of the tools except InSTEP will allow an internally stored model to be viewed and manipulated with an updated schema definition. However, they will all disregard data values in attributes which don't exist in the new schema, and there will be no checking that attributes which do transfer across match the type now specified for the attribute. All tools make the assumption that in order to move data models to a new version of a schema, a mapping is specified between the two schema versions and the data mapped across before being reloaded into the instance management tools. An extension to the integrated framework described in this thesis could alleviate this problem. The EPE tool already tracks all modifications to an evolving schema, a small extension to this tool would enable it to aggregate these modifications together to form the bases of a VML mapping between two states of the schema. Though not all mappings could be captured automatically it would be possible to represent the vast majority of changes with this extension.

Chapter 10

Mapping Controller

The development of a generic mapping controller has been a neglected area of research amongst the components of an integrated building design system. To date, the majority of integrated building design systems have employed hand-coded translators to move data from their IDM to the design tools used. This has been due partly to the lack of a language which allows a high-level specification of mappings between models, though VML, and other mapping languages, have now partly solved that problem.

Given a mapping specification between two schemas, there can be many implementations which will achieve the correct movement of data between the models. In many cases the style of implementation will be dependent upon the type of IBDS being developed. For example, an IBDS utilising many KBS's could require a very interactive connection between the IDM and each design tool. There are three levels of control that can be offered by a mapping controller for a particular mapping. These are:

Complete: the whole database is mapped every time data needs to be passed to or from the IDM, destructively overwriting the data in the data-store it is being mapped to. For this type of control no information about previous mappings needs be maintained, as the whole mapping is re-evaluated every time it is required. This is the scenario put forward in the ISO-STEP standard, where a single file representing the informational content of an AP is passed to the design tool requiring information (and the same to pass information back). While this is the simplest case to implement, it is also the most expensive computationally as the whole database must be mapped every time a design tool is invoked.

Modified: only the data changed since the previous mapping is passed to or from the IDM, where it merges with the previously mapped data. This method provides the same result as a complete mapping, but as only modified data is mapped it requires less computation. This

level of control assumes that there is a method of determining the changes between the model at the time of the previous mapping and the current state of the model. Information about previous mappings may be kept for this type of control to simplify the mapping of modifications, but is not absolutely necessary.

Interactive: an adaptation of the *modified* mapping controller where individual changes are mapped as they are recorded. As with the *modified* mapping controller, only modified data is mapped to or from the IDM to merge with the existing data. This level of control allows for very interactive design tools to be used in an IBDS, especially where the data requirements may not be ascertainable when starting the design tool (e.g., a KBS whose input is based on what it has previously seen as well as what its user is asking for). However, this level of mapping can lead to long periods of inconsistency in the models being mapped to, while information is entered by the actor using the design tool. The length of time over which actor changes are made is likely to be orders of magnitude greater than the time taken to map a set of changes as in the complete or modified scenarios above.

Though the result of a mapping using any of these three strategies would be identical, the time taken to achieve it, and the structure of the mapping controller required to implement it, will be very different. In the mapping controller described in this chapter, the second two control strategies (modified and interactive) are implemented. This is mainly due to the fact that they offer the greatest range of interactional possibilities for design tools in an IBDS. Implementation of a partial update mapping controller also allows modes of interaction for collaborative designers to be examined, as well as investigating strategies for tracking previous data mappings to reduce the computational workload in mapping data.

The remainder of this chapter details the working of a mapping controller. This starts with a consideration of what model changes need to be tracked and how this was achieved. The process of mapping is described, starting with considerations of ensuring the correct state of the data-stores to perform the mapping, and then the details of matching up objects and solving equations are defined.

10.1 Data-Store Modification Records

For a mapping controller to be able to handle partial mappings it must be able to determine the changes to a data-store between any two of its previously published states. When maintaining a mapping between two data-stores the mapping controller also needs to know when the state of one of the data-stores has changed. There are many ways that each of these requirements could be realised. In this section an implementation of persistent data-stores in Snart is detailed which provides all the information required by a mapping controller.

All versions of Snart, the implementation language of this system, have had an implementation of data-stores of varying complexity added to them (Grundy 1993). In its simplest form a data-store provides a named space inside which objects can be created. The most complex implementation of a data-store in Snart allows all objects created in a named space to be identified, and provides hooks to add data-store dependent functionality onto the creation and deletion of objects in the space.

These data stores have also been specialised to provide persistency. The persistent data-store developed for this project allows all the objects created in a space to be saved to a file, the name of which is specified when the space is created. It also allows the reloading of the persistent data-stores, with object identifier renaming if required, into the running Snart environment (see Appendix C for a complete description of object spaces and persistent stores).

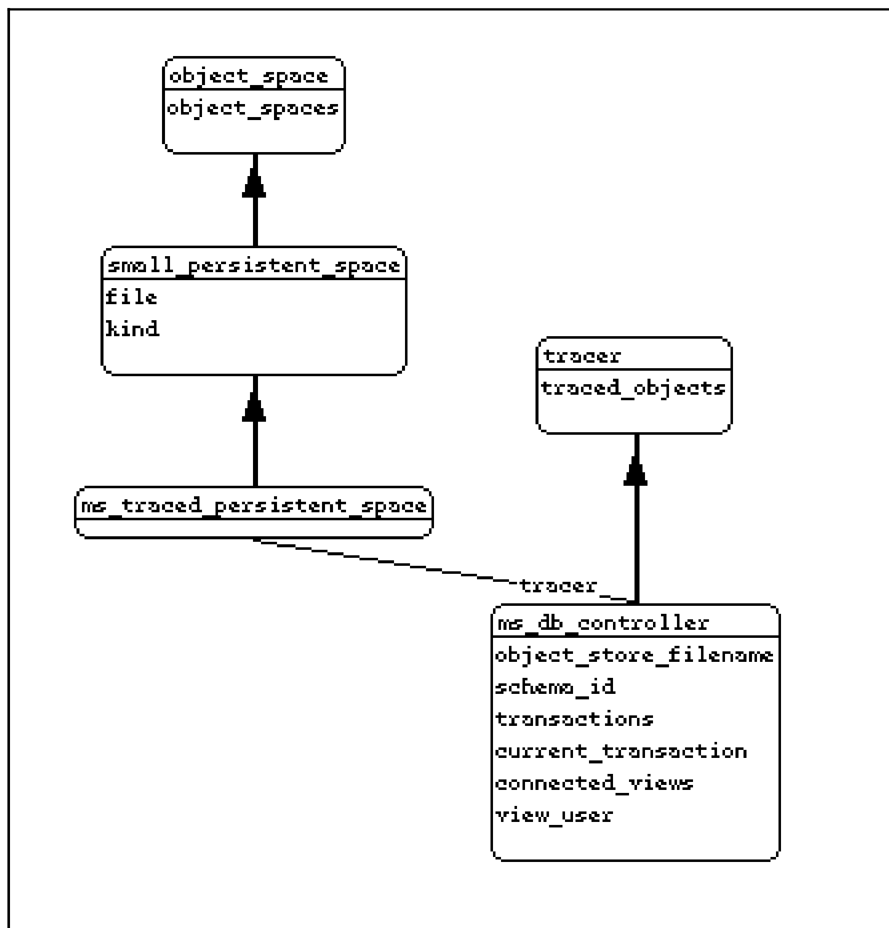


Figure 10.1 Structure of the traced persistent space in Snart

To provide the functionality required for a mapping controller, this persistent data-store allows all objects created in a data-store to be traced. This was achieved mainly by utilising the functionality of a tracer class, provided in Snart, specialised for use as a controller for the data-store (see the class hierarchies shown in Figure 10.1). Using this arrangement, all object creations are caught by the *ms_traced_persistent_space* object which ensures that they are traced by the *ms_db_controller* object for that data-store. All attribute value instantiations and method calls are caught by the

ms_db_controller object and, if necessary, recorded.

In each data-store there is a single *ms_db_controller* object which catches all events pertaining to traced objects. The *ms_db_controller* object collates modifications into a structure named a transaction, which contains all work performed for a particular task (e.g., specifying the space layout or completing a change request). The *ms_db_controller* has a notion of *working* and *finished* transactions. There may be many *working* transactions open at any time, one of which will be a default transaction in which modifications are recorded, unless specified otherwise. As each transaction collates information on a particular task in a project, having several working transactions allows an actor to manage several tasks at once, moving between them as ideas, deadlines or project managers dictate.

When the *ms_db_controller* is notified of an event that should be recorded, a new modification number is assigned (a monotonically increasing number) and a record of the event is stored in the default transaction. The raw events which are stored in a transaction are:

create_object:	creation of an object, with optional parameters
delete_object:	deletion of an object
add_value:	initial specification of a value of an attribute
change_value:	modification of a value of an attribute
delete_value:	deletion of a value of an attribute
invoked_method:	calling of a method of an object, with optional parameters
add_facet:	initial specification of a value of a facet of an attribute
change_facet:	modification of a value of a facet of an attribute
delete_facet:	deletion of a value of a facet of an attribute

The record of these events contains previous values, where these are known, so that a modification can be reversed (undone) or, at the transaction level, a whole transaction may be reversed. The transaction system also contains a notion of an aggregate transaction, which allows multiple transactions to be collated together and subsequently treated as a single transaction. These services are available to all design tools using the persistent traced space, though few have been developed to make use of them (e.g., implementing an undo/redo function inside a design tool).

When an actor has completed a transaction, a label for the transaction must be supplied. This label allows the actor to specify, in a human readable form, what work was undertaken during the course of the transaction. It is this label which will be used, initially by other actors, to deduce what work was completed in a given transaction. This label is used to form a unique identification for each transaction (See Figure 10.3 for examples of transaction labels) by appending a unique, to the data-store, transaction number (a monotonically increasing number) along with the filename of the data-store. This label is guaranteed to be unique for all data-stores on a system, and with a little padding out (e.g., written in URL form) could be unique over any number of machines.

When a transaction is complete, as signalled to the *ms_db_controller*, the raw event data is processed until there is at most one record for each object that was modified. In this form, multiple changes to a single attribute and its facets are stored as one item. Objects which were created, modified and then deleted inside the transaction have no record in the processed form. This compaction introduces constraints on the type of method that can be mapped between systems utilising a transaction-based mapping. Mapped methods are constrained to have no side-effects on other related objects. This constraint is not enforced, so the system will attempt to map any method specified in a mapping. Where methods have side-effects this could lead to failed method calls due to non-existent objects. Three types of processed events are stored:

create:	lists the modified attributes and method calls of the object which has been created
modify:	lists the modified attributes and method calls of the object which has been changed
delete:	records that the object has been deleted

Although the processing collates modifications which could be widely dispersed in the transaction, the initial ordering of object use remains constant. Therefore, the final ordering of processed modifications reflects the order in which object creation, the first modification of an object, and object deletion, were encountered.

When the processed form of a completed transaction has been calculated, the *ms_db_controller* informs all objects which have registered an interest in the data-store (usually mapping controllers) that a new transaction has been completed in the data-store. Interactive mapping controllers are also informed of each modification which occurs in the data-store.

As the description of the modification records might indicate, the number (and space requirements) of the stored modification records can build up very quickly, especially in an object-oriented application which requires much computation, and hence many method calls. To manage this storage requirement in the limited application space of the Prolog development environment a menu item allows the storage of these method call records to be turned on or off.

This data-store implementation provides the notification of modifications for each design tool and the IDM used in the IBDS. In this IBDS every design tool must interface through a data-store of this form to gain, or pass back, information from the IDM. This means that a mapped model of the IDM exists for every design tool used in the IBDS. In the examples used throughout this thesis most of the design tools are implemented in the Snart environment (i.e., PlanEntry, ThermalDesigner, FaceEditor). However, the VISION-3D design tool is a stand-alone tool which requires a data-file to be read in to supply the information for the tool to operate. Hence, the use of both independent and tightly coupled design tools is catered for using the described data-store implementation.

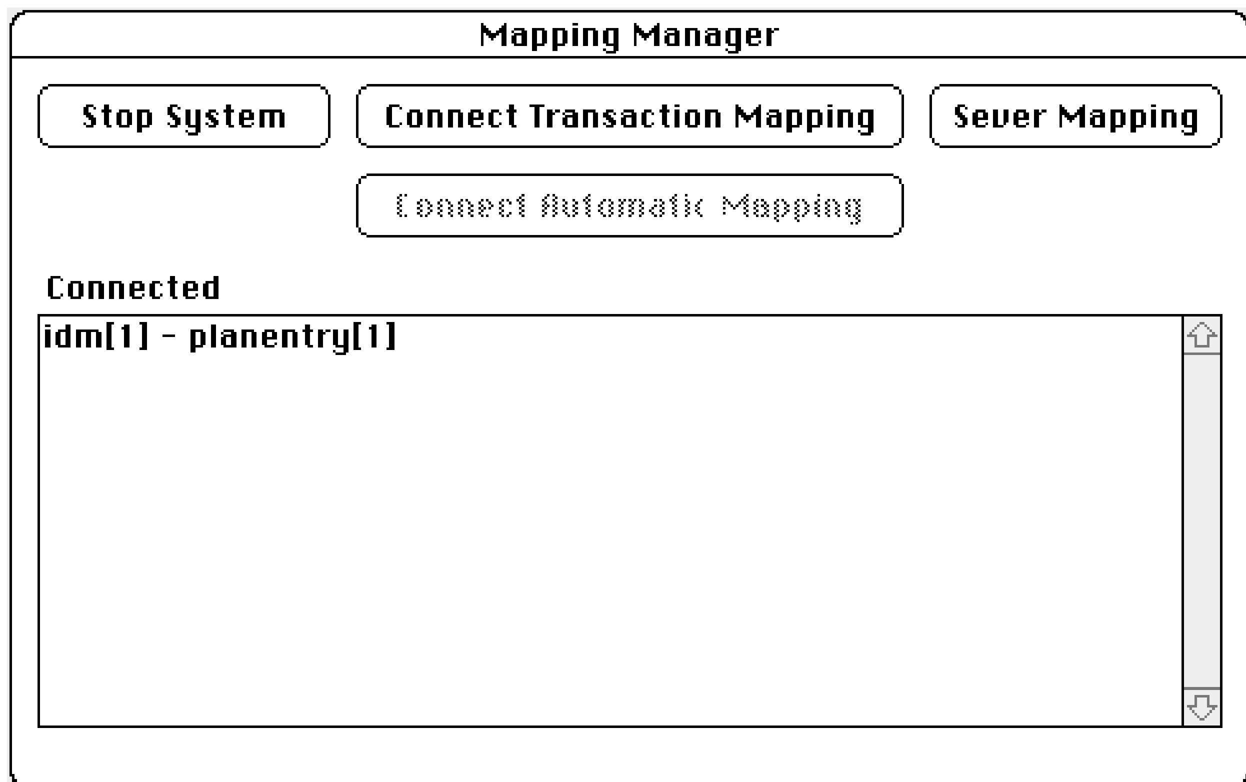


Figure 10.2 An actor's mapping controller interface

10.2 The Mapping Controller

The actor interface to the mapping system must allow the actor to manage all mappings between design tools that they require to perform their design role in a project window. To enable this each actor has a mapping controller interface as shown in Figure 10.2. This interface allows the actor to manage a design session where data is to be transferred between the IDM and a range of design tools. The range of functions available through the interface are:

Start System: initialises the mapping controller and allows the specification of a persistent data-store for the storage of information about the mappings. If an existing data-store is specified, the mapping state that was saved in it is restored and actors can continue from where they left off. If a new data-store is specified no mappings are loaded initially. When the mapping controller has been initialised the *Start System* button changes into the *Stop System* button as described below.

Stop System: allows the design session to be temporarily suspended. All data pertaining to the current set of mappings (and any which were previously connected and then severed) are saved to the persistent data-store.

Connect Transaction Mapping: starts up a transaction-based mapping between two data-stores which contain data models as specified during the creation of the mapping database as described in Chapter 6. The actor identifies the two data-stores to be used in the mapping. These can be existing data-stores or new data-stores, in which case a new file is created for the data-store. A mapping manager is created for each mapping which is connected into the

system. See Figure 10.3 for a transaction-based mapping manager interface.

Connect Automatic Mapping: starts up an automatic mapping between two data-stores which contain data models as specified during the creation of the mapping database as described in Chapter 6. The actor identifies the two data-stores to be used in the mapping. These can be existing data-stores or new data-stores, in which case a new file is created for the data-store. A mapping manager is created for each mapping which is connected into the system. See Figure 10.4 for an automatic mapping manager.

Sever Mapping: the actor can select one of the mappings which has been started up and remove it from the set of active mappings. When a mapping is severed the current mapping state is stored in the mapping controller data-store in case the mapping is ever required again.

When a mapping manager is established it informs the two data-stores it is connecting that it is interested in all transactions that are completed, and for the automatic manager all modifications as well. The data-store records the details of the interested mapping manager and adds it to its list of objects interested in its state. Several mappings may reference the same data-store and see all the changes made to that data-store, thus establishing a central IDM with multiple views for the IBDS. A collaborative environment with multiple concurrent actors is envisaged for the IBDS. However, because a Macintosh environment lacking a client-server-based object-oriented database has been utilised, the effort put into considerations of concurrency has been minimal. In the single machine environment, data-stores are locked only when a mapping is being applied to them. It is envisaged that this scheme could be used by an object-oriented database serving several actors on different machines. The only other capability the object-oriented database would require is the ability to pass a message through to the mapping manager objects on each machine which need to be informed of new transactions or modifications (a CORBA-like environment (Otte et al. 1996) would support this).

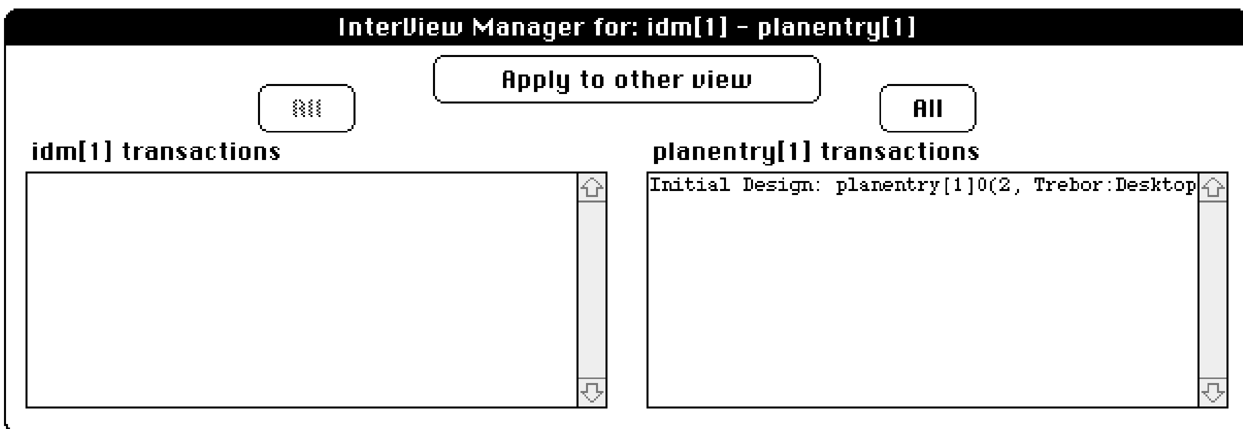


Figure 10.3 A transaction-based mapping manager

The mapping controller operates in its own persistent data-store, along with all the mapping managers that it oversees. Therefore, the set of loaded mapping managers and their state can be retained between sessions with the mapping controller. Having a persistent data-store for the mapping managers also means that all previous mappings between the two data-stores it manages

are retained and all the dynamically created indices, described in detail in later sections, can be saved between sessions. This ensures that all the work performed in determining previous mappings does not have to be duplicated when a new transaction is mapped in a later session.

10.2.1 Transaction-based mapping manager

When a transaction-based mapping manager is connected, it interrogates the two data-stores it is connecting to ascertain what transactions they have recorded in them. A difference list is constructed, and presented to the actor in the interface shown in Figure 10.3. The data-store's list of transactions can be used to ascertain which common transactions have been applied to each data-store as when a mapping is performed the full transaction name (actor label, data-store name and number) of the originator of the transaction is used to label the changes made in the opposing data-store. Using the two lists of outstanding transactions the actor determines which transactions to map between the two data-stores, when they should be mapped, and, within certain constraints, in which order they are mapped. As new transactions are completed in each data-store they are notified to the mapping manager and are added to the list of outstanding transactions for the particular data-store.

If one of the schemas in the mapping being used to maintain the connections between the data-stores is specified as *read_only* (see Chapter 5 for the different types of mapping available) none of the transactions in that data-store will be displayed in the mapping manager. When an actor selects one or more transactions to be mapped through to the other data-store, the mapping type is examined to check whether the other store is categorised as *integrated*, in which case a check is made to ensure that all outstanding transactions from the *integrated* data-store have been applied. This ensures that data-stores labelled as *integrated* are only updated by mappings from data-stores which are consistent with the *integrated* data-store's state. If all outstanding transactions from the *integrated* data-store have been applied then the mapping is allowed to proceed to the next stage of consideration, otherwise it is aborted with an appropriate message to the actor. When a set of transactions are organised to be mapped they are examined to ensure that there is no other outstanding transaction from the same data-store which must be mapped through before the selected one; for example, where a selected transaction depends upon a modification in an earlier transaction, such as the creation of an object for which an attribute has been specified. If there is no conflict the transaction mappings are started, otherwise an appropriate message is presented to the actor and the mapping aborted.

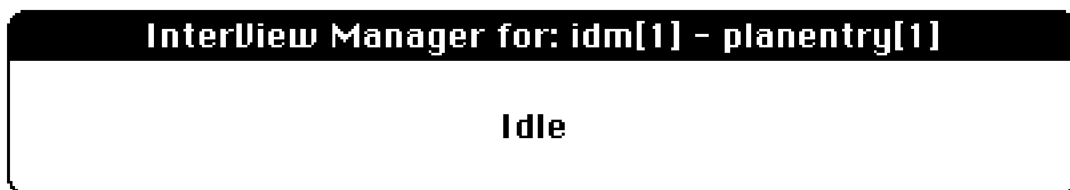


Figure 10.4 An automatic mapping manager

10.2.2 Automatic mapping manager

An automatic mapping manager can only operate between two data-stores which have had the same transactions applied to them. When an automatic mapping manager is connected, it interrogates the two data-stores it is connecting to ascertain what transactions they have recorded in them. If the two lists are different it will not allow a connection. When the conditions for an automatic mapping manager are met, an interface as in Figure 10.4 is created. This provides no functionality to the actor, but provides information about when data is being mapped between the two data-stores. As well as mapping individual modifications between the two data-stores, the automatic mapper recognises signals for a completed transaction and collates the corresponding modifications in the mapped store to become the modifications corresponding to the newly named transaction.

The mapping controller may be used to start up a mix of mapping managers, as transaction-based and automatic mapping managers readily coexist. However, the changes propagated by one type of mapping manager may be seen slightly differently by the other type of mapping manager. For example, if an IDM is connected to both transaction-based design tool models and automatically mapped design tools, all automatically mapped modifications made to the IDM will not be seen by the transaction-based design tools until an end of transaction is mapped through, at which time it will see the collated set of atomic modifications. The case of a full transaction mapped to the IDM will be seen by an automatic mapping manager as a large set of individual modifications queued for processing.

10.3 Performing a Mapping

The sections on data-stores and the mapping controller have laid the groundwork required to describe the actual mapping process. The reader should now have an idea of how modifications are gathered in a data-store and the compressed form in which they are presented to the mapping system. The reader should also understand the control options available to the actor and under what conditions they allow a mapping to proceed. In the following sections the process followed for a transaction-based mapping is described. It is almost identical to that followed for an automatic mapping.

To describe how a mapping is performed, a top down approach is adopted, looking at some general setup work, then the various cases presented by different types of objects, leading down to how individual equations are treated in the mapping system.

10.3.1 In preparation to map

Before a mapping can proceed it is necessary to ensure that the data-stores are in the state specified by the transactions being mapped (see Table 10.1 for the pseudo-code). To ensure this, all current work on the data-store and all outstanding transactions which are not being mapped must be reversed. This is possible because the persistent data-store controller tracks all transactions, both *working* and *finished*, and the transaction records all modifications (and the order in which they were performed). To bring the data-store into its correct state all *working* and unmapped transactions are rolled-back. Transactions to be mapped are checked against working and unmapped transactions to ensure that no dependencies exist between them. If the transactions to be mapped are dependant upon other unmapped data then the user in informed of the dependant transactions and the mapping is terminated. The roll-back leaves the data-store in the state it was at the completion of the specified transactions. As the data-store is locked during the mapping, rolling-back the data-store does not affect any running applications, apart from stalling them. At the end of the mapping the data-store is returned to its previous working state by rolling-forward all the rolled-back modifications.

```
performing a mapping
        determine outstanding transactions (working and unmapped)
        calculate dependencies between outstanding transactions and transactions to map
        if dependencies exist
        then
                cancel mapping and inform user
        else
                roll-back all outstanding transactions
                perform mappings
                roll-forward all rolled-back transactions
```

Table 10.1 Pseudo-code for performing a mapping

```
establish a mapping
        determine all inter_class definitions purely for the store being mapped to
        for each inter_class found above
                if object of specified class exists matching the invariants
                then
                        do nothing
                else
                        create object of named type
                        solve all initialisers
```

Table 10.2 Pseudo-code for establishing a mapping

10.3.2 The first mapping between two stores

The first time a mapping is performed to a particular data-store, some objects may need to be created in the data-store (see Table 10.2 for the pseudo-code). In a VML mapping this is specified by an *inter_class* which only lists classes for one of the schemas being mapped between. If this is the first known mapping to the particular data-store, the mapping system searches the mapping database for any *inter_class* definitions of this form. If any are found the data-store is searched for

objects of the class specified. If none are found new objects of the specified type are created and the equations, usually initialisers, calculated as defined later in this chapter.

```
perform mappings
        for each aggregated modification to be mapped
                case aggregated mapping of
                create:
                        find inter_class definitions referencing class of created object
                        for each inter_class identified
                                determine initial object groups
                                generate object combinations for the inter_class
                                for each remaining combination
                                        if created object is grouped in inter_class header
                                        then
                                                match with existing mappings of this type
                                                re-evaluate grouped objects
                                                re-evaluate invariants if object's class involved
                                                if invariants are violated
                                                then
                                                        dissolve existing mapping
                                                else
                                                        re-calculate affected equivalences
                                        else
                                                create required objects in other data store
                                                apply initialisers
                                                solve equivalences
                modify:
                        { see Table 10.11 }
                delete:
                        determine all existing affected mappings
                        for each affected mapping
                        if object's class is grouped in inter_class header
                        then
                                re-evaluate grouped objects
                                re-evaluate invariants if object's class involved
                                if invariants are violated
                                then
                                        dissolve existing mapping
                                else
                                        re-calculate affected equivalences
                        else
                                { see Table 10.10 }
                        find all other inter_class definitions with object's class grouped
                        for each inter_class identified
                                { perform mapping as for create above }
```

Table 10.3 Pseudo-code for performing mappings

10.3.3 Consideration of modification types

The three classified forms of object modification obtained from the data-store (create, modify and delete) are handled separately in the mapping system, though the create and modify forms are very similar. A general outline of what happens for each of these types of object modification is presented initially and then more detail will be presented on important aspects of their handling (see Table 10.3 for the pseudo-code).

create: all *inter_class* definitions which reference the class of the new object are identified. All combinations of the classes in the header of the *inter_class* specifications are created from objects in the data-store being mapped from, but using the created object as a placeholder for the class in the header. Then all the invariants pertaining to that side of the mapping are applied to the combinations to determine if any of them are applicable. If any pass the invariants then all the initialisers and equations are solved as described later in this section.

modify: all existing mappings which use the modified object are notified of a change to the object and must re-calculate any changed equations. If the modification affects invariants of *inter_class* definitions then they must be re-evaluated as for a newly created object, i.e., it may be possible to create a mapping with the new values, or it may be necessary to dissolve a mapping as for delete below.

delete: all mappings referencing the deleted object are notified and must re-calculate their equations. If the object was part of the header of an *inter_class* then the mapping is dissolved and any objects created in the mapped to data-store due to the now deleted object are themselves deleted.

10.3.4 Determining combinations of objects from an *inter_class* header

When instigating a mapping between two data-stores, or propagating changes between data-stores, it is necessary to determine which objects are going to be used with each *inter_class* definition. The first step towards this is determining the combinations of objects to be tested against the invariants for each *inter_class* definition (see Table 10.5 for the pseudo-code). This is calculated by examining each class in the header of an *inter_class* and returning a list of possibly matching objects. The only exception is for the class of the newly created, or modified, object where the returned list contains just the newly created, or modified, object (unless it is denoted as grouped). If the class of the newly created, or modified, object is in the header as grouped then the returned list is a list of a list of all objects matching the class defined. Table 10.4 shows some combinations of headers and the resulting lists of objects returned. CK is the class of the known (newly created or modified) object, C1 and C2 are arbitrary classes. The resultant object list is either a single object or a number of objects (shown as 1..n, or m or p), the O denotes an object and is subscripted with the class type from the *inter_class* header.

As the combinations of all objects in these lists grows exponentially based on the number of objects in each list, a pruning algorithm is employed to help minimise the number of combinations produced (see Table 10.6 for the pseudo-code). Note that in the worst case, where there are no invariants, the number of combinations will equal the cross-product of all objects in each list. The pruning algorithm about to be presented is not optimal computationally (the author is aware of optimal methods from relational database work, Ullman 1982), but is used to prototype the mapping manager implementation and is a workable tradeoff between programming time available

(minimal), program intricacy (simple), and efficiency (moderate).

inter_class header	Resultant object lists
[CK]	[[O_{CK}]]
[CK, C1]	[[O_{CK}], [$O1_{C1}$..On_{C1}]]
[CK, C1, C2]	[[O_{CK}], [$O1_{C1}$..On_{C1}], [$O1_{C2}$..Om_{C2}]]
[C1, CK]	[[$O1_{C1}$..On_{C1}], [O_{CK}]]
[CK, group(C1)]	[[O_{CK}], [[$O1_{C1}$..On_{C1}]]]
[CK, group(C1), C2]	[[O_{CK}], [[$O1_{C1}$..On_{C1}]], [$O1_{C2}$..Om_{C2}]]
[group(CK)]	[[[$O1_{CK}$..Op_{CK}]]]
[C1, group(CK)]	[[$O1_{C1}$..On_{C1}], [[$O1_{CK}$..Op_{CK}]]]

Table 10.4 Examples of *inter_class* headers and resultant object lists

```
determine initial object groups
        for each class in inter_class header
                if class is grouped
                then
                        find all objects of named type
                        return as a list of a list
                else if class is of key object
                        then
                                return the key object in a list
                        else
                                find all objects of the named type, return in a list
```

Table 10.5 Pseudo-code for determining initial object groups

To prune the number of object combinations at each stage the following algorithm is used. Starting with the last list of objects, the list is reduced by applying all invariants which apply purely to those objects. The next set of objects is similarly reduced, then the cross-product of the two sets is obtained. This combination is reduced by applying any invariants which pertain just to the two classes in the cross-product. Then the next set of objects is reduced by applying all invariants which apply purely to those objects. Then the cross-product of the reduced set and the previous cross-product is obtained. This combination is reduced by applying any invariants which pertain just to the classes in the new cross-product. This is repeated until the first set of objects has been incorporated into the cumulative cross-product. The result is a list of lists of objects which match all the invariants in the *inter_class* definition. Each of these lists can be used to create a mapping to the other data-store. In the worst case, where there are no invariants, the number of combinations is equal to the cross-product of all objects of all classes. However, assuming this is a valid specification of a mapping then it must be handled by the mapping system.

```
generate object combinations for the inter_class
    initialise current result set to empty
    for each set of objects relating to a class in the inter_class header
        determine invariants relating purely to this class
        apply selected invariants to the set
        create cross-product of reduced set with current result set
        determine invariants applying to classes incorporated in current result set
        apply selected invariants to the result set
```

Table 10.6 Pseudo-code for generating object combinations for an *inter_class*

The way in which an invariant is applied to objects of a class varies, dependent upon whether the class is grouped or not. If a class is not grouped, and an object fails an invariant, the whole combination is removed from the list of combinations to be further considered. If a class is grouped, then the invariant is applied to each object in the group and is used to reduce the number of objects in the group. The application of this algorithm is demonstrated in the following example based upon an *inter_class* definition for mapping between the IDM and PlanEntry. In this example, a mapping from PlanEntry to the IDM is being undertaken. The *pe_face* object is the newly created object, and there are six *pf_plane_object* objects and five *pe_opening* objects. The values of selected attributes and functions of the example objects are shown below the header and invariants section of the *inter_class* definition. Note that in the values column for objects of class *pf_plane_object* the result of calling the function *map_orientation_axis* is shown, rather than an attribute value, and for the objects of class *pe_opening* the result of calling the function *contained_in_face* is shown.

```
inter_class([idm_space_face], [pe_face, pf_plane_object, group(pe_opening)],
    invariants(
        type_of_face \= 'opening',
        member(pe_face.orientation, ['n', 's']),
        pe_face.offset = pf_plane_object.offset,
        map_orientation_axis(pe_face.orientation, pf_plane_object.axis),
        contained_in_face(pe_face, pe_opening),
        pf_plane_object.axis \= 'z'
    ),
……).
```

ObjID	Type	Attributes / Functions	Values / Results
o1	pe_face	[orientation, offset]	['n', 12.5]
o2	pf_plane_object	[axis, offset, map_orientation_axis('n', 'x')]	['x', -7, false]
o3	pf_plane_object	[axis, offset, map_orientation_axis('n', 'x')]	['x', 12.5, false]
o4	pf_plane_object	[axis, offset, map_orientation_axis('n', 'y')]	['y', -11, true]
o5	pf_plane_object	[axis, offset, map_orientation_axis('n', 'y')]	['y', 12.5, true]
o6	pe_opening	[contained_in_face(o1, o6)]	[true]
o7	pe_opening	[contained_in_face(o1, o7)]	[false]
o8	pe_opening	[contained_in_face(o1, o8)]	[false]
o9	pe_opening	[contained_in_face(o1, o9)]	[true]
o10	pe_opening	[contained_in_face(o1, o10)]	[false]
o11	pf_plane_object	[axis, offset, map_orientation_axis('n', 'z')]	['z', 0, false]
o12	pf_plane_object	[axis, offset, map_orientation_axis('n', 'z')]	['z', 2.3, false]

The initial list of list of objects for the header [pe_face, pf_plane_object, group(pe_opening)] is [[o1], [o2, o3, o4, o5, o11, o12], [[o6, o7, o8, o9, o10]]]. The first step is to try to reduce the grouped list [[o6, o7, o8, o9, o10]]. However, there are no invariants which apply purely to these objects, so the next list is selected, [o2, o3, o4, o5, o11, o12] and it is also reduced. In this case the invariant checking that the axis is not *z* can be applied, producing the list [o2, o3, o4, o5]. The two resultant lists are combined to create the following cross-product:

[[o2, [o6, o7, o8, o9, o10]],
 [o3, [o6, o7, o8, o9, o10]],
 [o4, [o6, o7, o8, o9, o10]],
 [o5, [o6, o7, o8, o9, o10]]]

There are no invariants affecting only *pe_opening* and *pf_plane_object* items so this initial cross-product can not be further reduced at this time. Then the next list is selected, [o1]. There is one invariant which applies purely to this object (member(pe_face.orientation, ['n', 's'])), this is checked against the object in the list to produce the same list, [o1]. This list is then combined with the other lists to create the cross-product below:

[[o1, o2, [o6, o7, o8, o9, o10]],
 [o1, o3, [o6, o7, o8, o9, o10]],
 [o1, o4, [o6, o7, o8, o9, o10]],
 [o1, o5, [o6, o7, o8, o9, o10]]]

Now the remaining three invariants on the PlanEntry side can be applied to each combination. The first list, [o1, o2, [o6, o7, o8, o9, o10]], is rejected as ‘pe_face.offset = pf_plane_object.offset’ is false, the second list, [o1, o3, [o6, o7, o8, o9, o10]], passes that invariant but fails the ‘map_orientation_axis(pe_face.orientation, pf_plane_object.axis)’ invariant. The third list, [o1, o4, [o6, o7, o8, o9, o10]], is rejected as ‘pe_face.offset = pf_plane_object.offset’ is false, the fourth list, [o1, o5, [o6, o7, o8, o9, o10]], passes that invariant and the ‘map_orientation_axis(pe_face.orientation, pf_plane_object.axis)’ invariant. The last invariant,

‘contained_in_face(pe_face, pe_opening)’ is used to reduce the objects in [o6, o7, o8, o9, o10] to the final list of [o6, o9]. Therefore the reduced list of lists from the given set of objects being applied to the invariants of this *inter_class* is [[o1, o5, [o6, o9]]], meaning that one mapping is performed across to the IDM for the newly created object o1.

10.3.5 Four pass mapping process

In recognition of the fact that the order in which objects are created in one data-store may not match the order objects should be created in another data-store, the mapping process is performed in four passes (see Table 10.7 for the pseudo-code). Using a four pass system obviates problems that may occur in transaction-based mapping due to the collating of all object modifications to one point, rather than the actual order they occurred. The work performed in each pass is detailed below:

First pass: determine all combinations to be mapped from the initial data-store. Where necessary, create new objects in the data-store being mapped to as described in Section 10.3.6. At this point not all objects, which could be connected together to match classes in the *inter_class* header, are created, so only specify values for the attributes of the newly created object. At the end of pass one it is guaranteed that the minimum state of the mapping is that all combinations to be mapped have been identified, and the object matching the first class of the *inter_class* header for the data store being mapped to has been identified. It is also guaranteed that all values that are directly identifiable for that first object have been calculated.

Second pass: determine all objects matching the header class specification. In mappings where there is more than one class specified in the header of the *inter_class* for the data-store being mapped to, try to connect to existing objects for the other classes rather than create new objects. At the end of pass two it is guaranteed that all objects matching the *inter_class* header for the data store being mapped to have been identified or created. It is also guaranteed that all values that are directly identifiable for these objects (i.e., specified in an *inter_class* where the object’s class is specified in the header) have been calculated, this includes initialisers, equivalences and invariants.

Third pass: attempt to solve equivalences for each affected object. Some equations with values that are accessed down a pointer chain may not be solvable at this time, as the values for the referenced object may not have been calculated. At the end of pass three it is guaranteed that all indirectly referenced objects (i.e., those specified through a pointer chain in equivalences) have either been associated to the *inter_class* (where they existed already), or created and initialisers solved. Due to the lack of control over the order that these indirectly referenced objects get created, and their initial values specified, there is a requirement for a further pass to solve outstanding equivalences. For example, the following equation o_{1l}=>o_{2l}=>o_{3l}=>a_{1l} = o_{1r}=>o_{2r}=>o_{3r}=>a_{1r} where each of the object references is solvable by another mapping (e.g., if mapping from left to right, the object ID of o_{1r} is determined

from the mapping of o_{11}) is not solvable until all associations have been completed. This is because the attribute references are not able to be determined until phase two is complete, and depending upon the order of solving mappings this equation might be attempted before all other attribute references have been matched.

Fourth pass: reassess all previously unsolvable equations involving referenced values to determine whether they are now calculable. At the end of pass four it is guaranteed that all equivalences which can be solved from the data available in the data store being mapped from, through the *inter_class* definitions, have been calculated.

```
four-pass inter_class resolution
        case pass number of
        one:
                determine all combinations to be mapped from data store
                create objects for first class in inter_class header specification
                apply appropriate initialisers, equivalences and invariants
        two:
                attach or create objects for all other classes in inter_class header
                apply appropriate initialisers, equivalences and invariants
        three:
                for each mapping combination apply all equivalences
        four:
                determine unsolved equivalences affected by pass three resolutions
                re-calculate determined equivalences
```

Table 10.7 Pseudo-code for the four-pass *inter_class* resolution

10.3.6 Mapping a new combination to the other data-store

When preparing to map a combination of objects it is checked against object combinations that have already been mapped to the other data-store (see Table 10.8 for the pseudo-code). If the combination of objects, except for those in grouped classes, is unique, then a new mapping is initiated. If the combination matches, except perhaps for objects in grouped classes, then all affected equations of the existing mapping are re-evaluated.

When a new mapping is created, an object whose class is that of the first class in the header for the side being mapped to is created. In the example above, an object of class *idm_space_face* would be created. Then all initialisers, equivalences, and invariants which determine values for the new object are solved. If the *inter_class* header for the side being mapped to contains more than one class, then a combination determining procedure, as outlined in Section 10.3.4, is followed in the data-store that is being mapped to during the second pass through the mappings. This provides a way of linking up with existing objects in the data-store being mapped to, rather than creating superfluous objects during the mapping. If a matching combination is found, then all equivalences which apply to the found objects are solved. If no matching combination is found, then a new object is created for the next class along in the header, all initialisers, equivalences, and invariants which determine values for the new object are solved and the matching process is retried. This process is duplicated until either a matching combination of existing objects is found for the rest of

the header classes, or until new objects have been created for every class defined in the header of the *inter_class*.

```
mapping all combinations of objects
        for each combination of objects for an inter_class header
                if combination matches an existing mapping combination (except grouped)
                then
                        re-calculate affected equivalences
                else
                        create object for first class in inter_class of side being mapped to
                        apply initialisers, equivalences and invariants purely for this object
                        for each following class in inter_class header (in pass 2)
                                search for matching objects
                                if no objects found
                                then
                                        create object for class no object could be found for
                                        apply initialisers, equivalences and invariants
                        solve all equivalences and invariants for the mapping (pass 3 and 4)
```

Table 10.8 Pseudo-code for mapping a new combination

10.3.7 Procedures followed when a new object is created

Whenever a new object is created during a mapping, the initialisers section of the *inter_class* definition being used is examined and any initialisers which apply to objects of that class are solved, setting values for attributes, or calling methods of the newly created object (see Table 10.9 for the pseudo-code).

Whenever an object is created in a mapping it is cross-referenced against the mapping in which it was created. In this manner the list of objects created in each individual mapping can be ascertained, and this information used when objects need to be deleted.

```
a new object has been created
        if an object clash was reported
        then
                find existing object which matches known criteria
                delete newly created object (if not already discarded by system)
                utilise the found object's ID as the new object ID
        else
                find all initialisers applying to the object
                for each identified initialiser
                        resolve initialiser
                cross-reference object creation against this mapping
```

Table 10.9 Pseudo-code for creating a new object

As the mapping manager tries to distance itself from the underlying implementation of the data-stores, it does not duplicate the functionality which would be provided by e.g., relational databases. To this extent, determining object uniqueness (i.e., key violations in a relational database) is not handled explicitly by the mapping manager. Instead, if an object is created which conflicts with existing objects, the mapping manager expects to be informed of this fact by the

underlying data-store. If such a clash is detected, the mapping manager searches the existing data-store to find the previous object. The object whose newly asserted value caused the clash is deleted and the existing object used for all further references. In this manner the mapping manager can reuse existing data-store objects, rather than creating multiple instances of objects containing the same data. However, this is not demonstrated here as the Snart persistent data-store used in the demonstrations of the mapping manager does not yet have an implemented key uniqueness verification system for objects created inside it. This remains for future work.

10.3.8 Tracking objects created and referenced in mappings

The fact that an object is referenced in a mapping is also tracked. This tracking encompasses more than just the objects collated together to match the header classes of the *inter_class*, but includes all objects referenced through the equations in the *inter_class*. Objects that can be included in this manner are those that are used to follow reference chains. In this way a list of mapping managers is compiled for every object used in a mapping. Then when an object is modified the set of mapping managers which may have to perform recalculations is quickly identified. These mapping managers are informed of the modified object and affected attributes, and using the pre-computed lists of objects and attributes in each equation determine which equations have to be re-computed.

```
an object was deleted requiring a mapping to be dissolved
        delete primary object from first class of header in the inter_class
        determine all other objects matching the header
        for each object
                if this mapping is only reference to the object
                then
                        delete object
                else
                        update mapping references to the object, deleting dissolved mapping
        determine all objects created when solving equivalences and initialisers
        for each object
                if this mapping is only reference to the object
                then
                        delete object
                else
                        update mapping references to the object, deleting dissolved mapping
```

Table 10.10 Pseudo-code for mapping an object deletion

10.3.9 Mapping the deletion of an object

The mapping of an object deletion in one data-store is handled using a type of garbage collection scheme (see Table 10.10 for the pseudo-code). If the deletion of an object causes a mapping to become non-viable (e.g., not enough objects for the classes in the header, or an invariant is violated) then all objects which were necessarily created when setting up that mapping (e.g., the object for the first class in the *inter_class* definition) are deleted. All the other objects which were created during the mapping are examined. If their only point of reference is the mapping which is being dissolved then they are deleted as well. Also, all objects referenced by equations in the

mapping are examined. If their only point of reference is the mapping which is being dissolved then they are deleted as well.

10.3.10 Mapping the modification of an object

Objects whose attributes have been modified use the lookup table (an AVL tree structure) to determine which mapping controllers need to be informed (see Table 10.11 for the pseudo-code). When the mapping controllers learn of the modification they initially check whether the modifications affect any of the invariants of the *inter_class*. If invariants are affected then they are re-evaluated to ensure that they still hold. If the invariants are violated the mapping is dissolved with deletions of objects as defined in Section 10.3.9. If the invariants are not affected, or if they still hold, then all affected equations are identified and re-calculated.

```
an object has been modified
    determine affected mappings
    for each affected mapping
        if modification affects an invariant specification
        then
            re-evaluate object combination
            if new combination results (except for grouped classes)
            then
                dissolve existing mapping
                create required objects in other data store
                apply initialisers
                solve equivalences
            else
                re-calculate affected equivalences
        else
            re-calculate all affected equivalences
    find inter_class definitions referencing class of modified object and also in invariants
    for each inter_class identified
        determine initial object groups
        generate object combinations for the inter_class
        for each remaining combination
            if combination matches existing mapping, or class is grouped in header
            then
                do nothing as will have been solved above
            else
                create required objects in other data store
                apply initialisers
                solve equivalences
```

Table 10.11 Pseudo-code for mapping an object modification

All *inter_class* definitions which reference the modified object in their class headers are also identified. They are checked to see if the modified attributes affect any of the invariants of these *inter_class* definitions. If they do, then the object combinations are computed as in Section 10.3.4 and any new combinations are used to initiate new mappings as detailed in Section 10.3.6.

10.3.11 Evaluating, or re-evaluating affected equations

All equivalences in an *inter_class* are passed through to the equation solver, along with all attributes affected by the object creation or modification. Prepended to all the equivalences are the invariants which apply to objects in the data-store being mapped to (i.e., what has to hold for the objects that are being created or modified). As each equation is pre-processed and all objects and attributes used in the equation identified, the first step in solving an equation is to identify values for all necessary object and attribute references in the equation (see Table 10.12 for the pseudo-code).

The result of this search for values is used to determine whether the equation can be solved currently or not. When performing a mapping, the controller is not allowed to make changes to the data-store the mapping is coming from. The assumption is that the transaction is not modifiable as it may have already been mapped to other data-stores. Therefore, any equation with unknowns on the side being mapped from is classed as unsolvable. Given that all values are known for the data-store being mapped from, the three different types of equation are handled slightly differently:

procedure: all values from the data-store being mapped from are matched to the procedure parameters. If there are any object references for the store-being mapped to, then either the existing object reference is supplied, or a new object is created to be passed in. If the parameter *$mapping_system$* is specified then the object ID of the mapping system is passed as a parameter. Any attribute unknowns are recorded and the procedure invoked. When it terminates, any of the unknowns which were calculated during the procedures calculations, and passed out in the parameters, are assigned to the specified attributes.

function: all values from the data-store being mapped from are matched to the function parameters. If there are any object references for the store being mapped to, then either the existing object reference is supplied, or a new object is created to be passed in (on the assumption that a function will not know how to create an object in a data-store). Any attribute unknowns are recorded and the function invoked. When it terminates, any of the unknowns which were calculated during the function's calculations, and passed out in the parameters, are assigned to the specified attributes.

equation: the mapping system contains an equation rearranger which can take most equations and rearrange to solve for a particular attribute. Therefore it is only necessary to identify the attribute that needs to be solved in the equation. If there is one unknown in the set of attributes of the data-store being mapped to, then this is the unknown attribute for which the equation is solved. If there are several unknown attributes, then the equation can not be solved at the current time, though it may be solvable after other equations have been solved (i.e., some of the unknowns are calculated by other equations), and is placed in a queue to be re-examined. If there are no unknowns, an alternate approach must be taken. Part of the meta-data recorded for each attribute, by way of facets, is a facet determining who supplied the data for a particular attribute. Where all attributes are known, this facet information is used to provide a ranking of the attributes in terms of the importance of where their data

was asserted, from default data through to actor defined data. This ranking is applied to try to identify the least significant piece of data known. The equation is then solved for this attribute. If no attribute can be identified to be solved then the equation is placed on a queue to be examined when the other equations have been solved, to see if they provide any further assistance in solving the difficult equation. After all equations have been attempted, the queue of unsolvable equations is re-examined, checking whether anything has changed which makes these equations solvable. This queue is iterated over until there is no change in the status of the equations in the queue, i.e., nothing was solved in a particular iteration. Any equations which can not be solved, after all the procedures outlined above, are rewritten as constraints and asserted against the particular object involved. In this way the values that should hold for a particular object, or set of objects, are recorded and are seen in the data-store even after the mapping has completed. Asserted constraints are specified in the constraint specification language implemented in Snart (Mugridge et al. 1995). The Snart constraint specification language has a similar syntax to the VML notation, enabling VML equations to be easily rewritten and asserted in Snart.

```
identify values for an equation
    for each reference in the equation to data-store being mapped from
        find value for the reference
    if any missing references in the equation to data-store being mapped from
    then
        terminate solving of equation
    else
        for each reference in the equation to data-store being mapped to
            if value is known for the reference
            then
                return known value
            else
                case equation type of
                procedure:
                    case reference type of
                    object:
                        { see Table 10.9 }
                    $mapping_system$:
                        return object ID of mapping system
                    attribute:
                        record as unknown
                function:
                    case reference type of
                    object:
                        { see Table 10.9 }
                    attribute:
                        record as unknown
                equation:
                    record as unknown
```

Table 10.12 Pseudo-code for identifying values for an equation

The ranking of importance of values assigned to an attribute is managed by the mapping system. The mapping system ensures that no low ranked values are able to overwrite high ranked attributes (e.g., an attribute defined through a default value being mapped over an attribute previously

defined by an actor). In most cases, values at the same ranking may overwrite each other if specified later in time (e.g., a design tool calculated value can overwrite a previous design tool calculated value). This is not automatically true for one case, that of actors. In this case, a dialogue is initiated to ensure that the new actor-specified data is allowed to overwrite the previous actor-specified data. If the overwriting is allowed, then that permission is maintained for the whole mapping of the transaction, before being rescinded again. The full set of update authorities is shown in Table 10.13.

New / Previous	User	Design Tool	Constraint	Default	Unknown
User	negotiate	no	no	no	no
Design Tool	yes	yes	yes	no	no
Constraint	yes	no	merge	no	no
Default	yes	yes	yes	yes	no
Unknown	yes	yes	yes	yes	yes

Table 10.13 Update authorities for attributes derived from different sources

When an equation is solved and a value instantiated for an attribute, it is necessary to instantiate the facet recording who specified the calculated value. In some cases this is easily calculated. Initialisers and invariants which specify a constant value for an attribute allow the value to be categorised as default. Equations which equate single attributes or pointers can be categorised utilising the same categorisation as the attribute from which the value or reference is mapped. Equations which involve several attribute values which may come from very different sources (e.g., default values and actor-specified) require a more sophisticated method of determination. In the mapping system the highest level specifier is used for complex equations. For example, in an equation involving default values, design tool calculated values, and actor-specified values the final attribute value will be defined as coming from an actor. Using this method, the values in a mapped system tend towards being actor-specified, seemingly imparting a high level of confidence in the data in the system. However, this also means that an actor must intervene more often during the mapping process to assert the right to overwrite values specified by other actors (assuming they are within the actor's schema specifying modification permission).

10.4 Appraisal of Mapping Controller

The mapping system described in this chapter provides an implementation of the VML language capable of both transaction-based and interactive mappings between data-stores. The implemented

system allows multiple data-stores to be connected to an IDM, with full maintenance of the consistency of the IDM at all times, as well as reflecting changes made in any connected data-store through to all affected data-stores. The mapping system has been tested with a wide set of mappings, including many small mappings drawn from other work in the area (see Appendix D and Clark 1992; Bailey 1994; Hardwick 1994), and a large example showing the integration of four design tools connected through an IDM (see Appendix E; Hosking et al. 1995; Mugridge et al. 1996). The large example described in Appendix E shows the use of the mapping system to maintain information updates and consistency back and forth between the interactive tools being integrated, as well as being output to a visualisation tool. The use of the different tools is controlled by a process model for the task being attempted, which establishes when actors can perform their design functions, and manages the mappings which are allowed. This example in Appendix E provides the main demonstration of how all the components of the thesis work together to create an integrated design system.

Although the mapping system is capable of maintaining the correspondences between two data-stores, it is not capable of linking together two independently developed data-stores (e.g., the same building specified independently in two different design tools). This is mainly due to the assumption that a new object is created in the data-store being mapped to for every *inter_class* mapping which is initiated with objects from the other data-store. The class matching algorithm could be extended to attempt a match of the initial class rather than creating one, but this may not provide the required solution in all cases. The problem is partially that there is no syntax to specify whether an object should be created or searched for in a mapping, and partially that there are cases when it is not possible to decide whether an object should be created or searched for. Part of the problem comes from object-oriented systems where there are not necessarily keys defining which objects must be unique. Instead it is assumed that the object-oriented system ensures the consistency of the model by the way in which it is developed (semantics of the application). These semantics may not be able to be specified in a mapping without implementing a large amount of the object-oriented application within the mapping.

The mapping system requires more work in the section which decides which attribute of an equation to solve for. The current method works well where there are small numbers of attributes in an equation, or where there is a previously asserted value which is a default or other low level of specification. However, where several attributes were all previously asserted at the same level there is no good way of deciding which to solve for. The addition of reasons, as implemented in the Snart language (Hosking et al. 1994), may provide a way of deciding which attribute to solve for depending upon which attribute, or attributes, changed in the object being mapped from. This could, however, make the equation definition much larger and more difficult to maintain, though the use of default directions would alleviate this problem. The real problem is dealing with an actor's intent in specifying information. If this intent could be captured, perhaps partly through the process specification, then it would be easier to determine what data could change. For example,

information derived from trialling a design variation should not take precedence over previous information asserted for a legally binding project sign-off stage. Capturing the level of intent would extend Table 10.13 to include classifications such as actor assumptions, derived information, informed specification, and constraints due to standards, codes of practice, best practice, etc.

Several speed improvements could be made to this system. Mapping of small building models (a building consisting of two spaces with doors and windows) between an application and the IDM takes twenty minutes, or more, on the Macintosh on which this was developed. Two sections of the mapping system are known to be inefficient.

- The persistent data-store which traces and records all modifications and method calls adds a very large burden on all applications in which it is utilised. This is especially true for the tracking of method calls as the calls in the running mapping system (written in Snart) are trapped as well as those in the running design tools (also written in Snart). However, the method calls of the running mapping system are discarded immediately. Rewriting the data and method call tracing in a lower level language would provide large speed improvements to design tools and the mapping system.
- The generation of combinations matching *inter_class* headers is very slow when large numbers of objects are involved. Two improvements are considered for this problem.
    - The first is to implement the more efficient combination generation algorithm from relational database work (Ullman 1982). Though the worst case combination generation is a polynomial of the same order as the number of classes referenced in a header (as in a relational database system), this is unlikely to ever be a practical problem. The reasons for this are that usually only a very small number of classes are grouped through a header, and where a larger number of classes are referenced it is usually to associate information from a single uniquely identifiable object. The relational database algorithms can ensure that the most efficient combinations are computed first and the resultant sets are reduced as soon as possible.
    - The second is to ensure that combinations are only generated once for any set of new, modified, or deleted objects. In the current system this is not the case and each new, modified, or deleted object forces the generation of its own set of combinations, before checking that the generated combinations are not already being used.

Other speed improvements are possible from the use of a low-level programming language like C rather than the LPA Prolog used in this thesis. Trials with a rewritten Snart have shown two orders of magnitude speed improvement for some operations (in the constraint processing). However, Prolog was the correct choice for prototype implementation for this PhD thesis in that more functionality was programmed than would have been possible in the same time with other languages.

Chapter 11

Flow Handling

A flow handler ties together all the work detailed in the rest of this thesis, providing a tool to manage the actual design process for individual projects. Earlier chapters describe methods of modelling: actors in a project; design tools used; the integrated data model; relationships between data in various schemas; and desired flow of control in a particular project. Chapter 10 describes a system which will move data between model instances of a particular building, and Chapter 8 refers to work on automating the invocation of design tools. The final step is to put the process control tool in place in order to manage the allowable design tasks at any particular time, and to ensure the design is accomplished according to the flows defined by the project manager. This chapter describes a particular implementation of a flow handler which provides managerial control to a project manager and task level control to the actors working on a particular project.

11.1 Requirements for Flow Handling

A flow handling system must provide two distinct categories of functionality. The first is for a project manager who oversees the running of a particular design project and must ensure a timely completion, as well as a final design which meets the initial design criteria. The second is for the actors, who need to see what design functions remain to be completed in their design roles and when they can pass the design on to another actor. The different requirements of the two main classes of user of the integrated design system are detailed in the following subsections.

11.1.1 Project manager requirements

The managerial requirements of a project manager are detailed below:

Project overview: the project manager must be able to determine the current state of the project; which stages of the project have been completed; which stages are still being worked upon; who is currently working on various stages of the project; and what design tasks are still to be completed.

Track design path: in most projects it is imperative to be able to determine who worked on what portion of the design, and to see when particular design tasks were completed.

Status: the project manager must be able to determine the current status of the project window; to see who is working, and on which parts of the design.

Dynamic project flow modification: the project manager must be able to intercede in the flow of a project if it is clear that new resources are needed to complete the project. The project manager must be able to move an actor to a new task; stop a task before its normal completion; stop an actor's design role work; and start a new actor working on a new design role, perhaps concurrently with other actors.

Dynamic flow diagram modification: if the project manager determines that a new path is required in the flow diagram (e.g., to force a particular task to be performed, or to add new functions into the design process, or to add paths requested by actors in the project) then modifications should be able to be made to the flow diagram, and they should be reflected in the running project.

11.1.2 Actor requirements

The requirements of actors are more closely related to their design roles, as detailed below:

Design role overview: actors must be able to manage the design functions required to complete their design roles, and they must be able to determine what is required to complete their design role.

Formal handover between actors: the completion of a design role and handover point between actors needs to be documented, both for the time of handover and the state of the project at the time of the handover.

Determination of allowable functions: the possible design functions for each actor are calculable based on the other actors currently working on the project and the design functions they are working on. Only currently invocable design functions should be allowed to start up, and the status of all currently available, or stalled, design functions should be re-evaluated upon the completion of any design function by any actor in the project.

This provides a level of control suitable for a strict project management regime. Actors have control over the work they perform towards their design roles, but limited to design functions specified in the project window definition. If an actor requires modifications to the project window this must be negotiated with the project manager. This ensures that specified design functions are

performed and there is no way to work around the requirements implicit in the workflow specification.

11.2 The Exchange Executive

The Exchange Executive (ExEx) provides the implementation of the flow of control definitions of Chapter 7 and implements the requirements of a project manager and actors as defined in Section 11.1. The main component of the ExEx is a simulator for the CombiNet specification, determining the design functions which could be next completed in a project. As well as simulating the CombiNet, the ExEx accesses the schema definitions of the various design tools and actors to determine possible restrictions between the design tools when invoked concurrently. Finally, to present the information to the various users there are separate user interfaces tailored for project managers and actors. The working of all of these components is detailed in the following subsections.

11.2.1 Simulation of flow of control

To describe the simulated flow of control in the ExEx, the flow through a net which comprises simple places and transitions will be described (as all CombiNets can be rewritten in this form). In this type of net there are four components: places, transitions, tokens, and connecting lines. The implementational semantics of these four components are described below:

Token: a token represents the current state of an individual workflow. When shown in a place it represents an actor working on the design function represented by that place. As the workflow progresses from design function to design function it may pass across actor boundaries and thereby denote a handover phase between actors. There may be several tokens in a simulated CombiNet representing different concurrent workflows. While a token in a place represents an actor performing a particular design function, an actor may be involved in several tasks at any one time. Each of these tasks would be defined by separate tokens in the CombiNet representing the particular workflow associated with the tasks being undertaken.

Place: a place represents a design function, which may or may not be realised through the use of a design tool. If the place represents a design tool, then the movement of a token into the place signifies the invocation of the design tool with data from the IDM, as described in its input schema definition. The movement of a token out of a place represents the termination of the design tool and the transfer of its resultant data (if any) through to the IDM. Between these two events the actor is free to perform whatever actions they wish with the design tool (as long as the action is within the scope of the area of responsibility of the actor, as defined by the actor's schemas).

Transition: a transition has no implementational semantics in the ExEx. It is purely a representational notation to describe a choice point between several places. Transitions are of most use when accessed by several places, reducing the number of lines in a diagram. The whole CombiNet definition could be redrawn without transitions and retain the same semantics, but many more lines could be required to represent flows between places.

Connecting lines: lines with an arrow at one end provide a path for the flow of control to proceed along. Each line represents a possible pathway, either from a place to a transition, or from a transition to a new place.

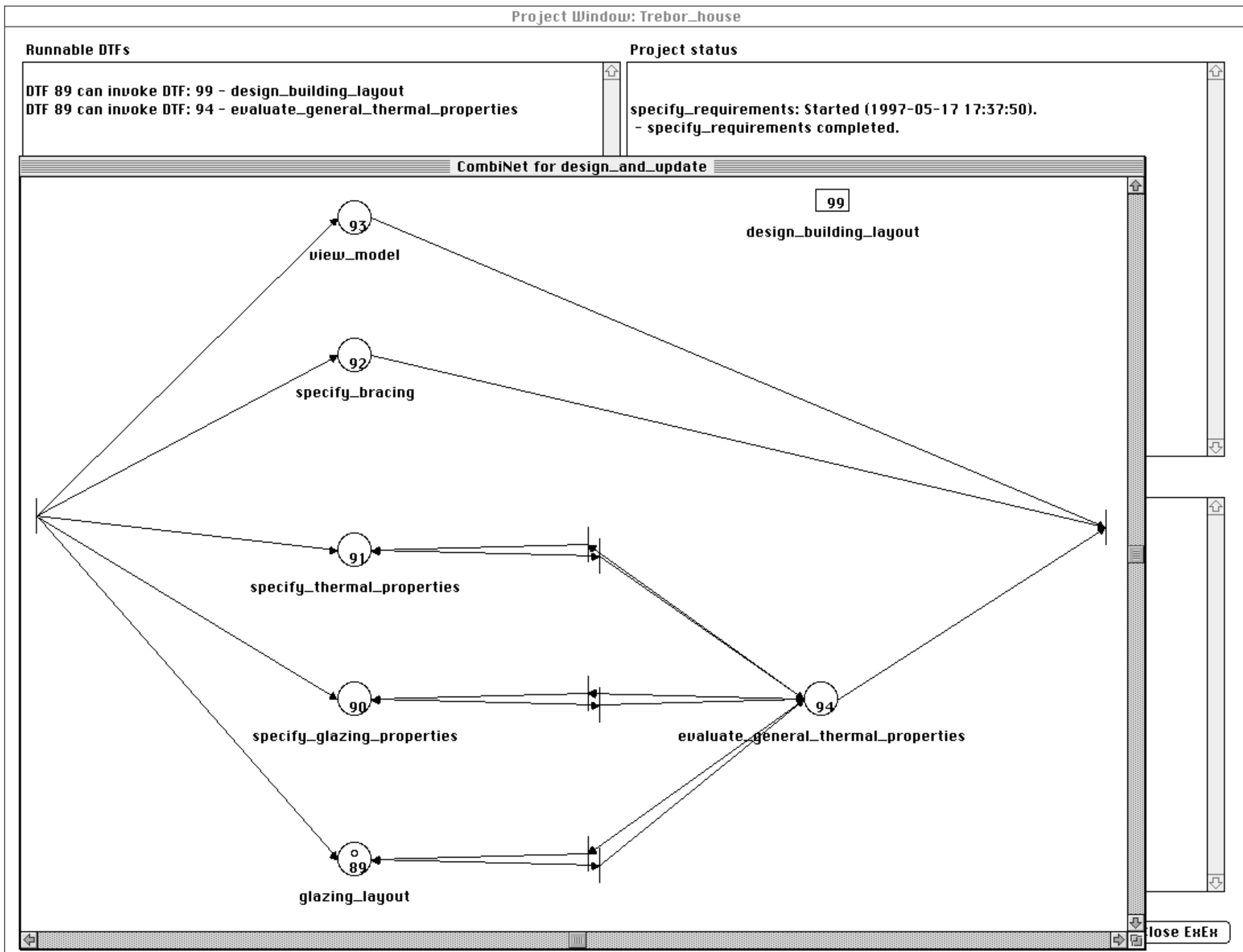


Figure 11.1 Calculating potential design functions from a CombiNet

The simulation of flow in a net comprising the above elements operates in the following manner:

- Identify each token in a place in the CombiNet.
- For each token in a place identify all transitions that can be exited to from that place.
- For each of these transitions identify the set of places which can be invoked from the transition.
- Determine the set of places which are candidates to be invoked from the current place by taking the union of all of the sets of places for each transition .

For example, Figure 11.1 shows a token (the small open circle) in the place representing the *glazing_layout* function. Upon completion of this function there is a single transition which can be exited to. This transition leads to the *evaluate_general_thermal_properties* place, however, there is

also a global place that can be invoked, called *design_building_layout.* The total set of design functions which can be invoked from this point is shown in the Runnable DTFs box of Figure 11.1.

There is a separate set of candidate places maintained by each token, representing possible continuations of the workflow, in the CombiNet. The set of candidates for each token is reduced by taking into account the following conditions:

Stalled place: this is derived from Constraint 1 in Chapter 7: if there is a currently running design function whose output could modify the input of the candidate place then it is labelled as stalled and removed from the list of candidates (e.g., Figure 11.2 shows the DTF *evaluate_general_thermal_properties* of Figure 11.1 running. The output of that DTF impacts on the *evaluate_thermal_code_compliance* DTF's input and for the actor shown in Figure 11.2, forces it to become stalled, as shown in the bottom right sub-window).

Insufficient data: if the data in the IDM is not sufficient to invoke the candidate design function then it is removed from the list of candidates.

Failed start: if there was enough data to start the tool but the tool would not start when invoked (usually through a failed constraint) then it is removed from the list of candidates.

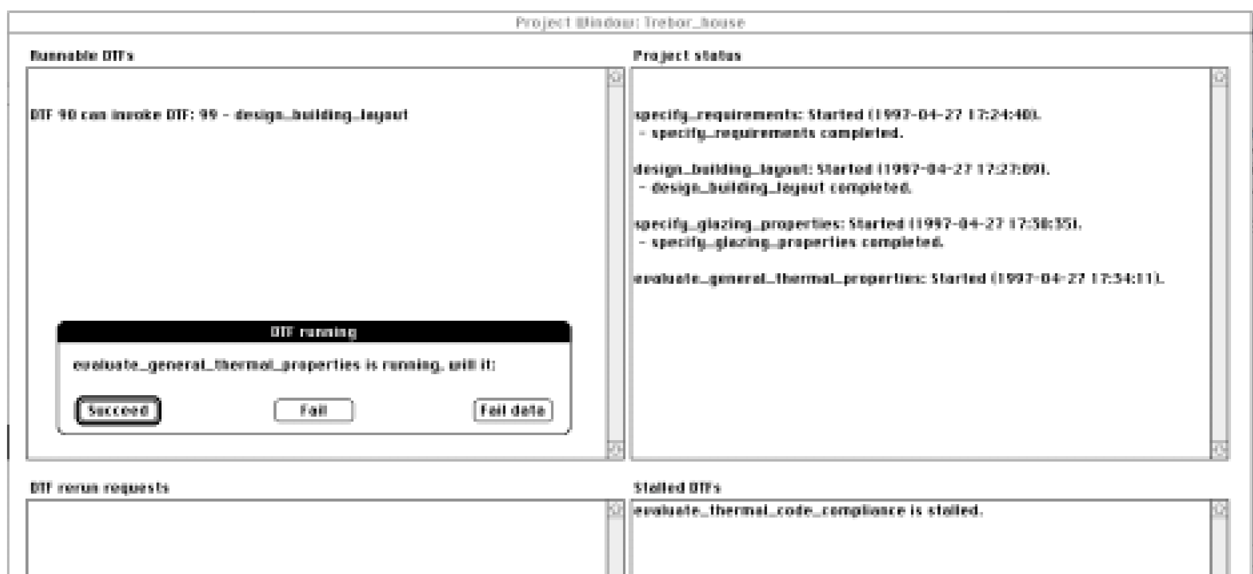


Figure 11.2 Multiple actors causing select design functions to become stalled

The set of candidates that can be run at any time is re-evaluated every time a running design function terminates or when a new design function is invoked. This re-evaluation may: add new places to the list that can be invoked; remove places from the list of candidates; or change the status of places whose status is un-runnable due to the conditions listed above.

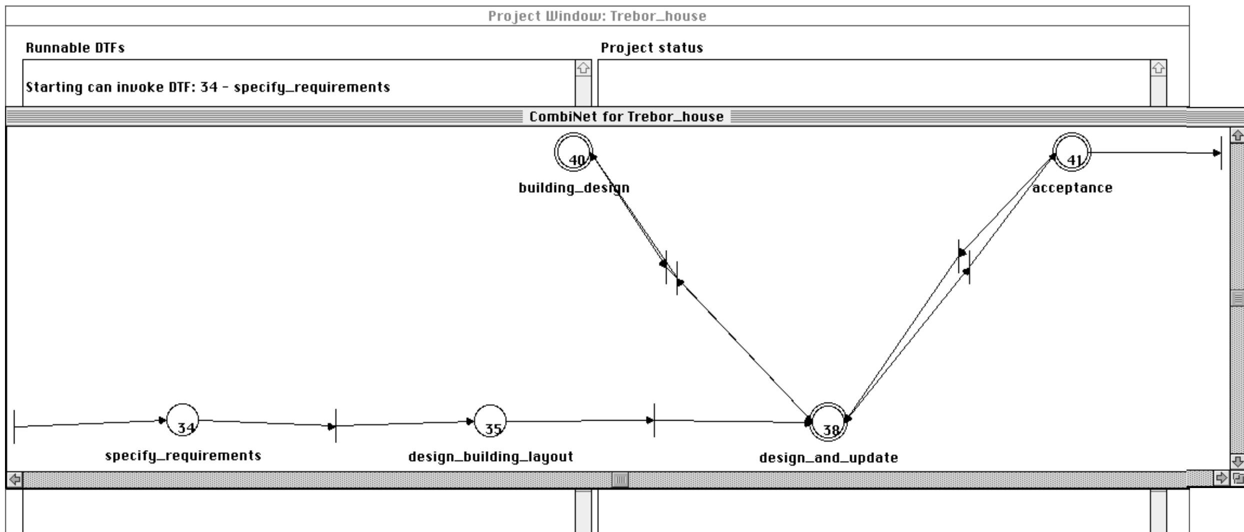


Figure 11.3 Initial CombiNet in a project window

We now extend the simulation semantics to cover several special states, and icons. Their flow conditions are described below:

Start of the project window: the start of the project window is determined by looking at the top level CombiNet (referenced by the user and function modelling diagram, see Chapter 7 and Figure 7.3). The start transition of this CombiNet is identified and the places reachable from it collated to become the initial candidates for the project window (e.g., Figure 11.3 shows the starting point for the examples used in Chapter 7. In this example there is only one place that may be invoked to start the project window, namely *specify_requirements*).

End of the project window: the end of the project window is found by identifying the end transitions in the top level CombiNet. Whenever one of these transitions is visible from a place with a token in it, there is the possibility of terminating the project window. In practice the completion of a project window is when the last token in the project window reaches the end transition (or the last token is terminated by the project manager) (e.g., Figure 11.3 shows a single end transition which can only be reached from the *acceptance* CombiNet).

Actor’s design role: when a token passes from one place to another, where the specified actors for the two places is different, this signifies the end of one actor’s design role and the start of a new actor’s design role. At this point the actor involved in the workflow represented by the token changes. This event is notified to the project manager who is responsible for permitting the new actor to start their design role (or who may terminate the token). At the end of an actor’s design role the actor may aggregate the work performed into an aggregate transaction, rather than single transactions for every design function they performed (as previously described in Sections 10.1 and 10.2). If this option is chosen, all their work since starting the current design role is collated under a single label in an aggregate transaction (e.g., Figure 11.3 has an actor changeover between *specify_requirements* and the *design_building_layout* design functions. At this point the *client* has completed the

requirement specification role and the *architect* starts the building design role. The whole requirement specification role could be recorded as a single aggregate transaction). At a hand-over point which offers the possibility of several actors working concurrently, it is up to the project manager to instigate any required concurrency, as is current practice in building design.

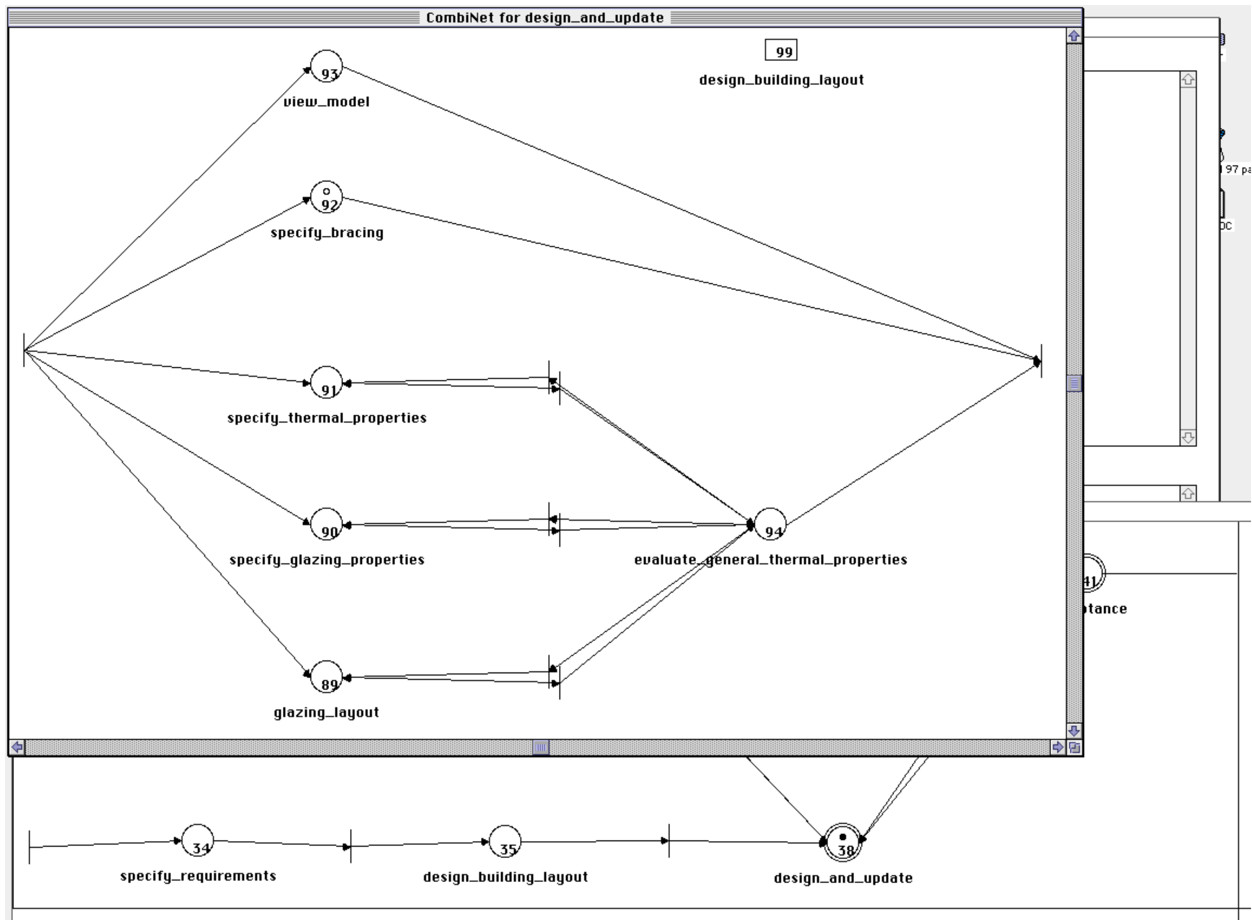


Figure 11.4 Multiple levels of aggregate functions

Aggregate place: an aggregate place represents a whole CombiNet, so when evaluating what can be invoked in an aggregate place, the start transition of the denoted CombiNet is identified, and the places which are reachable from that transition collated (e.g., in Figure 11.4 the *design_and_update* aggregate place, as shown in Figure 11.3, connects directly to the five places reachable from the starting transition, and can also access a global place). This evaluation process can be nested down several levels, as the start of a CombiNet can reference an aggregate place whose start transition would have to be identified, etc. When an end transition is reached in a CombiNet referenced by an aggregate place, it is tied to the output transitions of the aggregate place which referenced the CombiNet (e.g., when the client exits the acceptance CombiNet shown in Figure 11.5 (small CombiNet with two places) the end transition is tied to the output transitions of the aggregate place it was called from, in this case the *acceptance* aggregate place shown in Figure 11.3. In this example there are two output transitions, one is the end transition of the project window and the other leads back to *design_and_update*). Again, this evaluation may be recursive, as the

termination of a CombiNet at one level could lead to the end transition of the CombiNet above it, etc. The path travelled by a token is always maintained dynamically, as many CombiNets could reference the same aggregate place (representing the same CombiNet). As a result, the path taken to a particular design function is not necessarily unique (e.g., *design_and_update* can be accessed from the top level CombiNet shown in Figure 11.3, but is also accessible as a global net at other stages in the project window). These different points of access to a lower level CombiNet can also have different actors assigned to them through the actor overlays described in Section 7.3.2.

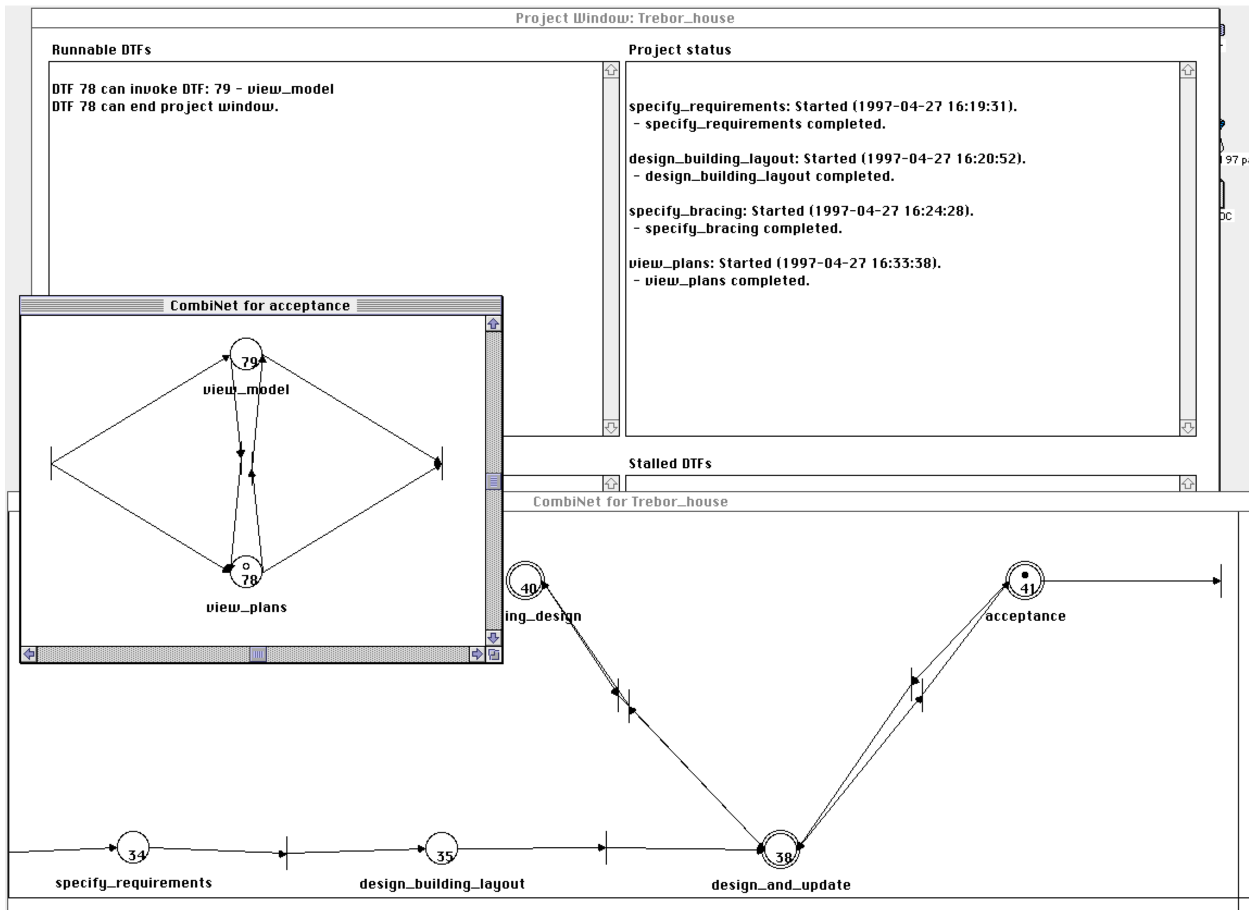


Figure 11.5 Exiting from a CombiNet representing an aggregate place

Global places: a global place is reachable from any place in the CombiNet in which it is defined, and from any CombiNet invoked through an aggregate place or global net (i.e., any descendant CombiNet). When exiting a global place, the token returns to the place from which the global place was invoked (e.g., the *design_building_layout* global place in Figure 11.4 can be reached from all the other places shown in that window. After completing the *design_building_layout* function the available design functions would be recalculated from the same place as the *design_building_layout* function was entered from). A global place is also accessible by all descendant CombiNets reachable from the CombiNet in which the global place resides.

Global net: a global net has the same functionality as a global place, except that, as it represents a whole CombiNet, its candidate places are calculated in the same manner as for an aggregate

place. As noted previously, when inside a global net it is not possible to reinvoke the same global net (it is removed from the list of candidate places).

Double transition: this shorthand notation is replaced with its two-transition equivalent in the ExEx, and handled in the same manner as other transitions (e.g., the transitions between *building_design* and *design_and_update* in Figure 11.3 are the expansion of a double transition).

When determining the set of candidate places from a given place it is possible to encounter the same design function along different paths. All occurrences are displayed for the actor to choose between, as each occurrence is part of a different workflow through the project window. Currently actors must navigate the CombiNets to ascertain the actual path traversed to particular instances of a design function.

11.2.2 Representation of design tool invocation

The ExEx, described in Section 11.2.1, simulates actors working in a particular project window. However, to perform this simulation, it requires information about the status of the design functions which are being performed. Though this ExEx implementation does not support design tool invocation (for the reasons described in Section 8.4), the point at which this would occur is simulated through a starting and ending dialogue for each design function. These two dialogues are described further below.

Figure 11.6 Design tool start up dialogue

At the time an actor decides to perform a particular design function which requires the use of a design tool a dialogue is initiated (see Figure 11.6). This dialogue is used to ensure that the actor performs a mapping from the IDM through to the design tool model, using the mapping manager described in Chapter 10. Dependant upon the outcome of the mapping process, and the ensuing design tool start up, there are two messages that can be passed back to the ExEx by selecting one of the buttons in the dialogue. These are:

Succeed: all of the required data existed in the IDM and was able to be mapped through to the design tool model. The data in the model was successfully translated into the form required by the design tool, and the design tool started without problems.

Fail: some initial constraints in the design tool model were invalidated, or it was not possible to start the design function due to the failure of the design tool to start up (e.g., due to design tool constraints not specified in the design tool's data model).

Figure 11.7 Design tool termination dialogue

After the design tool start up dialogue is completed, a new dialogue is presented (see Figure 11.7). This dialogue must be completed when the actor has finished work with the design tool and has mapped resultant data back to the IDM, using the mapping manager described in Chapter 10. Dependent upon the outcome of the mapping process there are three messages which can be sent back to the ExEx by selecting one of the buttons in the dialogue. These are:

Succeed: the design tool completed normally, and all data in the resultant data-files of the design tool were mapped back to the IDM through the design tool data model.

Fail: the design tool terminated abnormally or incorrectly.

Fail Data: the mapping system was not able to map the resultant data through to the IDM. This would be due to a violation of constraints specified in the IDM schema, which may be more constrained than the design tool's output model.

If the ExEx is informed of any of the three failure modes described above, it will roll the token back to the place at which the choice was made to invoke the design tool which just failed. The choices at that point are re-evaluated. The new list of available functions is presented to the actor and a new design function is chosen to continue work on that actor's design role. This list may still include the design function which previously failed if the failure was due to its operation rather than data constraints.

It is conceivable that failures of design functions could lead to a state where an actor has no further choices available. There are two possible methods of resolving this situation. First, if other actors are working on the project window, their completion of design functions will force the re-evaluation of stalled and failed tasks. Second, the project manager is informed of any actor who is stalled in this manner. The project manager has two options at this point, either: terminate the actor's design role (removing the token); or move the actor to a new point in the design process.

11.2.3 Project manager interface

To support his/her project management role, the project manager is provided with a comprehensive query, control, and browsing interface to the ExEx, allowing arbitrary movement around the flow definitions, and with the ability to easily modify the status of actors working in the project window. The main interface provided for the project manager is shown in Figure 11.8. This shows a list of all actor workflows currently executing in the project window, and the path which

brought them to their current position. The path for a particular actor workflow starts at their current position and steps back through all design functions invoked in their design role. Where an aggregate place or global net has been entered a bullet is placed before the place name to denote the point at which the lower level CombiNet was entered. For example, in Figure 11.8 the structural consultant is working on *define_wall_structures*, the path taken to this design function for the particular design role and workflow is from the *structural_work* aggregate place and before that the *building_design* aggregate place. Any actors who are currently stalled (i.e., having no possible design functions available at the current time) are denoted with a '•' before the actor name. Actor-specific functions supplied by the buttons in the interface become available when an actor is selected. The actor-specific functions offered by the buttons are:

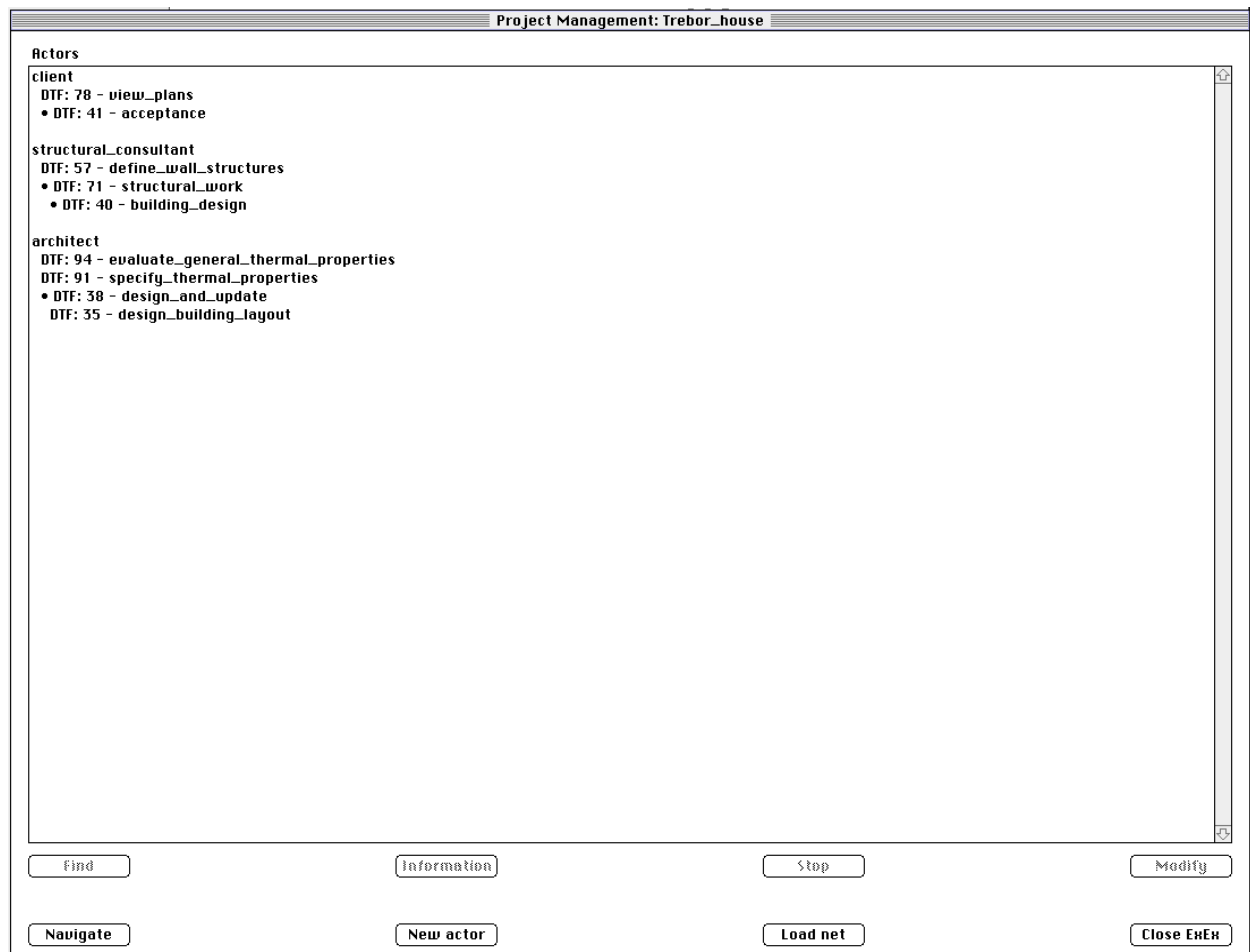


Figure 11.8 Project manager user interface

Find: opens up all CombiNets that were entered by the highlighted actor to reach the position at which the actor is currently working. The windows are opened in order, from the top level CombiNet through to the CombiNet in which the actor is currently working. Any aggregate places or global nets that an actor passed through to reach the current position are denoted with a filled token in the higher level diagram. A token is shown in these CombiNets for any other actor working in a CombiNet passed through to reach the selected actor.

Information: details the design functions available to the selected actor, shows any stalled design functions, and lists the work performed by the selected actor, using the interface provided to individual actors (see Figure 11.10 for an actor's user interface).

Stop: terminates the design function and workflow in which the selected actor is involved. The actor is removed from the project window at this point and must negotiate with the project manager if they wish to be re-involved in the current project window.

Modify: the selected workflow of an actor is modified to work in a new place in the project window. This is used to help an actor workflow which has no available design functions, or to make more resources available in a certain portion of the project window. The selected actor workflow is removed from the current location, and the project manager navigates through the CombiNets (as explained for the *Navigate* function below) to the place where the workflow is to continue from. When the project manager finally double clicks on a place, or global place, this is defined as the current location of the actor in that workflow. After placing the actor, the project manager must specify the place, or global place, from which it is assumed the actor reached the current place. This is in case the design function fails and the actor has to roll back one place.

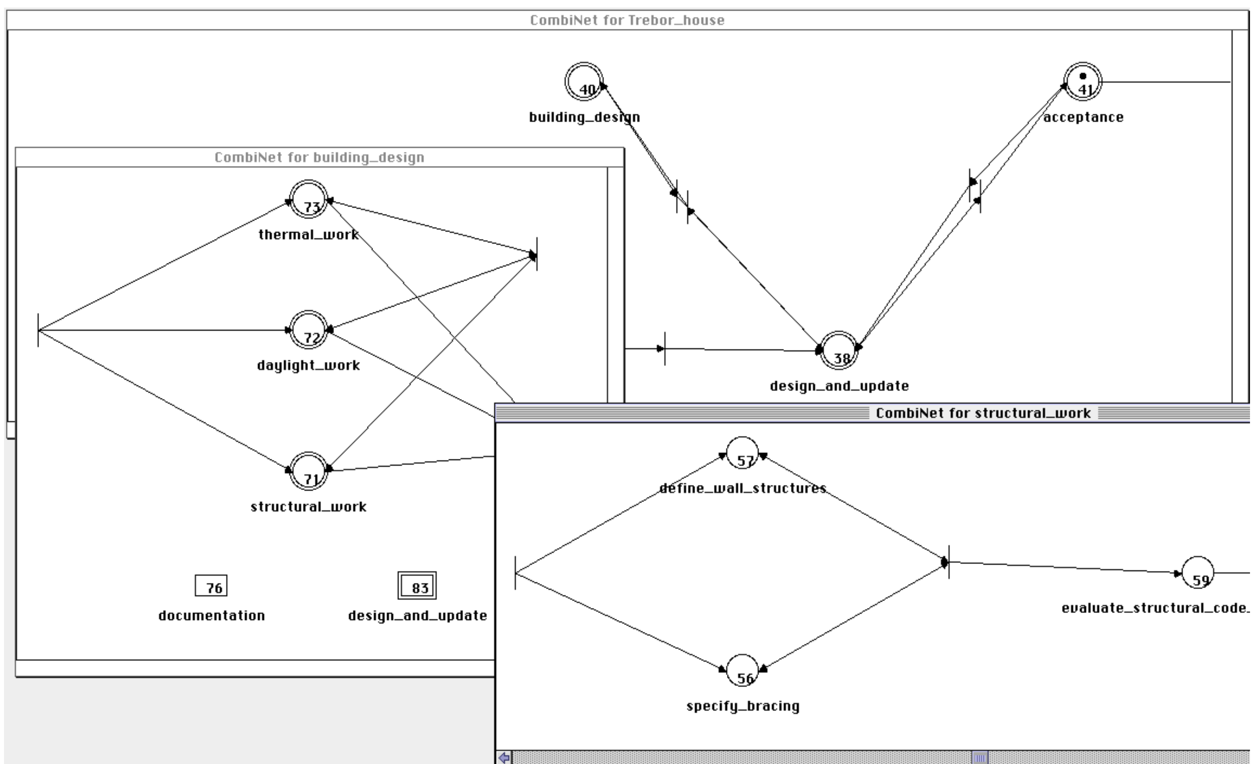


Figure 11.9 Project manager navigation through CombiNets

The project manager also has three function buttons available which are always invocable. These allow the following management functions to be performed:

Navigate: closes all currently visible CombiNets and displays the top level CombiNet. The project manager can double click on aggregate places, or global nets, to expand a level in the project window, displaying the new CombiNet. The project manager can backtrack to the previous CombiNet by closing windows (see Figure 11.9 for a navigated view from a

project manager interface).

New Actor: allows the placement of a new workflow for an actor in the project window. The project manager starts navigating from the top level CombiNet as explained in the *Navigate* function above. The aggregate places and global nets passed through during the navigation provide the path of the new actor. When the project manager finally double clicks on a place, or global place, this is defined as the current location of the new actor. After placing the new actor, the project manager must specify the place, or global place, from which it is assumed the new actor reached the current place. This is in case the design function fails and the new actor has to roll back one place.

Load Net: replaces the currently loaded project window with a modified version. The paths of the actors in the current project are used to try to place the actors in the new project window definition. If any actor can not be placed in the new project window (for example, if a place they passed through does not exist any more) then they must be placed in the new project window by the project manager in the manner described for the *Modify* function.

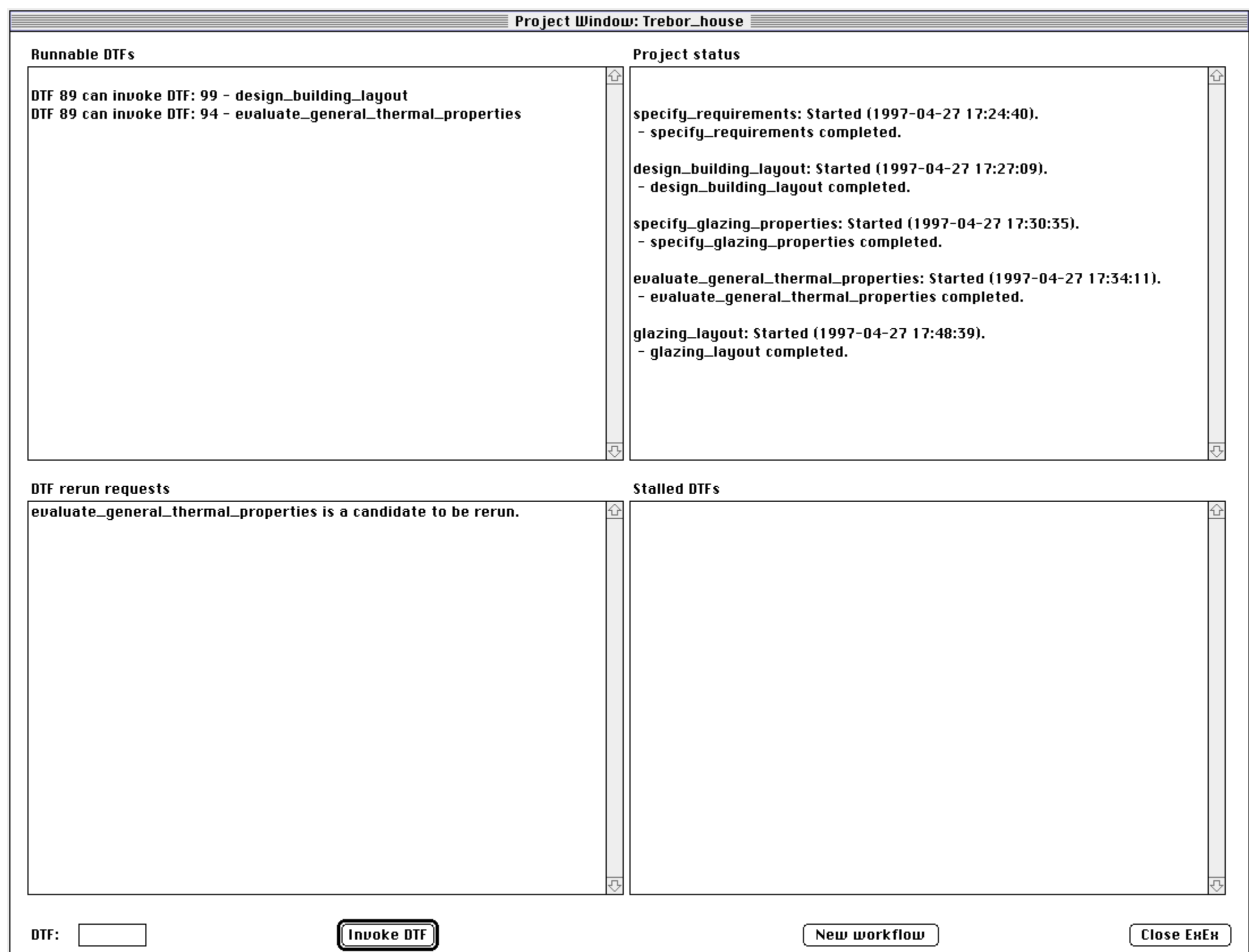


Figure 11.10 Actor's user interface

11.2.4 Actor interface

The actors are each provided with a task level interface to the ExEx, allowing specification of the design function they wish to work upon and determination of the status of the candidate design functions from their current position. This interface concentrates upon an actor's design role and

provides very little information about the status of other actors in the project window. However, any actors working in the same CombiNet (having followed the same path) will be displayed in the actor's interface. An actor will also see any filled tokens denoting an actor working down an aggregate place, or global net, which is represented in the actor's current CombiNet.

The main interface for an actor is shown in Figure 11.10. There are four main sections to this interface providing information about the actor's current state, as described below:

Runnable DTFs: provides a list of design tool functions which are currently invocable. This list is recalculated for the current actor every time an actor in the project window completes or starts a design function. This list is also calculated over the time that an actor is performing a design function, providing a list of design functions that the actor could perform concurrently (e.g., a documentation function at the same time as a simulation function). Apart from the design functions that an actor can perform there are two special states which may be shown in this window, they are:

end: denotes the end of the project window, usually only reachable by one or two actors in the project window. By specifying *end* the actor terminates the project window (assuming that no other tokens exist in the current project window).

hand-over: denotes the hand-over point from one actor to another. The hand-over specifies the actor who takes up the new design role and the design function they will perform to start that role. When choosing a *hand-over* the actor's design role is completed, and the token now represents the actor to whom the design has been passed. When an actor completes their design role they may choose to collect their work into an aggregate transaction representing all the work performed in that design role.

Status: details the work performed by the current actor in their design role. This list details when design functions were started, when design functions completed, if any design functions failed to start, or terminate, normally.

Stalled DTFs: provides a list of candidate design functions which are not currently invocable. This can be due to the functions being performed by other actors, or to the failure of design functions to start or terminate correctly when previously tried.

Rerun requests: lists all design functions previously performed by the actor whose input has changed due to the performance of subsequent design functions. This list is purely informative, and carries no obligation for the actor to re-perform a design function whose input has changed.

As well as being provided with the textual interface to current design functions and the design status, an actor is also presented with a graphical window showing the CombiNet they are currently working in, see Figure 11.5. These views can be navigated in the same way as described for a project manager. As the actor chooses new design tasks to perform, the token representing their position in the CombiNet moves through the graphical view. As they move up or down

between levels in the project window, new graphical views are provided showing their current location.

11.3 Appraisal of Flow Handling

The ExEx provides an implementation of the flow of control specification from Chapter 7, with enough checking and control to manage project windows of the small size demonstrated in the figures in this chapter, through to large project windows (containing several hundred places) as demonstrated in the COMBINE project (Flynn 1994). The interfaces to the ExEx provided for its users provide the required access to the state of a project window and the control required to manage the running of the project window. The project manager has full control over who works in the project window and what functions they perform, but, assuming that the actors work well together and the design progresses without problems, the project manager may not have to do anything to aid the completion of the project window. The actors only have control over their design functions, but can manage all tasks pertaining to their design role with the guarantee that they are not impinging on other actors working in the system.

There are a few aspects of the ExEx which do not provide the level of support or control which was envisaged at the start of the project. These points are elaborated below.

The project manager's ability to dynamically modify the running project window is limited. What is required is a close link between the project specification tool and the ExEx, so that the project manager may make modifications to the project window definition and have them appear in the running project window, without the need to reload the whole project specification as is the case currently.

There is a gap between the set of available design functions displayed to an actor in their control interface and what they can determine from the CombiNet display of their current place. This gap is due to not being able to see the path to all the design functions which are available at any one time. However, there seems to be no easy way to resolve this problem, as available design functions could have been inherited from several CombiNets above the current CombiNet as well as from several CombiNets below through aggregate places and global nets. Place navigation aids are available to provide navigation similar to that offered to the project manager, but these still do not provide a good view of the relationship between the current design function and all the available options.

The mechanism used to calculate stalled design functions provides an overly conservative determination of what can be performed concurrently. This is due to some portions of the IDM being generic (e.g., the geometric model). Almost all design tools draw data from the geometric

definition, so any design function which outputs geometric data will stall almost every other design function in the project window. In some cases this may be sensible, but in the majority it is not necessary. For example, consider a design function which defines the paving around a building. When using this design function, all design functions which require only internal aspects of the building will be stalled as they access the geometry portion of the IDM. One solution would be to mark portions of the IDM as being generic (e.g., geometry, documentation) and not consider these portions when determining stalled design functions. However, this may be problematic for design functions where the use of generic portions of a model does impinge on their operation (e.g., a visualisation tool whose input is almost purely geometry). Another solution is to consider the actual objects used by running design functions when determining what is stalled in collaboration with the intersection of models. However, it is impossible to know whether a design function will produce output which is going to affect a design function which wishes to start up. The method used in the ExEx does guarantee consistency of the model, with the trade-off that design functions may stall unnecessarily.

In a similar vein, the current model of intersection checking does not consider two design functions whose output may partially overwrite each other, for the reasons described in Chapter 7. However, assuming concurrent design in a project window, the current ExEx does not stop a design function writing its data back to the central store, even if it may be overwritten by a design function which does not have permission to overwrite it (e.g., a running design function which should be completed before the design function which wishes to write the data). Stalling design functions whose outputs overlap with currently running design functions would provide a conservative strategy (for the same reasons as mentioned above), but would guarantee consistency of the central model. This problem and the one above indicate that further work is required in determining strategies to manage concurrency and consistency in project windows.

Chapter 12

Conclusions

The research presented in this thesis examines the type of development framework required to plan and implement an integrated design system. The central premise is that existing stand-alone tools lack the functionality required for large or medium sized integration projects. Research in this thesis also recognises that insufficient aspects of an integrated project are modelled, resulting in stand-alone, or poorly connected tools. Although many projects and many hundreds of person years have been spent developing integrated design systems, their development frameworks all have major shortcomings. These shortcomings range from the limited amount of information modelled through to the nonexistence, or low capability, of tools needed to implement prototype or commercial systems. This thesis analyses and identifies the full range of information which needs to be modelled for an integrated design system and demonstrates a range of tools to support this modelling.

The above analysis shows that modelling and integrating purely data aspects in a project is not enough. To be usable in a development project, an integrated design system must take into account the processes involved in the development and final use of the system. Thus a very wide range of modelling methods covering data, processes, documents, legal aspects, activities, etc. (c.f. the IDEF family of models, Mayer et al. 1994) is needed. Support tools are required to facilitate the creation of an integrated design system, and this thesis demonstrates the benefit offered to a project when they are available. Examples in this thesis show the application of these ideas to an integrated building design system. The need for the tools and integrated systems shown in this thesis is currently growing more urgent. Many small to medium sized integrated design environments are under consideration, or even under development. To ensure that these projects complete successfully, and operate as required by those in the industry, tools and modelling as described in this thesis need to be developed and introduced to the industry.

The major framework advances proposed in this thesis fall into three areas. First, more capable modelling tools are required for the development and testing of the very large models which must be specified for many information types (e.g., product and process). This thesis demonstrates the benefits of such environments. Second, formal mapping languages are required to define the correspondences between information in the many views of a domain found in integrated design systems. The thesis defines one such language and shows that such languages can be supported by automated implementations (rather than using the mapping as a coding specification). Third, formal process specification languages need to be adopted to model the intent and usage of tools inside an integrated design system. This thesis defines a process formalism and demonstrates an implementation which can be used to test and control a running integrated design system.

The remainder of this chapter examines the contributions made by this project and the conclusions that can be drawn from this work, and then proposes further work required in this area.

12.1 The Project Development Environment

This thesis has highlighted the need for development environments to help with the creation of integrated design systems. The tools currently available to developers of such environments, and the model specifiers, are totally inadequate for a well managed project. Inadequacies highlighted in this thesis are: an inability to guarantee the consistency and correctness of a model and the data used to test that model; an inability to manage and present overlapping views which make the models more understandable than canonical forms; and an inability to cope with development teams larger than a single person. This makes existing tools poorly suited for use in development projects where models with hundreds of classes are the norm and where the size of the development team is usually greater than ten people. The development tools presented in this thesis show capabilities which tackle and resolve all of these problems.

The use of several modelling methods to capture different aspects of a project, each supported by a development tool and environment, highlights the need for connections between models. Where several models of information in design tools and central models exist it is imperative to provide a method of mapping between the representations. Where processes and activities are defined they must be linked with the data aspects that support the processes and activities. When information in one model changes there are a range of inter-related models which are in some way dependent upon the changed information and require updating. This requires that all of the development environments communicate between themselves when dependent data in their models is modified. All of the development tools demonstrated in this thesis are built upon a platform which allows related modelling systems to be attached with notification of changes to dependant objects. Even during the schema development stages it is clear that all aspects of the project domain need to be consulted to ensure that the schemas capture the domain sufficiently. The best way to ensure this

sufficiency is to model also the related aspects. This has led to models of data, correspondences between data model, processes, activities, actors, and design tool parameters being incorporated into the demonstrated system. Other related aspects also need to be incorporated, including documents and their associated legal aspects for a project, possible conflicts and their management, though these two aspects are both implicit in the constraints which are seen in the process models.

The development environments created in this thesis all provide a common set of functions much needed in the area. This includes the ability to use multiple overlapping views of components, enabling subset views of a schema to be presented for various design processes, thus highlighting different aspects of a schema and the relationships between entities in the schema. This helps ensure the correctness of schemas by allowing concentration of particular aspects of the schema at any one time and makes the schemas more understandable to audiences with different interests. The other main ability is to be able to work with both textual and graphical representations in the same environment. Again this enables views of varying levels of complexity and completeness to be presented to different audiences for different tasks. Supporting these abilities in existing commercial and research tools would have a marked impact in terms of greater certainty of the correctness of the models being developed, a greater ability for them to be understood, and the reduced time that would be required to develop them.

Though there are only simple ties between the schema development and the testing environments in this project, they show the utility of a fast and automated path between specification and testing. The testing environment tools developed for this thesis allow schemas to be instantiated and then visualised, both in terms of values corresponding to data structures and for renditions of graphical definitions, to perform quick checking of schema sufficiency and validity. The tools developed allow for an early and continuous feedback loop into the model development phase of a project. In this way the sufficiency of the models can be demonstrated and checked from early in the model development. These types of tools and their close linkages with modelling environments need to be more widely used to reduce the number of large schemas which are developed from theoretical foundations without the ability to test against actual working needs until late in the project. It is recommended that instantiations of schemas should be developed almost in a direct parallel with the schema development. This usage of test data alongside the model development does however introduce an additional requirement on the linked tools, that being the ability to move test data sets forward to new schema representations as modifications are made. The use of the VML mapping language is recommended for this phase, especially if tied to the semantics of modification functions in the modelling environments, which allow mappings to be built automatically as schemas are modified.

12.2 The Mapping System

Previous research work into integrated design systems has highlighted the necessity to perform mappings between representations of information in a domain, usually between central models and those utilised by the different actors and design tools used in the integrated system. Existing mapping languages and techniques have, however, proved too restrictive for this task, mainly due to assumptions of commonality of semantics between models being mapped. It is certainly not possible to guarantee this between the models of existing design tools, let alone those employed by the actors from different disciplines in a project. This thesis presents an analysis of the types of mapping required between various models and leads to a set of requirements for a mapping langauge. The VML language was specified to meet these requirements and is demonstrated through an interpreted implementation.

The VML language provides a mechanism to describe the correspondences between two schemas. One of the main premises of the language is that bidirectional mappings must be supported (as most non-trivial mappings must be performed in both directions). This led to a high-level declarative language with an in-built assumption that any implementation of the language will provide a mechanism to run mappings in the required direction. VML provides a language of far greater range than allowed in RDBMS views, with the obvious drawback that not all described mappings are invertible (which is guaranteed in an updatable RDBMS view). However, the power of automatically invertible declarations is also greater than that offered in RDBMS views, and demonstrated to work through an implementation which must track all modifications and previous mapping connections. To support non-invertible mapping definitions the VML language offers a fall-back position of defining two procedural mappings to enact the mapping in both directions.

A base VML mapping component offers three main functionalities: a specification of constraints to determine when the mapping can be applied; a set of mappings to be applied; and initial values to be instantiated when new objects are created during a mapping. A VML mapping provides unique functionality in allowing methods in object-oriented systems to be referenced and invoked as part of a normal mapping. For example, it enables the semantics of method calls to be mapped between various systems, especially those which have side-effects outside the scope of the data in the model, e.g., screen display calls.

VML allows a mapping to be specified between two schemas. It also describes what type of mapping can be performed, i.e., read only, read-write or an integrated mapping which forces a consistent state to be achieved before application. A network of VML mappings can be specified between a range of design tools. The examples of this project demonstrate a star topology through a central integrated model. However, many other topologies can be specified.

The implementation of VML in this thesis demonstrates the ability to implement complex VML specifications and run them bidirectionally, as illustrated for a set of tools in the building and construction domain. To enable mappings to be made in either direction the mapping system tracks a large amount of meta-data about the mappings which were previously applied. This allows updates to a mapping to be identified and applied with greater efficiency in the implementation. However, it is clear that the amount of meta-data tracked soon greatly outweighs the amount of data in the individual repositories, especially if many small mappings are applied between two data stores. However, this is a minor cost to be paid for the ability to perform complex mappings bidirectionally between two data-stores. It is clear that any system required to perform efficient mappings between two stores will need to track this level of meta-data. This is the only way to avoid problems of trying to match structures in two existing stores through the mapping definitions.

Trying to compare VML to other mapping languages highlighted the need for a methodology to compare mapping languages, and, underlying this, the need for a formal description of the mapping domain. With a description of the mapping domain it would be possible to describe the power of individual mapping languages, their strengths and weaknesses, and their sufficiency for different types of applications. This has been attempted to some extent in the thesis by selecting features of the mapping problem that needed to be supported for the particular domain. However, these features are overlapping in that they provide different views of similar functions and do not necessarily represent a comprehensive set of mapping language requirements.

As previously noted, it is not possible to apply all declarative mappings bidirectionally. In fact, it is not possible to determine which mappings are possible in a particular integrated environment until the power of the mapping implementation is determined (not all VML specifications need to be implemented in all mapping implementations), and some techniques make more mappings possible, e.g., incorporation of a simultaneous equation solver. So, though a VML specification can describe how a mapping could be performed, it may or may not be calculable in a particular integrated design system. One partial solution to this problem, utilised in this thesis, is to allow constraints to be imposed on particular data sets following a mapping. In this way, though the value for an attribute may not be calculable, it will be possible to determine whether a newly specified value equals the value in the store from which it should have been calculated, or if a new mapping is required to change values in the original store. For example, with *area=height*length* it is not possible to determine *height* or *length* from *area*, but the two attributes can be constrained to equal the value of *area* when multiplied, or if they do not equal *area* at some later stage it requires *area* to be updated with a new application of the mapping.

12.2.1 Additional applications of the mapping language

The VML mapping language is designed to be domain independent, as is the relational database language SQL. Thus it should be applicable to any situation which requires views of a similar

domain to be kept consistent, and where the system needs to be easily extended without having to compile in new features, functions, or tools. Some areas where VML would be immediately applicable are:

RDBMS Views: VML has already been shown to be at least equivalent to the operators in a RDBMS (see Appendix A.3). Therefore, any existing RDBMS view can be described in VML. However, VML allows more powerful views to be defined as there is no assumption that the view and base are semantically equivalent. Therefore, VML could allow more sophisticated RDBMS views to be defined and updated automatically.

Schema Integration: as with RDBMS views, schema integration techniques for RDBMS assume that the schemas being integrated are semantically consistent before integration. In many real-world applications this is not possible to ensure and hence VML would provide the ability to integrate non-consistent schemas, but still maintain the semantics of the individual schemas. Appendix A.3 shows that VML is at least equivalent to RDBMS schema integration operators.

Object-Oriented Views: some object-oriented languages and distributed object environments allow multiple public interfaces to be defined for an object (e.g., the interface description language (IDL) in CORBA, Otte et al. 1996), equivalent to database views except enhanced with method calls. VML provides an alternative mechanism to specify these views and also allows access to methods to allow methods calls to be tracked or invoked during a mapping.

Data Store Migration: some data repositories need to migrate to new schema definitions over their lifetime. Where simple changes are made, such as adding a new attribute, there is no great problem, but where the schema change involves existing attributes in a new form it is often difficult to create the new version of the data store. VML provides a specification and migration environment which can handle this problem.

Loosely Connected Toolkits: many systems exist as a set of loosely connected components or tools. Where each tool has a well defined set of functions and provides services which may be of use to many other tools not always known in advance. VML can be used in such systems to provide the glue to connect each of these tools as needed using a network approach to defining interfaces between associated tools, or through a mapping to a global and common data store.

12.3 The Flow of Control System

The specification of processes and the flows between them has received scant attention in the development of integrated design systems, mainly due to a focus on data and its transmission. However, as previously stated, it has been recognised that this focus on pure data transmission does not provide a system which can integrate with the processes required in the domain that the integrated systems sit in. Though the more recent European integration projects have considered

process and workflow in their systems, this modelling has often been of processes independent of the rest of the system. In this thesis, the full set of interconnections between various aspects of a project's information was considered. This shows dependencies between a process and data, documents, actors, design tools and the influence on process invocation due to legal constraints, standards and regulations, and the state of the executing system.

The CombiNet process modelling formalism addresses many of these issues by defining a two level specification. At one level this defines the actors in a project, their design tasks, and design functions which are tied to data specifications and design tools. The second level of specification provides a hierarchical specification of design functions and their possible flows of control. Varying levels of constraint in the process flows can be defined using either global processes at different levels of the hierarchy to provide unconstrained process flows, or totally connected processes to force a strict flow of control. This level allows the point of actor handover between processes to be further specified, and allows the user to place constraints on the invocation and completion of processes. One of the major benefits of the flow of control specification is that it allows looping or iteration between processes which is not allowed in the major process modelling formalisms due to the difficulty in simulating processes with loops (unless the number of iterations can be pre-determined). However, in the integrated design systems in which this process model is utilised, it is not important to simulate time taken to complete a project (though that is useful), but it is necessary to define the possible flows between design tasks of the actors to control the project. Though CombiNet ties together many aspects of a project there are still links that could be defined. One of these is a link to documents and documentation which feed into a process, or are required at the termination of a process. This would enhance the ability to specify constraints on processes by allowing legality aspects to brought into the project (e.g., whether a stage has been signed off properly).

The implementation of the CombiNet formalism provides a control system for a running project. This includes project manager type overview functions for controlling the work done on the project. It also provides interfaces for actors to specify the design functions on which they are working, and to notify others of their progress. It also allows actors to identify design functions that they are next able to work on, or to specify a handover to another actor in the project. Due to the connections between the different aspects of a project, the flow of control system can track and manage many functions of a running project. It can determine what design functions can run at any time in the project, providing help to those attempting concurrent engineering. It can also identify which actor in a project is hindering other work, allowing the project manager to smooth the running of a project, or to examine business process re-engineering of the processes. As the implemented flow of control system tracks all processes undertaken, it provides a record of task progression to allow actors to show they completed all design tasks in their programme of work.

12.4 Future Work

The work presented in this thesis has examined requirements for an integrated design environment and provided prototypes of components required to make this environment reality. However, not all of the requirements could be addressed within the scope of this work, and several areas for further work have been identified. A selection of the more important areas for future work are addressed below.

12.4.1 Tighter system integration

The integrated design system framework developed in this thesis provides for communication between several of the modelling and testing tools. However, there is benefit in providing for even closer integration of these tools. One approach envisaged for achieving this integration is to extract the underlying model requirements of each of the tools, and develop an integrated schema incorporating these models. In this way, the development framework could provide services similar to those being created in the integrated design system itself. For example, if the schema modelling and process modelling tools utilise a common data store containing the combined models, then by specifying the model portions in which each tool is interested it would be possible for a tool to be notified when changes are made to information it is referencing. This would provide for greater consistency in the total set of models developed for an integrated design system.

Providing these meta-models for tools would also enable a wider range of modelling paradigms to be more easily integrated into the design framework. If generic model concepts are encapsulated in the developed meta-models, then a variety of modelling methods (e.g., NIAM, EXPRESS and ER) could be used, as long as the tools which model with a given methodology can translate their models into the meta-model, or accept models in the meta-model format.

Tighter integration could also apply between the tools defining models and those manipulating and populating those models. In this way, automatic migration of test data sets between versions would be more easily achieved. Modelling of the functionality available in the different tools, and describing how this maps to the associated data manipulation tool is of considerable benefit. For example, when an attribute’s type is modified in a schema development tool, it must identify what function, or set of functions, this maps onto in the data model manipulation tool.

The final outcome of this environment integration would be a set of tools which communicates with all other tools upon which it has an impact. This would apply whether it is a modelling tool or a data manipulation tool. In this environment designers would be sure they are working with the most up-to-date models and that all affected models would be notified of changes made.

12.4.2 Distributed environment

Though all of the tools developed in this system are stand-alone and could be used by a number of concurrent users, multi-user development was not supported in this thesis. The connected tools in this thesis have a very simple view of the information logistics server with which they communicate (i.e., the Macintosh OS). This should, however, be easily scalable to incorporate information logistic servers which handle multi-platform, multi-user and multi-organisational environments. It is envisaged that interfaces to CORBA-like platforms (Otte et al. 1996) would enable the handling of distributed data and distributed functionality for the tools defined in this thesis. The 'mapping through transaction-based updating' approach provides a model which would enable multiple users to work on the same underlying model through a distributed environment and guarantee consistency (though at a high level of granularity).

Providing this extension would make the framework developed in this thesis more palatable to the types of teams which are used today to develop integrated building design systems. These teams consist of several modellers and testers in several organisations, usually also in several countries (for the EU funded projects). For these teams, a distributed environment which manages model and data consistency would be in itself of enormous benefit.

12.4.3 Wider incorporation of project aspects

Throughout this thesis it is argued that integrated design systems must enable all aspects of a project to be encompassed in the final system. The system developed for this thesis shows the integration of data and process aspects with some computing environment information also drawn in. However, there are other areas which need to be brought into the system. Documents and documentation are the most important aspect not currently tied into integrated design systems, though there are many commercial systems available (IIC Consulting and Cimtech Limited, 1996). Documents and their legal aspects are closely examined in the ToCEE project (Amor and Clift, 1997) and the following points noted:

- Documents are the main communication mechanism in any construction project.
- All project information is passed through documents and they are the input to all processes, as well as the output of all design processes.
- Documents are the only legally binding information medium in a project.

Therefore, by incorporating documents, it is possible to track and manage legal aspects of a project as they impact on processes and the data being manipulated (e.g., contracts and sign-off points). The development of models and frameworks to handle wider project aspects can be seen in the recent work on the BCCM in STEP (Building Construction Core Model, ISO/TC184, 1996) and the EU-ESPRIT funded ToCEE project (Towards a Concurrent Engineering Environment, Katranuschkov et al. 1996).

Other aspects which could be considered for incorporation include conflict management and negotiation. In past construction projects, this was limited to clash detection (e.g., does a pipe's position clash with a column's placement). This needs to be extended to allow negotiation between project partners when designers fail to agree on the state of overlapping portions of a design. In some instances this can be managed by providing the project manager with tools to arbitrate disputes and apply decisions to the project team, though in others a dialogue between partners must be supported before decisions are enacted.

A final aspect which could be considered, and which certainly has an impact on the whole system, is that of constraints on a project due to standards and regulations. In some cases they determine what processes can be enacted, or in what order. In many cases they impose pre- and post-conditions on invocation of a process as well as on aspects of the data model and documentation which must be produced.

12.4.4 Formal definitions of the mapping domain

In Chapter 5 it is noted that there is no method available to provide a rigorous comparison between different mapping languages. Part of the reason for this is a lack of understanding in the research community of the scope of the mapping problem and few formal methods to help describe mappings (though Ainsworth et al. 1996 seem to provide a way to approach this problem). Further work is required to define the semantics of mappings and mapping languages as the initial step to allowing mappings to be validated and languages to be compared. Though Chapter 5 details the main requirements of a mapping language, these are drawn from analysis of mappings in a single domain (i.e., construction). The scope of these requirements needs to be validated for a wider set of domains and specified in a more formal way. This more formal specification would then allow the power of related techniques to be gauged (e.g., constraint specification systems and the Interface Description Language (IDL) of CORBA, Otte et al. 1996).

A formal specification of the semantics of mapping languages would also help clarify the use of methods and transactions in a mapping. Currently VML allows method calls to be mapped between models, though the time dependency of the method calls is collapsed through the use of transactions to amalgamate multiple model modifications into a single mapping. A formal treatment of methods and their mapping requirements would provide an insight into where transaction boundaries should be drawn for a mapping.

12.4.5 Alternate mapping language implementations

In this thesis an interpreter for VML was constructed. However, VML could be translated into many other forms. An interesting project would be to examine the requirements for translating VML into other languages. This would clarify the trade-off that has been made between the specification language power and implementation complexity. For example, the translation of

VML into a procedural language is likely to be very difficult even when supported by a powerful set of backing libraries to provide many of the mapping functions. Examining the requirements would enable a decision on the technical feasibility and commercial viability of this translation to be made. The translation requirements would also determine the possibility of extending other systems to handle mappings, for example, constraint solving systems.

12.4.6 Incorporation of measure tools

Once all project aspects are incorporated into an integrated design system, there exists a central repository of all information relevant to a particular project and its progress. This would provide an exciting opportunity to data-mine the workings of a project, either to provide feedback for new projects, or as a monitor while a project is running.

A set of tools incorporated into the integrated design system and development environment could provide measures of the state of the running or developing system. These measures could identify aspects of a project which are: outside of best-practice (e.g., an inheritance hierarchy over twenty entities deep); or bottlenecks (e.g., the structural and HVAC engineers are both waiting for the architect to finish the redesign, they have been waiting 2 days, and this is the third time this month); or performance indicators (e.g., how much various parts of the project deviate from schedule).

Tools to examine completed projects would examine common aspects of a series of projects to identify possible improvements. For example, any possibilities for concurrent engineering in similar projects could be identified, or the feasibility of applying business process re-engineering to reduce the development time, and hence the project cost.

12.4.7 Dissemination and exploitation

The integrated design system development environment proposed in this thesis has been demonstrated for the construction domain. However, the system was devised to be generic and applicable to any domain with similar problems. To this extent, the system should be transferable to most engineering domains as well as medical domains and software engineering domains, all of which require the development of similar integrated systems. In many cases it is likely that such a system would make more impact, as these domains are further advanced in terms of data models developed and used, their understanding of process, the legal implications of getting it wrong, and the profits to be made by improvements.

In terms of general software systems there are several existing data management systems which could be improved through the use of a system as described in this thesis. RDBMS specification, development and maintenance could be subsumed by an integrated design system such as demonstrated, providing the same wide range of benefits as well as providing more powerful data

views for the database users. The same approach could be taken for heterogeneous database systems as long as a more sophisticated information logistics system was incorporated. Again this would provide greater power in the data views supported.

Appendix A

The View Mapping Language

VML is a major component of this thesis. This appendix provides the technical specification of its syntax, the graphical notation, and comparisons to other methods to show equivalence.

A.1 VML Syntax

The syntax of the mapping language is detailed below in two parts. The first part provides high-level structures in the language, and the second part provides low level definitions for basic structures such as strings and numbers. The starting point of the syntax below, which is listed alphabetically, is the *mapping* specification.

```
and_op = ',' .
attribute_id = simple_id | variable_id .
attribute_name = class_name '.' ( attribute_ref | 'SELF' ) .
attribute_ref = attribute_spec [ attribute_ref_tail ] .
attribute_ref_tail = '=>(' expression ')' | '=>' attribute_spec [ attribute_ref_tail ] .
attribute_spec = attribute_id [ '[' [ list_id { ',' list_id } ] ']' ] .
attribute_value = [ [ model_id ':' ] class_id '.' ] attribute_ref | variable_id .
bijection_expr = class_name | attribute_name | invariant_simple_expr .
class_id = simple_id .
class_list = '[' [ class_list_name { ',' class_list_name } ] ']' .
class_list_name = 'group(' class_name ')' | class_name .
class_name = ( [ model_id ':' ] class_id | variable_id )
       [ '[' [ list_id { ',' list_id } ] ']' ] .
constant_values = 'pi' | atom_literal | integer_literal | real_literal | string_literal
       | boolean_literal .
equivalences_def = 'equivalences(' equivalent { ',' equivalent } ')' .
equivalent = expression '=' expression | 'map_to_from(' predicate ',' predicate ')'
       | 'bijection(' bijection_expr ',' bijection_expr ')' | predicate .
expression = term { add_like_op term } .
factor = simple_factor { '^' simple_factor } .
function = function_1_arg | function_2_arg .
function_1_arg = 1_arg predicate_expr ')' .
```

```
function_2_arg = 2_arg predicate_expr ',' predicate_expr ')' .
group_expression = '(' expression { ',' expression } ')' .
inherits = 'inherits(' inherit_list ')' .
inherit_list = inherit_map { ',' inherit_map } .
inherit_map = 'inter_class(' class_list ',' class_list ')' .
initialiser = expression '=' expression | method | predicate .
initialisers_def = 'initialisers(' initialiser { ',' initialiser } ')' .
inter_class_def = 'inter_class(' class_list ',' class_list [ ',' inherits ]
       [ ',' invariants_def ] [ ',' equivalences_def ] [ ',' initialisers_def ] ')' '.' .
inter_view_def = 'inter_view(' model_id ',' model_type ',' model_id ',' model_type ','
       map_type ')' '.' .
invariants_def = 'invariants(' invariant_expr { or_op invariant_expr } ')'.
invariant_expr = invariant_simple_expr { and_op invariant_simple_expr } .
invariant_simple_expr = '(' invariant_expr { or_op invariant_expr } ')'
       | expression rel_op expression | predicate | function | method
       | 'group(' attribute_name { ',' attribute_name } ')' .
list_expression = '[' expression { ',' expression } ']' .
list_id = integer_literal | attribute_value .
mapping = inter_view_def { inter_class_def } .
map_type = 'complete' | 'partial' .
method = ( [ [ model_id ':' ] class_id '.' ] method_attribute_ref | [ model_id ':' ] class_id
       | ϶ ) '@' ( method_id | predicate ) .
method_attribute_ref = attribute_spec [ method_attribute_ref_tail ] .
method_attribute_ref_tail = '=>' attribute_spec [ method_attribute_ref_tail ] .
method_id = simple_id .
model_id = simple_id [ '{' version '}' ] .
model_type = 'integrated ' | 'read_only ' | 'read_write ' .
or_op = ';' .
predicate = predicate_id '(' predicate_expr { ',' predicate_expr } ')' .
predicate_expr = '[' predicate_expr { ',' predicate_expr } ']' | expression .
predicate_id = simple_id .
simple_factor = group_expression | list_expression | function | predicate | attribute_name
       | method | constant_values.
term = factor { multiplication_like_op factor } .
version = integer_literal | real_literal | atom_literal | string_literal .
1_arg = 'abs(' | 'average(' | 'cos(' | 'cos_1(' | 'count(' | 'deg_rad(' | 'exp(' | 'int('
       | 'ln(' | 'maximum(' | 'minimum(' | 'rad_deg(' | 'sign(' | 'sin(' | 'sin_1(' | 'sqr('
       | 'sqrt(' | 'sum(' | 'tan(' | 'tan_1(' .
2_arg = 'pwr(' | 'pwr_1(' .

%------------------------------------------------------------
% Low level definitions for strings, numbers, comments, etc.
%------------------------------------------------------------
add_like_op = '+' | '-' .
atom_literal = \q { atom_literal_char } \q .
atom_literal_char = character | ' ' | '"' .
boolean_literal = 'false' | 'true' .
character = digit | letter | special .
digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' .
digits = digit { digit } .
embedded_remark = '/*' { embedded_remark_el } '*/'.
embedded_remark_el = not_lparen_star | lparen_not_star | star_not_rparen | embedded_remark .
integer_literal = digits .
letter = lower_case | upper_case .
logical_literal = 'false' | 'true' .
lower_case = 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm'
       | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z' .
lparen_not_star = '/' not_star .
multiplication_like_op = '*' | '//' | '/' | 'mod' .
not_lparen_star = not_paren_star | ')' .
not_paren_star = letter | digit | not_paren_star_special .
not_paren_star_special = '!' | '@' | '#' | '$' | '%' | '^' | '&' | '_' | '(' | ')' | '-'
       | '+' | '=' | '{' | '}' | '[' | ']' | '~' | ':' | ';' | '''' | '<' | '>' | ','
       | '.' | ' ' | \t | \n | '?' | '/' | '|' | '\' .
not_rparen = not_paren_star | '*' .
```

```
not_star = not_paren_star .
real_literal = digits '.' [ digits ] ( 'e' | 'E' ) [ sign ] digits .
rel_op = '<=' | '>=' | '<>' | '<' | '>' | '=' | '=\=' | '=:=' .
remark = embedded_remark | tail_remark .
sign = '+' | '-' .
simple_id = lower_case { simple_id_char } .
simple_id_char = letter | digit | '_' .
special = not_paren_star_special | '*' | '/' .
star_not_rparen = '*' not_rparen .
string_literal = '"' { string_literal_char } '"' .
string_literal_char = character | ' ' | \q .
tail_remark = '%' { tail_remark_char } \n .
tail_remark_char = character | ' ' .
upper_case = 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K' | 'L' | 'M'
       | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z' .
variable_id = upper_case { simple_id_char } | '_' ( letter | digit ) { simple_id_char } .
whitespace = { whitespace_char } .
whitespace_char = ' ' | \n | \t | remark .
```

A.2 VML Graphical Notation

The graphical notation for VML was created in order to specify high-level over-views of mappings. It comprises few components and is basically a wiring notation. Figure A.1 shows the main components of the notation.

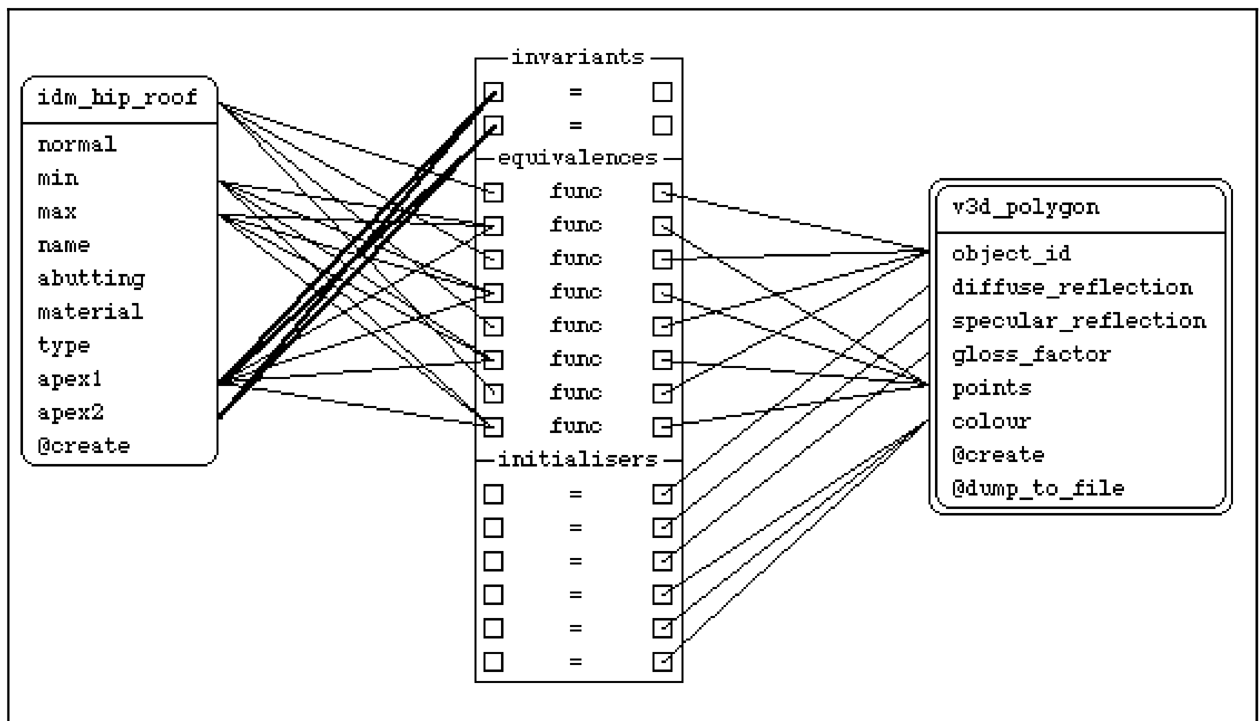


Figure A.1 VML graphical notation

Classes being mapped between are denoted by rounded rectangles. Normally mapped classes are shown with a single line (e.g., *idm_hip_roof*) whilst those which are grouped in the mapping are shown with a double line (e.g., *v3d_polygon*). An *inter_class* definition is shown by a rectangular box broken into three sub-components, one each for *invariants*, *equivalences* and *initialisers*. Individual objects and attributes involved in a single equation are wired to the same line in the

inter_class specification. Four different types of equation can be specified in this notation, straight equivalence (=), more complicated equations (eqn), functions (func), and procedures (proc).

A.3 VML Comparison to other Notations

VML provides greater functionality than current languages for specifying views of data, though these languages are based on a formal specification able to guarantee integrity of data. To help the claim of VML's power, it is necessary to show its equivalence to existing methods. Here the equivalence to relational operators and those used in one method of database integration are shown.

A.3.1 Comparison to database operators

In this section it is shown that VML has an equivalent expressive power to the five basic operators required in a relational database system (Ullman 1982), i.e., projection, selection, union, set difference and cartesian product. After this, the emulation of some of the more useful higher order functions used in relational databases is shown.

A.3.1.1 The five basic operators from relational databases

1) Project

Given a source entity with attributes $S_1..S_m$ and a target entity with attributes $T_1..T_k$, then a project is accomplished with the following mapping:

```
inter_class([S], [T],
        equivalences(...,
                        Si = Tj, ...)
        ).
```

2) Select

A select statement is accomplished through the use of the invariants section of a mapping. Given a source entity S, with attributes $S_1..S_m$, and some set of boolean formulas F which work on attributes in S to give a target entity T then a select is accomplished as follows:

```
inter_class([S], [T],
        invariants(F),
        equivalences(S1 = T1,
                        ...,
                        Sm = Tm)
        ).
```

3) Union

The union of entities R and S, with attributes $R_1..R_m$ and $S_1..S_m$, to a target entity T is performed by specifying two mapping definitions with no invariant definition such as:

```
inter_class([R], [T],
        equivalences(R1 = T1,
                     ...,
                     Rm = Tm)
        ).

inter_class([S], [T],
        equivalences(S1 = T1,
                     ...,
                     Sm = Tm)
        ).
```

4) Set difference

The difference between entities R and S, which have key attributes $R_1..R_k$ and $S_1..S_k$ and attributes $R_1..R_m$, to a target entity T is specified by checking that R does not exist in the set of S entities. To do this a function called *same_key* is introduced. This function is assumed to return true if an object has the same key as any of a set of objects passed as the second parameter. In this case S is grouped in the header which means that the set of all S are associated with each mapping. Therefore, the reference to S in the *same_key* function will be substituted with the whole set of S objects when called.

```
inter_class([R, group(S)], [T],
        invariants(not(same_key(R, S))),
        equivalences(R1 = T1,
                     ...,
                     Rm = Tm)
        ).
```

5) Cartesian product

The cartesian product of entities R and S, with attributes $R_1..R_m$ and $S_1..S_n$, to a target entity T with attributes $T_1..T_{m+n}$ is simply specified as below:

```
inter_class([R, S], [T],
        equivalences(R1 = T1,
                     ...,
                     Rm = Tm,
                     S1 = Tm+1,
                     ...,
                     Sn = Tm+n)
        ).
```

A.3.1.2 Higher order operators from relational databases

Intersection

The intersection between entities R and S, which have key attributes $R_1..R_k$ and $S_1..S_k$ and attributes $R_1..R_m$, to a target entity T is specified with an invariant to find all the intersecting objects in the consideration set and then projecting the attributes of R as follows:

```
inter_class([R, S], [T],
        invariants(R1 = S1, ..., Rk = Sk),
        equivalences(R1 = T1,
                        ...,
                        Rm = Tm)
        ).
```

Join

This is basically a cartesian product coupled with an invariant. The join of entities R and S, with attributes $R_1..R_m$ and $S_1..S_n$, under the conditions F to a target entity T with attributes $T_1..T_{m+n}$ is simply specified as below:

```
inter_class([R, S], [T],
        invariants(F),
        equivalences(R1 = T1,
                        ...,
                        Rm = Tm,
                        S1 = Tm+1,
                        ...,
                        Sn = Tm+n)
        ).
```

Natural join

Again this is very similar to a cartesian product, but this time the invariant is composed of a key in the source entities. The natural join of entities R and S, with attributes $R_1..R_m$ and $S_1..S_n$, and keys $R_1..R_k$ and $S_1..S_k$ to a target entity T with attributes $T_1..T_{m+n-k}$ is simply specified as below:

```
inter_class([R, S], [T],
        invariants(R1 = S1, ..., Rk = Sk),
        equivalences(R1 = T1,
                        ...,
                        Rm = Tm,
                        Sk+1 = Tm+1,
                        ...,
                        Sn = Tm+n-k)
        ).
```

A.3.2 Comparison to Motro virtual integration operators

In this section it is shown that VML has equivalent expressive power to the ten operators defined in Motro (1987) for the definition of superviews for multiple databases. Some of these mappings will look rather simple (for example the rename operator) as the main scope of Motro is the

definition of a superview model, whereas VML is more concerned with the definition of equivalences between existing models. However, as discussed in the thesis, it is possible to use the mapping system to perform model integration so the comparison to an existing model integration language is of benefit.

1) Meet

The meet of two entities S and T with attributes $S_1..S_n$ and $T_1..T_p$, which share a common key $S_1..S_k$ and $T_1..T_k$ and have shared attributes $S_1..S_m$ and $T_1..T_m$ (where $m > k$ and $n > m$ and $p > m$) to three entities S', T' and U where U is the common generalisation of S and T is as follows (note that U is not seen, as both S' and T' inherit from it):

```
inter_class([S], [S'],
        equivalences(S1 = S'1,
                     ...,
                     Sn = S'n
        ).

inter_class([T], [T'],
        equivalences(T1 = T'1,
                     ...,
                     Tp = T'p
        ).
```

2) Join

The join of two entities S and T with attributes $S_1..S_n$ and $T_1..T_p$, which share a common key $S_1..S_k$ and $T_1..T_k$ (where $n > k$ and $p > k$) to the entity U where U is the generalisation of S and T is defined as follows:

```
inter_class([S, T], [U],
        invariants(S1 = T1, ..., Sk = Tk),
        equivalences(S1 = U1,
                     ...,
                     Sn = Un,
                     Tk+1 = Un+1,
                     ...,
                     Tp = Un+p-k
        ).
```

3) Fold

The fold of two entities S and T with attributes $S_1..S_n$ and $T_1..T_p$, where T is a generalisation of S ($n > p$ and $S_1 = T_1$ .. $S_p = T_p$) to the entity T' where T' contains all of S and T is defined as follows:

```
inter_class([T], [T'],
        invariants(var(T'p+1), ..., var(T'n)),
        equivalences(T1 = T'1,
                     ...,
                     Tp = T'p
        ).

inter_class([S], [T'],
        invariants(nonvar(T'p+1); ...; nonvar(T'n)),
        equivalences(S1 = T'1,
                     ...,
                     Sn = T'n
        ).
```

4) Rename

Renaming the entity S to the entity T is achieved as follows:

```
inter_class([S], [T],
        equivalences(S1 = T1,
                     ...,
                     Sn = Tn
        ).
```

5) Combine

Combining the equivalently typed entities S and T to the entity U is defined as follows:

```
inter_class([S], [U],
        equivalences(S1 = U1,
                     ...,
                     Sn = Un
        ).

inter_class([T], [U],
        equivalences(T1 = U1,
                     ...,
                     Tn = Un
        ).
```

6) Connect

Connecting the entities S and T with attributes $S_1..S_n$ and $T_1..T_p$, where the type of S is contained in T (so $n < p$) to the entity T' which is equivalently typed to T is defined as follows:

```
inter_class([T], [T'],
        invariants(nonvar(T'n+1); ...; nonvar(T'p)),
        equivalences(T1 = T'1,
                     ...,
                     Tp = T'p
        ).
```

```
inter_class([S], [T'],
        invariants(var(T'n+1), ..., var(T'p)),
        equivalences(S1 = T'1,
                        ...,
                        Sn = T'n
        ).
```

7) Aggregate

The aggregation of S to classes S' and T is defined as follows:

```
inter_class([S], [S', T],
        equivalences(S1 = S'1,
                        ...,
                        Sm = S'm,
                        Sm+1 = S'.T=>T1,
                        ...,
                        Sn = S'.T=>Tn-m
        ).
```

8) Telescope

Telescoping the entities S and T into S' is defined as follows:

```
inter_class([S, T], [S'],
        equivalences(S1 = S'1,
                        ...,
                        Sn = S'n,
                        S.T=>T1 = S'n+1,
                        ...,
                        S.T=>Tp = S'n+p
        ).
```

9) Add

The addition of an implied attribute S'_{n+1} to the entity S with attributes $S_1..S_n$ to give the entity S' is defined as:

```
inter_class([S], [S'],
        invariants(S'n+1 = function_result),
        equivalences(S1 = S'1,
                        ...,
                        Sn = S'n
        ).
```

10) Delete

The removal of a non relevant attribute S_n from the entity S with attributes $S_1..S_n$ to give the entity S' is defined as:

```
inter_class([S], [S'],
        equivalences(S1 = S'1,
                        ...,
                        Sn-1 = S'n-1
        ).
```

Appendix B

Project Specification Language

The project specification language as described in Sections 7.2 and 7.3 of this thesis consist of a graphical notation, as all information is ascertainable in this form. However, to be transferred to the ExEx described in Chapter 11 there is a schema specification which is used to define what should be transferred. The schema definition of the CombiNet is included below, followed by a description of the graphical icons used in the notation.

B.1 Project Model Transfer Syntax

The schema definition for project window specification is given below in the EXPRESS modelling language.

```
SCHEMA pw_reference_model;

TYPE schema_filename = STRING;
END_TYPE;

TYPE design_tool = STRING;
END_TYPE;

ENTITY project_window;
        called : STRING;
        requires : SET [1:?] OF actor;
        scheduled_by : combi_net;
        UNIQUE called;
END_ENTITY;
```

```
ENTITY actor;
	has_name : STRING;
	performs : SET [1:?] OF design_role;
	viewable_schema : schema_filename;
	modifiable_schema : schema_filename;
	UNIQUE has_name;
END_ENTITY;

ENTITY design_role;
	has_role : STRING;
	requires : SET [1:?] OF design_function;
	UNIQUE has_role;
END_ENTITY;

ENTITY design_function;
	has_function : STRING;
	is_performed_by : OPTIONAL design_tool;
	has_input_schema : OPTIONAL schema_filename;
	has_output_schema : OPTIONAL schema_filename;
	UNIQUE has_function;
END_ENTITY;

ENTITY combi_net;
	called : STRING;
	places : SET [1:?] OF abstract_place;
	transitions : SET [2:?] OF abstract_transition;
	actors : OPTIONAL SET [1:?] OF actor_overlay;
END_ENTITY;

ENTITY abstract_place
	ABSTRACT SUPERTYPE OF (
		ONEOF (invocable_place, global_place, global_net));
	x_loc : INTEGER;
	y_loc : INTEGER;
END_ENTITY;

ENTITY invocable_place
	ABSTRACT SUPERTYPE OF (
		ONEOF (place, aggregate_place))
	SUBTYPE OF (abstract_place);
	exits_to : SET [0:?] OF transition;
END_ENTITY;

ENTITY place SUBTYPE OF (invocable_place);
	represents : design_function;
END_ENTITY;

ENTITY aggregate_place SUBTYPE OF (invocable_place);
	represents : combi_net;
END_ENTITY;

ENTITY global_place SUBTYPE OF (abstract_place);
	represents : design_function;
END_ENTITY;

ENTITY global_net SUBTYPE OF (abstract_place);
	represents : combi_net;
END_ENTITY;
```

```
ENTITY abstract_transition
        ABSTRACT SUPERTYPE OF (transition, double_transition);
        x_loc : INTEGER;
        y_loc : INTEGER;
END_ENTITY;

ENTITY transition SUBTYPE OF (abstract_transition);
        invokes : SET [0:?] OF invocable_place;
END_ENTITY;

ENTITY double_transition SUBTYPE OF (abstract_transition);
        invokes : SET [2:2] OF invocable_place;
END_ENTITY;

ENTITY actor_overlay;
        represents : actor;
        covers : SET [1:?] OF abstract_place;
END_ENTITY;

END_SCHEMA;
```

B.2 Project Modelling Graphical Notation

The project specification is detailed in two separate types of diagram. The icons of the user and function modelling are described below, followed by the icons in the flow of control specification.

B.2.1 User and function specification

There are three icons available for user and function specification at this level. These icons are shown in Figure B.1. From left to right they allow specification of actors involved in a project, their design roles, and the design functions associated with those design roles. A fuller description is provided below:

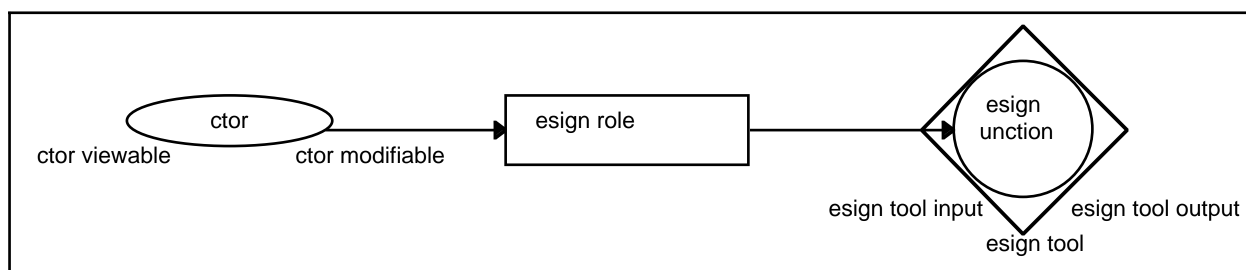


Figure B.1 Icons used for user and function specification

actor: every person involved in a project must be specified at this level. The actor icon allows the name or type of person to be identified along with two schema definitions which provide the specification of the viewable portion of the integrated data model for the actor, and the modifiable portion of the integrated data model for the actor. The actor can be connected to multiple design roles which defines their sphere of influence and activities in a project.

design role: the design role is a high-level definition of a particular role required by an actor. The design role icon allows the name of the role to be specified. Design roles are broken into several design functions which define the tool supported activities which are required to complete the design role. Multiple actors can access the same design role and the design role can be connected to multiple design functions.

design function: a design function is an identifiable portion of work, usually associated with a single design tool. The design function allows the name of the function to be specified and, if a design tool is used to perform the design function, the name of the design tool to be used along with the input schema for the design tool and its output schema. Several design functions can use the same design tool. However, in this case the input and output schemas are likely to be different to reflect the work performed by the design tool for different design functions (i.e., limit what can be written back to the IDM to reflect the function being performed by the design tool).

A convention used in the CGE tool created to support this formalism is to have a title block for each diagram. In the CGE tool there may only be one diagram for user and function specification. The project name specified in the title block is treated as the name of the top level CombiNet for the project window.

B.2.2 Flow of control specification

There are seven icons available for flow of control specification at this level. These icons are shown in Figure B.2. A set of the icons shown below can be grouped together to create what is called a single CombiNet. As most project window specifications will require hundreds of design functions to be specified it is likely that multiple CombiNets will be required to show the whole project specification. To allow this, the notation allows for the aggregation of items into named CombiNets. Again, in the CGE tool, each CombiNet has a title block. The name of a CombiNet is specified in this title block, and becomes the name used to reference the whole set of icons in any other CombiNet. A fuller description is provided below:

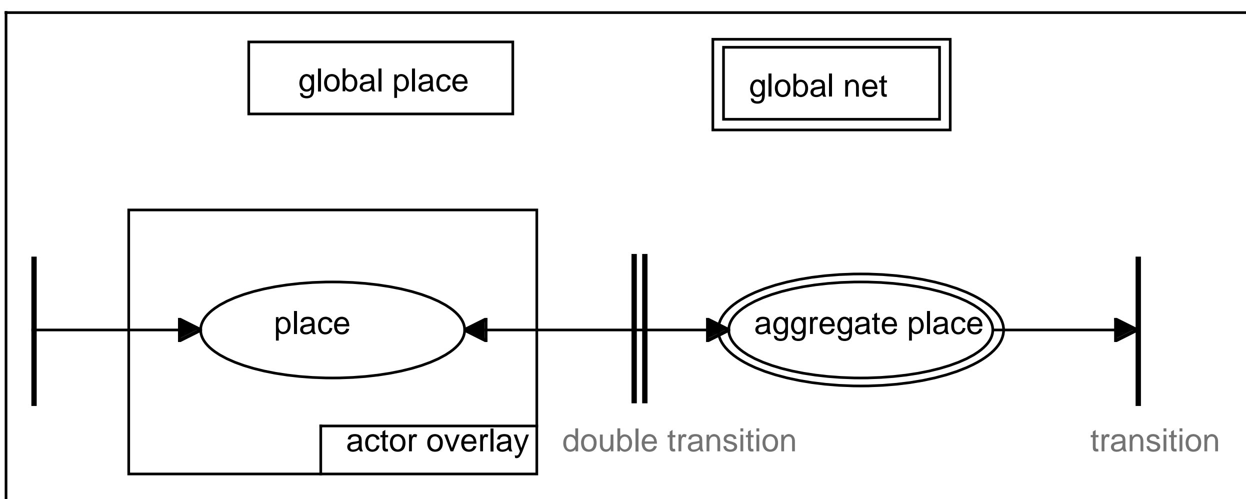


Figure B.2 Icons used for flow of control specification

place: a place represents one of the design functions specified in the user and function diagram. Each place icon is labelled with the name of the design function. If the design function is only used by a single actor then it can be calculated who is able to use the design function. If several actors can use the design function then an actor overlay (see specification below) is needed to define who will be using the design function in this part of the workflow. Places are joined to other places or aggregate places through transitions as described below. Multiple arrows may leave a place to connect to transition icons (see description below), and define possible flows of control. Multiple arrows may point to a place representing multiple paths to the design function.

aggregate place: these icons, very similar to a place, represent a whole CombiNet. The label in the aggregate place defines the name of an existing CombiNet whose components will be substituted for the icon in an implemented system. Arrows entering an aggregate place are implicitly tied to the starting transition of the named CombiNet, whilst arrows leaving an aggregate place come from the end transitions of the named CombiNet (see transitions below). Multiple arrows may leave an aggregate place to connect to transition icons (see description below), and define possible flows of control. Multiple arrows may point to an aggregate place representing multiple paths to the CombiNet.

global place: a global place represents a design function that can be accessed at any time (i.e., it is not sequenced). Therefore it is visible from any point in the CombiNet in which it is specified, and all CombiNets that are accessible from this CombiNet. When a global place terminates, the flow of control is returned to the point from which it was called.

global net: this represents a CombiNet which can be accessed at any time (i.e., it is not sequenced) and its visibility is the same as for a global place.

actor overlay: This box is drawn over all design functions and CombiNet specifications which are utilised by a particular actor (where this is not uniquely defined by the user and function diagram). When drawn over global nets and aggregate places it affects everything in the CombiNet that those global nets and aggregate places represent. Actor overlays may overlap to specify design functions or CombiNets which can be followed by different actors.

transition: these are used to sequence the workflow in a CombiNet. Many arrows may lead to a transition representing flow of control from the termination of design functions and CombiNets. Many arrows may leave from a transition representing the possible flows of control available. A transition with no arrows leading to it is called a starting transition and represents a possible starting point for a CombiNet. A transition with no arrows leaving from it is called an end transition and represents the completion point of a CombiNet. If the CombiNet is at the top level for a project then its end transitions define the end of a project, or project window. For lower level CombiNets, the end transition represents the interface back to lines leading from the point at which the CombiNet was referenced.

double transition: this is a shorthand notation to represent two transitions forming a loop between places or aggregate places.

Appendix C

Snart

The Snart language, in which the majority of this thesis has been implemented, was under continuous development over the life of this research. Started for use in another PhD thesis (Grundy 1993), it was taken up for teaching purposes and other research projects (Mugridge et al. 1995). Work on this thesis highlighted support areas which did not exist in Snart at the time. To enable Snart to be used efficiently for this thesis several extensions were proposed and implemented. Some of these are now part of the basic Snart release (Mugridge et al. 1995). This appendix describes extensions to Snart for the specification of facets for attributes, a query language, object spaces, persistency, and a dynamic object viewer.

C.1 Facets

Previous research work by the author (Amor 1991) identified the requirement to specify additional meta-information about attributes in a schema (like a data dictionary in relational database systems). This allowed information to be associated with an attribute such as units for values, default values, constraints, textual description, and who asserted a value. The frames-based system developed for ICAtect (Amor 1991) supported this concept. When working on this thesis it was recognised that facet information would again be required, and this was not supported by Snart. To accommodate this requirement, the syntax of the language was extended to allow arbitrary facet information to be specified in a schema definition. The snippet of syntax diagram below shows what was added to allow facets to be represented.

```
attribute =  name '(' 'facets(' facet_list ')' ')' | name ':' 'facets(' facet_list ')' |
       name '(' attribute_type ')' | name ':' attribute_type .
facet_list = '[' [ facet { ',' facet } ] ']' .
facet = facet_name '(' value ')' .
/* The list below is not complete */
facet_name = 'type' .
facet_name = 'unit' .
facet_name = 'default' .
facet_name = 'asserted_by' .
facet_name = 'constraints' .
facet_name = 'description' .
facet_name = prolog_atom .
```

It was also necessary to extend the language to allow facet values to be set and to allow facet values to be retrieved. The following snippet of syntax diagram shows the additions to the Snart language to support these requirements.

```
snart_statement = attribute_value | facet_value | method_call | attribute_assignment |
       facet_assignment .
facet_value = object_id '@' name '@@' facet_name '(' value ')' .
facet_assignment = object_id '@' name '@@' facet_name ':=' value .
```

The set of facets able to be defined for a single attribute is not constrained to those specified in the schema. New facets can be added to an attribute dynamically. Facet information was used heavily in the mapping system (see Section 10.3.11) to determine what values in a mapping took precedence when choices had to be made about how to solve equations referencing several attributes. The instance management system developed for EPE (see Section 9.2.1) also allowed an attribute's facet information to be viewed and modified.

C.2 Query Language

This sub-appendix describes the query language provided for the object-oriented language Snart (Mugridge 1994, following on from Grundy 1993). This query language provides a very powerful method to determine sets of objects which match certain criteria. Development of this language is partially in response to the Snart language supporting multiple persistent object-spaces as well as normal non-persistent object-spaces (where a persistent space may be used in a similar manner to a relational database system), and partly through the development of an inter-schema mapping language (see Chapter 5) which requires groups of objects to be identified in order to use a mapping specification.

C.2.1 Introduction

There are many occasions when it is useful to find a set of objects which match a certain criteria. In some circumstances all objects of a certain class are needed, in others objects whose attributes have a particular value are required. So it is of no surprise that many object-oriented database systems have followed the path of their relational counterparts and offer a query language, in most cases a variant of SQL. However, a straightforward adaption of SQL to the object-oriented world

is not always possible. Notions of tables, relations and allowable operations in the relational database field often have no fixed counterpart in the object-oriented field. Methods developed to evaluate conditions and optimise queries in the relational database field can not be taken directly through to the object-oriented field. New methods of evaluating queries for object-oriented database systems have been developed and some of these ideas have been incorporated into this query language (see the description of language implementation in a later section).

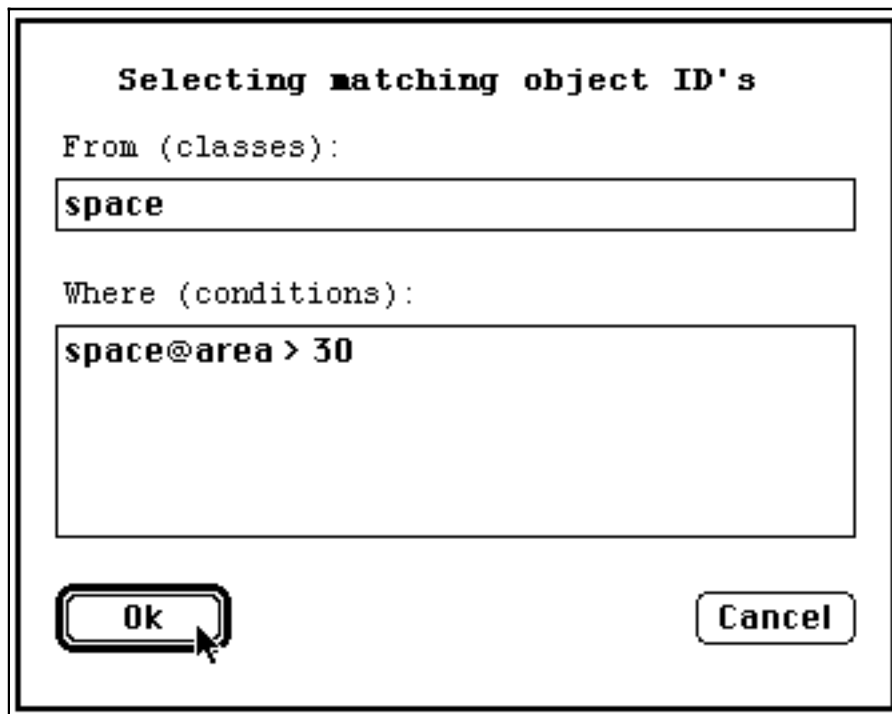


Figure C.1 Snart query language interface

The query language is accessible in two forms. For the users of the Snart programming environment, there is a query window interface invoked from the pull-down menus. This allows the user to specify queries and have the resultant objects ID’s displayed (see Figure C.1 which displays a query to return all space objects with an area greater than 30m^2). For Snart programmers, there is a search predicate through which a programmer can specify a search condition and be returned a list of matching object ID’s. This second form of access to the query system is an important feature for the work on providing multiple views of a base model through the inter-schema mapping language of Chapter 5. One critical requirement of a view maintenance system is to be able to define specific criteria which must be met before a mapping is performed. The query language is the vehicle through which this portion of the mapping system can be achieved.

C.2.2 Example Schemas

To illustrate the querying system throughout this section I will define two simple classes which can be used in all further examples. These are a space and wall class as detailed below:

```
class(space,
      features(
            height : float,
            area : float,
            connect_space : space,
            walls : list(wall)
      )
).

class(wall,
      features(
            height : float,
            width : float,
            wall_type : atom,
            my_space : space,
            position : [float, float, float],
            calc_ratio(float),
            is_exterior
      )
).
```

C.2.3 Old Style Queries in Snart

The original Snart querying system was tiered in three levels, this structure has been modified in the new system. One could query by the exact object identifier, in which case one object was returned, if it existed. Alternatively, one could query through a combination of the class name and a set of attributes value pairs. The class name and conditions could be applied simultaneously or independently to give differing sets of objects in return. One could also select objects in a given object space, which narrowed the search even further. The structure of the query for class and attribute querying was:

```
sn_find_objects(+ClassName(s), +Conditions, -TheObjects)
sn_find_objects(+ObjectSpace, +ClassName(s), +Conditions, -TheObjects)
```

The previous form of *Conditions* was either a single attribute with a value, or a list of attributes with values which were evaluated in a conjunctive form. For example:

```
:- sn_find_objects(end_of_file, height(2.4), X).
:- sn_find_objects(end_of_file, position([1,2,0]), X).
:- sn_find_objects(end_of_file, [height(2.4),position([1,2,0])], X).
```

These queries would return all objects whose attributes matched exactly the given values. The *end_of_file* value for *ClassName* signifies that it was not to be included in the query. In these examples, query one would return both *wall* and *space* objects since they both had a height attribute, while query two and three could only return *wall* objects.

C.2.4 New Style Queries in Snart

The simple equivalence offered in the old style Snart query language is quite limited and a more flexible description of the objects to be selected is useful in many applications. In fact for the

mapping language it is imperative to have a very flexible and powerful query language, and the requirements of the mapping language has driven the functionality offered in this query language. The call for the new query system is almost identical to previously, i.e.,

```
sn_find_objects(+ClassList, +Conditions, -TheObjectTuples)
sn_find_objects(+ObjectSpace, +ClassList, +Conditions, -TheObjectTuples)
```

Conditions must be passed as a term of the individual conditions to be satisfied. There are two special modes of use of this predicate that users should be aware of. If the *ClassList* contains a list of classes and there are no conditions which bind objects from different classes, then the result is a list of all objects of the classes defined (rather than the cross-product of the objects of the named classes). If no classes are specified in the *ClassList*, but there are conditions, then these conditions are applied to all objects.

In the previous query language it was only possible to describe queries pertaining to one class type, in this query language any number of classes can be collected and searched as part of the query. To this extent, the result of a query will be a set of tuples of objects of each class specified in the query and which matches the specified conditions. Put in another way, a multi-class query forms the cross product of all the objects from all the classes and returns only those which match the specified query. As this initial cross-product could be enormous, we employ several methods of reducing the size of tuples which must be examined at any one time (see the later discussion of the language implementation). The various components of the query language are demonstrated in the remainder of this section.

Class, Attribute and Method Referencing

To reference a particular attribute of a class in a query the user should specify both the class and attribute names, though where a query references just one class, or where the attribute name is unique in the classes referenced, the class name can be omitted (though this is strongly recommended against). The object identifier of the individual objects being matched can also be specified in the query through the use of the self keyword. The methods attached to a class can also be invoked and are expected to be in one of two forms. Either the method is called with all parameters specified in the condition, in which case the method must succeed or fail; or the last parameter of the method is not specified in the query and this parameter is used to return the result of the method invocation. For example:

ClassName@Attribute	e.g., `wall@height > 2`
Attribute	e.g., `height = 2.2`
ClassName@self	e.g., `space@connect_space = space@self`
ClassName@Method()	e.g., `wall@calc_ratio > 1.5`
ClassName@Method()	e.g., `wall@is_exterior`

Simple Comparisons

At its most fundamental level, a query is concerned with evaluating a condition to be either true or false. To enable us to make that decision we write a query which evaluates a certain condition through the use of the comparison operators =, \=, >, <, >=, =< (and also all the other Prolog comparison operators). For example:

```
wall@height = 2.4
space@area >= 30
```

Compound Terms

To enable complex compound queries the user may string together a set of simple comparisons through the use of both conjunctive (,) and disjunctive (;) operators teamed with brackets. For example:

```
wall@height > 2.4, wall@width > 4.5 ; wall@height > 7.5
wall@height > 2.4, (wall@width > 4.5 ; wall@width < 2.2)
```

Arithmetic Expressions

Comparisons need not be based purely on the existing value of an attribute, queries may also contain arbitrary mathematical expressions which will be evaluated to determine the outcome of the query. The allowable arithmetic operators are +, -, *, /, // (integer quotient), mod (remainder after integer division), ^ (to the power of), and several bit operators which can be found in the Prolog reference manual. For example:

```
space@area > space@height * 5
space@height > space@area / space@height * 2.4
```

Extended Arithmetic Expressions

Along with the standard arithmetic shown above the user can use arithmetic functions in an expression. The allowable arithmetic functions are abs (absolute value), acos, aln (base e), alog (base 10), asin, atan, cos, deg_rad (change from degrees to radians), exp (e to the power of), fp (the fractional part of a float), int (integer equal to or less than value), ip (integer part of a float), ln (base e), log (base 10), max (of two values), min (of two values), rad_deg (change from radians to degrees), rand (a random number in the given integer range), sign (-1 if negative, 0 if 0, 1 if positive), sin, sqr, sqrt, tan. For example:

```
sqrt(space@area) > ln(space@height)
sqr(int(sqrt(space@area))) > tan(deg_rad(space@height))
max(wall@height, wall@width) < 2.4
```

Prolog Predicates

Queries using standard arithmetic may be extended by using any other standard Prolog predicates or user defined predicates. These predicates may be of two types, those which either pass or fail in their execution, or those which return their result as the last parameter of the predicate. For

example:

```
integer(wall@height)
member(wall@wall_type, [interior, partition]), wall@height =< 2
distance_from_origin(wall@position) < 20
```

Reference Pointers

As many class definitions in an object-oriented schema are concerned with references to other objects, it is useful to be able to check values by following references in the object structure. We introduce the operator => to denote an attribute of a referenced object. For example:

```
wall@my_space=>area > 25
wall@my_space=>connect_space=>height > wall@my_space=>height
wall@my_space=>(height * area) > wall@height * sqr(wall@width)
wall@my_space=>(ln(height) * area) > wall@height * sqr(wall@width)
```

Aggregation Functions

In keeping with the relational query language support for aggregation it is also possible to include aggregation functions in an expression. While these operators are usually used to aggregate values in a table in a relational system, in an object-oriented system they are normally used to aggregate over a list of references found in an object. The allowable aggregation functions are sum, maximum, minimum, count and average. For example:

```
maximum(space@walls=>height) > 2.4
average(space@walls=>(height * width)) > 12
count(space@walls) >= 4
```

List or Array Indexing

As the type of attributes can be a list of values, or a list of lists, etc., it can be useful to have a method of accessing individual items in the list or if we treat lists as a n-dimensional array then any point in that array. To achieve this there is a list index operator ~ to specify which item to retrieve from the list, or ~[] if the user wishes to reference an item in a multi-dimensional array (eg toy_array~[4, 19, 28]). For example:

```
wall@position~[1] > 50
wall@my_space=>walls~[1] = wall@self
```

Multiple Class References in a Single Condition

Though all the examples thus far have shown individual conditions dealing with just one class type, it is quite possible to reference several class types in a condition. This is extremely useful in associating two objects which otherwise have no connection between them (or limiting the number of combinations of classes that get returned from a multi-class query). For example:

```
wall@height * wall@width = space@area
wall@my_space = space@connect_space
```

C.2.5 Implementation of the Query Language in Snart

As may be obvious from the examples shown above, the query system built on top of Snart takes full advantage of many of the features present in Prolog to manage the evaluation of a particular query. For example, the *and* and *or* conditions are handled directly using , and ; from Prolog. Also, the order of the query evaluation is determined through the precedence of the various operators in Prolog. At the lowest level of the evaluation all attribute references are replaced by their value and all arithmetic terms replaced with the result of their evaluation. Therefore at the top level of the evaluation a conditional term either fails or succeeds, thereby determining whether an object should be added to the list of objects which satisfy the query.

To improve the efficiency of a query evaluation there is some juggling of the queries separated by the conjunctive operator (,). Initially each part of the query is classified as to whether it references a single class, or whether it references multiple classes. When processing the query, the object ID's for all objects of the named classes are grouped by class. Then all conditions acting on a single class are evaluated on the objects belonging to that class. With the reduced lists of objects the cross-product of objects from all classes is built up. This is achieved by starting from the first class specified in the class list and creating a cross-product with the next class from the class list. This cross-product is reduced by applying all conditions which apply purely to these two classes. The result of this is used to create a cross-product with the third class from the class list, and all conditions which apply to these three classes are applied to reduce the number of tuples to consider, and so on, until we have covered all the classes in the class list, and all conditions have been applied.

C.3 Object Spaces

This sub-appendix discusses the method used in the Snart language (Grundy 1993) to allow multiple object spaces to exist in the Snart model space. Multiple object spaces provide a mechanism for having several distinct class name spaces working in the same environment. This allows schemas with the same class names to be used without conflict in the same environment. For example, as the name wall is common to almost all schemas dealing with buildings, it would be inconvenient and user unfriendly to be required to have separate class names for a wall in each schema. The modifications required to implement multiple object spaces on top of the existing Snart system are discussed, as are the various predicates provided for creating and manipulating various object spaces. The object space additions described here were added to the original Snart. The newest versions of Snart (Mugridge et al. 1995) incorporate a newer mechanism to handle object spaces, though the management of name spaces is not addressed. The use of the new Snart and its object spaces is described in Appendix C.4 in association with persistence and persistent spaces.

C.3.1 Introduction

The provision of multiple object spaces allows new levels of class management and manipulation to be achieved in the Snart object-oriented environment. They can also lead to better user interaction with the Snart system by allowing the user to name classes without having undue worry about name generation and naming conflicts. For example, in a multiple view scenario there may be several participants who have a class they wish to call a *wall*, even though the definition of these classes may all be quite distinct. With multiple object spaces each participant works in their own object space and may name classes as they wish. The name they define, however, is mapped to a distinct name at the Snart class management level.

There are several levels of management which are imposed by the addition of object spaces. At one level it is necessary to identify the object space that any defined class belongs to. With this addition it becomes possible to manipulate a defined set of classes in one operation. One can remove a subset of the system classes in one go, or identify which classes are in a particular object space, etc.

At another level it must be possible to distinguish all objects which were created in a particular object space. With this addition it is possible to manipulate a defined set of objects in one operation. One can remove a whole model in one operation, map a function over a whole set of objects in one space, merge two object spaces into one, etc.

C.3.2 Defining an object space in Snart

It is necessary to identify which object space a class definition belongs to when that class is first seen and again when it is modified. This is due to the term expansion which is used by Snart to translate a class definition into a form useable by the Snart system. At this point it is necessary to map the class name from that defined in the object space to a unique name inside the Snart object space. At the initialisation of the Snart system a default user space called *base* is defined. If there has been no other object space referenced before a class definition then the class will become a part of the *base* object space. To specify the object space that a particular set of classes belongs to it is necessary to use a predicate call of the form:

:- *object_store(+ObjectSpaceName).*

This must appear in each window containing class definitions before the actual class definitions. All classes parsed after this predicate call will belong to the named object space and it will become the default object space. At the parsing stage a register of classes which have been defined in an object space is constructed. The definition of object spaces is stored through property lists with the form defined below:

(sn_os_current_type, ?ObjectSpaceName)
Defines the current default object space

(sn_os_class_register, +ObjectSpaceName, ?Register)
Keeps track of all classes defined in this object space. The register contains a list of objects of the form:

ClassObjectSpaceName(ClassSnartName)

(sn_os_name, +ObjectSpaceName, ?OS_Tag)
Defines the atom which is prepended to the class name to give a unique class name in the Snart system.

(sn_os_inherit, +ObjectSpaceName, ?ObjectSpaceParent)
Defines a parent object space whose class definitions can be seen from the current object space.

To work with the object spaces there are a number of predicates which have been added to the Snart system. They are described below.

sn_os_create(+ObjectSpaceName)
Creates a new object space of the given name, or changes to the named object space if it already exists.

sn_os_change, sn_os_change(+ObjectSpaceName)
Changes the default object space to the one named or creates it if it doesn't already exist.

sn_os_delete(+ObjectSpaceName)
Deletes an object space from the Snart system. This removes all class definitions and all objects in an object space of this name.

sn_os_find_name(+ObjectSpaceClass, -SnartClassName, -ObjectSpaceName)
Finds the mapped name of a class in an object space. This search proceeds from the current default object space and if such a class is not found there then looks up the object space hierarchy, if there is one defined for this object space. If the class is still not found then it searches all other object spaces in no particular order until it finds a matching class name.

sn_os_find_name_rev(+SnartClassClass, -ObjectSpaceClass)
The reverse for the predicate above. Given a class name in the Snart class space, it will find the corresponding name in its object space.

sn_os_find_all_os(+Class, -ObjectSpaceNameList)
Returns a list of all object spaces which contain a class definition matching Class.

It is also possible to define a default method for a whole object store. If there is a definition of the form:

> *object_store(ObjectStoreName)::MethodName:- Body*
> *object_store(ObjectStoreName)::MethodName(Arguments) :- Body*

with the class definitions for an object store then this method will be applied to every object in the named object store when *MethodName* is called for an object. Uses for this form of definition are a create method which can specify special conditions for the creation of any object in a given object-store, and similarly for a delete method.

C.3.3 Working with an object space model in Snart

Currently the creation of objects works in a similar method to classes in an object space in the sense that any object created must be created in an object space model of the same type as the class from which it is being created. Extra information required to manage models of an object space is kept in property lists of the following form. This extends to the persistency system in Snart (Amor 1993b) which must also keep track of which model a particular object was created in.

ObjectSpace&Class@create([Parameters,]ObjectID)
Allows an object to be created in the named object space and for the specified class.

(sn_os_current_model, ?ObjectSpaceNumber)
Defines the current object space model being worked with.

(sn_os_model, +ObjectSpaceNumber, ?ObjectSpaceName)
Defines the type of object space that a particular model was created for.

(sn_os_object_register, +ObjectSpaceNumber, ?Register)
Holds the list of objects which were created in the object space model. The register will hold information about all objects and is in the form:

> [1,'p(12)','p(15)',6,27,'p(2)']

(sn_os_def_model, +ObjectSpaceName, ?ObjectSpaceNumber)
Defines the model which was last used when working with an object space of a given type. This is used to ascertain which object space an object should be inserted into.

To work with objects in the object space model there are a set of predicates which have been added to the Snart system. They are described below.

sn_create_os(+ObjectSpaceName, -ObjectSpaceModelNumber)
Creates a new object space model of the given type and returns the unique number associated with that model.

sn_os_model_change(+ObjectSpaceModelNumber)
Allows the user to change the default object space model to the named one.

sn_find_os_name(+ObjectID, -ObjectSpaceName)
Finds the object space type that a given object was created in.

sn_find_os_number(+ObjectID, -ObjectSpaceModelNumber)
Finds the object space number that a given object was created in.

sn_os_delete_model(+Model)
Deletes a model space and all the objects defined in that model. It then updates all properties to reference another valid model. If the base model is being cleared it is left as an empty model space.

sn_os_merge(+FromModel, +ToModel)
Merges two object space models, and then deletes all references to the *FromModel*.

sn_os_map(+ObjectSpaceModelNumber, +Call)
Sends a method call defined by Call to all objects in a given object space model.

C.4 Persistency

This sub-appendix discusses two methods used in different versions of the Snart language to allow objects to be persistent. Both methods are introduced as some modules in this thesis are written in the old Snart (e.g., EPE and Cerno-II extensions). The mechanisms involved in declaring and checking persistent classes are discussed, as are the methods used to load and save data to a persistent store.

C.4.1 Introduction

Persistency is required in almost all programs which manage and manipulate quantities of data. In this thesis almost every component has a requirement to manage and manipulate large amounts of data, whether they be the schema models, mappings, process specifications, or even the data in models being used by the various design tools. This requirement led to a very early decision to extend the capabilities of Snart to include persistency in some form. The two resulting systems are described further below.

There has recently been a discussion on the Internet in the *comp.databases.object* newsgroup about persistency and its implementation in object-oriented languages. The conclusions of this discussion are definitions of the various models of persistency which are presented below.

OO-DBMS theory says there are 4 models of persistence. Either classes are persistent or individual objects of classes are persistent. Persistent objects may be stored implicitly without any explicit store operation, or they may be stored explicitly. In the net discussion one further model of persistence was suggested which is based on a notion of reachability. This yields the following models:

Implicit class level persistence
Your class declaration says that this class is persistent. Any object belonging to this class is automatically stored in the database.

Implicit object level persistence
Individual objects are declared to be persistent when they are allocated. These persistent objects are automatically stored in the database.

Explicit class level persistence
Your class declaration says that this class is persistent. Any object belonging to this class can be stored in the database with a store operation.

Explicit object level persistence
Individual objects are declared to be persistent when they are allocated. These persistent objects can be stored in the database with a store operation.

Persistency by reachability
Any object reachable from a persistent root(s) at commit time becomes persistent. The programmer never says anything about persistence; persistence is implicit in the structure of the object graph. Persistence by reachability was the model used in the Argus distributed persistent programming language (Atkinson et al. 1983), and was retained in the design for the Thor distributed object-oriented database. Persistence by reachability (based on garbage collection) was also the model used by PS-Algol (Atkinson et al. 1983) and Napier (Morrison et al. 1988). PS-Algol is not, however, based on an object-oriented data model.

C.4.2 Persistency in the old Snart

The model of persistency used in previous implementations of the Snart programming language (Grundy 1993) was that of explicit class level persistence. In this form a persistent class was described through a variation of the normal class definition syntax of Snart. This previous general form of a persistent class definition in Snart, as opposed to the normal class definition, is detailed

below:

persistent_class(ClassName, ClassVersion, ClassParents, ClassFeatures).
class(ClassName, ClassParents, ClassFeatures).

The current version of Snart has been changed to explicit object level persistence, where persistent stores are created and only objects explicitly created in this store are saved when the store is instructed to save itself. With potentially multiple types of object spaces available, a persistent object space keeps track of the object ID's in its store. This enables persistent models to be kept distinct from other models in the system.

C.4.3 Manipulating persistent objects in the old Snart

Once created, persistent objects in Snart are accessed in exactly the same way as any other type of Snart object. Where a persistent object is referenced and it hasn't been loaded from the object store then the object will be loaded automatically. Any change to an feature of a persistent object marks that object as modified and a candidate for saving back to the persistent object store upon the store closure or a forced update. The only visible difference between the persistent object and a normal object is the index number which is generated for the object. A normal object has integer values generated during object creation while a persistent object has an index value of the form *p(IndexNumber)* indicating to the Snart system that the object is persistent.

The management and access methods for a persistent store are fairly rudimentary, stores can be created, opened or closed, and the objects in the store can be manually loaded or saved. The major access methods are described below:

sn_create_object_store(File, Path) or *sn_create_object_store*
Creates and initialises a new object store in the specified location.

sn_open_object_store(File, Path) or *sn_open_object_store*
Opens a previously saved object store and sets the new object index above the maximum index used in the object store. Upon opening the object spaces existing in the store are loaded and new object spaces are defined in the Snart system for each of the object spaces in the persistent store.

sn_close_object_store(File) or *sn_close_object_store*
Closes an object store after saving all modified objects. A register of all object spaces which contain persistent objects is also saved to the object store at this point.

sn_load_object_store
Loads all objects from the object store, unless they have already been previously loaded.

sn_load_object_store(ClassName)
Loads all objects of the type *ClassName* from the object store, unless they have already been previously loaded.

sn_write_objects
Saves all modified objects back to the object store. A register of all object spaces which contain persistent objects is also saved to the object store at this point.

sn_merge_object_store(File, Path) or *sn_merge_object_store*
Merges all objects from the named persistent store into the current persistent store. Duplicate or clashing object ID's are mapped to a new ID so they can be saved in the current store.

C.4.4 Persistency in the new Snart

The model of persistency used in the newer implementations of the Snart programming language (Mugridge et al. 1995) is that of explicit object level persistence. In this form an object is created in a named space. If this space happens to be a persistent space then when the space is instructed to save itself it will save all objects created inside itself. The basic structure of object spaces and the persistent space is detailed in the Snart definitions below:

```
class(object_space,
      metas(
            object_spaces:list(object_space), % All but default, called 'o'
            create(+id, args),
            remove_object_space(+objectSpace),
            all_object_spaces(-list(object)),
            load_object(+object),
            trace_objects
      ),
      features(
            locate_object(+object),
            load_object(+object),
            dispose,
            new_object(+object),
            deleted_object(+object),
            all_objects(-list(object))
      )
).

class(small_persistent_space,
      inherits(object_space),
      features(
            file : atom,
            kind : atom,
            create(+atom),
            create(+atom, +atom),
            open,
            close,
            save,
            delete
      )
).
```

This small persistent space is implemented by an ASCII file containing all objects created in the object space. This is a very simple implementation suitable for small amounts of data. The complete object space is written to the file every time it is saved, and loaded in its entirety when the space is opened. The only optimising feature implemented (by John Grundy) was the addition of a line size at the start of each line to allow the remainder of the line to be read in one go, rather than byte by byte. The basic structure of a persistent file is shown in the file below (note: this is formatted to show what would appear on one line as an indented piece of text):

```
traced_persistent_space.
96.
data_store_1.
239. obj(data_store_1, [('#traced', false), (file, 'Trebor:Desktop Folder:VML Test:idm.db'),
     (kind, traced_persistent_space), ('#class', ms_traced_persistent_space), (tracer,
     data_store_1_1), ('#original_class', ms_traced_persistent_space)]).
180. obj('data_store_1(ms_db_controller', [('#original_class', meta_ms_db_controller),
     ('#facets', []), ('#traced', false), ('#triggered', false), ('#class',
     meta_ms_db_controller)]).
168. obj('data_store_1(idm_3d_point', [('#original_class', meta_idm_3d_point), ('#facets',
     []), ('#traced', false), ('#triggered', false), ('#class', meta_idm_3d_point)]).
159. obj('data_store_1(idm_space', [('#class', meta_idm_space), ('#original_class',
     meta_idm_space), ('#facets', []), ('#traced', false), ('#triggered', false)]).
```

The major functionality available for the persistent object space is described below:

sn_init_persistent_space(+Name, -Space, -File)
Creates a persistent space with the specified *Name*. The user is prompted to specify a filename for the space and this file is used to save all objects into. The returned *Space* atom is the name of the space which was created. This atom is required to name the space when an object is created as with persistent spaces in the old versions of Snart.

small_persistent_space::create(+File [, +Kind])
Creates a persistent space with a specified filename and if requested with a specified type. The type enables specialised forms of persistent spaces to be recognised. For example, in the example shown above the space type is a *traced_persistent_space*.

small_persistent_space::open
Opens the space and loads all objects in the file into main memory. If the name of the space in the file being loaded from is different from the one it is being loaded into all object references in the file are renamed to suit the new space name.

small_persistent_space::save
Determines all objects which exist in the space and then writes them all to file. In the process of doing this the old file is overwritten and all objects rewritten.

small_persistent_space::close
Closes a persistent space, without saving, and removes all the object definitions from the main working memory.

small_persistent_space::delete
Performs a close as defined above and then physically removes the file from the file system.

C.5 ObjectViewer

While LPA Prolog provides a debugging system, and Snart objects all have a print method associated with them, this does not provide the ideal interface for interactive development and testing of Snart programs. What is often required is a dynamic view of the changing state of specific objects as a program runs. The ObjectViewer provides a general purpose debugging tool for the Snart language as well as a data model viewing and navigation tool. The basic premise of the ObjectViewer is that it spies upon an object, and always reflects the current state of the object. It is more powerful than that, as it also allows attributes to be modified, and methods of the class to be invoked, from the object view window.

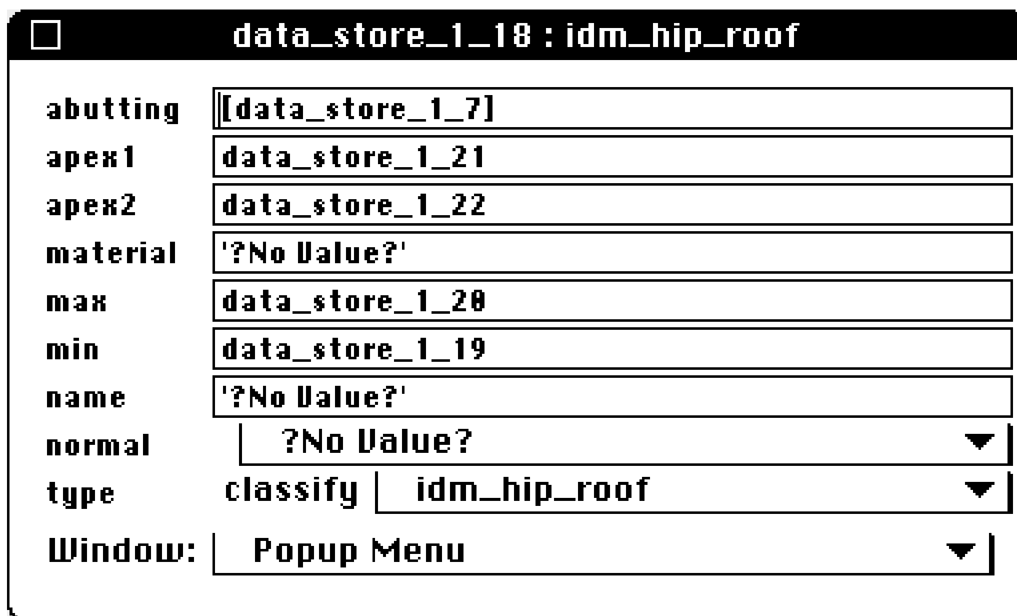


Figure C.2 ObjectViewer interface

The ObjectViewer is built into Snart and can be invoked either through an extended menu item in the LPA Prolog environment, or through a hot-key. Whichever way it is invoked the ObjectViewer collects the highlighted text in the current window and parses this to extract all object identifiers.

For each identifier found in the highlighted text it will create an ObjectViewer window for them. To be informed of the state of the named object, the ObjectViewer places a trace on the object and is then informed of all modifications made to the object through the Snart system (e.g., attribute assignment, re-classification, deletion). For each class of object the ObjectViewer spies upon, it creates a new class definition which provides the mechanism to display an object and manage messages passed back from the viewing window. The window created for an object allows values for attributes to be modified, and then set for the underlying object. Attribute values in a window change at the same time that they are set in the underlying object.

By default the ObjectViewer uses the set of attributes of a class to determine the layout of a window representing that object. The resulting layout is as seen in Figure C.2.

This default layout displays all attributes in alphabetic order, followed by a set of functions in the Window Popup Menu line at the bottom. This default layout can be modified by specifying a layout template for objects of a given class. This layout comprises a list of four types of component which will be rendered in the display window in the order they are named. This provides a way of ordering and reducing the attributes which are displayed, of changing the style of layout of the window specific functions, and of allowing method calls of the object to be invoked. The four components are described below:

Attribute: names an attribute of the class to be displayed in the window. Attributes are displayed according to their type. As shown in the diagram above, classifiers are shown as a popup item which allows the object to be reclassified. Attributes of enumerated types will also be shown as a popup where any of the enumeration can be selected.

Button: defines a button to be displayed in the window. Buttons declared in sequence are strung side-by-side, up to three across a window. Button specifications take the form *ButtonID(ButtonLabel)*, where *ButtonID* is the name of the method invoked when the button is pressed. *ButtonLabel* provides the name that will be seen in the button.

Popup: defines a popup menu which can contain several items. When the popup menu is accessed, the item selected when the mouse button is released is the method which is invoked. Popup specifications take the form *PopupID(PopupLabel, [ItemID1(ItemLabel1), ItemID2(ItemLabel2), ...])*, where *PopupLabel* is the name given to the popup in the window (e.g., *Window* in the figure above) and *ItemID* is the name of the method invoked when the popup item is selected. *ItemLabel* provides the name that will be seen in the popup. A special form of popup item is allowed called *nop(+Char)*. This allows a separating item to be placed in a list which performs no operation. *Char* is used as the character used to create the separating line and is drawn across the maximum width of the popup.

Window Functions: this is a keyword of either '$popup_system$' or '$button_system$' which defines where the window system functions are placed and in what form they appear. Window functions can be placed as a popup, as shown in the diagram above, or as a set of

buttons in the window. If the window function keyword is not included in the layout specification then it will be added automatically at the end as a popup.

The set of window functions which are supplied for every object are as follows:

update('Apply Modifications')
Modifies the object to reflect the values which have been set in the ObjectViewer window. If this includes a reclassification then the object type is reclassified.

view_all('View All Objects')
Runs the ObjectViewer over everything in the attribute that the cursor resides in. This is the same as calling ObjectViewer with highlighted text, except in this case the text does not need to be highlighted.

view_selected('View Selected')
Runs the ObjectViewer over everything highlighted in the attribute that the cursor resides in. This is the same as calling ObjectViewer with highlighted text.

print_object('Print Object')
Invokes the print method of the underlying object.

dispose('Discard View')
Removes the window and tracers on the object from the system. This is in contrast to clicking the go-away box in the window title-bar which merely hides the window.

redraw('Re-Draw')
Redraws the window to take account of the newly specified window size or font specifications.

restructure('Re-Structure')
Totally redraws the window to take into account a new layout specification for the class the object represents.

There are also a set of global parameters which can be set for the ObjectViewer. These are described below:

sn_ov_set_width
sn_ov_set_width(+Width)
Sets the maximum width of the window. This is then used to calculate the size required for attribute names and the remainder is used for displaying the attribute values. Without a parameter *sn_ov_set_width* sets the value back to its default. This call only affects windows which are

created after the call is made, not existing windows. There are constraints upon the maximum and minimum window width based on screen size and the font specification described below.

sn_ov_set_font
sn_ov_set_font(+Font)
sn_ov_set_font(+Font, +Size)
sn_ov_set_font(+Font, +Style, +Size)
Sets the display font, style and size. By default this is Courier 10 point. This call only affects windows which are created after the call is made, not existing windows.

Appendix D

Small Examples Models and Mappings

These twenty small examples are taken from examples in papers describing other mapping languages (Clark 1992, Bailey 1994, Hardwick 1994) as well as the author's own work. While there is some considerable overlap between some of the following examples, the author feels it is useful to provide VML definitions of each example for comparison purposes. The layout of the following examples is a definition of the schemas used for the IDM model in the left column and the view schema in the right column. The required mapping definition for the two schemas is given below the schema definitions.

1) Example from Clark (1992).

This example shows the mapping between a single entity in one schema and multiple classified entities in the second schema. The invariant gender = male ensures that only entities describing a male in the idm are mapped to the male entity in view1. The entity reference of male for view1 in the inter_class definition ensures that only male entities get mapped to person entities in the idm schema with gender being set to male.

Note: In this example the entity person in view1 can not be mapped to person in the idm as the gender attribute of person in the idm is not optional.

```
1)
SCHEMA idm;

TYPE sex_type = ENUMERATION OF
      (male, female);
END_TYPE;

ENTITY person;
      name : STRING;
      age : INTEGER;
      gender : sex_type;
      inity : INTEGER;
END_ENTITY;

END_SCHEMA;
```

```
2)
SCHEMA view1;

ENTITY person
      SUPERTYPE OF (ONEOF (male,
female));
      name : STRING;
      age : INTEGER;
END_ENTITY;

ENTITY male
      SUBTYPE OF (person);
      masculinity : INTEGER;
END_ENTITY;

ENTITY female
      SUBTYPE OF (person);
      femininity : INTEGER;
END_ENTITY;

END_SCHEMA;
```

Mapping

```
inter_view(idm, integrated, view1, read_write, complete).

inter_class([person],[male],
      invariants(gender = 'male'),
      equivalences(name = name,
                  age = age,
                  inity = masculinity)
      ).

inter_class([person],[female],
      invariants(gender = 'female'),
      equivalences(name = name,
                  age = age,
                  inity = femininity)
      ).
```

2) Example from Clark (1992).

This example shows the mapping between schemas whose inheritance hierarchies have a different distribution of attributes. This mapping shows how mappings may be defined for parent entities and seen by children entities when they need to perform their mappings.

Note: the attribute size for entity person in schema view1 will be required when mapping from the idm to view1. However, this does not force a reclassification as would be the case in the previous example, so a mapping between person and person is possible in this case.

```
1)
SCHEMA idm;

ENTITY person
      SUPERTYPE OF (ONEOF (man,
woman));
      name : STRING;
      age : INTEGER;
END_ENTITY;

ENTITY man
      SUBTYPE OF (person);
      masculinity : INTEGER;
      size : INTEGER;
END_ENTITY;

ENTITY woman
      SUBTYPE OF (person);
      femininity : INTEGER;
      size : INTEGER;
END_ENTITY;

END_SCHEMA;
```

```
2)
SCHEMA view1;

ENTITY person
      SUPERTYPE OF (ONEOF (man,
woman));
      name : STRING;
      age : INTEGER;
      size : INTEGER;
END_ENTITY;

ENTITY man
      SUBTYPE OF (person);
      masculinity : INTEGER;
END_ENTITY;

ENTITY woman
      SUBTYPE OF (person);
      femininity : INTEGER;
END_ENTITY;

END_SCHEMA;
```

<u>Mapping</u>

```
inter_view(idm, integrated, view1, read_write, complete).

inter_class([person],[person],
      equivalences(name = name,
                   age = age)
      ).

inter_class([man],[man],
      inherits(inter_class([person],[person])),
      equivalences(size = size,
                   masculinity = masculinity)
      ).

inter_class([woman],[woman],
      inherits(inter_class([person],[person])),
      equivalences(size = size,
                   femininity = femininity)
      ).
```

3) Example from Clark (1992).

This example is an extension of the previous example which shows a derived entity in view1. In Clark (1992) there is an explicit mapping to generate this derived entity. We believe that this is an erroneous schema and that defining a mapping in this case is invalid. This is a derived entity and as such should be described so in the schema definition, and implemented as one in the database system (i.e., a database view).

```
1)
SCHEMA idm;

ENTITY person
      SUPERTYPE OF (ONEOF (man,
woman));
      name : STRING;
      age : INTEGER;
END_ENTITY;

ENTITY man
      SUBTYPE OF (person);
      masculinity : INTEGER;
      size : INTEGER;
END_ENTITY;

ENTITY woman
      SUBTYPE OF (person);
      femininity : INTEGER;
      size : INTEGER;
END_ENTITY;

END_SCHEMA;
```

```
2)
SCHEMA view1;

ENTITY person
      SUPERTYPE OF (ONEOF (man,
woman));
      name : STRING;
      age : INTEGER;
      size : INTEGER;
END_ENTITY;

ENTITY man
      SUBTYPE OF (person);
      masculinity : INTEGER;
END_ENTITY;

ENTITY woman
      SUBTYPE OF (person);
      femininity : INTEGER;
END_ENTITY;

ENTITY couple;
      him : man;
      her : woman;
      compatibility : INTEGER;
END_ENTITY;

END_SCHEMA;
```

<u>Mapping</u>

```
inter_view(idm, integrated, view1, read_write, complete).

inter_class([person],[person],
      equivalences(name = name,
                   age = age)
      ).

inter_class(inherits([man],[man],
      inter_class([person],[person])),
      equivalences(size = size,
                   masculinity = masculinity)
      ).

inter_class(inherits([woman],[woman],
      inter_class([person],[person])),
      equivalences(size = size,
                   femininity = femininity)
      ).
```

4) Example from Clark (1992).

This example shows the mapping definition between two conceptually different representations of a point on a plane. The idm represents a point using an angle and radius, view1 by absolute geometrical coordinates.

```
1)
SCHEMA idm;

ENTITY point;
      r : REAL;
      theta : REAL;
END_ENTITY;

END_SCHEMA;
```

```
2)
SCHEMA view1;

ENTITY point;
      x_coord : REAL;
      y_coord : REAL;
END_ENTITY;

END_SCHEMA;
```

```
Mapping
inter_view(idm, integrated, view1, read_write, complete).

inter_class([point],[point],
      equivalences(r * cos(theta) = x_coord,
                  r * sin(theta) = y_coord,
                  r = sqrt(sqr(x_coord) + sqr(y_coord)),
                  theta = tan_1(y_coord / x_coord))
      ).
```

5) Example from Bailey (1994).

This example shows a simple mapping between two structurally similar representations of a circle. Note: If comparing to Bailey (1994) notice that we define the mapping between references to other entities explicitly (e.g., circle.centre and circ.centre). Bailey (1994) assumes that this can be handled automatically.

```
1)
SCHEMA idm;

ENTITY circle;
      centre : point;
      radius : REAL;
END_ENTITY;

ENTITY point;
      x, y, z : REAL;
END_ENTITY;

END_SCHEMA;
```

```
2)
SCHEMA view1;

ENTITY circ;
      centre : point;
      diameter : REAL;
END_ENTITY;

ENTITY point;
      x_coord, y_coord, z_coord :
REAL;
END_ENTITY;

END_SCHEMA;
```

Mapping

```
inter_view(idm, integrated, view1, read_write, complete).

inter_class([circle],[circ],
      equivalences(radius * 2 = diameter,
                   centre = centre)
      ).

inter_class([point],[point],
      equivalences(x = x_coord,
                   y = y_coord,
                   z = z_coord)
      ).
```

6) Example from Bailey (1994).

This example shows how the mapping between two structurally dissimilar representations of a circle can be described. The invariant provides the requisite information to create a point for each circle when mapping from the idm to view1 and gives the selection strategy when creating idm circles from view1 objects.

```
1)
SCHEMA idm;

ENTITY circle;
      radius : REAL;
      centre_x, centre_y, centre_z
: REAL;
END_ENTITY;

END_SCHEMA;
```

```
2)
SCHEMA view1;

ENTITY circ;
      centre : point;
      diameter : REAL;
END_ENTITY;

ENTITY point;
      x_coord, y_coord, z_coord :
REAL;
END_ENTITY;

END_SCHEMA;
```

Mapping

```
inter_view(idm, integrated, view1, read_write, complete).

inter_class([circle],[circ],
      equivalences(radius * 2 = diameter,
                  centre_x = centre=>x_coord,
                  centre_y = centre=>y_coord,
                  centre_z = centre=>z_coord)
      ).
```

7) Example from Bailey (1994).

This example is structurally identical to the previous example and is included for comparison to Bailey (1994).

1)

```
SCHEMA idm;

ENTITY location;
      point : cartesian_point;
END_ENTITY;

ENTITY cartesian_point;
      x, y, z : REAL;
END_ENTITY;

END_SCHEMA;
```

2)

```
SCHEMA view1;

ENTITY location;
      p, q, r : REAL;
END_ENTITY;

END_SCHEMA;
```

Mapping

```
inter_view(idm, integrated, view1, read_write, complete).

inter_class([location],[location],
      equivalences(point=>x = p,
                   point=>y = q,
                   point=>z = r)
      ).
```

8) Example from Bailey (1994).

This example shows the mappings required for a classification problem.

```
1)
SCHEMA idm;

ENTITY circle;
      centre : cartesian_point;
      diameter : OPTIONAL REAL;
      radius : OPTIONAL REAL;
      p1, p2, p3 : OPTIONAL
cartesian_point;
END_ENTITY;

ENTITY cartesian_point;
      x, y, z : REAL;
END_ENTITY;

END_SCHEMA;
```

```
2)
SCHEMA view1;

ENTITY point;
      x, y, z : REAL;
END_ENTITY;

ENTITY radius_circle;
      centre : point;
      radius : REAL;
END_ENTITY;

ENTITY diam_circle;
      centre : point;
      diameter : REAL;
END_ENTITY;

ENTITY three_p_circle;
      p_1, p_2, p_3 : point;
END_ENTITY;

END_SCHEMA;
```

```
Mapping
inter_view(idm, integrated, view1, read_write, complete).

inter_class([cartesian_point],[point],
      equivalences(x = x,
                   y = y,
                   z = z)
      ).

inter_class([circle],[radius_circle],
      invariants(exists(circle.radius)),
      equivalences(centre = centre,
                   radius = radius)
      ).

inter_class([circle],[diam_circle],
      invariants(exists(circle.diameter)),
      equivalences(centre = centre,
                   diameter = diameter)
      ).

inter_class([circle],[three_p_circle],
      invariants(exists(p1),exists(p2),exists(p3)),
      equivalences(p1 = p_1,
                   p2 = p_2,
                   p3 = p_3)
      ).
```

9) Example from Bailey (1994).

This example shows the mapping for two structurally different representations of an arc.

```
1)
SCHEMA idm;

ENTITY three_point_arc;
      pnt1, pnt2, pnt3: point;
END_ENTITY;

ENTITY point;
      x, y : REAL;
END_ENTITY;

END_SCHEMA;
```

```
2)
SCHEMA view1;

ENTITY angle_arc;
      centre : point;
      radius : REAL;
      theta : REAL;
      phi : REAL;
END_ENTITY;

ENTITY point;
      x, y : REAL;
END_ENTITY;

END_SCHEMA;
```

Mapping

```
inter_view(idm, integrated, view1, read_write, complete).

inter_class([three_point_arc],[angle_arc],
      equivalences(pnt2=>x = centre=>x,
                   pnt2=>y = centre=>y,
                   pnt1=>x = centre=>x - radius * cos(theta),
                   pnt1=>y = centre=>y + radius * sin(theta),
                   pnt3=>x = centre=>x - radius * cos(phi),
                   pnt3=>y = centre=>y - radius * sin(phi))
      ).
```

10) Example from Amor (1994).

This example shows the merging of two entities in view1 to one entity in the idm, or vice versa. The creation of trombe_type entities is regulated by the unique clause on name which guarantees that only one of any trombe_type exists in the system.

Note: The matching of trombe_wall and trombe_type is done by equivalence of the name field to the trombe_type.

```
1)
SCHEMA idm;

ENTITY trombe_wall;
      height, width : REAL;
      glazing_area : REAL;
      vent_area : REAL;
      trombe_type : STRING;
      perf_ratio : REAL;
END_ENTITY;

END_SCHEMA;
```

```
2)
SCHEMA view1;

ENTITY trombe_wall;
      height, width : REAL;
      glazing_area : REAL;
      vent_area : REAL;
      trombe_type : STRING;
END_ENTITY;

ENTITY trombe_type;
      name : STRING;
      perf_ratio : REAL;
      UNIQUE name;
END_ENTITY;

END_SCHEMA;
```

Mapping

```
inter_view(idm, integrated, view1, read_write, complete).

inter_class([trombe_wall],[trombe_wall, trombe_type],
      invariants(trombe_wall.trombe_type = trombe_type.name),
      equivalences(height = height,
                  width = width,
                  glazing_area = glazing_area,
                  vent_area = vent_area,
                  trombe_type = trombe_type,
                  perf_ratio = perf_ratio)
      ).
```

11) Example from Amor (1994).

This example is the same as the previous example except the link between trombe_wall and trombe_type is made through a pointer to an object of trombe_type.

```
1)
SCHEMA idm;

ENTITY trombe_wall;
      height, width : REAL;
      glazing_area : REAL;
      vent_area : REAL;
      trombe_type : STRING;
      perf_ratio : REAL;
END_ENTITY;

END_SCHEMA;
```

```
2)
SCHEMA view1;

ENTITY trombe_wall;
      height, width : REAL;
      glazing_area : REAL;
      vent_area : REAL;
      trombe_type : trombe_type;
END_ENTITY;

ENTITY trombe_type;
      name : STRING;
      perf_ratio : REAL;
      UNIQUE name;
END_ENTITY;

END_SCHEMA;
```

<u>Mapping</u>

```
inter_view(idm, integrated, view1, read_write, complete).

inter_class([trombe_wall],[trombe_wall],
      equivalences(height = height,
                  width = width,
                  glazing_area = glazing_area,
                  vent_area = vent_area,
                  trombe_type = trombe_type=>name,
                  perf_ratio = trombe_type=>perf_ratio)
      ).
```

12) Example from Amor (1994).

This example shows the merging of several entities based on a key into one entity in the idm.

```
1)
SCHEMA idm;

ENTITY wall;
      results : perf_result;
END_ENTITY;

ENTITY perf_result;
      a1, a2, a3 : REAL;
      b1, b2, b3 : INTEGER;
      c1, c2, c3 : BOOLEAN;
END_ENTITY;

END_SCHEMA;
```

```
2)
SCHEMA view1;

ENTITY wall;
      id : INTEGER;
      UNIQUE id;
END_ENTITY;

ENTITY wperf1;
      wall_ID : INTEGER;
      a1, a2, a3 : REAL;
      UNIQUE wall_ID ;
END_ENTITY;

ENTITY wperf2;
      wall_ID : INTEGER;
      b1, b2, b3 : INTEGER;
      UNIQUE wall_ID ;
END_ENTITY;

ENTITY wperf3;
      wall_ID : INTEGER;
      c1, c2, c3 : BOOLEAN;
      UNIQUE wall_ID ;
END_ENTITY;

END_SCHEMA;
```

Mapping

```
inter_view(idm, integrated, view1, read_write, complete).

inter_class([wall],[wall, wperf1, wperf2, wperf3],
      invariants(view1:wall.id = wperf1.wall_ID,
                 view1:wall.id = wperf2.wall_ID,
                 view1:wall.id = wperf3.wall_ID),
      equivalences(results=>a1 = a1,
                 results=>a2 = a2,
                 results=>a3 = a3,
                 results=>b1 = b1,
                 results=>b2 = b2,
                 results=>b3 = b3,
                 results=>c1 = c1,
                 results=>c2 = c2,
                 results=>c3 = c3)
      ).
```

13) Example from Amor (1994).

This example shows the definition of an equivalence which can not be mapped automatically in two directions. Trying to create the idm wall from a view1 wall will result in a constraint being imposed on height and width without it being possible to calculate their exact values.

```
1)
SCHEMA idm;

ENTITY wall;
      construction : LIST OF
material;
      height : REAL;
      width : REAL;
END_ENTITY;

END_SCHEMA;
```

```
2)
SCHEMA view1;

ENTITY wall;
      construction : LIST OF
material;
      area : REAL;
END_ENTITY;

END_SCHEMA;
```

Mapping

```
inter_view(idm, integrated, view1, read_write, complete).

inter_class([wall],[wall],
      equivalences(width * height = area,
                  construction = construction)
      ).
```

14) Example from Amor (1994).

This example shows the mapping required for a view1 entity which contains summary information rather than explicit representations of component entities. It is obvious in this example that there is no way to generate idm windows directly from the view1 wall entity. However, mapping from view1 to the idm will impose several constraints on the properties of the windows list.

```
1)
SCHEMA idm;

ENTITY wall;
      materials : LIST OF material;
      windows : LIST OF window;
      height : REAL;
      width : REAL;
END_ENTITY;

ENTITY window;
      offset : position;
      height : REAL;
      width : REAL;
      materials : LIST OF
window_mat;
END_ENTITY;

END_SCHEMA;
```

```
2)
SCHEMA view1;

ENTITY wall;
      materials : LIST OF material;
      wall_area : REAL;
      glazing_area : REAL;
END_ENTITY;

END_SCHEMA;
```

```
Mapping
inter_view(idm, integrated, view1, read_write, complete).

inter_class([wall],[wall],
      equivalences(wall.materials = materials,
            sum(wall.windows=>(height * width)) = glazing_area,
            wall.height * wall.width = wall_area + glazing_area)
      ).
```

15) Example from Amor (1994).

This example is similar to the previous example except that there is more information which can be gleaned from the idm objects.

```
1)
SCHEMA idm;

ENTITY window;
      panes : LIST OF pane;
      offset : position;
      framing : frame;
END_ENTITY;

ENTITY pane;
      offset : position;
      height : REAL;
      width : REAL;
      material : LIST OF
glazing_mat;
END_ENTITY;

END_SCHEMA;
```

```
2)
SCHEMA view1;

ENTITY window;
      offset : position;
      height : REAL;
      width : REAL;
      material : LIST OF
glazing_mat;
END_ENTITY;

END_SCHEMA;
```

Mapping

```
inter_view(idm, integrated, view1, read_write, complete).

inter_class([window],[window],
      equivalences(offset = offset,
            panes[1]=>material = material,
            maximum(panes=>(offset=>y + height))- minimum(panes=>offset=>y) =
height,
            maximum(panes=>(offset=>x + width))- minimum(panes=>offset=>x) =
width)
      ).
```

16) Example from Amor (1994).

This example shows the relative complexity required to manage the mapping between a general schedule representation in the idm, and a more specific representation in view1 (where schedule information is grouped into bunches of three in each entity). This example comes from representations found in SUNCODE and DOE-2. In this example it is useful to introduce a temporary entity which captures a list of lists comprising up to three *time_val* entities as a half way point between the two schema entities. This temporary entity has an implied schema as follows:

```
ENTITY _temp_schedule;
	name : STRING;
	splitvals : LIST OF LIST [1:3] OF time_val;
END_ENTITY;
```

Also note the use of a variable parameter *I* in the inter_class definition, this provides the mechanism for determining how to merge the multiple *schedule* objects into a single *_temp_schedule* object with the list being constructed in the correct order.

```
1)
SCHEMA idm;

ENTITY schedule;
      name : STRING;
      vals : LIST OF time_val;
      UNIQUE name;
END_ENTITY;

ENTITY time_val;
      time : INTEGER;
      val : REAL;
END_ENTITY;

END_SCHEMA;
```

```
2)
SCHEMA view1;

ENTITY schedule;
      name : STRING;
      position : INTEGER;
      sched1 : time_val;
      sched2 : OPTIONAL time_val;
      sched3 : OPTIONAL time_val;
      UNIQUE name, position;
END_ENTITY;

ENTITY time_val;
      time : INTEGER;
      val : REAL;
END_ENTITY;

END_SCHEMA;
```

Mapping

```
inter_view(idm, integrated, view1, read_write, complete).

inter_class([time_val],[time_val],
      equivalences(time = time,
                  val = val)
      ).

inter_class([schedule],[_temp_schedule],
      equivalences(name = _temp_schedule.name,
                  list_splitter(vals, _temp_schedule.splitvals))
      ).
```

```
inter_class([_temp_schedule],[schedule],
      invariants(group(schedule.name)),
      equivalences(name = name ,
                  splitvals[schedule.position, 1] = sched1,
                  splitvals[schedule.position, 2] = sched2,
                  splitvals[schedule.position, 3] = sched3)
      ).

list_splitter([], []).
list_splitter([A], [[A]]).
list_splitter([A, B], [[A, B]]).
list_splitter([A, B, C|Rest], [[A, B, C]|SplitRest) :-
      list_splitter(Rest, SplitRest).
```

17) Example from Amor (1994).

This example shows the mapping required to provide representations of a wall as a positioned rectangle in the idm and as a stretched and rotated column in view1.

Note: The definition of the mapping for lists of equivalent types can be performed without recourse to individual elements in the list. However, in the next example we have a case where accessing the individual elements of a list is necessary for the specification of the mappings.

```
1)
SCHEMA idm;

ENTITY wall;
      name : STRING;
      height : REAL;
      width : REAL;
      azimuth : REAL;
      position : LIST [3:3] OF
REAL;
      UNIQUE name;
END_ENTITY;

END_SCHEMA;
```

```
2)
SCHEMA view1;

ENTITY column;
      name : STRING;
      height : REAL;
      radius : REAL;
      azimuth : REAL;
      position : LIST [3:3] OF
REAL;
      UNIQUE name;
END_ENTITY;

END_SCHEMA;
```

<u>Mapping</u>

```
inter_view(idm, integrated, view1, read_write, complete).

inter_class([wall],[column],
      equivalences(name = name,
                  height = height,
                  width = radius * 2,
                  azimuth = azimuth,
                  position[1] = position[1] - cos(azimuth) * radius,
                  position[2] = position[2] - sin(azimuth) * radius,
                  position[3] = position[3])
      ).
```

18) Example from Amor (1994).

This example shows the mapping required to provide representations of a wall as a positioned rectangle in the idm and as a set of four 3D points in view1.

```
1)
SCHEMA idm;

ENTITY wall;
      name : STRING;
      height : REAL;
      width : REAL;
      azimuth : REAL;
      position : LIST [3:3] OF
REAL;
      UNIQUE name;
END_ENTITY;

END_SCHEMA;
```

```
2)
SCHEMA view1;

ENTITY wall;
      name : STRING;
      corners : LIST [4:4] OF LIST
[3:3] OF REAL;
      UNIQUE name;
END_ENTITY;

END_SCHEMA;
```

Mapping

```
inter_view(idm, integrated, view1, read_write, complete).

inter_class([wall],[wall],
      equivalences(name = name,
            position[1] = corners[1, 1],
            position[1] = corners[4, 1],
            position[2] = corners[1, 2],
            position[2] = corners[4, 2],
            position[3] = corners[1, 3],
            position[3] = corners[2, 3],
            position[3] + height = corners[3, 3],
            position[3] + height = corners[4, 3],
            position[1] + width * cos(azimuth) = corners[2, 1],
            position[1] + width * cos(azimuth) = corners[3, 1],
            position[2] + width * sin(azimuth) = corners[2, 2],
            position[2] + width * sin(azimuth) = corners[3, 2])
      ).
```

19) Example from Amor (1994).

This example shows the mapping between a global coordinate system and an offset coordinate system. This example is rather contrived as it is more likely that there would be a space entity above the wall of the idm which would also be in global coordinates, and hence the calculation of the wall location would be considerably easier. The use of variables greatly simplifies the size of the expressions that would otherwise need to be written in this mapping.

```
1)
SCHEMA idm;

ENTITY wall;
      name : STRING;
      x_pos : REAL;
      y_pos : REAL;
      z_pos : REAL;
      height : REAL;
      width : REAL;
      azimuth : REAL;
      UNIQUE name;
END_ENTITY;

END_SCHEMA;
```

```
2)
SCHEMA view1;

ENTITY building;
      x_pos : REAL;
      y_pos : REAL;
      z_pos : REAL;
      azimuth : REAL;
END_ENTITY;

ENTITY space;
      x_offset : REAL;
      y_offset : REAL;
      z_offset : REAL;
      azimuth : REAL;
      from_building : building;
END_ENTITY;

ENTITY wall;
      name : STRING;
      x_offset : REAL;
      y_offset : REAL;
      height : REAL;
      width : REAL;
      azimuth : REAL;
      from_space : space;
      UNIQUE name;
END_ENTITY;

END_SCHEMA;
```

Mapping

```
inter_view(idm, integrated, view1, read_write, complete).

inter_class([wall],[wall],
      equivalences(name = name,
            height = height,
            width = width,
            WallTheta = tan_1(y_offset / x_offset),
            WallDist = sqrt(sqr(x_offset) + sqr(y_offset)),
            WallX = WallDist * cos(WallTheta + azimuth),
            WallY = WallDist * sin(WallTheta + azimuth),
            SpaceLocalX = WallX + from_space=>x_offset,
            SpaceLocalY = WallY + from_space=>y_offset,
            SpaceTheta = tan_1(SpaceLocalY / SpaceLocalX ),
            SpaceDist = sqrt(sqr(SpaceLocalX) + sqr(SpaceLocalY)),
            SpaceX = SpaceDist * cos(SpaceTheta + from_space=>azimuth),
            SpaceY = SpaceDist * sin(SpaceTheta + from_space=>azimuth),
```

```
        BuildingLocalX = SpaceX + from_space=>from_building=>x_pos,
        BuildingLocalY = SpaceY + from_space=>from_building=>y_pos,
        BuildingTheta = tan_1(BuildingLocalY / BuildingLocalX ),
        BuildingDist = sqrt(sqr(BuildingLocalX) + sqr(BuildingLocalY)),
        x_pos = BuildingDist*cos(BuildingTheta
              + from_space=>from_building=>azimuth),
        y_pos = BuildingDist*sin(BuildingTheta
              + from_space=>from_building=>azimuth),
        z_pos = from_space=>from_building=>z_pos + from_space=>z_offset,
        azimuth = BuildingTheta)
  ).
```

20) Example from Hardwick (1994).

This example shows the mapping required to provide a representation of a block as defined in AP 203 from STEP in a less complex form (i.e., collapsing the structure).

```
1)
SCHEMA easy_203;

ENTITY cube;
      x : REAL;
      y : REAL;
      z : REAL;
      size : REAL;
END_ENTITY;

END_SCHEMA;
```

```
2)
SCHEMA ap_203;

ENTITY block;
      position : axis2_placement;
      x : REAL;
      y : REAL;
      z : REAL;
END_ENTITY;

ENTITY axis2_placement;
      axis : direction;
      ref_direction : direction;
      location : cartesian_point;
END_ENTITY;

ENTITY direction;
      vector : LIST [3:3] OF REAL;
END_ENTITY;

ENTITY cartesian_point;
      coordinates : LIST [3:3] OF
REAL;
END_ENTITY;

END_SCHEMA;
```

Mapping

```
inter_view(easy_203, read_write, ap_203, integrated, complete).

inter_class([cube],[block],
      invariants(block.x = block.y,
                 block.y = block.z),
      equivalences(size = x,
                 x = position=>location=>coordinates[1],
                 y = position=>location=>coordinates[2],
                 z = position=>location=>coordinates[3]),
      initialisers([0,0,1] = position=>axis=>vector,
                 [0,0,1] = position=>ref_direction=>vector)
      ).
```

Appendix E

Large Example Models and Mappings

To provide an example of the use of the type of integrated design system and hence a more realistic example than those shown in Appendix D, the work performed in previous BRANZ contracts is highlighted (Hosking et al. 1995 and Mugridge et al. 1996). As part of these contracts a range of small tools were connected through an integrated data model (see Figure E.1). Several screen-dumps of the use of this system are shown in the following section, followed by descriptions of the tools and their schemas. The mappings for each tool are listed as well as a project window to show how the use of these tools could be managed in a project. Much of the description below is drawn from Hosking et al. (1995) and Mugridge et al. (1996).

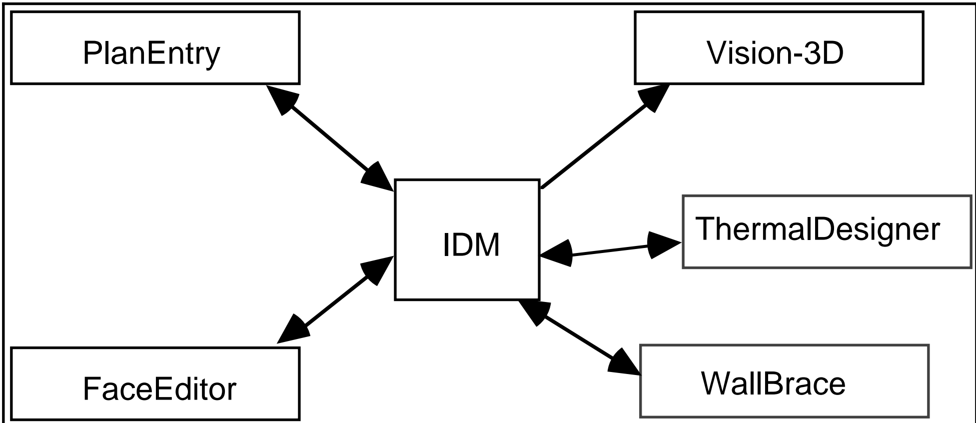


Figure E.1 Integration of tools in example

E.1 Description of Large Example

In this section we describe an example of the integrated system in action. Figure E.2 shows a building design for a simple L shaped building constructed using PlanEntry. Having constructed

this model, the user is currently in the process of mapping the design (as a whole, in this case) to the IDM and then on to the other tools. The additional windows provide information about the transactions involved in the various mappings, and allow the user to instigate the mapping process.

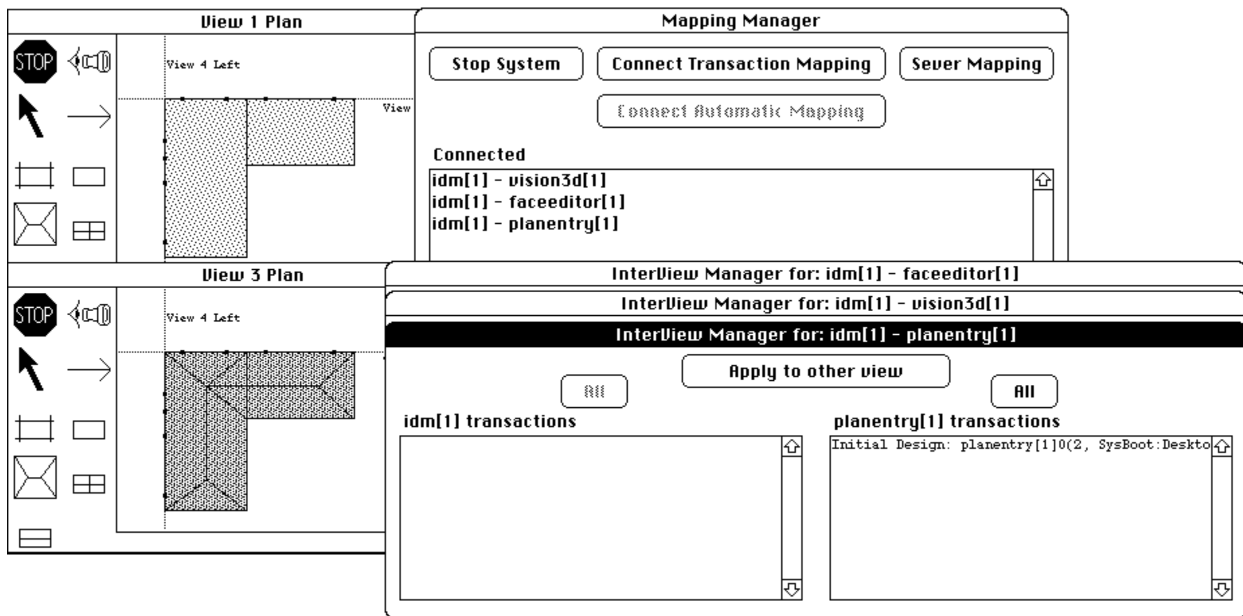


Figure E.2 Building design in PlanEntry with mapping controlers

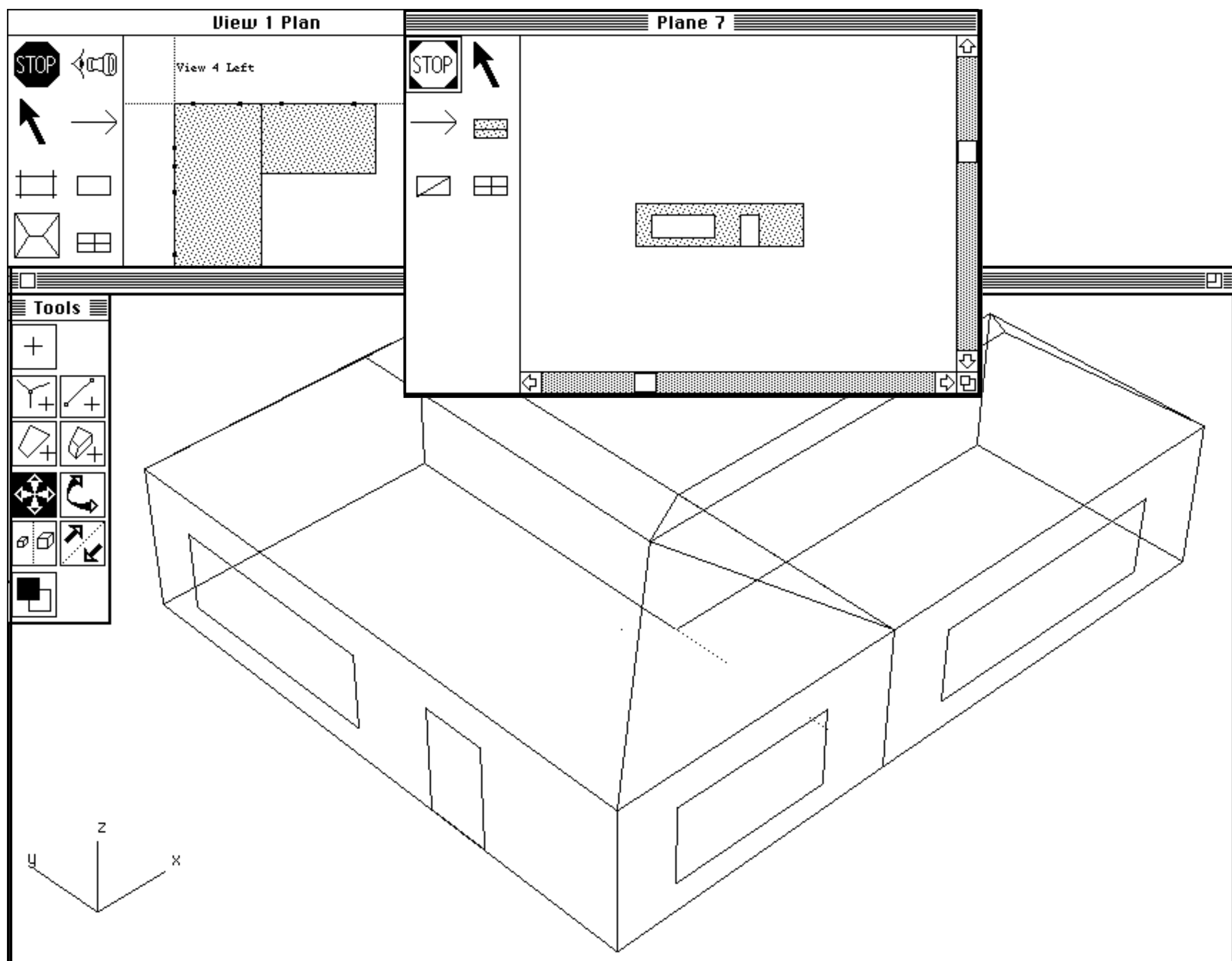


Figure E.3 Result of mapping to VISION-3D and FaceEditor

Figure E.3 shows the result of mapping the model through to VISION-3D (at rear) and mapping one of the faces through to the face editor to allow addition of materials.

Figure E.4 shows another design represented in three of the tools.

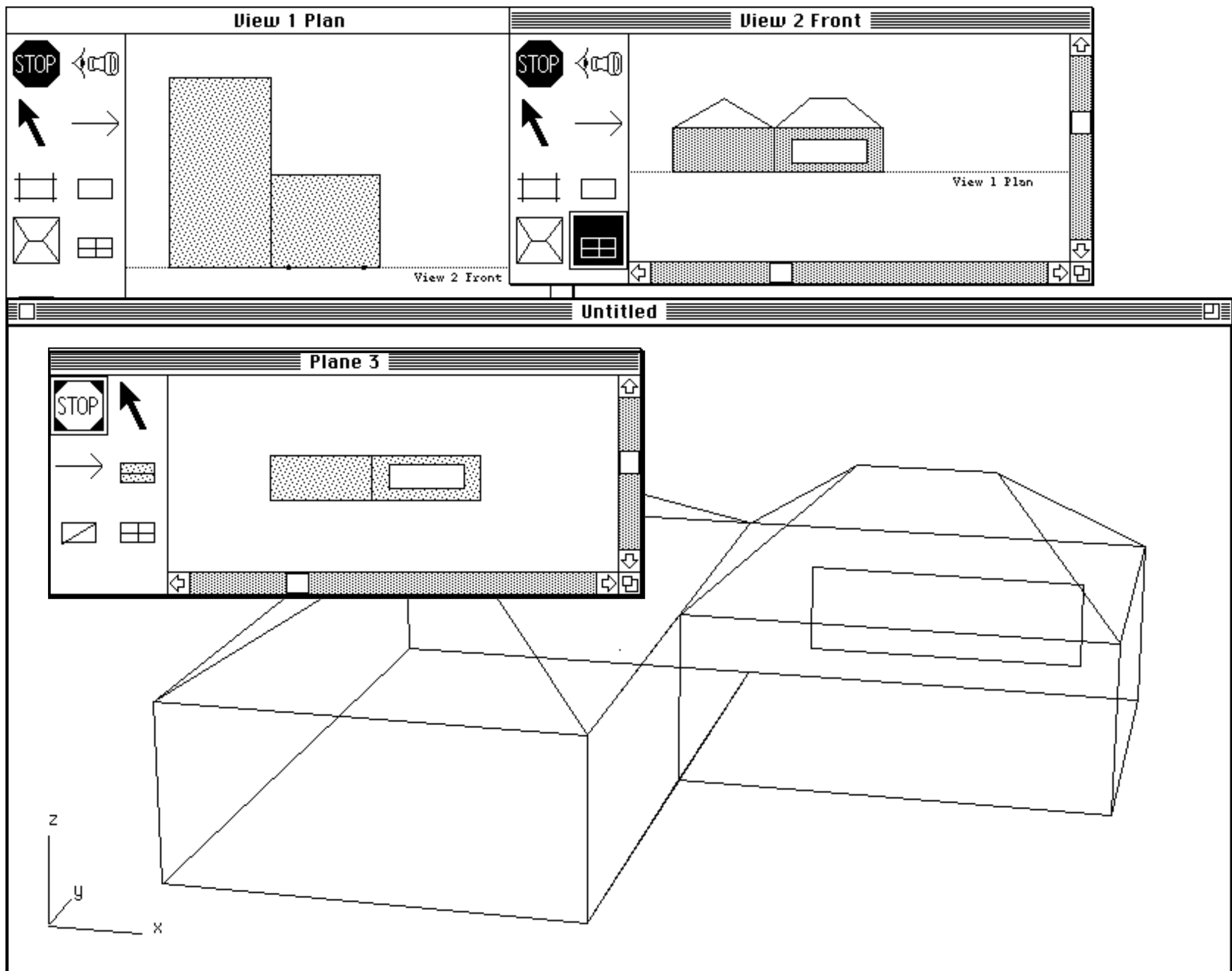


Figure E.4 Building design after mapping to three tools

A layout change for this design is shown in Figure E.5, where one of the spaces is moved so that the building changes from being an L-shape to being a T-shape.

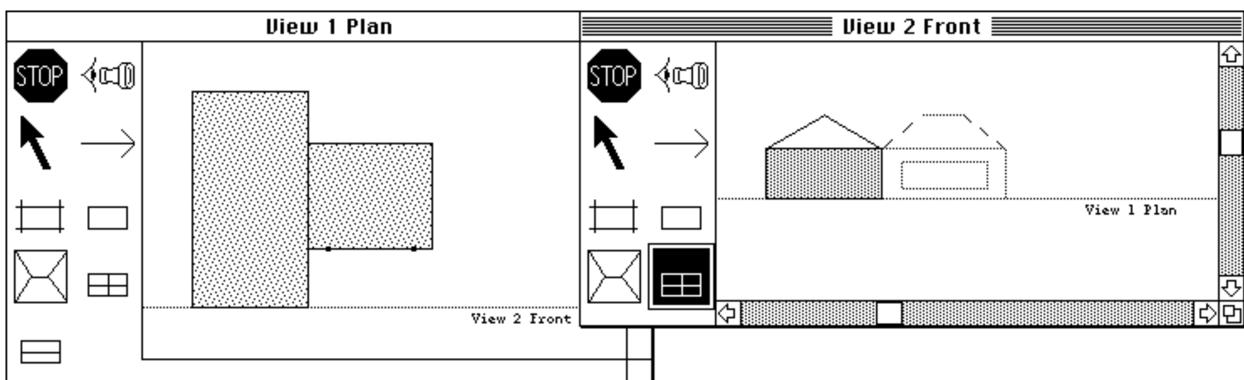


Figure E.5 Layout change to building in Figure E.4

The result of propagating this change to the other tools is shown in Figure E.6.

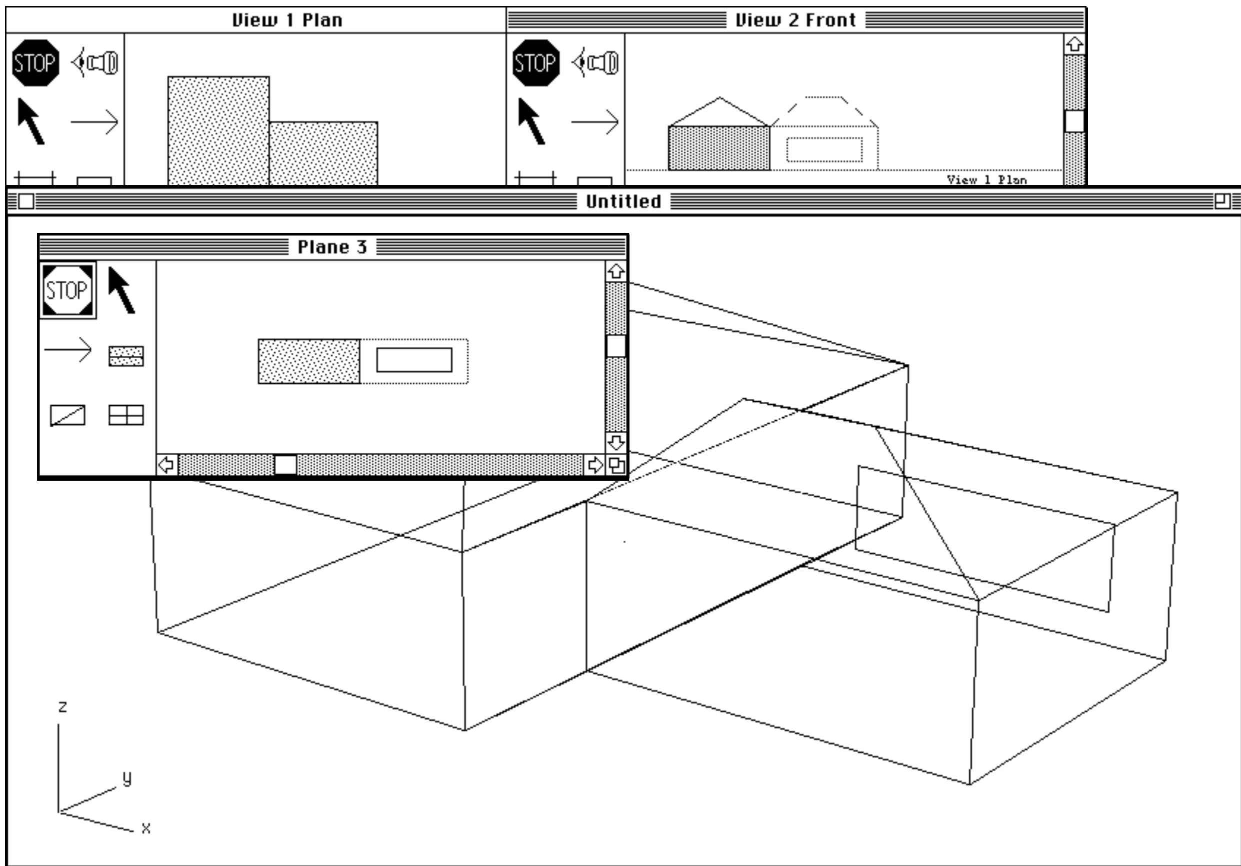


Figure E.6 Result of propagating changes shown in Figure E.5

E.2 Schemas for the Large Example

Detailed below are the schemas for the IDM, PlanEntry, FaceEditor, VISION-3D, and ThermalDesigner. The schemas are described utilising a combination of SPE diagrams and the full code of some of the schemas. The code for PlanEntry and FaceEditor is not presented, as the class definitions are tied very closely to the implementation, giving unnecessary complexity for this appendix.

E.2.1 IDM schema

The schema description for the IDM is shown below. This incorporates elements from PlanEntry and the FaceEditor, as well as including building attributes that are relevant to ThermalDesigner and WallBrace. Following the approach taken in (Mugridge and Hosking 1995) the IDM is composed of several views of the building components. A geometric or space oriented view specifies the geometric properties of the building components. A materials view specifies thermally-oriented materials characteristics. A bracing view specifies bracing-related characteristics.

Redundancy is managed from outside the IDM, consequently, any tool that changes space information, for example, must ensure that plane and face information that is dependent on that change is also changed to retain consistency.

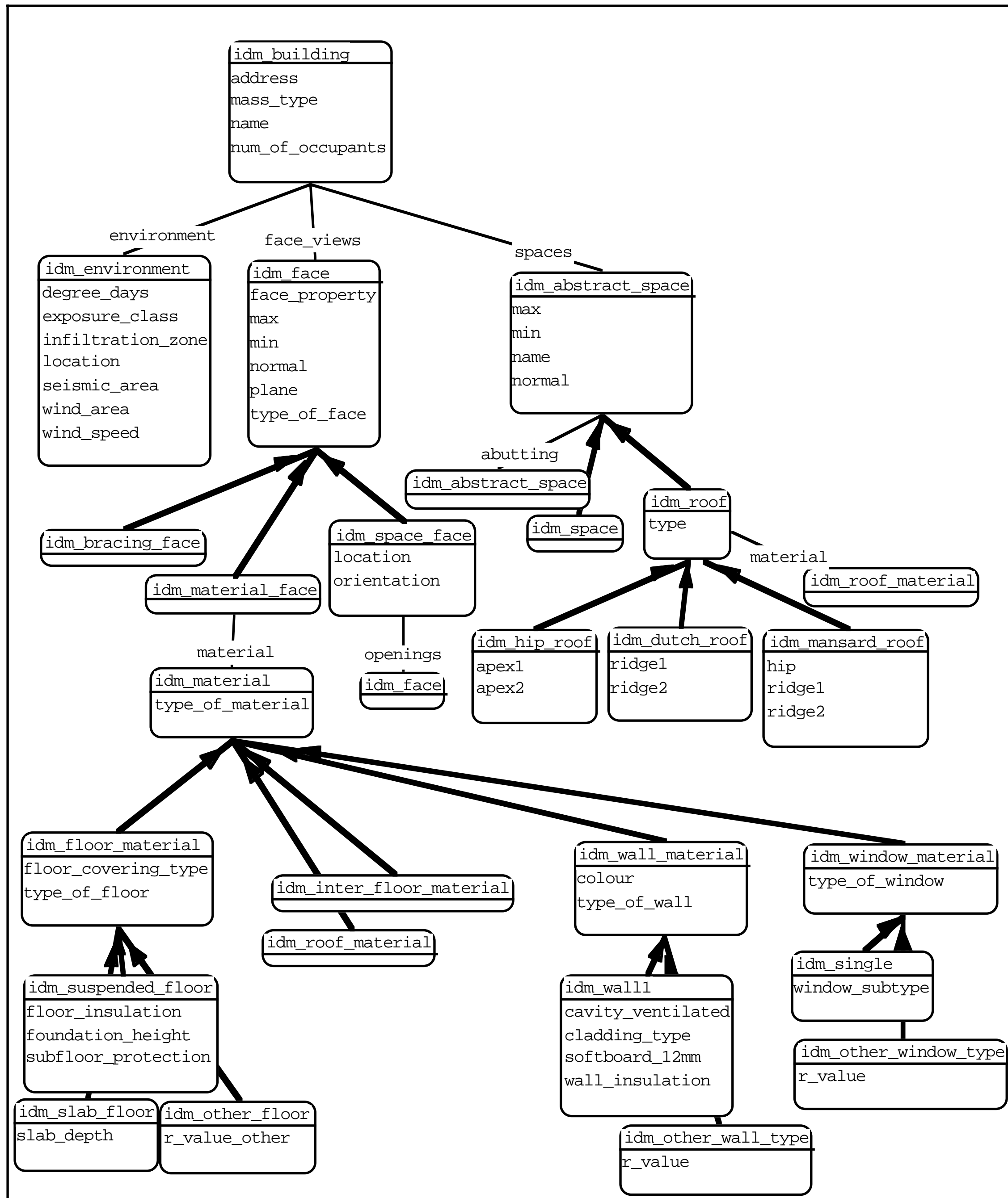


Figure E.7 IDM schema

```
class(idm_building, inherits(idm_cuboid), features(
        name                : text ,
        address             : text,
        mass_type           : [spp,  sppb, other],
        num_of_occupants    : integer,
        spaces              : list(idm_abstract_space),
        face_views          : list(idm_face),
        environment         : idm_environment
)).
```

```
class(idm_environment, features(
        wind_area         : [low, medium, high],
        seismic_area      : [a, b, c],
        wind_speed        : float, %   affects wind_area
        exposure_class    : [sheltered, medium_sheltered,medium_exposed,  exposed],
        % Some relation between exposure_class and wind_area/ local_acceleration
        location          : [auckland,  hamilton,  napier, new_plymouth, wellington,
                             christchurch, dunedin, invercargill, other],
        infiltration_zone : [a, b, c, d], % uses location if it's <> other
        degree_days       : integer       % uses location if it's <> other
)).

class(idm_2d_point, features(
        x    : float,
        y    : float
)).

class(idm_3d_point, features(
        x    : float,
        y    : float,
        z    : float
)).

class(idm_line, features(
        p1    : idm_3d_point,
        p2    : idm_3d_point
)).

class(idm_rectangle, features(
        normal   : [x, y, z],
        min      : idm_2d_point,
        max      : idm_2d_point
)).

class(idm_cuboid, features(
        normal   : [x, y, z],
        min      : idm_3d_point,
        max      : idm_3d_point
)).

class(idm_plane, features(
        name     :  text,
        axis     : [x, y, z],
        offset   : float,
        view_plane
)).

idm_plane::view_plane :- true. % so we can pass messages about planes

class(idm_abstract_space, inherits(idm_cuboid), features(
        name     : text,
        abutting : list(idm_abstract_space)
)).

class(idm_space, inherits(idm_abstract_space), features(
)).

class(idm_face, inherits(idm_rectangle), features(
        type_of_face   : [wall, floor, inter_floor, opening],
        face_property : classifier(idm_bracing_face,  idm_material_face,  idm_space_face),
        plane                        : idm_plane
)).

class(idm_bracing_face, inherits(idm_face), features(
)).
```

```
class(idm_material_face, inherits(idm_face), features(
        material : idm_material
)).

class(idm_space_face, inherits(idm_face), features(
        location    : [int, ext],
        orientation : [n, ne, e, se, s, sw, w, nw, up, down],
        openings    : list(idm_face)  % Includes windows, doors, sky-lights, etc
)).

class(idm_material, features(
        type_of_material  : classifier(idm_floor_material, idm_inter_floor_material,
                 idm_roof_material, idm_wall_material, idm_window_material)
)).

class(idm_floor_material, inherits(idm_material), features(
        floor_covering_type : ['carpet and underlay', 'cork tile',   'other covering'],
        type_of_floor     : classifier(idm_suspended_floor, idm_slab_floor, idm_other_floor)
)).

class(idm_suspended_floor, inherits(idm_floor_material), features(
        foundation_height   : float,
        subfloor_protection : [a, b, c],
        floor_insulation    :  [uninsulated, 'perforated foil 25mm',   'lined 75mm blanket']
)).

class(idm_slab_floor, inherits(idm_floor_material), features(
        slab_depth        : float
)).

class(idm_other_floor, inherits(idm_floor_material), features(
        r_value_other : float
)).


class(idm_roof, inherits(idm_abstract_space), features(
        material : idm_roof_material,
        type     : classifier(idm_hip_roof, idm_dutch_roof, idm_mansard_roof)
)).

class(idm_hip_roof, inherits(idm_roof), features(
        apex1          : idm_3d_point,
        apex2          : idm_3d_point
)).

class(idm_dutch_roof, inherits(idm_roof), features(
        ridge1      : idm_line,
        ridge2      : idm_line
)).

class(idm_mansard_roof, inherits(idm_roof), features(
        ridge1      : idm_line,
        hip         : idm_line,
        ridge2      : idm_line
)).

class(idm_roof_material, inherits(idm_material), features(
        colour         : [dark, light],
        type_of_roof   : classifier(idm_roof1_material, idm_other_roof_material)
)).
```

```
class(idm_roof1_material, inherits(idm_roof_material), features(
        building_paper    : boolean,
        cladding_primed : boolean,
        roof_insulation : [nil,'fibre 75','fibre 100',  'fibre 150',  'paper 150'],
        softboard_12mm    : boolean
)).

class(idm_other_roof_material, inherits(idm_roof_material), features(
        r_value : float
)).

class(idm_inter_floor_material, inherits(idm_material), features(
        type_of_inter_floor : classifier(idm_inter_floor1_material,
           idm_other_inter_floor_material)
)).

class(idm_inter_floor1_material, inherits(idm_inter_floor_material), features(
         ceiling_type           :  ['panel ceiling', 'slab ceiling', 'other ceiling'],
        inter_floor_insulation : [nil,  'fibre 75','fibre 100',  'fibre 150',  'paper 150'],
        softboard_12mm         : boolean
)).

class(idm_other_inter_floor_material, inherits(idm_inter_floor_material), features(
        r_value : float
)).

class(idm_wall_material, inherits(idm_material), features(
        colour         : [dark, light],
        type_of_wall   : classifier(idm_wall1, idm_other_wall_type)
)).

class(idm_wall1, inherits(idm_wall_material), features(
        cavity_ventilated : boolean,
        cladding_type  : ['bevelback timber', 'lightweight ventilated PVC',
                          'fibre reinforced planks'],
        softboard_12mm    : boolean,
        wall_insulation   : [nil,  foil, 'fibre 75','fibre 94','paper 94']
)).

class(idm_other_wall_type, inherits(idm_wall_material), features(
        r_value : float
)).

class(idm_window_material, inherits(idm_material), features(
        type_of_window : classifier(idm_single, idm_other_window_type)
)).

class(idm_single, inherits(idm_window_material), features(
        window_subtype : [clear,  'clear with eaves', tinted,  reflective,'emit 0.6',
                          'emit 0.2']
)).

class(idm_other_window_type, inherits(idm_window_material), features(
        r_value : float
)).
```

E.2.2 PlanEntry schema

PlanEntry is a constraint-based CAD tool for constructing and manipulating three-dimensional building models from multiple two-dimensional projections. It was developed as part of Uniservices project 4376.01 (Hosking and Mugridge, 1994), using the Snart object-oriented constraint language.

Figure E.8 (from Hosking et al, 1994) shows PlanEntry in use. Four views of a building are shown: two plan views, at different heights; and front and side elevations. Each view has a tool palette for manipulating and extending the building model via its two-dimensional projection.

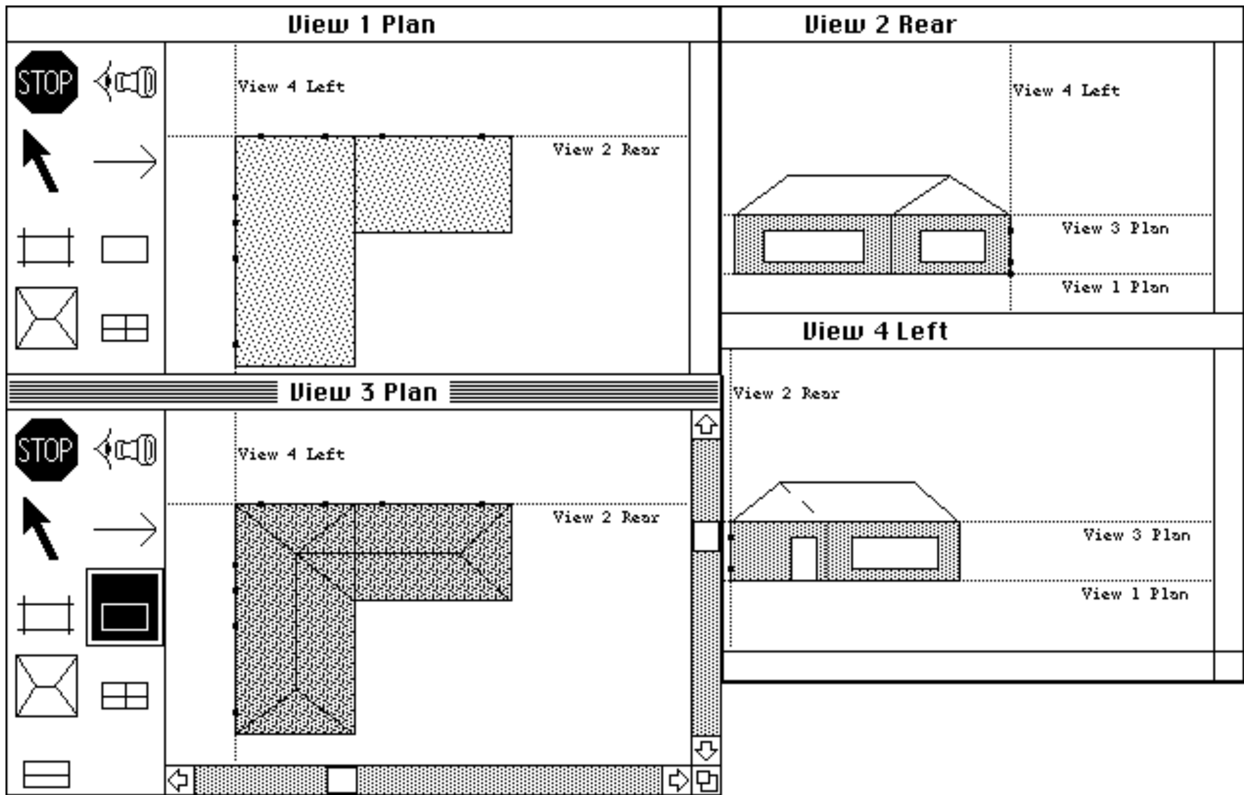


Figure E.8 PlanEntry in use

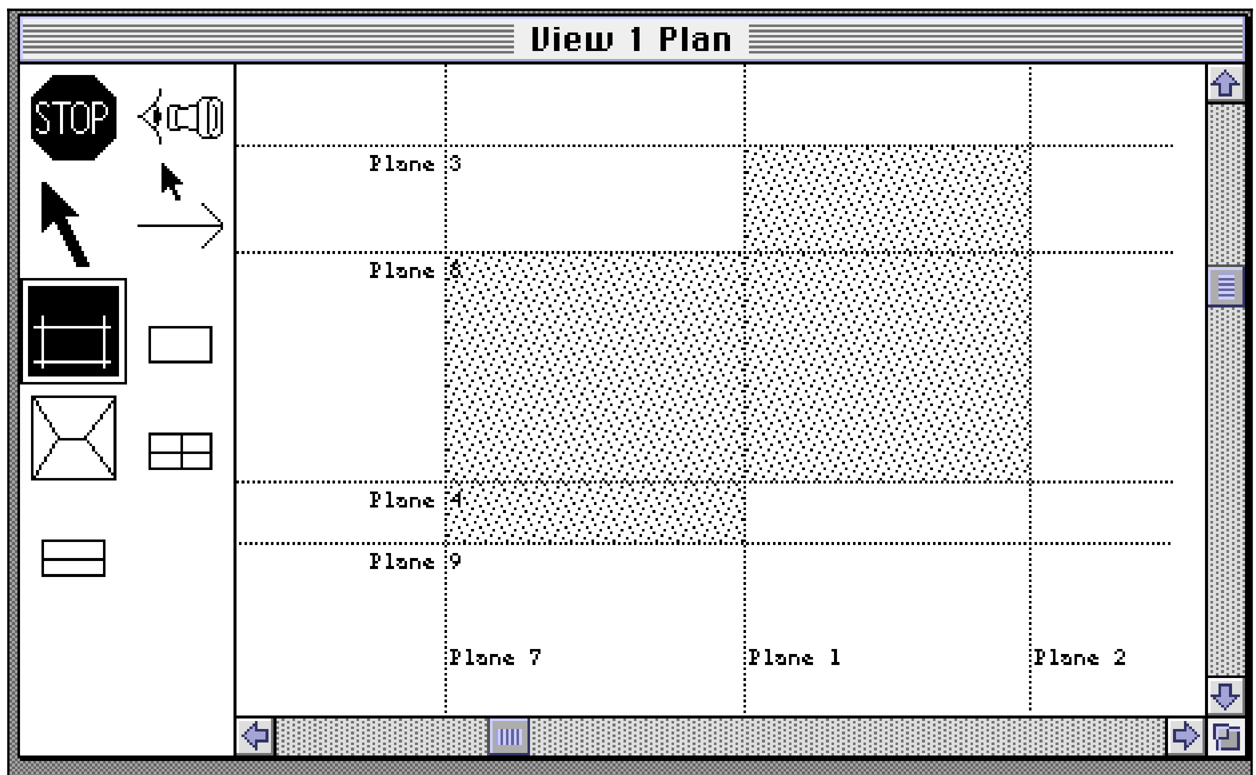


Figure E.9 Planes calculated for a building

For the purposes of the BRANZ project, the PlanEntry tool was extended with planes. Each plane is coincident with one or more faces of spaces of the building. Planes are made visible by the user selecting the plane tool (see Figure E.9).

Clicking on a plane using the PlanEntry face tool causes the FaceEditor to provide a view of the faces of the building that lie on that plane, ready for specifying or modifying additional information about those faces (as shown in Figure E.11).

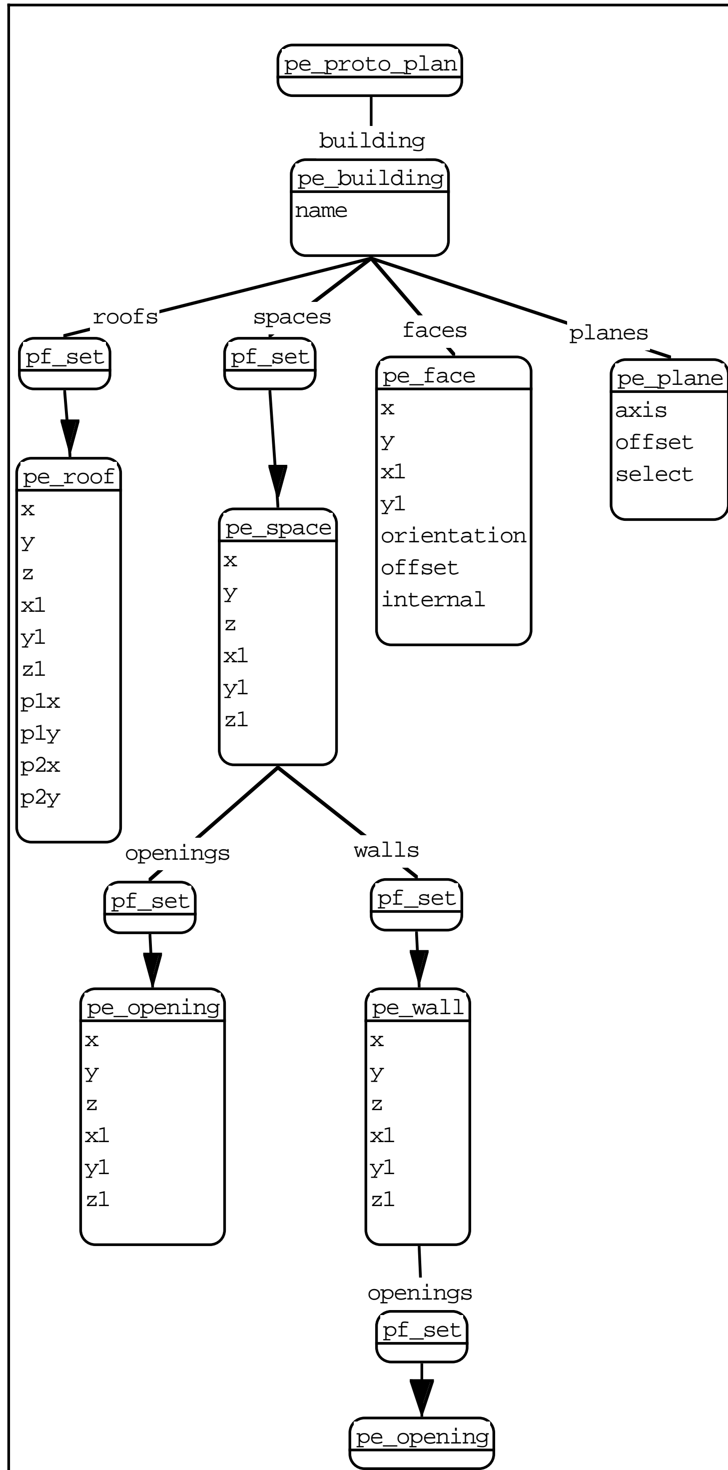


Figure E.10 PlanEntry schema

The FaceEditor requires geometric information concerning the faces of a building that lie on a plane. As these planes and faces are completely determined by the juxtaposition of spaces in PlanEntry, the PlanEntry model has been extended to include this plane and face information. Consistency between the spaces, planes and faces is maintained by PlanEntry.

Each space has six or more associated faces. Where spaces abut, the partially overlapping faces are divided according to whether they are internal or external. For example, in the figures above, a wall of the larger space partially abuts another space so it is divided in two; one face is an internal wall while the other is exterior. Faces have no rendered representation in PlanEntry; they are constructed for use by other tools, via the IDM.

The resulting schema for the PlanEntry building model, including all features of interest to the integrated data model, is shown in below. While spaces, faces, and planes are represented independently and redundantly in the schema, Plan Entry is responsible for maintaining their consistency. There is a functional dependency from the set of spaces to the set of faces and to the set of planes. When the set of spaces is altered (such as through adding or deleting a space, or through moving or resizing an existing space), PlanEntry ensures that the set of planes and faces is updated appropriately. Additional user-supplied information may be associated with the original faces, as they were before the change. Thus any updates to the faces (and planes) have to be managed carefully to avoid loss of such information.

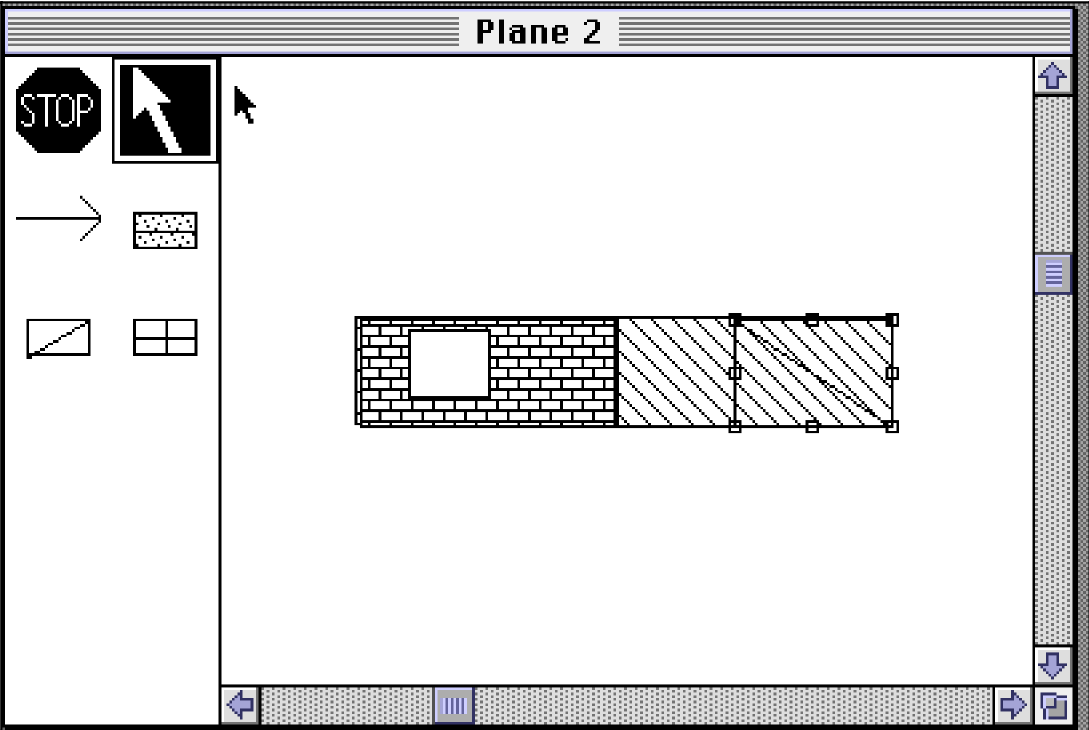


Figure E.11 FaceEditor in use

E.2.3 FaceEditor schema

The FaceEditor tool was developed specifically for the BRANZ project. When a plane is selected by the PlanEntry face tool, geometric information about the plane is passed through to the FaceEditor. The FaceEditor allows materials and bracing information to be overlaid on faces that lie on the selected plane, including the material of openings (such as windows). This permits, for example, a single wall material to be specified for the faces of several spaces that lie together on a plane (e.g., several storeys) as well as for several different wall materials to be specified for different parts of a single face.

Figure E.11 shows an example of the FaceEditor in operation, specifying materials and bracing for one face of a building.

To allow for the later integration of the ThermalDesigner and WallBrace tools, the FaceEditor can be used to specify wall materials both in terms of their thermal properties or their bracing properties. For example, the user may specify a timber framed weatherboard/plasterboard wall which has diagonal braces. However, at this stage the FaceEditor does not ensure that overlapping thermal and bracing properties are consistent; this will be the subject of future work.

The FaceEditor allows multiple planes to be selected and edited. Plane views may be hidden when not needed.

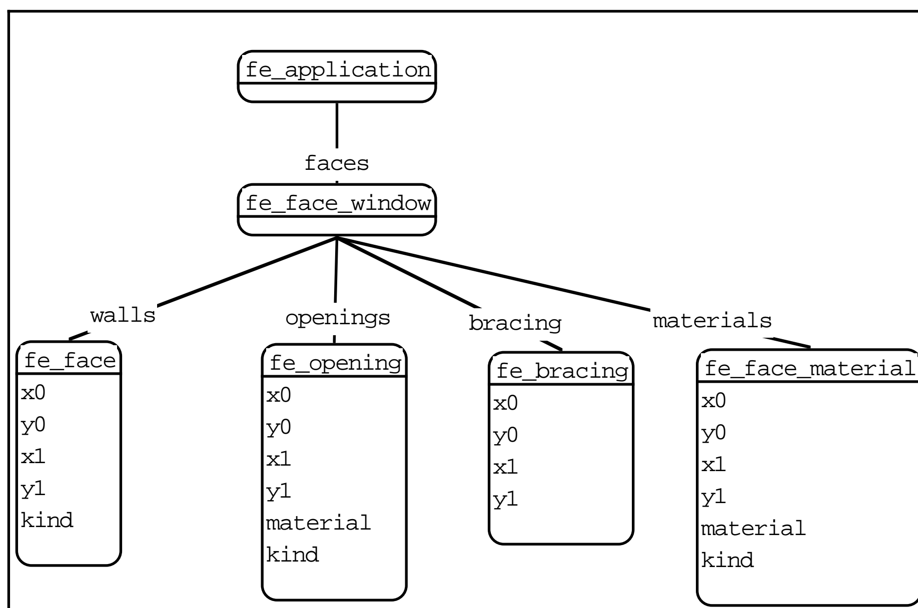


Figure E.12 FaceEditor schema

The schema for the FaceEditor model, including all features of interest to the integrated data model, is shown in the figure below. Geometric information of walls and openings that lie on a plane are provided to the FaceEditor from PlanEntry, via the IDM. These may not be altered in the FaceEditor.

In this schema, both fe_opening and fe_face_material entities have an associated material attribute. This is a string specifying the material type, and is associated in the mapping with an object defining the material attributes in the IDM. The kind attributes are used to specify the type of rendering to be associated with the various FaceEditor components.

E.2.4 VISION-3D schema

VISION-3D is a 3D model creation, editing, and rendering system for the Macintosh, developed by staff at the University of Auckland School of Architecture (Bourke, 1989). It can be supplied with a variety of scene description formats, which can be rendered in several different ways (wire frame, hidden line, coloured, shaded, etc). It is possible to observe the scene from any angle, and to create fly-through animations.

For the purposes of this project, VISION-3D is only used to provide a three dimensional rendering of a building model. An example of such a rendering is shown in Figure E.13. Data entry for this application is via a formatted file, which is constructed according to the mapping defined between the VISION-3D and IDM schemas.

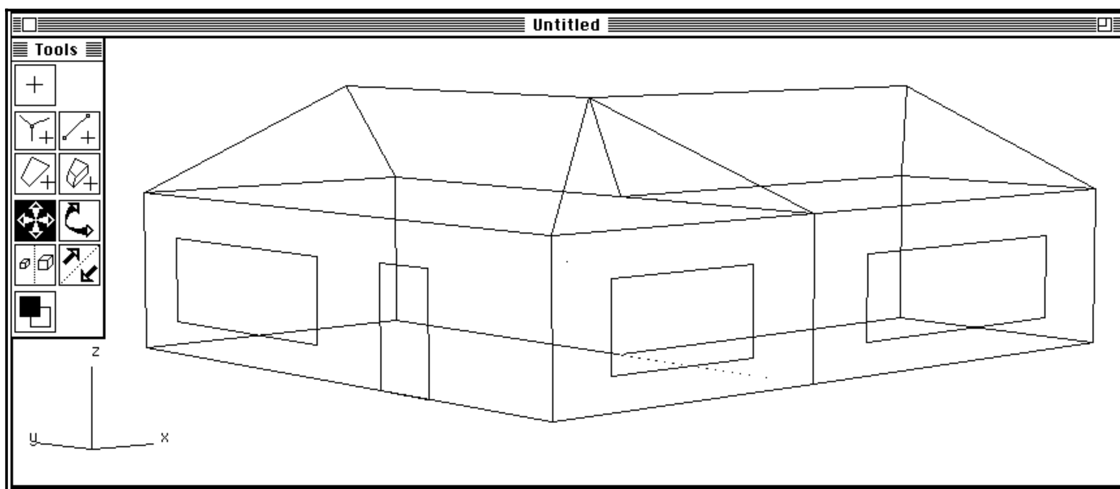


Figure E.13 VISION-3D in use

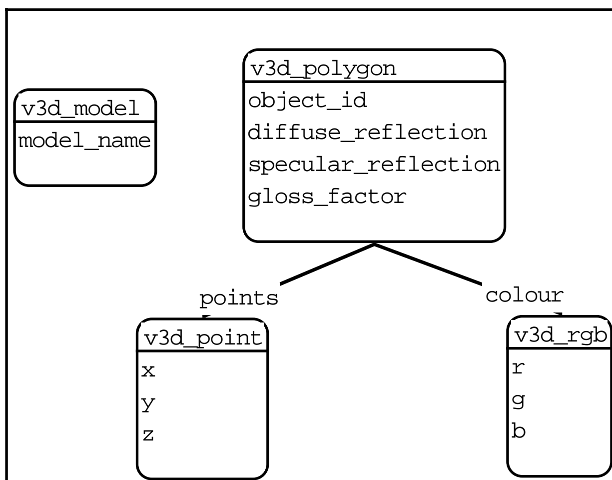


Figure E.14 VISION-3D schema

A schema for VISION-3D, corresponding to the descriptors in the file, is shown in Figure E.14 and below. A small application has been written which takes models in the form of this schema and generates the formatted file representation (see *dump_to_file*).

```
/*******************************************************
 * © Copyright Robert W. Amor 1995
 *
 * Department of Computer Science
 * University of Auckland
 * Private Bag 92019
 * Auckland
 * New Zealand
 *
 * This software may be duplicated and used for research
 * purposes as long as this copyright message remains.
 *
 ******************************************************/
/*******************************************************
 * Purpose: Collate information on the whole model
 ******************************************************/
class(v3d_model,
 features(
  model_name : string,
  create,
  dump_to_file
 )
).

/*******************************************************
 * Purpose: Print out info on create
 ******************************************************/
v3d_model::create :-
   cat([self, '@dump_to_file'], Meth, _),
   writeseqnl(['To dump the Vision3D model to file type: ',Meth]).

/*******************************************************
 * Purpose: Allow the whole model to be dumped to disk for use
 *          in Vision 3D
 ******************************************************/
v3d_model::dump_to_file :-
   new(File, 'Vision-3D file to create?', '*.v3d'),
   open(File, write),
   self@'#space'(Space),
   Space@all_objects(Objs),
   findall(P, (member(P, Objs), P@class(v3d_polygon)), Polygons),
   forall(member(Poly, Polygons), Poly@dump_to_file(File)),
   close(File).

/*******************************************************
 * Purpose: Collate information on a single polygon in an object.
 *          The object is determined by the object_id
 ******************************************************/
class(v3d_polygon,
 features(
  object_id : int,
  diffuse_reflection : float,
  specular_reflection : float,
  gloss_factor : float,
  points : list(v3d_point),
  colour : v3d_rgb,
  dump_to_file(+file)
 )
).
```

```
/********************************************************
 * Purpose: Output all information on a polygon in the correct
 *          format and order to the named file
********************************************************/
v3d_polygon::dump_to_file(File) :-
   self@object_id(OID),
   self@diffuse_reflection(DR),
   self@specular_reflection(SR),
   self@gloss_factor(GF),
   self@points(Points),
   self@colour(Colour),
   length(Points, NumPoints),
   writeseqnl([NumPoints, OID, DR, SR, GF]) ~> File,
   forall(member(Point, Points), Point@dump_to_file(File)),
   Colour@dump_to_file(File).

/********************************************************
 * Purpose: Definition of a point in 3D space
********************************************************/
class(v3d_point,
 features(
  x : float,
  y : float,
  z : float,
  dump_to_file(+file)
 )
).

/********************************************************
 * Purpose: Write the points location to the named file
********************************************************/
v3d_point::dump_to_file(File) :-
   self@x(X),
   self@y(Y),
   self@z(Z),
   writeseqnl([X, Y, Z]) ~> File.

/********************************************************
 * Purpose: Definition of a colour in RGB vector
********************************************************/
class(v3d_rgb,
 features(
  r : float,
  g : float,
  b : float,
  dump_to_file(+file)
 )
).

/********************************************************
 * Purpose: Write the colour vector to the named file
********************************************************/
v3d_rgb::dump_to_file(File) :-
   self@r(R),
   self@g(G),
   self@b(B),
   writeseqnl([R, G, B]) ~> File.
```

E.2.5 ThermalDesigner schema

ThermalDesigner (Amor et al, 1992) helps a designer to check that a building design meets the requirements of the New Zealand thermal insulation standard for residential buildings, NZS4218P (SANZ, 1977). It is based on an approach developed by the Building Research Association of

New Zealand (BRANZ) as a paper design guide (Bassett et al, 1990). Figure E.15 shows the forms-based interface to the application.

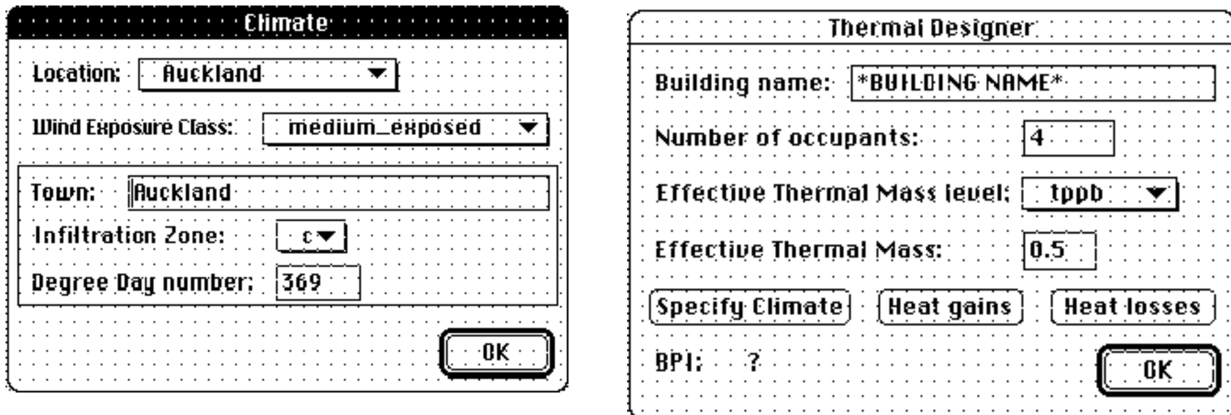


Figure E.15 ThermalDesigner in use

```
/*
   File:    ThermalDesigner.sn
   Purpose: Prototype of the ALF thermal design system.
   First Developed in Kea by:
         Robert Amor
         Department of Computer Science,University of Auckland
   Date:    April 1991

   Then redeveloped in Snart by:
         Rick Mugridge
         Department of Computer Science, University of Auckland
   Date:    July 1995
   Copyright  (C) 1991,1995 Building Research Association of New Zealand

*/

%___________ Climate ___________

class(td_climate,
  features(   % All Defined from IDM
   exposure_class: [sheltered, medium_sheltered, medium_exposed, exposed],
   infiltration_zone: [a, b, c, d],
   degree_days: int,
 create % Set up the local dialog for Climate
)).

td_climate::create :-
  set_prop(td,climate,self),
  climate_dialog.

%___________ Space ___________

class(td_space,
  features(
   floor: list(td_floor), % Defined from IDM
   roof: list(td_roof),      % Defined from IDM
   walls: list(td_wall)), % Defined from IDM
  constraints(
   space_volume: float := sum(collect(f in floor, f@area)) * max(collect(w in walls,
w@height)),
   walls_heat_loss: float   := sum(collect(w in walls, w@heat_loss)),
   floor_heat_loss: float   := sum(collect(f in floor, f@heat_loss)),
   roof_heat_loss: float   := sum(collect(r in roof, r@heat_loss)),
   windows_heat_loss: float := sum(collect(w in walls, w@solar_loss_total)),
   windows_heat_gain: float := sum(collect(w in walls, w@solar_gain_total)),
```

```
   solar_glazing_area: float
      := sum(collect(w in walls, w@window_area_total where (w@orientation in [n, ne, nw]))),
   joint_length: float
      := sum(collect(w in walls, w@joint_length)) + sum(collect(f in floor, f@joint_length))
   + sum(collect(r in roof, r@joint_length))
)).

%___________ Floor ___________

class(td_floor(degree_days: int),
   features( % All defined from IDM, except r_value
      construction_type: classifier(td_suspended_floor, td_concrete_floor),
      length: float,
      width: float,
      floor_covering_r_value: float,
      r_value: float), % Calculated in subclasses
   constraints(
      area: float := length * width,
      perimeter_length: float := 2.0 * (length + width),
      area_perimeter_ratio: float := area / perimeter_length,
      alf: float := 3.5343 + 0.045608 * degree_days - 1.1001e-5 * degree_days * degree_days,
      heat_loss: float := area * alf / r_value,
      joint_length: float := length + width
)).

%___________ Suspended Floor ___________

class(td_suspended_floor, inherits(td_floor),
   features(  % All from IDM
   subfloor_protection: [a,b,c],
   foundation_height: float,
   floor_insulation_r_value: float),
   constraints(
   perimeter_wall_area: float := area * foundation_height,
   area_perimeter_wall_area_ratio: float := area / perimeter_wall_area,
   r_value  :=  subfloorR(subfloor_protection,area_perimeter_wall_area_ratio)  +
floor_insulation_r_value + floor_covering_r_value
)).

subfloorR(a,_,0.01).
subfloorR(b,Ratio,V) :- V is 0.05714 * Ratio.
subfloorR(_,Ratio,V) :- V is 0.1 * Ratio.

%___________ Concrete Floor ___________

class(td_concrete_floor, inherits(td_floor),
   features(  % Defined from IDM
   insulation_depth: float),
   constraints(
   slab_r_value: float := 0.4 + 0.2 * insulation_depth + area_perimeter_ratio *
           (0.464 + 0.052 * insulation_depth),
        % Formula derived from the graphs given on ALF p16.
        % Extrapolation beyond insulation depth = 1m is unwise
   r_value := slab_r_value + floor_covering_r_value
)).

%___________ Roof ___________

class(td_roof(degree_days: int),
   features(  % Defined from IDM
      length: float,
      width: float,
      r_value: float,
      colour: [light, dark]),
```

```
   constraints(
      area: float := length * width,
      alf: float := roof_alfR(colour,degree_days),
      heat_loss : float := area * alf / r_value,
      joint_length : float := length + width
)).

roof_alfR(dark,Days,V) :-
  V is -9.7212 + 0.065364 * Days - 1.9697e-5 * Days * Days.
roof_alfR(light,Days,V) :-
  V is -2.8380 + 0.063549 * Days - 1.9522e-5 * Days * Days.

%___________ Wall ___________

class(td_wall(degree_days: int),
   features( % All from IDM
      width: float,
      height: float,
      orientation : [n, ne, nw, w, e, sw, se, s],
      colour: [dark, light],
      windows: list(td_window),
      r_value: float),
   constraints(
      alf: float := alfR(colour, orientation, degree_days),
      heat_loss: float := area * alf / r_value,
      window_area_total : float := sum(collect(j in windows, j@area)),
      area: float := width * height - window_area_total,
      solar_gain_total: float := sum(collect(w in windows, w@heat_gain)),
      solar_loss_total: float := sum(collect(w in windows, w@heat_loss)),
      % a hack for joint length, presume that it is half the joint length of a wall
      % as each wall joint touches the wall joint of another wall or ceiling or floor.
      joint_length: float := width + height
)).

alfR(light, Or, Days, V) :-
   member(Or,[n, ne, nw]),
   V is -5.9841 + 0.060934 * Days - 1.8473e-5 * Days * Days.
alfR(light, Or, Days, V) :-
   member(Or,[e, w]),
   V is -3.9841 + 0.060934 * Days - 1.8473e-5 * Days * Days.
alfR(light, Or, Days, V) :-
   member(Or,[s, se, sw]),
   V is -2.6186 + 0.060696 * Days - 1.8240e-5 * Days * Days.
alfR(dark, n, Days, V) :-
   V is -16.875 + 0.065122 * Days - 1.9639e-5 * Days * Days.
alfR(dark, Or, Days, V) :-
   member(Or,[ne, nw]),
   V is -13.005 + 0.063322 * Days - 1.8765e-5 * Days * Days.
alfR(dark, Or, Days, V) :-
   member(Or,[e, w]),
   V is -11.529 + 0.067010 * Days - 2.0688e-5 * Days * Days.
alfR(dark, Or, Days, V) :-
   member(Or,[s, se, sw]),
   V is -7.0047 + 0.063322 * Days - 1.8765e-5 * Days * Days.

%___________ Window ___________

class(td_window(facing: orientation_type, degree_days: int),
   features(
      r_value: float,
      shading_coef: float,
      height: float,
      width: float),
```

```
   constraints(
      area: float := height * width,
      alf: float := 12.0 + 0.02143 * degree_days,
      heat_loss: float := area * alf / r_value,
      heat_gain: float := area * shading_coef * agfR(facing,degree_days)
)).

agfR(n,Days,V) :- V is 429.41 - 0.241220 * Days + 4.1833e-5 * Days * Days.
agfR(Or,Days,V) :- member(Or,[ne,nw]), V is 354.59 - 0.234590 * Days + 5.8192e-5 * Days *
Days.
agfR(Or,Days,V) :- member(Or,[e,w]), V is 221.18 - 0.158000 * Days + 4.1833e-5 * Days * Days.
agfR(Or,Days,V) :- member(Or,[se,sw,s]), V is 99.468 - 0.071612 * Days + 2.4488e-5 * Days *
Days.


%___________ Building ___________

class(td_building,
   features(
        building_name: text,   % Defined from IDM
        num_of_occupants: int,    % Defined from IDM
        effective_thermal_mass: float,  % Defined from IDM
        spaces: list(td_space),   % Defined from IDM
        climate: td_climate,   % Defined from IDM
        create),   % Used to set up local dialogs
   constraints(
   floor_area_total: float := sum(collect(s in spaces, sum(collect(f in s@floor, f@area)))),
   building_volume_total: float := sum(collect(r in spaces, r@space_volume)),
   joint_length: float := sum(collect(s in spaces, s@joint_length)),
   joint_volume_ratio: float := joint_length / building_volume_total,
   air_leak_temp: float := air_leak(climate@infiltration_zone) * joint_volume_ratio,
   air_leakage_rate: float := air_leak_rate(climate@exposure_class) * air_leak_temp,
   air_leakage_alf: float := -3.3678 + 0.022838 * climate@degree_days
           - 6.7557e-6 * climate@degree_days * climate@degree_days,
   air_heat_loss: float := building_volume_total * air_leakage_alf * air_leakage_rate,
   total_seasonal_heat_losses: float
      := air_heat_loss
      + sum(collect(s in spaces, s@floor_heat_loss + s@roof_heat_loss
              + s@walls_heat_loss + s@windows_heat_loss)),
   internal_heat_gain: float := 900.0 + 150.0 * num_of_occupants,
   total_seasonal_gain: float
      := internal_heat_gain + sum(collect(s in spaces, s@windows_heat_gain)),
   gain_loss_ratio: float := total_seasonal_gain / total_seasonal_heat_losses,
   glr: float := gain_loss_ratio,
      % Equations for the graphs in ALF: use the closest graph for finding
      % a point instead of a trickier interpolation to the correct value -
      % things are so vague anyway that this slight inaccuracy won't matter
      % a damn
   useful_fraction: float := useful_fract(effective_thermal_mass,glr),
   useful_heat_gain: float := total_seasonal_gain * useful_fraction,
   net_heating_EU: float := total_seasonal_heat_losses - useful_heat_gain,
   solar_glazing_area: float := sum(collect(s in spaces, s@solar_glazing_area)),
   solar_glazing_floor_area_ratio: float := solar_glazing_area / floor_area_total,
   est_max_winter_indoor_temp: float
      := 20.0 + 142.857 * (1.0 - 0.38333 * effective_thermal_mass)
         * solar_glazing_floor_area_ratio,

   bpi: float := net_heating_EU / (climate@degree_days * floor_area_total),
   window_N_gain: float
      := sum(collect(s in spaces, sum(collect(w in s@walls, w@solar_gain_total where
         (w@orientation = n))))),
   window_NE_NW_gain: float
      := sum(collect(s in spaces, sum(collect(w in s@walls, w@solar_gain_total where
         (w@orientation in [ne, nw]))))),
```

```
   window_E_W_gain: float
      := sum(collect(s in spaces, sum(collect(w in s@walls, w@solar_gain_total where
         (w@orientation in [e, w]))))),
   window_SE_SW_S_gain: float
      := sum(collect(s in spaces, sum(collect(w in s@walls, w@solar_gain_total where
         (w@orientation in [se, sw, s]))))),
   slab_heat_loss: float
      := sum(collect(s in spaces, sum(collect(f in s@floor, f@heat_loss where
            (f@construction_type = td_suspended_floor))))),
   suspended_heat_loss: float
      := sum(collect(s in spaces, sum(collect(f in s@floor, f@heat_loss where
            (f@construction_type = td_concrete_floor))))),
   wall_N_heat_loss: float
      := sum(collect(s in spaces, sum(collect(w in s@walls, w@heat_loss where
         (w@orientation = n))))),
   wall_NE_NW_heat_loss: float
      := sum(collect(s in spaces, sum(collect(w in s@walls, w@heat_loss where
         (w@orientation in [ne, nw]))))),
   wall_E_W_heat_loss: float
      := sum(collect(s in spaces, sum(collect(w in s@walls, w@heat_loss where
         (w@orientation in [e, w]))))),
   wall_SE_SW_S_heat_loss: float
      := sum(collect(s in spaces, sum(collect(w in s@walls, w@heat_loss where
         (w@orientation in [se, sw, s]))))),
   roof_heat_loss: float
      := sum(collect(s in spaces, s@roof_heat_loss)),
   window_heat_loss : float
      := sum(collect(s in spaces, sum(collect(w in s@walls, w@solar_loss_total))))),
  demons(
    cw_set_item('Thermal Designer',bpiText,number_atom(bpi))
)).

air_leak(a,15.00000).  air_leak(b,13.33333).
air_leak(c,11.81818).  air_leak(d,10.43478).

air_leak_rate(exposed,0.09091).           air_leak_rate(medium_exposed,0.07857).
air_leak_rate(medium_sheltered,0.06666).  air_leak_rate(sheltered,0.05714).

useful_fract(ETM,GLR,V) :- ETM =< 0.15, uf(1.07870, 0.42465, 0.0518650, 0.0043706, GLR, V).
useful_fract(ETM,GLR,V) :- ETM =< 0.45, uf(1.10220, 0.38544, 0.0075756, 0.0147310, GLR, V).
useful_fract(ETM,GLR,V) :- ETM =< 0.80, uf(1.11800, 0.36126,-0.0059524, 0.0113678, GLR, V).
useful_fract(ETM,GLR,V) :- ETM =< 1.25, uf(1.06130, 0.12136,-0.2161800, 0.0673400, GLR, V).
useful_fract(ETM,GLR,V) :- ETM =< 1.75, uf(0.97333, 0.23106,-0.5170500, 0.1420500, GLR, V).
useful_fract(ETM,GLR,V) :- ETM  > 1.75, uf(0.95071,-0.33541,-0.5955100, 0.1578300, GLR, V).

uf(A,B,C,D,GLR, V) :- V is A - B * GLR + C * GLR * GLR + D * GLR * GLR * GLR.

td_building::create :-
  set_prop(td,building,self),
  td_dialog.  % Set up the top-level TD dialog
```

E.3 Mappings for the Large Example

Listed below are the VML specifications for the mappings between the IDM and PlanEntry, FaceEditor, VISION-3D, and ThermalDesigner.

E.3.1 IDM <-> PlanEntry mapping

Only a small range of classes need to be mapped between the IDM and PlanEntry as shown below. When first creating a mapping to PlanEntry it is important that a pe_proto_plan object gets created to manage the running of PlanEntry. Building objects map straight across, though there is a mismatch between the objects in sets for both of those classes requiring a bijection to split out the right objects. The mapping of planes utilises method mappings to ensure that object select methods in PlanEntry get mapped through to connected applications (e.g., the FaceEditor to allow specification of materials for the selected face). All mappings with geometry flip the y-axis between the two systems as a different axis setup is utilised in both systems.

```
inter_view(idm, integrated, planentry, read_write, complete).

inter_class([],  [pe_proto_plan]
).

inter_class([idm_building], [pe_building],
   equivalences(
      bijection(idm_building.spaces[]@class('idm_space'), spaces=>list[]),
      bijection(idm_building.spaces[]@class('idm_roof'), roofs=>list[]),
      bijection(idm_building.face_views[]@class('idm_space_face'), faces[])
   ),
   initialisers(
      name = 'PlanEntry building',
      pe_building.plan=>building = pe_building
   )
).

inter_class([idm_plane], [pf_plane_object],
   equivalences(
      name = planename,
      axis = axis,
      offset = offset,
      @view_plane = @select
   )
).

inter_class([idm_space], [pe_space],
   equivalences(
      min=>x = x,
      min=>y = 0 - y,
      min=>z = z,
      max=>x = x1,
      max=>y = 0 - y1,
      max=>z = z1,
      bijection(abutting[], faces=>list[]=>spaces[] \= pe_space) % Only works from PE to IDM
   )
).

inter_class([idm_hip_roof], [pe_roof],
   equivalences(
      min=>x = x,
      min=>y = 0 - y,
      min=>z = z,
      max=>x = x1,
      max=>y = 0 - y1,
      max=>z = z,
      apex1=>x = p1x,
      apex1=>y = 0 - p1y,
      apex1=>z = z1,
      apex2=>x = p2x,
```

```
      apex2=>y = 0 - p2y,
      apex2=>z = z1,
      bijection(abutting[], faces=>list[]=>spaces[] \= pe_roof) % Only works from PE to IDM
   )
).

inter_class([idm_space_face], [pe_face, pf_plane_object, group(pe_opening)],
   invariants(
      type_of_face \= 'opening',
      member(pe_face.orientation, ['up', 'down']),
      pe_face.offset = pf_plane_object.offset,
      map_orientation_axis(pe_face.orientation, pf_plane_object.axis),
      contained_in_face(pe_face, pe_opening)
   ),
   equivalences(
      min=>x = pe_face.x,
      min=>y = 0 - pe_face.y,
      max=>x = pe_face.x1,
      max=>y = 0 - pe_face.y1,
      plane=>name = pf_plane_object.planename,
      map_orientation_axis(pe_face.orientation, idm_space_face.plane=>axis),
      plane=>offset = pe_face.offset,
      location = internal,
      orientation = orientation,
      map_face_type_from_orientation(type_of_face, pe_face.orientation, pe_face, spaces),
      openings = pe_opening
   )
).

inter_class([idm_space_face], [pe_face, pf_plane_object, group(pe_opening)],
   invariants(
      type_of_face \= 'opening',
      member(pe_face.orientation, ['n', 's']),
      pe_face.offset = pf_plane_object.offset,
      map_orientation_axis(pe_face.orientation, pf_plane_object.axis),
      contained_in_face(pe_face, pe_opening)
   ),
   equivalences(
      min=>x = pe_face.x,
      min=>y = pe_face.y,
      max=>x = pe_face.x1,
      max=>y = pe_face.y1,
      plane=>name = pf_plane_object.planename,
      map_orientation_axis(pe_face.orientation, idm_space_face.plane=>axis),
      plane=>offset = 0 - pe_face.offset,
      location = internal,
      orientation = orientation,
      map_face_type_from_orientation(type_of_face, pe_face.orientation, pe_face, spaces),
      openings = pe_opening
   )
).

inter_class([idm_space_face], [pe_face, pf_plane_object, group(pe_opening)],
   invariants(
      type_of_face \= 'opening',
      member(pe_face.orientation, ['e', 'w']),
      pe_face.offset = pf_plane_object.offset,
      map_orientation_axis(pe_face.orientation, pf_plane_object.axis),
      contained_in_face(pe_face, pe_opening)
   ),
   equivalences(
      min=>x = 0 - pe_face.x,
      min=>y = pe_face.y,
      max=>x = 0 - pe_face.x1,
      max=>y = pe_face.y1,
```

```
      plane=>name = pf_plane_object.planename,
      map_orientation_axis(pe_face.orientation, idm_space_face.plane=>axis),
      plane=>offset = pe_face.offset,
      location = internal,
      orientation = orientation,
      map_face_type_from_orientation(type_of_face, pe_face.orientation, pe_face, spaces),
      openings = pe_opening
   )
).

inter_class([idm_space_face], [pe_wall],
   invariants(
      location = 'int',
      type_of_face = 'wall'
   ),
   equivalences(
      map_to_from(map_polar_to_3D_rect(min, max, plane, pe_wall),
                  map_3D_rect_to_polar(x, y, z, x1, y1, z1, min, max, plane)),
      openings = openings=>list
   )
).

inter_class([idm_space_face], [pe_opening],
   invariants(
      type_of_face = 'opening'
   ),
   equivalences(
      map_to_from(map_polar_to_3D_rect(min, max, plane, pe_opening),
                  map_3D_rect_to_polar(x, y, z, x1, y1, z1, min, max, plane))
   )
).

% Auxiliary functions
map_face_type_from_orientation(wall, Orientation, _, _) :-
   member(Orientation, [n, s, e, w]).
map_face_type_from_orientation(floor, down, FromOID, [FromOID]).
map_face_type_from_orientation(inter_floor, Orientation, _, _) :-
   member(Orientation, [up, down]).

map_orientation_axis(n, y).
map_orientation_axis(s, y).
map_orientation_axis(e, x).
map_orientation_axis(w, x).
map_orientation_axis(up, z).
map_orientation_axis(down, z).

map_axis_orientation(y, Orient) :-
   member(Orient, [n, s]).
map_axis_orientation(x, Orient) :-
   member(Orient, [e, w]).
map_axis_orientation(z, Orient) :-
   member(Orient, [up, down]).

map_3D_rect_to_polar(X, Y, Z, X1, Y1, Z1, Min, Max, Plane) :-
   FY is 0 - Y,
   FY1 is 0 - Y1,
   (Z = Z1 ->
      Plane@axis := z,
      Plane@offset := Z,
  concat('Plane z-', Z, PlaneName),
      Plane@name := PlaneName,
      Min@x := X,
      Min@y := FY,
      Max@x := X1,
      Max@y := FY1
```

```
   ;(Y = Y1 ->
         Plane@axis := y,
         Plane@offset := FY,
     concat('Plane y-', FY, PlaneName),
         Plane@name := PlaneName,
         Min@x := X,
         Min@y := Z,
         Max@x := X1,
         Max@y := Z1
      ;Plane@axis := x,
         Plane@offset := X,
     concat('Plane x-', X, PlaneName),
         Plane@name := PlaneName,
         Min@x := FY,
         Min@y := Z,
         Max@x := FY1,
         Max@y := Z1
      )
   ).

map_polar_to_3D_rect(Min, Max, Plane, Rect) :-
   Plane@axis(Axis),
   Plane@offset(Offset),
   Min@x(MinX),
   Min@y(MinY),
   Max@x(MaxX),
   Max@y(MaxY),
   (Axis = z ->
      Rect@x := MinX,
      FMinY is 0 - MinY,
      Rect@y := FMinY,
      Rect@z := Offset,
      Rect@x1 := MaxX,
      FMaxY is 0 - MaxY,
      Rect@y1 := FMaxY,
      Rect@z1 := Offset
   ;(Axis = y ->
         FOffset is 0 - Offset,
         Rect@x := MinX,
         Rect@y := FOffset,
         Rect@z := MinY,
         Rect@x1 := MaxX,
         Rect@y1 := FOffset,
         Rect@z1 := MaxY
      ;Rect@x := Offset,
         FMinX is 0 - MinX,
         Rect@y := FMinX,
         Rect@z := MinY,
         Rect@x1 := Offset,
         FMaxX is 0 - MaxX,
         Rect@y1 := FMaxX,
         Rect@z1 := MaxY
      )
   ).

contained_in_face(Face, Opening) :-
   Face@orientation(Orient),
   Opening@axis(Axis),
   map_orientation_axis(Orient, Axis),
   Face@offset(Radius),
   Face@x(BX),
   Face@y(BY),
   Face@x1(BX1),
   Face@y1(BY1),
   Opening@x(X),
```

```
   Opening@y(Y),
   Opening@z(Z),
   Opening@x1(X1),
   Opening@y1(Y1),
   Opening@z1(Z1),
   ((Axis = z, Z = Z1, Z = Radius) ->
      contained_in_rect(X, Y, X1, Y1, BX, BY, BX1, BY1)
   ;((Axis = y, Y = Y1, Y = Radius) ->
         contained_in_rect(X, Z, X1, Z1, BX, BY, BX1, BY1)
      ;Axis = x,
         X = X1,
         X = Radius,
         contained_in_rect(Y, Z, Y1, Z1, BX, BY, BX1, BY1)
      )
   ).

contained_in_rect(X, Y, X1, Y1, BX, BY, BX1, BY1) :-
   value_between(X, BX, BX1),
   value_between(Y, BY, BY1),
   value_between(X1, BX, BX1),
   value_between(Y1, BY, BY1).

value_between(Val, P1, P2) :-
   (P1 < P2 ->
      P1 =< Val,
      Val =< P2
   ;P2 =< Val,
      Val =< P1
   ).
```

E.3.2 IDM <-> FaceEditor mapping

The mapping between the IDM and FaceEditor is more complicated than that for PlanEntry as more work is required in identifying the right objects for a mapping. Every plane has a one-to-one correspondence, but there is a requirement for data from associated objects in the model when mapping. FaceEditor has complicated create methods for most of its objects, so requires create methods to be specified in the initialisers of most mappings. Again the y-axis is measured differently from the IDM and must be swapped around during mapping of geometry.

```
inter_view(idm, integrated, faceeditor, read_write, complete).

inter_class([idm_building], [fe_application]).

inter_class([idm_plane, idm_building, group(idm_space_face), group(idm_material_face),
group(idm_bracing_face)], [fe_face_window, fe_application],
   invariants(
      plane_equivalence(idm_plane, idm_space_face.plane),
      plane_equivalence(idm_plane, idm_material_face.plane),
      plane_equivalence(idm_plane, idm_bracing_face.plane)
   ),
   equivalences(
      bijection(idm_space_face[].type_of_face \= 'opening', walls[]),
      bijection(idm_space_face[].type_of_face = 'opening', openings[]),
      idm_material_face = materials,
      idm_bracing_face = bracing,
      idm_plane.name = name,
      idm_plane@view_plane = fe_application@create_view(_, idm_plane.name)
   ),
   initialisers(
      fe_face_window@create(idm_building, idm_plane.name, idm_plane.axis, 0, '+', [])
   )
).
```

```
inter_class([idm_space_face], [fe_face, fe_face_window],
   invariants(
      type_of_face \= 'opening',
      member(fe_face, fe_face_window.walls)
   ),
   equivalences(
      min=>x = x0,
      min=>y = 0 - y0,
      max=>x = x1,
      max=>y = 0 - y1
   ),
   initialisers(
      face_property = 'idm_space_face',
      fe_face@create(idm_space_face.plane, idm_space_face.plane, 'space', 0, 0,
idm_space_face.min=>x, 0 - idm_space_face.min=>y,
          idm_space_face.max=>x, 0 - idm_space_face.max=>y)
   )
).

inter_class([idm_space_face, idm_material_face], [fe_opening, fe_face_window],
   invariants(
      idm_space_face.type_of_face = 'opening',
      idm_material_face.type_of_face = 'opening',
      point_equivalence(idm_space_face.min, idm_material_face.min),
      point_equivalence(idm_space_face.max, idm_material_face.max),
      plane_equivalence(idm_space_face.plane, idm_material_face.plane),
      member(fe_opening, fe_face_window.openings)
   ),
   equivalences(
      idm_space_face.min=>x = x0,
      idm_space_face.min=>y = 0 - y0,
      idm_space_face.max=>x = x1,
      idm_space_face.max=>y = 0 - y1,
      idm_space_face.plane = fe_face_window,
      idm_material_face.min=>x = x0,
      idm_material_face.min=>y = 0 - y0,
      idm_material_face.max=>x = x1,
      idm_material_face.max=>y = 0 - y1,
      idm_material_face.material=>r_value = material=>r_value,
      idm_material_face.material=>shading_coefficient = material=>shading_coef,
      idm_material_face.material=>construction_name = material=>name
   ),
   initialisers(
      idm_space_face.face_property = 'idm_space_face',
      idm_material_face.face_property = 'idm_material_face',
      idm_material_face.material=>type_of_material = 'idm_window_material',
      fe_opening@create(idm_space_face.plane, idm_space_face.plane, 'space', 0, 0,
idm_space_face.min=>x, 0 - idm_space_face.min=>y,
          idm_space_face.max=>x, 0 - idm_space_face.max=>y,
idm_material_face.material=>construction_name)
   )
).

inter_class([idm_material_face], [fe_face_material, fe_face_window],
   invariants(
      type_of_face = 'wall',
      kind = 'wall',
      member(fe_face_material, fe_face_window.materials)
   ),
   equivalences(
      min=>x = x0,
      min=>y = 0 - y0,
      max=>x = x1,
      max=>y = 0 - y1,
      plane = fe_face_window,
```

```
      material=>r_value = material=>r_value,
      material=>colour = material=>colour,
      material=>construction_name = fe_face_material.material=>name
   ),
   initialisers(
      face_property = 'idm_material_face',
      idm_material_face.material=>type_of_material = 'idm_wall_material',
      idm_material_face.material=>colour = 'dark',
      fe_face_material@create(idm_material_face.plane, idm_material_face.plane, 'wall', 0, 0,
idm_material_face.min=>x, 0 - idm_material_face.min=>y,
            idm_material_face.max=>x, 0 - idm_material_face.max=>y,
idm_material_face.material=>construction_name)
   )
).

inter_class([idm_material_face], [fe_face_material, fe_face_window],
   invariants(
      type_of_face = 'floor',
      kind = 'floor',
      member(fe_face_material, fe_face_window.materials)
   ),
   equivalences(
      min=>x = x0,
      min=>y = 0 - y0,
      max=>x = x1,
      max=>y = 0 - y1,
      plane = fe_face_window,
      material=>r_value = material=>r_value,
      material=>construction_name = material=>name
   ),
   initialisers(
      face_property = 'idm_material_face',
      material=>type_of_material = 'idm_floor_material',
    material=>type_of_floor = 'idm_suspended_floor',
      fe_face_material@create(idm_material_face.plane, idm_material_face.plane, 'floor', 0,
0, idm_material_face.min=>x, 0 - idm_material_face.min=>y,
            idm_material_face.max=>x, 0 - idm_material_face.max=>y,
idm_material_face.material=>construction_name)
   )).

inter_class([idm_material_face], [fe_face_material, fe_face_window],
   invariants(
      type_of_face = 'inter_floor',
      kind = 'inter_floor',
      member(fe_face_material, fe_face_window.materials)
   ),
   equivalences(
      min=>x = x0,
      min=>y = 0 - y0,
      max=>x = x1,
      max=>y = 0 - y1,
      plane = fe_face_window,
      material=>r_value = material=>r_value,
      material=>construction_name = material=>name
   ),
   initialisers(
      face_property = 'idm_material_face',
      material=>type_of_material = 'idm_inter_floor_material',
      material=>type_of_inter_floor = 'idm_inter_floor1_material',
      fe_face_material@create(idm_material_face.plane, idm_material_face.plane,
'inter_floor', 0, 0, idm_material_face.min=>x, 0 - idm_material_face.min=>y,
            idm_material_face.max=>x, 0 - idm_material_face.max=>y,
idm_material_face.material=>construction_name)
   )
).
```

```
inter_class([idm_bracing_face], [fe_bracing, fe_face_window],
   invariants(
      member(fe_bracing, fe_face_window.bracing)
   ),
   equivalences(
      min=>x = x0,
      min=>y = 0 - y0,
      max=>x = x1,
      max=>y = 0 - y1,
      plane = fe_face_window
   ),
   initialisers(
      face_property = 'idm_bracing_face',
      fe_bracing@create(idm_bracing_face.plane, idm_bracing_face.plane, 'bracing', 0, 0,
idm_bracing_face.min=>x, 0 - idm_bracing_face.min=>y,
            idm_bracing_face.max=>x, 0 - idm_bracing_face.max=>y)
   )
).
```

Auxiliary functions

```
point_equivalence(Point, Point).
point_equivalence(Point1, Point2) :-
   Point1@x(X),
   Point2@x(X),
   Point1@y(Y),
   Point2@y(Y),
   !.

plane_equivalence(Plane, Plane).
plane_equivalence(Plane1, Plane2) :-
   Plane1@axis(A),
   Plane2@axis(A),
   Plane1@offset(O),
   Plane2@offset(O),
   !.
```

E.3.3 IDM <-> VISION-3D mapping

The mapping to VISION-3D is only maintained in a single direction as VISION-3D is utilised purely as a viewer and navigation system. Only geometry needs to be mapped through to VISION-3D from the IDM objects, though default material types are set for objects with little material information in the IDM.

```
inter_view(idm, integrated, vision3d, read_only, complete).

inter_class([idm_building],[v3d_model],
   equivalences(
      name = model_name
   )
).

inter_class([idm_space_face],[v3d_polygon],
   invariants(
      type_of_face = 'opening'
   ),
   equivalences(
      map_id_to_num(idm_space_face, object_id),
      map_polar_rect_to_polygon(min=>x, min=>y, max=>x, max=>y, plane=>axis, plane=>offset,
points[1], points[2], points[3], points[4])
   ),
   initialisers(
      diffuse_reflection = 0.1,
      specular_reflection = 0.1,
```

```
      gloss_factor = 90.0,
      colour=>r = 0.88,
      colour=>g = 0.88,
      colour=>b = 0.88
   )
).

inter_class([idm_space_face],[v3d_polygon],
   invariants(
      type_of_face \= 'opening' % Could check for inter_floor, floor, wall to set different
material types
   ),
   equivalences(
      map_id_to_num(idm_space_face, object_id),
      map_polar_rect_to_polygon(min=>x, min=>y, max=>x, max=>y, plane=>axis, plane=>offset,
points[1], points[2], points[3], points[4])
   ),
   initialisers(
      diffuse_reflection = 0.8,
      specular_reflection = 0.5,
      gloss_factor = 2.0,
      colour=>r = 0.6,
      colour=>g = 0.6,
      colour=>b = 0.6
   )
).

inter_class([idm_hip_roof],[group(v3d_polygon)],
   invariants(
      apex1=>x = apex2=>x,
      apex1=>y = apex2=>y
   ),
   equivalences(
      map_id_to_num(idm_hip_roof, v3d_polygon[1].object_id),
      map_triangle_to_polygon(min=>x, min=>y, min=>z, min=>x, max=>y, min=>z, apex1=>x,
apex1=>y, apex1=>z,  v3d_polygon[1].points[1], v3d_polygon[1].points[2],
v3d_polygon[1].points[3]),
      map_id_to_num(idm_hip_roof, v3d_polygon[2].object_id),
      map_triangle_to_polygon(min=>x, min=>y, min=>z, max=>x, min=>y, min=>z, apex1=>x,
apex1=>y, apex1=>z,  v3d_polygon[2].points[1], v3d_polygon[2].points[2],
v3d_polygon[2].points[3]),
      map_id_to_num(idm_hip_roof, v3d_polygon[3].object_id),
      map_triangle_to_polygon(max=>x, min=>y, min=>z, max=>x, max=>y, min=>z, apex1=>x,
apex1=>y, apex1=>z,  v3d_polygon[3].points[1], v3d_polygon[3].points[2],
v3d_polygon[3].points[3]),
      map_id_to_num(idm_hip_roof, v3d_polygon[4].object_id),
      map_triangle_to_polygon(max=>x, max=>y, min=>z, min=>x, max=>y, min=>z, apex1=>x,
apex1=>y, apex1=>z,  v3d_polygon[4].points[1], v3d_polygon[4].points[2],
v3d_polygon[4].points[3])
   ),
   initialisers(
      diffuse_reflection = 0.8,
      specular_reflection = 0.5,
      gloss_factor = 2.0,
      colour=>r = 0.6,
      colour=>g = 0.3,
      colour=>b = 0.3
   )
).

inter_class([idm_hip_roof],[group(v3d_polygon)],
   invariants(
      apex1=>x \= apex2=>x,
      apex1=>y = apex2=>y
   ),
```

```
   equivalences(
      map_id_to_num(idm_hip_roof, v3d_polygon[1].object_id),
      map_triangle_to_polygon(min=>x, min=>y, min=>z, min=>x, max=>y, min=>z, apex1=>x,
apex1=>y, apex1=>z,  v3d_polygon[1].points[1], v3d_polygon[1].points[2],
v3d_polygon[1].points[3]),
      map_id_to_num(idm_hip_roof, v3d_polygon[2].object_id),
      map_quad_to_polygon(min=>x, min=>y, min=>z, max=>x, min=>y, min=>z, apex2=>x, apex2=>y,
apex2=>z, apex1=>x, apex1=>y, apex1=>z,  v3d_polygon[2].points[1], v3d_polygon[2].points[2],
v3d_polygon[2].points[3], v3d_polygon[2].points[4]),
      map_id_to_num(idm_hip_roof, v3d_polygon[3].object_id),
      map_triangle_to_polygon(max=>x, min=>y, min=>z, max=>x, max=>y, min=>z, apex2=>x,
apex2=>y, apex2=>z,  v3d_polygon[3].points[1], v3d_polygon[3].points[2],
v3d_polygon[3].points[3]),
      map_id_to_num(idm_hip_roof, v3d_polygon[4].object_id),
      map_quad_to_polygon(max=>x, max=>y, min=>z, min=>x, max=>y, min=>z, apex1=>x, apex1=>y,
apex1=>z, apex2=>x, apex2=>y, apex2=>z,  v3d_polygon[4].points[1], v3d_polygon[4].points[2],
v3d_polygon[4].points[3], v3d_polygon[4].points[4])
   ),
   initialisers(
      diffuse_reflection = 0.8,
      specular_reflection = 0.5,
      gloss_factor = 2.0,
      colour=>r = 0.6,
      colour=>g = 0.3,
      colour=>b = 0.3
   )
).

inter_class([idm_hip_roof],[group(v3d_polygon)],
   invariants(
      apex1=>x = apex2=>x,
      apex1=>y \= apex2=>y
   ),
   equivalences(
      map_id_to_num(idm_hip_roof, v3d_polygon[1].object_id),
      map_quad_to_polygon(min=>x, min=>y, min=>z, min=>x, max=>y, min=>z, apex2=>x, apex2=>y,
apex2=>z, apex1=>x, apex1=>y, apex1=>z,  v3d_polygon[1].points[1], v3d_polygon[1].points[2],
v3d_polygon[1].points[3], v3d_polygon[1].points[4]),
      map_id_to_num(idm_hip_roof, v3d_polygon[2].object_id),
      map_triangle_to_polygon(min=>x, min=>y, min=>z, max=>x, min=>y, min=>z, apex1=>x,
apex1=>y, apex1=>z,  v3d_polygon[2].points[1], v3d_polygon[2].points[2],
v3d_polygon[2].points[3]),
      map_id_to_num(idm_hip_roof, v3d_polygon[3].object_id),
      map_quad_to_polygon(max=>x, min=>y, min=>z, max=>x, max=>y, min=>z, apex2=>x, apex2=>y,
apex2=>z, apex1=>x, apex1=>y, apex1=>z,  v3d_polygon[3].points[1], v3d_polygon[3].points[2],
v3d_polygon[3].points[3], v3d_polygon[3].points[4]),
      map_id_to_num(idm_hip_roof, v3d_polygon[4].object_id),
      map_triangle_to_polygon(max=>x, max=>y, min=>z, min=>x, max=>y, min=>z, apex2=>x,
apex2=>y, apex2=>z,  v3d_polygon[4].points[1], v3d_polygon[4].points[2],
v3d_polygon[4].points[3])
   ),
   initialisers(
      diffuse_reflection = 0.8,
      specular_reflection = 0.5,
      gloss_factor = 2.0,
      colour=>r = 0.6,
      colour=>g = 0.3,
      colour=>b = 0.3
   )
).
```

Auxiliary functions

```
map_quad_to_polygon(P1x, P1y, P1z, P2x, P2y, P2z, P3x, P3y, P3z, P4x, P4y, P4z,  Points1,
Points2, Points3, Points4) :-
   Points1@x := P1x,
   Points1@y := P1y,
   Points1@z := P1z,
   Points2@x := P2x,
   Points2@y := P2y,
   Points2@z := P2z,
   Points3@x := P3x,
   Points3@y := P3y,
   Points3@z := P3z,
   Points4@x := P4x,
   Points4@y := P4y,
   Points4@z := P4z.

map_triangle_to_polygon(P1x, P1y, P1z, P2x, P2y, P2z, P3x, P3y, P3z,  Points1, Points2,
Points3) :-
   Points1@x := P1x,
   Points1@y := P1y,
   Points1@z := P1z,
   Points2@x := P2x,
   Points2@y := P2y,
   Points2@z := P2z,
   Points3@x := P3x,
   Points3@y := P3y,
   Points3@z := P3z.

map_polar_rect_to_polygon(X, Y, X1, Y1, Orientation, Radius, Points1, Points2, Points3,
Points4) :-
   ((Orientation = up ; Orientation = down) ->
    Points1@x := X,
    Points1@y := Y,
    Points1@z := Radius,
    Points2@x := X1,
    Points2@y := Y,
    Points2@z := Radius,
    Points3@x := X1,
    Points3@y := Y1,
    Points3@z := Radius,
    Points4@x := X,
    Points4@y := Y1,
    Points4@z := Radius
   ;((Orientation = w ; Orientation = e) ->
     Points1@x := Radius,
     Points1@y := X,
     Points1@z := Y,
     Points2@x := Radius,
     Points2@y := X,
     Points2@z := Y1,
     Points3@x := Radius,
     Points3@y := X1,
     Points3@z := Y1,
     Points4@x := Radius,
     Points4@y := X1,
     Points4@z := Y
    ;Points1@x := X,
     Points1@y := Radius,
     Points1@z := Y,
     Points2@x := X1,
     Points2@y := Radius,
     Points2@z := Y,
     Points3@x := X1,
     Points3@y := Radius,
     Points3@z := Y1,
```

```
     Points4@x := X,
     Points4@y := Radius,
     Points4@z := Y1
    )
   ).

map_rect_to_polygon(X, Y, Z, X1, Y1, Z1, Points1, Points2, Points3, Points4) :-
   (Z = Z1 ->
    Points1@x := X,
    Points1@y := Y,
    Points1@z := Z,
    Points2@x := X1,
    Points2@y := Y,
    Points2@z := Z,
    Points3@x := X1,
    Points3@y := Y1,
    Points3@z := Z,
    Points4@x := X,
    Points4@y := Y1,
    Points4@z := Z
   ;(Y = Y1 ->
     Points1@x := X,
     Points1@y := Y,
     Points1@z := Z,
     Points2@x := X1,
     Points2@y := Y,
     Points2@z := Z,
     Points3@x := X1,
     Points3@y := Y,
     Points3@z := Z1,
     Points4@x := X,
     Points4@y := Y,
     Points4@z := Z1
    ;Points1@x := X,
     Points1@y := Y,
     Points1@z := Z,
     Points2@x := X,
     Points2@y := Y1,
     Points2@z := Z,
     Points3@x := X,
     Points3@y := Y1,
     Points3@z := Z1,
     Points4@x := X,
     Points4@y := Y,
     Points4@z := Z1
    )
   ).

map_id_to_num(ID, Num) :-
   ID@'#space'(Space),
   name(Space, SpaceList),
   length(SpaceList, LenSpace),
   LenDivider is LenSpace + 1,
   cat(Bits, ID, [LenDivider]),
   Bits = [_, NumAtom],
   pname(Num, NumAtom).
```

E.3.4 IDM <-> ThermalDesigner mapping

The mapping between the IDM and ThermalDesigner is again fairly complicated, mostly due to the need to associate related objects together in a mapping to provide enough information for the mapping. There is also much information in ThermalDesigner which can not be found in the IDM and is asserted through as defaults in the initialisers section of the mappings. The mappings here

require complex invariants to ensure that the right plane and material definitions are associated together to enable the mapping to take place.

```
inter_view(idm, integrated, thermaldesigner, read_write, complete).

inter_class([idm_building], [td_building],
   equivalences(
      name = building_name,
      num_of_occupants = num_of_occupants,
      effective_thermal_mass = effective_thermal_mass,
      spaces = spaces,
      environment = climate
   ),
   initialisers(
      td_building.building_name = '*BUILDING NAME*',
      td_building.num_of_occupants = 4,
      td_building.effective_thermal_mass = 0.5
   )
).

inter_class([idm_environment], [td_climate],
   equivalences(
      exposure_class = exposure_class,
      infiltration_zone = infiltration_zone,
      degree_days = degree_days
   ),
   initialisers(
      td_climate.exposure_class = 'medium_exposed',
      td_climate.infiltration_zone = 'c',
      td_climate.degree_days = 369
   )).

inter_class([group(idm_space_face), group(idm_roof)], [td_space],
   invariants(
      idm_space_face.location = 'ext'
   ),
   equivalences(
      bijection(idm_space_face[].type_of_face = 'wall', walls[]),
      bijection(idm_space_face[].type_of_face = 'floor', floor[]),
      idm_roof = roof
   )).

inter_class([idm_space_face, idm_material_face, idm_building], [td_wall],
   invariants(
      idm_space_face.type_of_face = 'wall',
      idm_space_face.location = 'ext',
      idm_material_face.type_of_face = 'wall',
      plane_equivalence(idm_space_face.plane, idm_material_face.plane),
      point_equivalence(idm_space_face.min, idm_material_face.min),
      point_equivalence(idm_space_face.max, idm_material_face.max)
   ),
   equivalences(
      idm_space_face.max=>x - idm_space_face.min=>x = width,
      idm_space_face.max=>y - idm_space_face.min=>y = height,
      idm_space_face.orientation = orientation,
      bijection(idm_space_face.openings[]@class(idm_space_face), windows[]),
      idm_material_face.material=>colour = colour,
      idm_material_face.material=>r_value = r_value
   ),
   initialisers(
      td_wall.colour = 'light',
      td_wall.r_value = 1.7,
      td_wall@create(idm_building.environment=>degree_days)
   )).
```

```
inter_class([idm_space_face, idm_material_face, idm_building], [td_window],
   invariants(
      idm_space_face.type_of_face = 'opening',
      idm_space_face.location = 'ext',
      idm_material_face.type_of_face = 'opening',
      plane_equivalence(idm_space_face.plane, idm_material_face.plane),
      point_equivalence(idm_space_face.min, idm_material_face.min),
      point_equivalence(idm_space_face.max, idm_material_face.max)
   ),
   equivalences(
      idm_space_face.max=>x - idm_space_face.min=>x = width,
      idm_space_face.max=>y - idm_space_face.min=>y = height,
      idm_material_face.material=>shading_coefficient = shading_coef,
      idm_material_face.material=>r_value = r_value
   ),
   initialisers(
      td_window.shading_coef = 1.0,
      td_window.r_value = 0.23,
      td_window@create(idm_space_face.orientation, idm_building.environment=>degree_days)
   )
).

inter_class([idm_roof, idm_building], [td_roof],
   equivalences(
      idm_roof.max=>x - idm_roof.min=>x = width,
      idm_roof.max=>y - idm_roof.min=>y = length,
      idm_roof.material=>colour = colour,
      idm_roof.material=>r_value = r_value
   ),
   initialisers(
      td_roof.colour = 'light',
      td_roof.r_value = 3.6,
      td_roof@create(idm_building.environment=>degree_days)
   )
).

inter_class([idm_space_face, idm_material_face, idm_building], [td_suspended_floor],
   invariants(
      idm_space_face.type_of_face = 'floor',
      idm_space_face.location = 'ext',
      idm_material_face.type_of_face = 'floor',
      idm_material_face.material@class('idm_suspended_floor'),
      plane_equivalence(idm_space_face.plane, idm_material_face.plane),
      point_equivalence(idm_space_face.min, idm_material_face.min),
      point_equivalence(idm_space_face.max, idm_material_face.max)
   ),
   equivalences(
      idm_space_face.max=>x - idm_space_face.min=>x = width,
      idm_space_face.max=>y - idm_space_face.min=>y = length,
      idm_material_face.material=>floor_covering_r_value = floor_covering_r_value,
      idm_material_face.material=>subfloor_protection = subfloor_protection,
      idm_material_face.material=>foundation_height = foundation_height,
      idm_material_face.material=>floor_insulation_r_value = floor_insulation_r_value
   ),
   initialisers(
      td_suspended_floor.floor_covering_r_value = 0.4,
      td_suspended_floor.subfloor_protection = 'c',
      td_suspended_floor.foundation_height = 1,
      td_suspended_floor.floor_insulation_r_value = 0.3,
      td_suspended_floor@create(idm_building.environment=>degree_days)
   )
).
```

```
inter_class([idm_space_face, idm_material_face, idm_building], [td_concrete_floor],
   invariants(
      idm_space_face.type_of_face = 'floor',
      idm_space_face.location = 'ext',
      idm_material_face.type_of_face = 'floor',
      idm_material_face.material@class('idm_slab_floor'),
      plane_equivalence(idm_space_face.plane, idm_material_face.plane),
      point_equivalence(idm_space_face.min, idm_material_face.min),
      point_equivalence(idm_space_face.max, idm_material_face.max)
   ),
   equivalences(
      idm_space_face.max=>x - idm_space_face.min=>x = width,
      idm_space_face.max=>y - idm_space_face.min=>y = length,
      idm_material_face.material=>floor_covering_r_value = floor_covering_r_value,
      idm_material_face.material=>insulation_depth = insulation_depth
   ),
   initialisers(
      td_concrete_floor.floor_covering_r_value = 0.4,
      td_concrete_floor@create(idm_building.environment=>degree_days)
   )
).
```

Auxiliary functions

```
point_equivalence(Point, Point).
point_equivalence(Point1, Point2) :-
   Point1@x(X),
   Point2@x(X),
   Point1@y(Y),
   Point2@y(Y),
   !.

plane_equivalence(Plane, Plane).
plane_equivalence(Plane1, Plane2) :-
   Plane1@axis(A),
   Plane2@axis(A),
   Plane1@offset(O),
   Plane2@offset(O),
   !.
```

E.4 Project Window for the Large Example

The following set of diagrams shows a project window devised as the main example for this thesis. The diagrams are broken into two sections, those concerned with the user and function specification, followed by those concerned with flow of control specification.

E.4.1 User and function specification

Though originally there was a single diagram showing all actors, design roles, and design functions, this could not be read reproduced on a single sheet. The following diagrams are extracted views from the original highlighting the roles of individual actors.

E.4.1.1 Client

The client has two roles in this project window (see Figure E.16), the initial specification of requirements and then the final acceptance of the building design. Defining the building requirements can be performed by a single design function, that of specifying requirements. This

design function is associated with a design tool called WordPerfect, in this case there is no input to WordPerfect from the IDM, and no output from WordPerfect to the IDM, hence no schemas are specified for the design function. The role of final acceptance can be achieved through two design functions, those of viewing the plans and viewing the model (in 3D). Both of these design functions have associated design tools with input from the IDM. However, there is no output from VISION-3D, meaning that no changes can be made in that design function and passed through to the IDM.

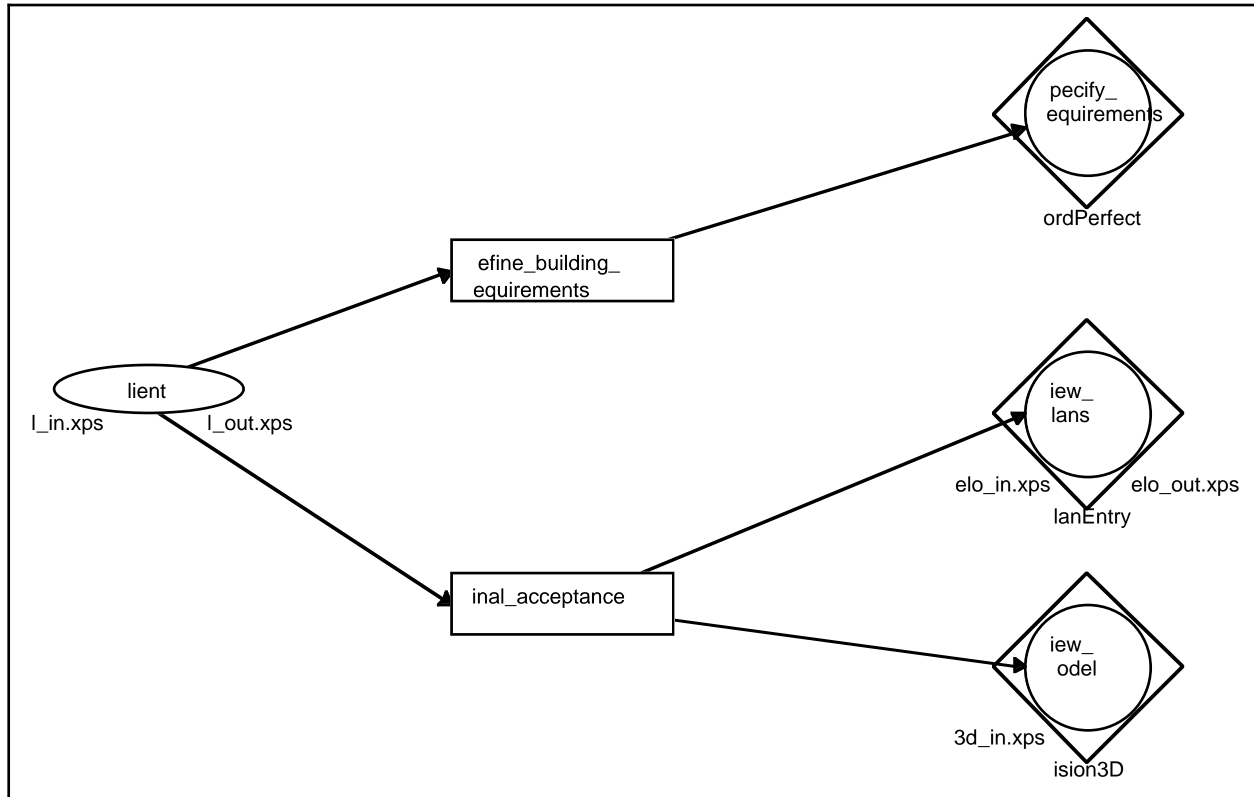


Figure E.16 Client and design roles

E.4.1.2 Architect

The architect plays three main roles in this project window (see Figure E.17), those of layout design, specifying building properties, and evaluation of the design. Each of these roles is supported by a range of design functions, including a documentation function for all roles.

E.4.1.3 Structural consultant

While the architect plays a fairly broad role in the project window (see Figure E.18), the structural consultant has very specialised design roles. These comprise the design of the structural system and ensuring the building's structural integrity.

E.4.1.4 Daylighting consultant

The daylighting consultant also plays a very specialised role in the project window (see Figure E.19). They can specify glazing for the building (note the overlap with the architect's design role of specifying glazing properties) and must ensure that there is no serious overheating in the

building due to the glazing specified.

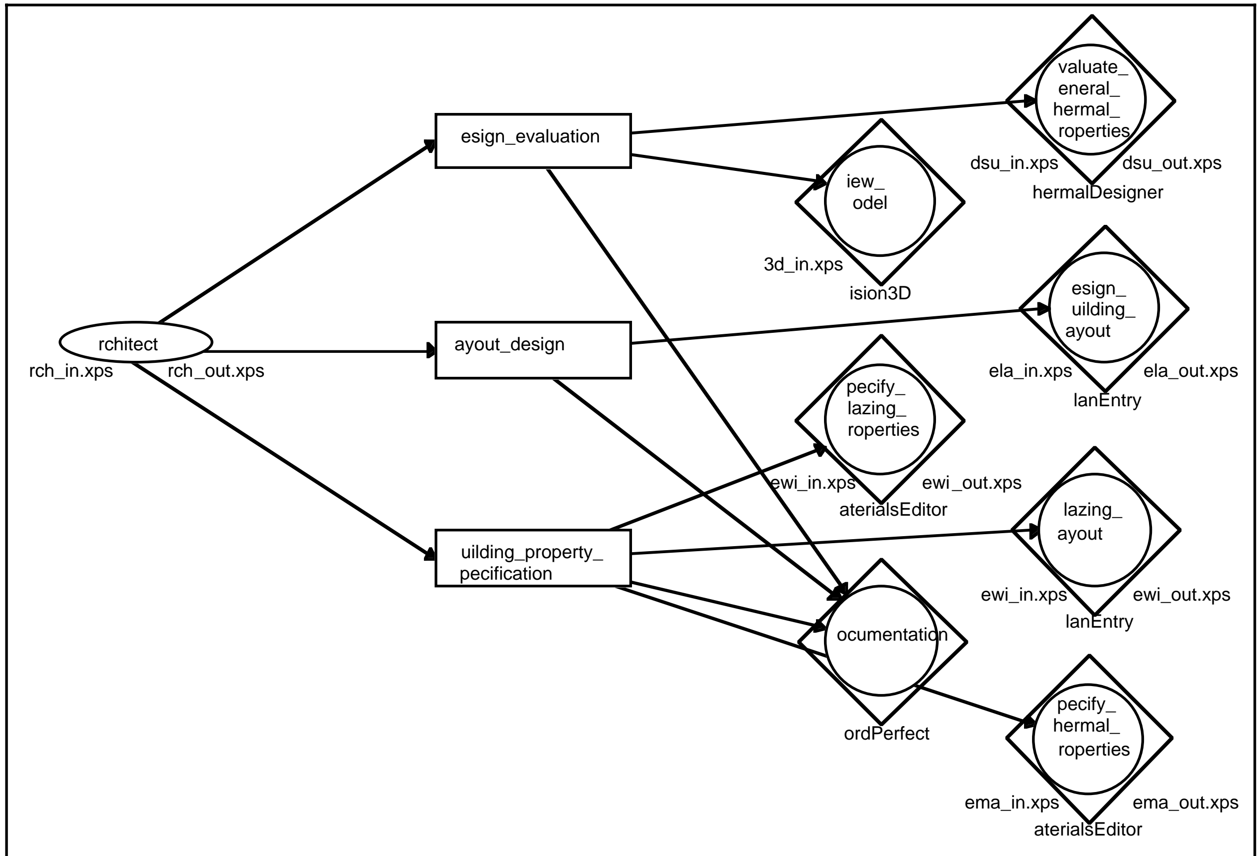


Figure E.17 Architect and design roles

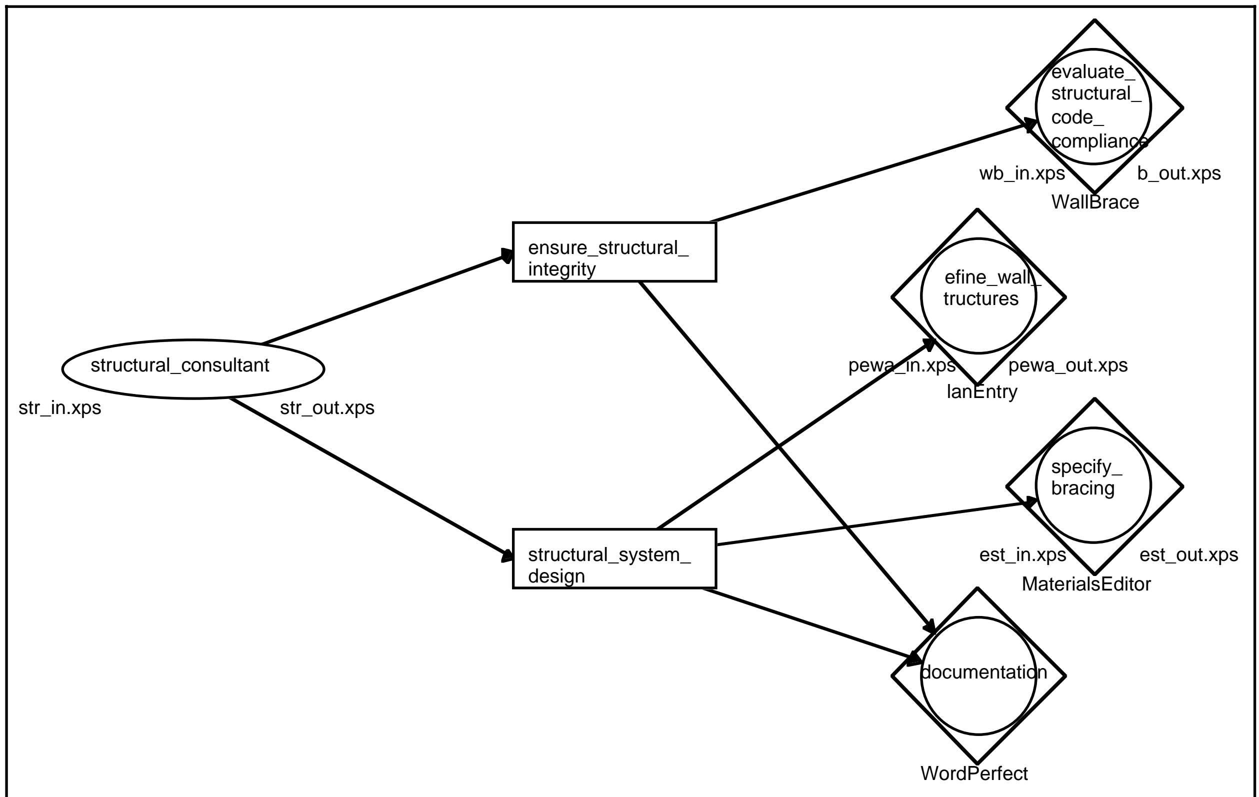


Figure E.18 Structural consultant and design roles

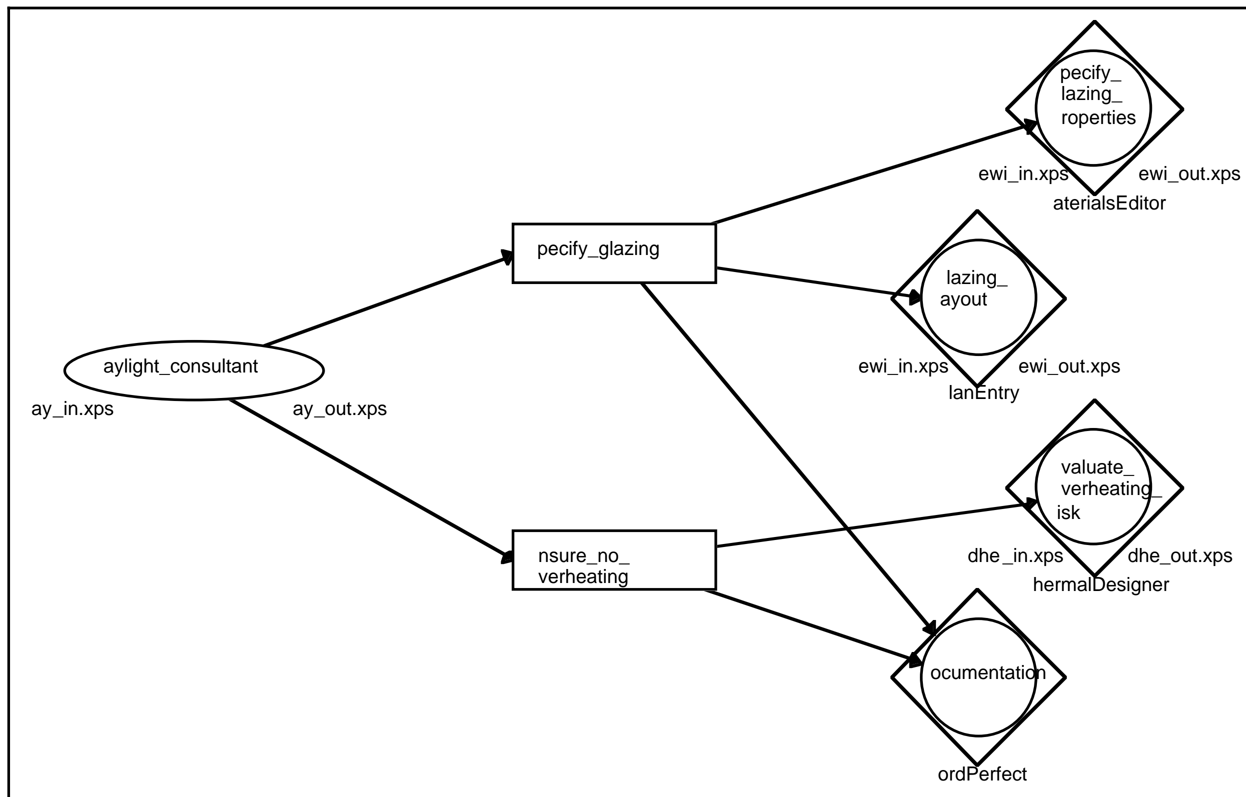


Figure E.19 Daylighting consultant and design roles

E.4.1.5 Thermal consultant

The thermal engineering consultant also has a very specialised role (see Figure E.20). They can specify materials for the building (note the overlap with the architect's design function to specify thermal properties) and must ensure that the proposed building meets the code on thermal compliance.

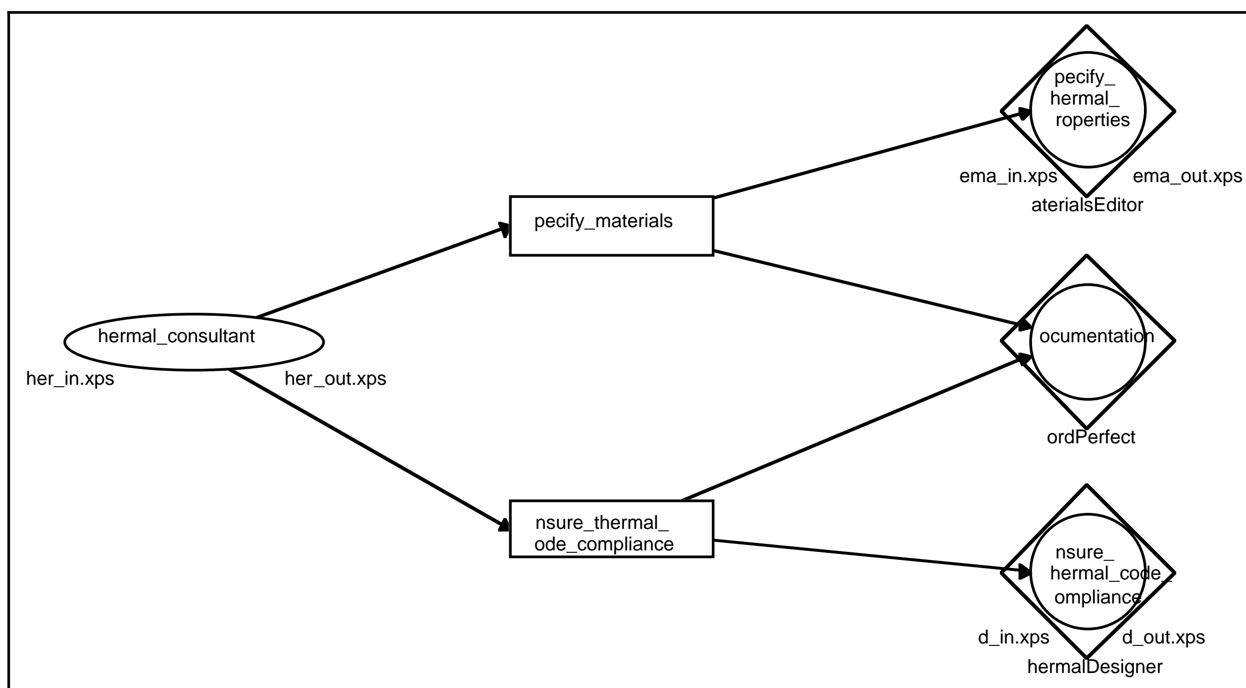


Figure E.20 Thermal consultant and design roles

E.4.2 Flow of control specification

The following set of diagrams show the flow of control for the example project window, starting from the top level CombiNet.

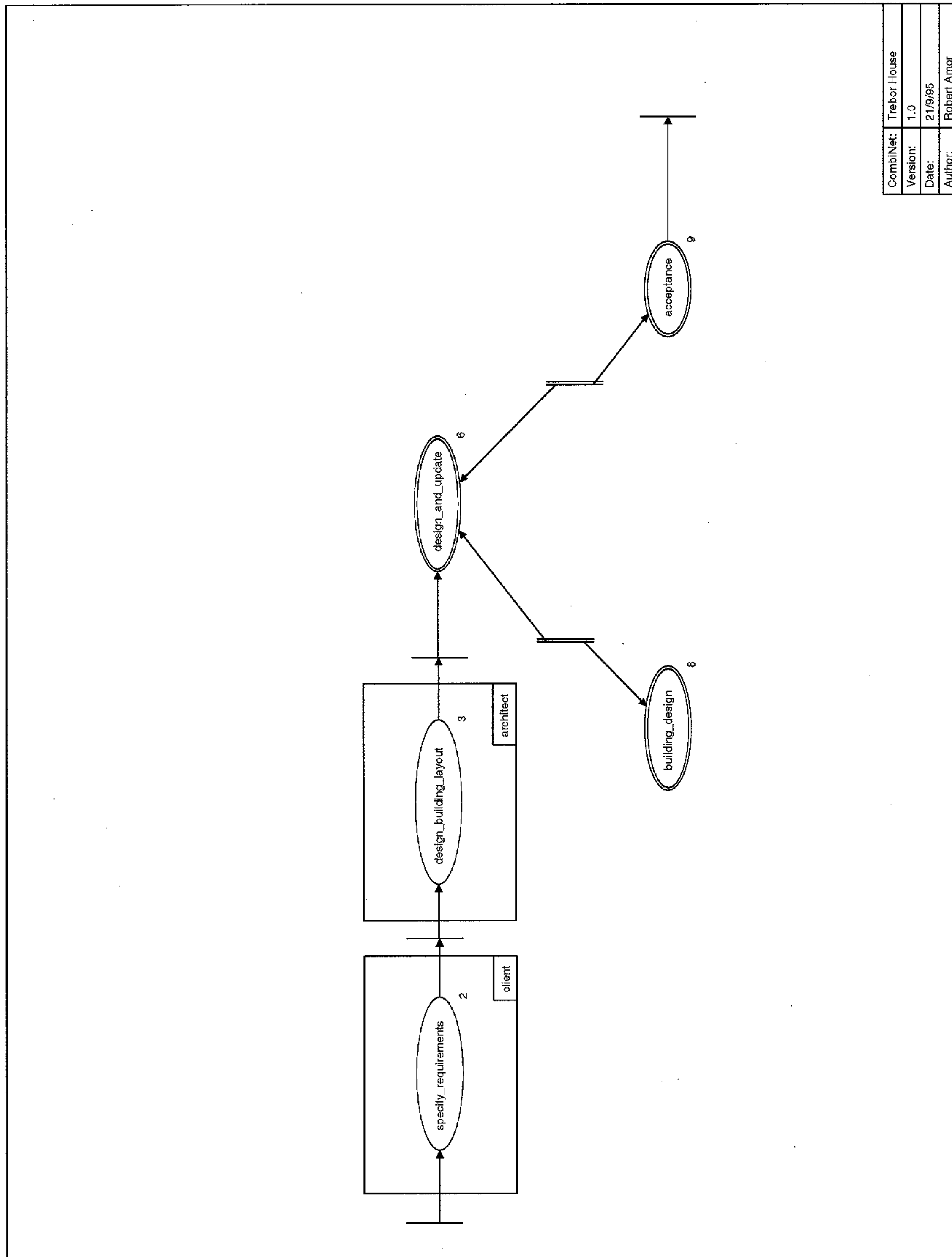


Figure E.21 Top-level CombiNet

E.4.2.1 Top-level CombiNet

The top level CombiNet (see Figure E.21) provides a high-level view of the flow in this project window. Work starts with the client specifying requirements, followed by the architect performing

an initial building design layout. Following these two sequential stages is an iterative stage of designing and updating looping with building design and acceptance CombiNets until the design is accepted and the project window terminates.

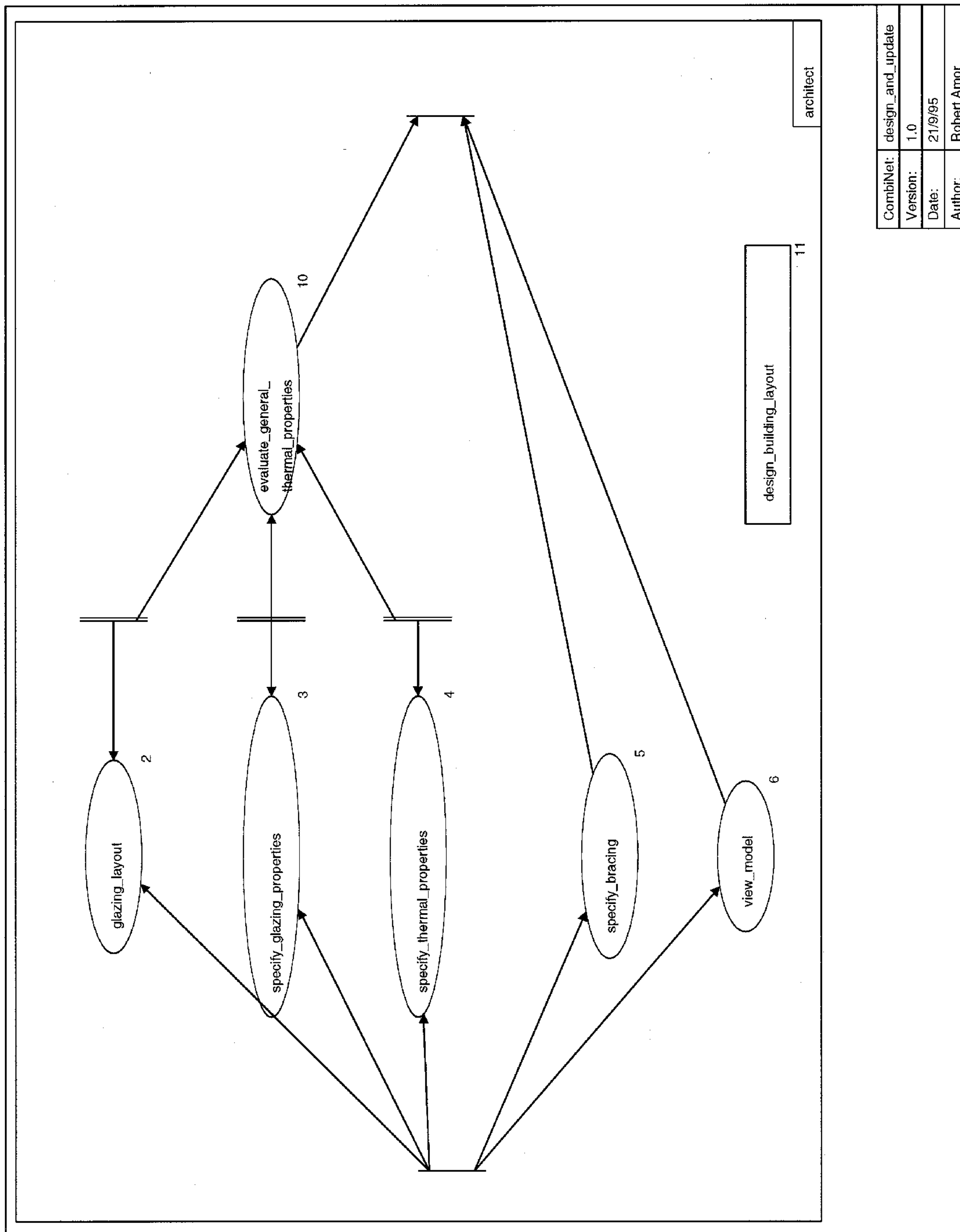


Figure E.22 Design and update CombiNet

E.4.2.2 Design and update CombiNet

The design and update CombiNet (see Figure E.22) allows the architect to specify a range of properties of the building. There is a looping between the specification of thermal properties and evaluating their impact on the building. When the architect is happy with the properties, or with the

bracing, or having navigated the model, they may exit the CombiNet. Note that designing the building layout may be entered at any stage in this CombiNet.

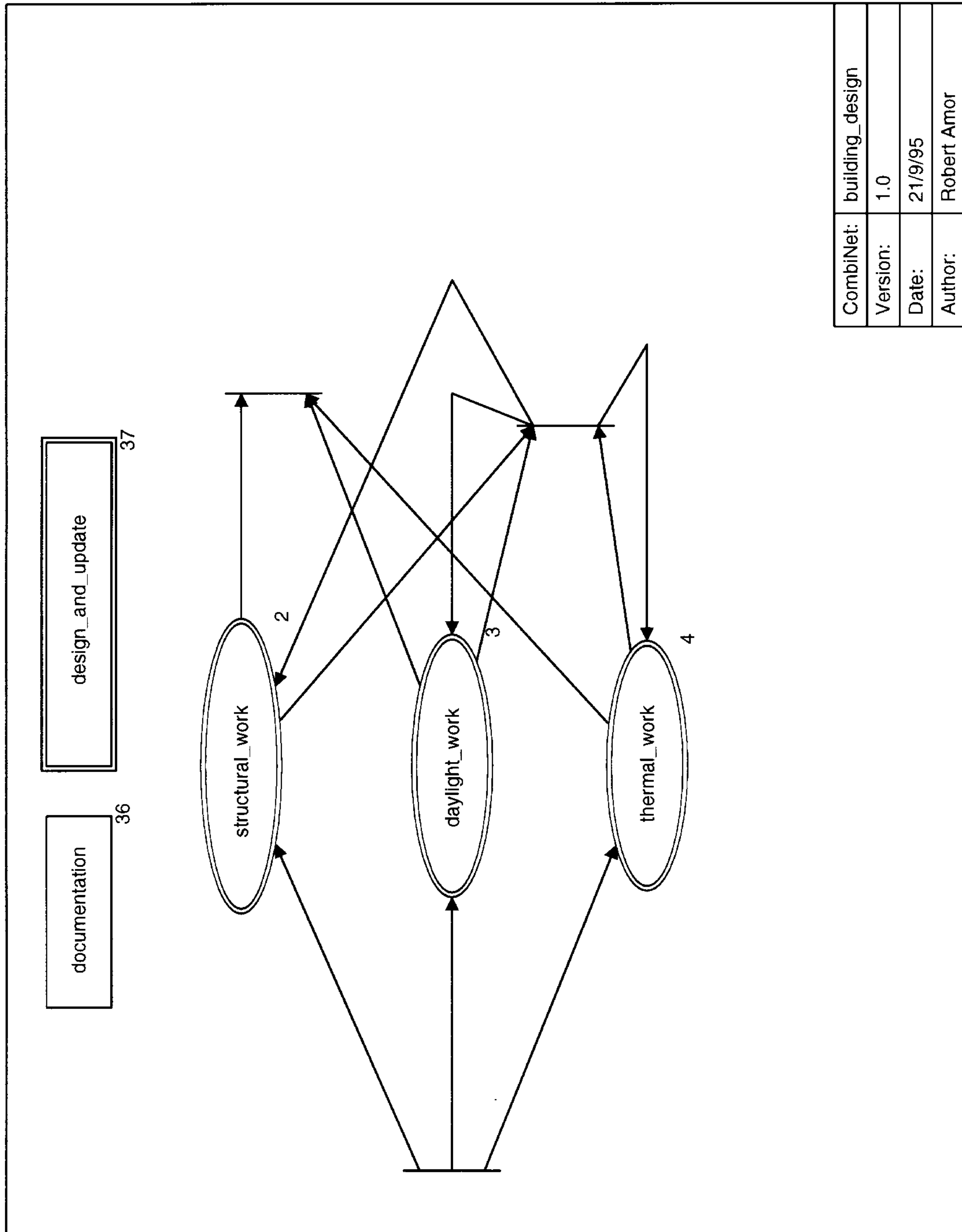


Figure E.23 Building design CombiNet

E.4.2.3 Building design CombiNet

The building design CombiNet (see Figure E.23) allows the various consultants to be drawn into the project with iterations between structural, daylighting and thermal work being supported. Documentation may be performed at any time, and the architect can be called in to perform design

and update work at any time by any of the consultants.

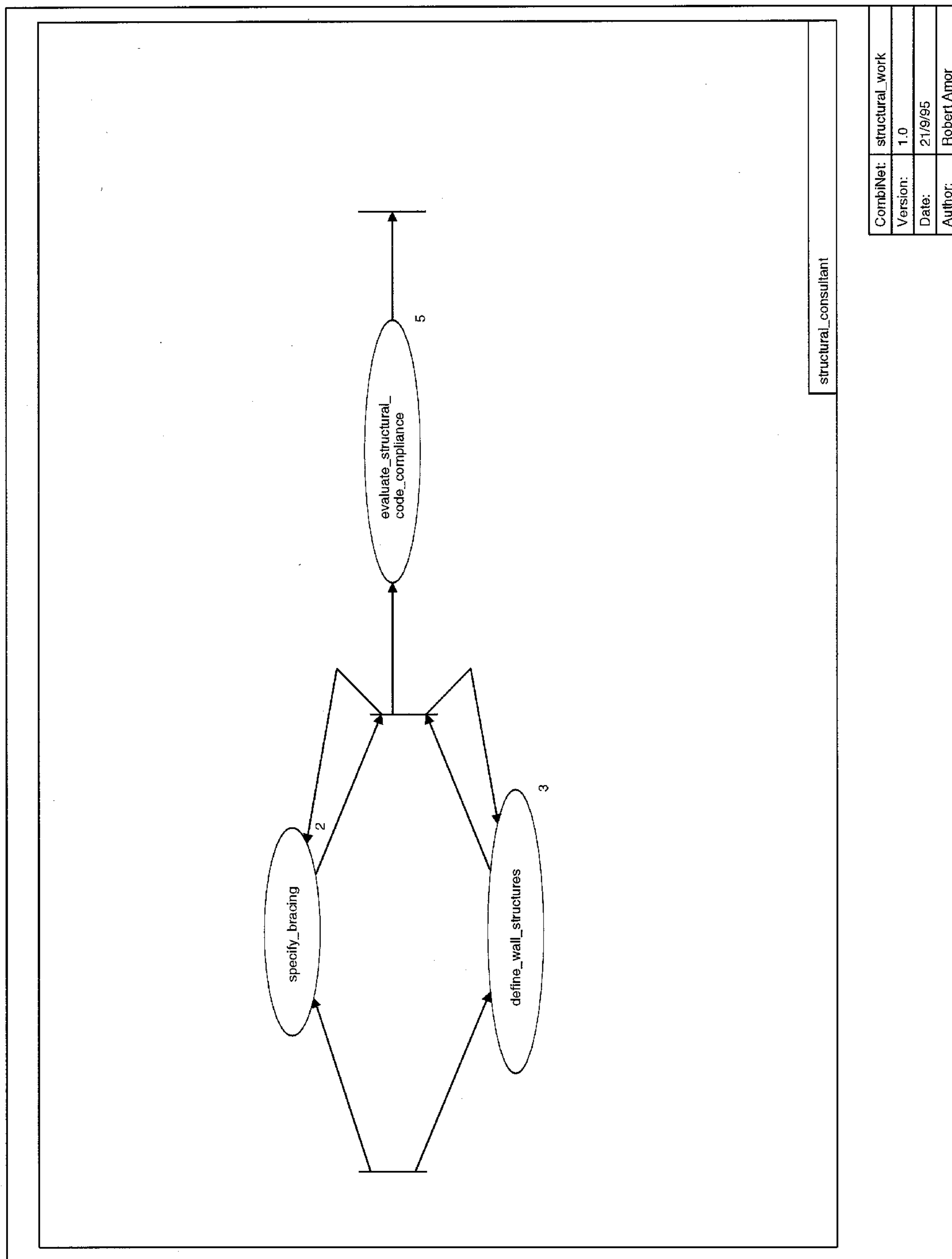


Figure E.24 Structural work CombiNet

E.4.2.4 Structural work CombiNet

This CombiNet allows the structural consultant to specify bracing and structural walls in an iterative manner until they complete their work by evaluating against structural code compliance (see Figure E.24). Note this structuring forces code compliance checks before the structural consultant can complete their task.

E.4.2.5 Daylight work CombiNet

This CombiNet allows the daylighting consultant to specify glazing in an iterative manner until they complete their work by evaluating the overheating risk of their specification (see Figure E.25). Note this structuring forces a check on overheating implications before the daylighting consultant can complete their task.

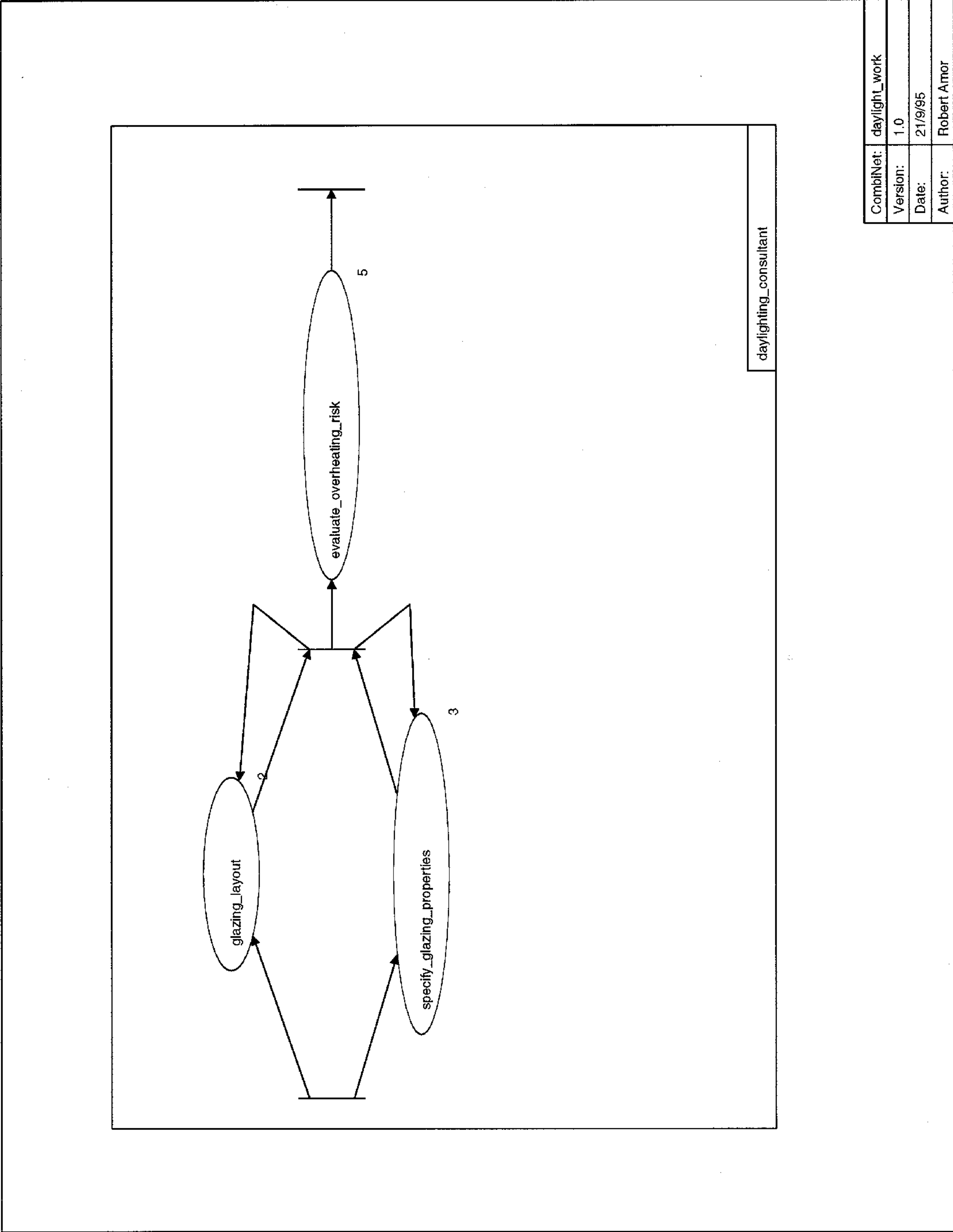


Figure E.25 Daylight work CombiNet

E.4.2.6 Thermal work CombiNet

This CombiNet allows the thermal consultant to specify thermal and glazing properties in an iterative manner until they complete their work by evaluating against thermal code compliance (see Figure E.26). Note this structuring forces code compliance checks before the thermal consultant can complete their task.

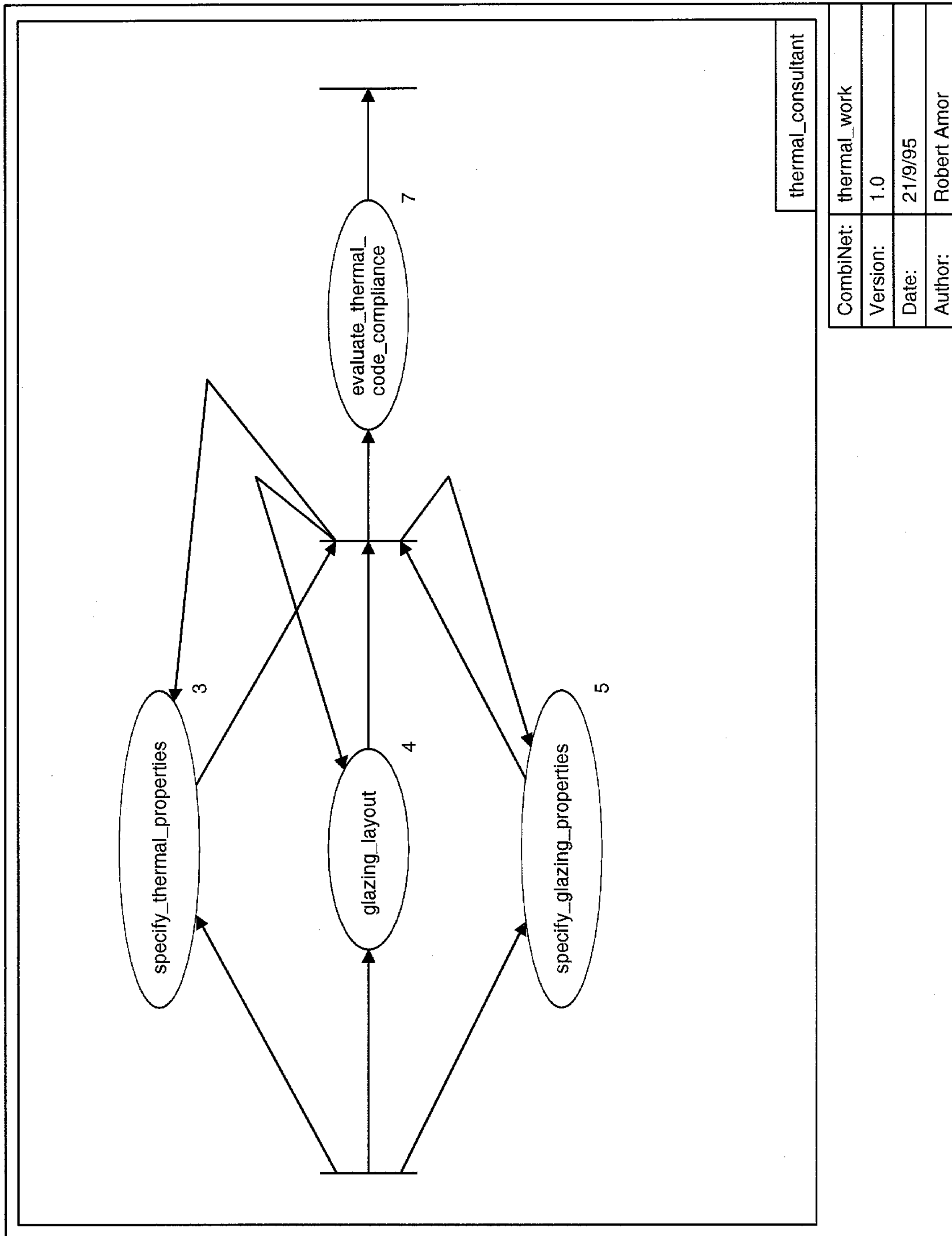


Figure E.26 Thermal work CombiNet

E.4.2.7 Acceptance CombiNet

The acceptance CombiNet (see Figure E.27) is the final work section in the project window and must be completed before the project window can complete. In this CombiNet the client reviews the plans and 3D model before giving approval, at which point the project window terminates (likely to lead to another project window for the construction phase); or rejection, in which case another round of design and update is performed by the architect.

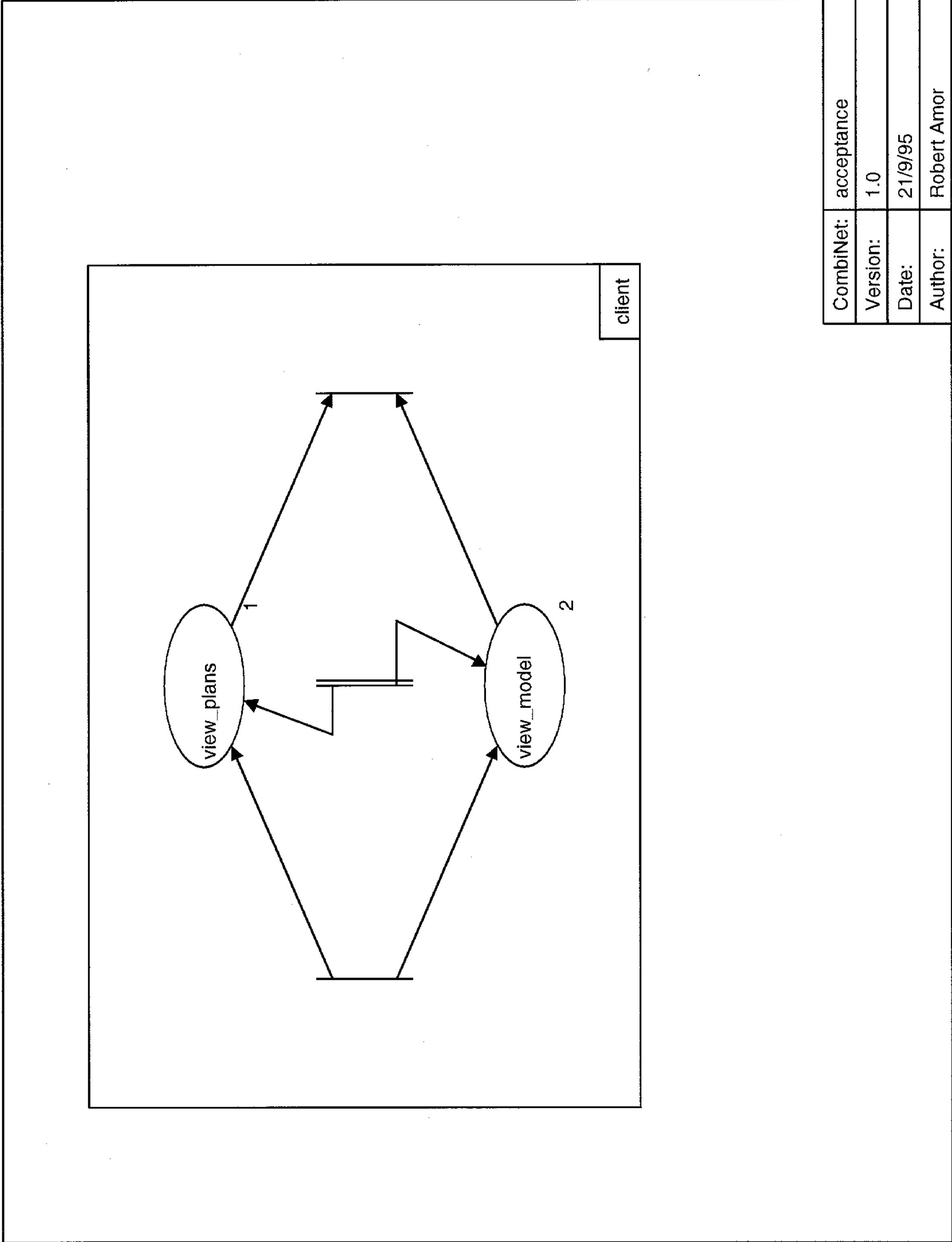


Figure E.27 Acceptance CombiNet

Appendix F

The Parsers

This appendix briefly describes the parsers and language translators which were written as part of this thesis. A range of methods were used to implement the parsers, ranging from DCGs through to a hand-coded parser.

F.1 ISO-10303:11 EXPRESS Parser

The vast majority of schemas developed for domains in the construction industry are written in the EXPRESS modelling language (ISO/TC184 1992) which is part of the ISO-STEP standards. To be able to utilise models developed in the industry, and to remain compatible with efforts underway, it was felt necessary to be able to accept schema definitions in this language. The EXPRESS standard document contains a BNF grammar (Wirth 1977) for the language. The standard is available in electronic format and the grammar could be extracted from this document without rekeying. Starting with a BNF grammar, it is an easy task to create a DCG form, which returns a parse tree of the schema being handled. This is what was implemented for this parser. The parser reads the whole schema into memory (for ease of programming, though a buffered approach would be more capable for very large schemas) and then passes it to the DCG grammar to return a parse tree for the schema. The main predicates developed in the EXPRESS parser are presented below:

exp_schema_block(-Schema, +Chars, [])

This is the top level DCG predicate which takes a set of characters (*Chars*) and returns a parse tree in *Schema*. The *[]* denote that the whole set of characters must be used when parsing. As many existing schemas were found to have incorrect syntax, the DCG provides status messages in the standard output window indicating which entity, type definition, procedure, etc it is attempting to

parse. This aids in identifying the point in a schema where a syntax error has been encountered.

express2parse(-Parsed)
This top level predicate prompts the user for the file(s) containing the EXPRESS schema to be parsed, and reads the file before passing it through to the *exp_schema_block* predicate to return the parse tree in *Parsed*.

express_checker
To check an EXPRESS schema's syntax prior to use, this predicate loads in the file(s) and ensures that it can be processed by the DCG.

An example of the syntax tree returned for a simple schema definition is shown below:

An EXPRESS schema
```
SCHEMA idm;

ENTITY wall;
      name : STRING;
      height : REAL;
      width : REAL;
      azimuth : REAL;
      position : LIST [3:3] OF REAL;
      UNIQUE name;
END_ENTITY;
END_SCHEMA;
```

Parsed representation of an EXPRESS schema
```
Parsed = schema_block(idm,
 schema_body([], nil,
  [
  entity_block(
   entity_head(wall, subsuper(nil, nil)),
    entity_body(
     [
     explicit_attribute([name], nil, simple_type('STRING'(nil))),
     explicit_attribute([height], nil, simple_type('REAL'(nil))),
     explicit_attribute([width], nil, simple_type('REAL'(nil))),
     explicit_attribute([azimuth], nil, simple_type('REAL'(nil))),
     explicit_attribute([position], nil, list_type( bound_specification(
      simple_expression([term([factor(simple_factor(literal(3)), nil)])]),
      simple_expression([term([factor(simple_factor(literal(3)), nil)])])),
      nil, simple_type('REAL'(nil))))
     ],
     [],
     [],
     unique_clause('UNIQUE', [labelled_attribute_list(nil,
      [referenced_attribute(name)])]),
     nil
    )
   )
  ]
 )
)
```

Initial tests with this parser produced many failed attempts at parsing existing schemas which caused some concern. However, investigations showed that this was not due to the parser, but due to the tested schemas being incorrectly specified according to the EXPRESS standard. This appears to be a hold-over from draft versions of the EXPRESS standard whose syntax were more flexible than the final version (e.g., allowing any order of SUBTYPE and SUPERTYPE specification in earlier definitions). Parsers developed from the time of the draft specification always allowed these more flexible syntax specifications and hence a strict checker failed on schemas which were being passed through to ISO for evaluation as draft standards.

Being able to parse a schema specification is an initial step, to be used in the environments developed in this thesis the parsed schema has to be translated into an implementable language. Two such translators are described below.

F.1.1 EXPRESS to Snart translator

The Snart language is the implementation language for this thesis. Therefore, to be able to utilise EXPRESS schemas in this thesis it was necessary to translate the parsed EXPRESS definition into Snart class definitions. At the time that this translator was written a conscious decision was taken to limit the amount of EXPRESS which was translated into Snart. This was to limit the amount of time required to produce the translator, and in recognition of the fact that all schemas produced to date do not use the majority of the EXPRESS language. The portions of EXPRESS which are not translated are the procedure definitions and some complicated *where* conditions. All inheritance structures, type definitions, attribute specifications, uniqueness constraints, and many simple *where* conditions are translated into Snart form. The main predicate available to perform this translation is described below:

express2snart

Prompts the user for the file(s) containing the EXPRESS schema to be translated to Snart. The files are passed through the parser described above and the resultant parse trees passed to a pretty printer which creates a file of Snart class definitions.

The translated Snart equivalent of the EXPRESS code above is shown below:

```
% Snart class definitions created from the EXPRESS file for the schema:  idm

class(wall,
   inherits([]),
   features([
      name: facets([type(string), relationship(values), relationship(key)]),
      height: facets([type(real), relationship(values)]),
      width: facets([type(real), relationship(values)]),
      azimuth: facets([type(real), relationship(values)]),
      position: facets([type(list(real)), attribute_cardinality([3, 3]),
                relationship(values)])
   ])
).
```

F.1.2 Snart to EXPRESS translator

While Snart is the implementation language of this thesis, it has previously been stated that EXPRESS is the specification language used by modellers in this domain. Therefore, it was felt necessary to provide a mechanism to translate Snart-based schemas through to EXPRESS form. This translator provides a limited translation ability as EXPRESS has no ability to record method definitions as exist in Snart. Therefore, what gets translated is restricted to class specifications with inheritance and attribute definitions. The predicate that provides this translation is described below:

snart2express

Prompts the user for the file(s) containing the Snart schema definition. These files are loaded into the Snart system (if not already loaded) and all class definitions extracted. A pretty printer is then invoked to write a new file containing the EXPRESS representation of the Snart definitions.

F.1.3 Snart to Re*f*lex translator

The Re*f*lex CAD tool provides a language to specify new objects which can be placed in the CAD environment. To take advantage of its use as a model specification tool, as described in Section 9.2.4, it was necessary to be able to translate schema specifications into its object definition language. The object definition language of Re*f*lex is very simple, utilising a single inheritance model and with an associated form definition language to enable attributes of an object to be specified when the object is laid down using the CAD tool. However, what was not possible to specify directly in the translation was the graphical form of the object. This has to be hand-edited into the object specification. In most cases, existing object definitions from the Re*f*lex library can be used and the parameters of these library objects tied to the attributes specified for the translated objects. The predicate provided to translate schemas into Re*f*lex form is described below:

snart2reflex

Prompts the user for the file(s) containing the Snart schema definition. These files are loaded into the Snart system (if not already loaded) and all class definitions extracted. A pretty printer is then invoked to write a new file containing the Re*f*lex representation of the Snart definitions.

The Re*f*lex object and form definitions for the EXPRESS schema above are shown below (prior to the hand-editing to insert existing library object definitions):

A Reflex object specification

```
///////////////////////////////////////////////////////////////////////////////
//
// Element Name:    wall
//
//    Library Name: IDM
//
//    Written by:   Robert Amor - Building Research Establishment
//
//    Date:         1997-5-24
//
//    Version No:   1.0
//
//    Version Date: 1997-5-24
//
//
//    Description:
//
//    (c) Copyright 1997 Robert Amor and the Building Research Establishment
//
///////////////////////////////////////////////////////////////////////////////

#include "objlib.vel"
#include "wall_draw.vel"

element wall
{

saved
     float azimuth
     float height
     string name
     float position[3]
     float width

private
     void OnEdit()
     void OnSave()
     void draw_shape(integer)

views
     plan
     View3d

dialog
     wallDlg
}

link wallDlg
{
     azimuth Tazimuth
     height Theight
     name Tname
     position Tposition
     width Twidth
}
```

```
void OnEdit()
{
}

void OnSave()
{
}

view plan
{
     draw_shape(0)
}

view View3d
{
     draw_shape(1)
}
```

A *Reflex* dialogue specification

```
///////////////////////////////////////////////////////////////////////////////
//
// Dialog Name:     wallDlg
//
//    Library Name: IDM
//
//    Written by:   Robert Amor - Building Research Establishment
//
//    Date:         1997-5-24
//
//    Version No:   1.0
//
//    Version Date: 1997-5-24
//
//
//    Description:
//
//    (c) Copyright 1997 Robert Amor and the Building Research Establishment
//
///////////////////////////////////////////////////////////////////////////////

dialog wallDlg
{

     Help "IDM/wall"
     Title "wall - IDM"

     Wgrid GRgrid
     {
          Dimension 5 2

          Wlabel Lazimuth { XY 1 1; Label "azimuth" }
          Wtext Tazimuth { Width 150 ; XY 1 2 }

          Wlabel Lheight { XY 2 1; Label "height" }
          Wtext Theight { Width 150 ; XY 2 2 }

          Wlabel Lname { XY 3 1; Label "name" }
          Wtext Tname { Width 150 ; XY 3 2 }
```

```
        Wlabel Lposition { XY 4 1; Label "position" }
        Wtext Tposition[3] { Width 150 ; XY 4 2 }

        Wlabel Lwidth { XY 5 1; Label "width" }
        Wtext Twidth { Width 150 ; XY 5 2 }

    }

}
```

F.2 ISO-10303:21 STEP data-file Parser

As well as being able to handle schema definitions in the EXPRESS specification language, it was necessary to be able to handle data associated with these schema definitions. ISO-STEP have defined a simple data-file representation to enable models to be transported (ISO/TC184 1994). The STEP Part 21 standard document contains a BNF grammar (Wirth 1977) for the language. The standard is available in electronic format and the grammar could be extracted from this document without rekeying. Starting with a BNF grammar it is an easy task to create a DCG form, which returns a parse tree of the schema being handled. This is what was implemented for this parser. The parser reads the whole data-file into memory (for ease of programming, though a buffered approach would be more capable for very large files) and then passes it to the DCG grammar to return a parse tree for the data-file. The main predicates developed in the STEP data-file parser are presented below:

step_exchange_file(-Schema, +Chars, [])
This is the top level DCG predicate which takes a set of characters (*Chars*) and returns a parse tree in *Schema*. The *[]* denote that the whole set of characters must be used when parsing. The DCG prints out the record number being processed at any stage as an aid to identifying syntax errors in data-files being parsed.

step2parse(-Parsed)
This top level predicate prompts the user for the file containing the STEP data-file to be parsed and reads the file before passing it through to the *step_exchange_file* predicate to return the parse tree in *Parsed*.

step_read(+FName, -Schema)
This programming level predicate provides a silent way of performing the function offered by *step2parse*. This predicate is used inside other predicates to import a STEP data-file for further processing, without having to involve the user.

step_checker

To check a STEP data-file's syntax prior to use this predicate loads in the file and ensures that it can be processed by the DCG.

An example of the syntax tree returned for a simple STEP data-file (based on the wall schema of Appendix F.1) is shown below:

A STEP data-file

```
ISO-10303-21;

HEADER;
FILE_DESCRIPTION(('This file contains a simple set of wall examples'),'idm');
FILE_NAME($,'1997-04-20 17:05:23',('RWA'),('Comp Sci, University of
Auckland'),'Hand-coded','idm',$);
FILE_SCHEMA(('idm'));
ENDSEC;

DATA;
#1 = WALL('North ground-floor', 2.4, 6.5, 0.0, (0.0, 0.0, 0.0));
#2 = WALL('East ground-floor', 2.4, 4.5, 90.0, (6.5, 0.0, 0.0));
#3 = WALL('South ground-floor', 2.4, 6.5, 180.0, (6.5, 4.5, 0.0));
#4 = WALL('West ground-floor', 2.4, 4.5, 270.0, (0.0, 4.5, 0.0));
ENDSEC;

END-ISO-10303-21;
```

Parsed representation of a STEP data-file

```
Parsed = exchange_file(
 header_section(
  [
  header_entity(standard_keyword('FILE_DESCRIPTION'),
   [untyped_parameter(list([untyped_parameter(
   string('This file contains a simple wall example'))])),
   untyped_parameter(string(idm))]),
  header_entity(standard_keyword('FILE_NAME'), [untyped_parameter(missing),
   untyped_parameter(string('1997-04-20 17:05:23')),
   untyped_parameter(list([untyped_parameter(string('RWA'))])),
   untyped_parameter(list([untyped_parameter(string('Comp Sci, University of
   Auckland'))])), untyped_parameter(string('Hand-coded')),
   untyped_parameter(string(idm)), untyped_parameter(missing)]),
  header_entity(standard_keyword('FILE_SCHEMA'),
   [untyped_parameter(list([untyped_parameter(string(idm))]))])
  ]
 ),
 data_section(
  [
  entity_instance(entity_name(1), nil,
   simple_record(standard_keyword('WALL'),
   [
   untyped_parameter(string('North ground-floor')),
   untyped_parameter(real(2.4)),
   untyped_parameter(real(6.5)),
   untyped_parameter(real(0)),
   untyped_parameter(list([untyped_parameter(real(0)),
    untyped_parameter(real(0)), untyped_parameter(real(0))]))
   ]
  )),
```

```
  entity_instance(entity_name(2), nil,
   simple_record(standard_keyword('WALL'),
   [
   untyped_parameter(string('East ground-floor')),
   untyped_parameter(real(2.4)),
   untyped_parameter(real(4.5)),
   untyped_parameter(real(90)),
   untyped_parameter(list([untyped_parameter(real(6.5)),
    untyped_parameter(real(0)), untyped_parameter(real(0))]))
   ]
  )),
  entity_instance(entity_name(3), nil,
   simple_record(standard_keyword('WALL'),
   [
   untyped_parameter(string('South ground-floor')),
   untyped_parameter(real(2.4)),
   untyped_parameter(real(6.5)),
   untyped_parameter(real(180)),
   untyped_parameter(list([untyped_parameter(real(6.5)),
    untyped_parameter(real(4.5)), untyped_parameter(real(0))]))
   ]
  )),
  entity_instance(entity_name(4), nil,
   simple_record(standard_keyword('WALL'),
   [
   untyped_parameter(string('West ground-floor')),
   untyped_parameter(real(2.4)),
   untyped_parameter(real(4.5)),
   untyped_parameter(real(270)),
   untyped_parameter(list([untyped_parameter(real(0)),
    untyped_parameter(real(4.5)), untyped_parameter(real(0))]))
   ]
  ))
  ]
 )
)
```

F.2.1 STEP data-file to Snart translator

The Snart language is the implementation language for this thesis. Therefore, to be able to utilise data associated with EXPRESS schemas in this thesis it was necessary to translate the parsed STEP data-file definition into Snart objects. The objects are created in the current object space, and if this space happens to be persistent then the created objects will be saved when the persistent store is saved. The main predicate available to perform this translation is described below:

step2snart

Prompts the user for the file containing the STEP data-file to be translated to Snart objects. The file is passed through the parser described above and the resultant parse tree passed to an instantiator which will create objects as required and assert the specified values for the object's attributes. STEP data-file style object references are changed to Snart style object references to ensure that relationships between objects are still maintained.

F.2.2 Snart to STEP data-file translator

While Snart is the implementation language of this thesis it has previously been stated that STEP is the environment used by modellers in this domain. Therefore, it was felt necessary to provide a mechanism to translate Snart-based models through to STEP data-file form. The predicate that provides this translation is described below:

snart2step(+OSNum)
All objects in the space specified by *OSNum* are identified and the user prompted for a STEP data-file to create. Each object is written directly to the file with Snart style object references modified to STEP data-file references.

F.3 CGE Parser

To allow modelled processes to be used to drive the workflow of an integrated design system, as described in Chapter 7, it was necessary to extract relevant process information from the files created by the CGE modelling tool (Vogel 1991). The CGE program uses a single directory for a set of files representing one model. Each diagram in a model has its own file in this directory. The file representing a diagram is stored in plain ASCII format (see the example below) and is named according to the formalism used in the diagram. Therefore it is relatively simple to identify all files associated with a project window, and using the file-name it is possible to identify the top level diagram which describes actors and their design roles as well as identifying the top level CombiNet.

Due to the simple fixed-format, attribute per line syntax of CGE files it was felt unnecessary to develop a CGE parser for this formalism. Instead a simpler hand-coded parser was written which processes the file line-by-line. Based on the object types that are found in the file, the parser extracts only those attributes which are named as necessary for that object type (e.g., the *tmp, x, and y* variables in the *DesignFunction* objects shown below are not required).

A portion of a CGE data-file

```
Name_of_st_def_used : "PW"
Diagram_name        : "Trebor_house"
Diagram_title       : "No_Title"
Name_of_creator     : "marcel"
Date_of_creation    : September 19 1995
Time_of_creation    : 13:04:40
Date_last_updated   : September 21 1995
Time_last_updated   : 10:12:16
Modified_by         : "marcel"
Release_number      : 0
Version_number      : 7
```

```
Diagram "Trebor_house"
Begin

	Class "Actor" With

	Class "DesignRole" With

	Class "DesignFunction" With

	Class "Connector" With
		Var "show_amt" = 0

	Class "brpnt" With

	Class "Legend" With
		Var "x" = 300
		Var "y" = -1000
		Var "w" = 2900
		Var "h" = 2700
		Var "version" = "1.0"
		Var "date" = "21/9/95"
		Var "author" = "Robert Amor"

	Object <14> Is_of_class "DesignFunction" With
		Var "x" = 880
		Var "y" = -680
		Var "design_function" = "specify_
bracing"
		Var "tmp" = 42
		Var "input_tool" = "mest_in.xps"
		Var "output_tool" = "mest_out.xps"
		Var "tool_name" = "MaterialsEditor"

	Object <13> Is_of_class "DesignFunction" With
		Var "x" = 1180
		Var "y" = -1200
		Var "design_function" = "specify_
glazing_
properties"
		Var "tmp" = 36
		Var "input_tool" = "mewi_in.xps"
		Var "output_tool" = "mewi_out.xps"
		Var "tool_name" = "MaterialsEditor"
```

There is a single predicate provided in the CGE parser to translate CGE projects, this is described below:

pw_read

This predicate prompts the user for the directory in which the project window files are stored. The set of files in this directory are collated and the top level project window description identified. This file is parsed to identify the top level CombiNet as well as actors, design roles, and design functions with their associated design tools. The CGE parser then processes all of the flow of control diagrams and builds up a single consistent name space for all workflow items in these diagrams. The parsed CGE project window is then written out into a STEP data-file representation of the whole project window. An example for the CGE file above is shown below:

A portion of a STEP data-file representing a CGE project window

```
ISO-10303-21;

HEADER;
FILE_DESCRIPTION(('This file contains a PW model translated from CGE'),'PW');
FILE_NAME($,'1997-04-20 17:05:23',('RWA'),('Comp Sci, University of Auckland'),'CGE to STEP','PW',$);
FILE_SCHEMA(('PW'));
ENDSEC;

DATA;
#1 = PROJECT_WINDOW('Trebor_house', (#11, #28, #29, #30, #31), #32);
#2 = DESIGN_FUNCTION('specify_bracing', 'MaterialsEditor', 'mest_in.xps', 'mest_out.xps');
#3 = DESIGN_FUNCTION('specify_glazing_properties', 'MaterialsEditor', 'mewi_in.xps', 'mewi_out.xps');
```

F.4 VML Parser

VML has a syntax very similar to that of Prolog, as much of its expressive capability is based on that of Prolog. However, there are some syntactic constructs which make VML specifications not directly parsable by the Prolog system. Therefore, a DCG was constructed based on the BNF grammar shown in Appendix A.1. The parser reads the whole data-file into memory (for ease of programming, though a buffered approach would be more capable for very large files) and then passes it to the DCG grammar to return a parse tree for the mapping. The main predicates developed in the VML parser are presented below:

map_mapping(-Schema, +Chars, [])
This is the top level DCG predicate which takes a set of characters (*Chars*) and returns a parse tree in *Schema*. The *[]* denote that the whole set of characters must be used when parsing. The DCG prints out the mapping header being processed at any stage as an aid to identifying syntax errors in data-files being parsed.

map2parse(-Parsed)
This top level predicate prompts the user for the file containing the VML specification to be parsed and reads the file before passing it through to the *map_mapping* predicate to return the parse tree in *Parsed*.

map_read(+FName, -Schema)
This programming level predicate provides a silent way of performing the function offered by *map2parse*. This predicate is used inside other predicates to import a VML specification for further processing, without having to involve the user.

map_checker
To check a VML file's syntax prior to use this predicate loads in the file and ensures that it can be processed by the DCG.

Appendix G

Generalised Schema Representation Notation

This notation allows the definition of general schemas from a wide range of modelling languages in both relational and object-oriented forms. The notation allows for versioning control in schema specification and holds redundant data to allow easy application or undoing of modifications.

The definitions below use a notation derived from the Prolog specification of the representation. Lower-case symbols are terminal tokens in the definition. Upper-case symbols represent variables which are to be specified for each record. The use of square brackets [] denotes a list of zero or more of the object defined between the brackets.

G.1 Version Tree

The version tree is defined as a directed acyclic graph with a single root node, denoted by an empty list in the *ParentVersion* definition. Version information is stated as shown in Table G.1.

```
version(VersionNumber, [ParentVersion], DateStarted, ReasonStarted, ModSequenceNumber)
```

Table G.1 Specification of a schema version

The variables in this definition are used for the following information: *VersionNumber* is the unique identifier for the version; *ParentVersion* is a list of all *VersionNumber*s that this version builds upon; *DateStarted* is the date this version was started; *ModSequenceNumber* is the highest specified sequence number of the modifications that have been made in this version; *ReasonStarted* provides further information about the reason for the versions creation. The range of allowable values for *ReasonStarted* is shown in Table G.2.

```
new_schema(SchemaFileName)
user_modification(UserName, Description)
merging(UserName, Description)
mapping_dependant(MapName, MapFileName)
```

Table G.2 Specification of version creation reasons

The four values for *ReasonStarted* are used in the following cases: *new_schema* defines the initialisation of the version tree and schema from an existing file of schema definition; *user_modification* describes a user initiated modification to a schema (including starting a schema); *merging* allows for the merging of two or more versions with no other reason than the bringing together of modifications in the merged versions; *mapping_dependant* describes a new version which is created due to specifications in a mapping definition which extend the schema version the mapping is defined upon.

G.2 Schema

The schema definition has two parts: a definition of the schema represented in the model; followed by a set of sequential modifications which describe the construction of the schema definition in a particular version. The form of these two specifications is shown in Table G.3.

```
schema(SchemaName, VersionTreeRoot, VersionSequenceNumber)
schema_mod(Version, SequenceNumber, Modification)
```

Table G.3 Specification of schema information

The variables in the definitions are used for the following information: *SchemaName* is a unique descriptive identifier of the schema; *VersionTreeRoot* specifies the starting version number for the schema (usually number one); *VersionSequenceNumber* represents the highest specified version number in the schema definition; *Version* is the schema version that the specified modification is applied to; *SequenceNumber* is the position in the list of modifications which make up the version that this modification occurred in; *Modification* describes the atomic change that was applied to the schema during this modification. The *Modification* specification can take the values shown in Table G.4.

The variables in the definitions shown in Table G.4 are used for the following information: *EntityName* is the name of an entity in a schema, this is unique to each schema; *EntityType* defines whether the entity is abstract or normal (for object-oriented schemas); *Position* allows the specification of a position in a list of inherited entities, position may affect the methods which will be called in an object-oriented schema; *AttributeName* is the name of a unique attribute in an entity definition; *Facet* specifies an associated piece of information describing an attribute (e.g., attribute type, uniqueness constraints, range constraints); *Value* specifies the current value of the *Facet* it is defined in; *Method* is a method interface specification (for object-oriented schemas).

```
add_entity(EntityName, EntityType)
rename_entity(OldName, NewName)
delete_entity(EntityName, [Parent], [Attribute], [Method])
add_parent(ParentName, EntityName)
add_parent(ParentName, Position, EntityName)
rename_parent(OldName, NewName, EntityName)
delete_parent(ParentName, Position, EntityName)
re_order_parents(OldOrder, NewOrder, EntityName)
add_attribute(AttributeName, EntityName)
add_attribute(AttributeName, [Facet(Value)], EntityName)
rename_attribute(OldName, NewName, EntityName)
delete_attribute(AttributeName, [Facet(Value)], EntityName)
change_attribute_type(AttributeName, OldType, NewType, EntityName)
add_facet(AttributeName, Facet(Value), EntityName)
rename_facet(AttributeName, OldName, NewName, EntityName)
delete_facet(AttributeName, Facet(Value), EntityName)
modify_facet_value(AttributeName, Facet, OldValue, NewValue, EntityName)
add_method(Method, EntityName)
rename_method(OldName, NewName, EntityName)
delete_method(Method, EntityName)
change_method_body(OldMethod, NewMethod, EntityName)

retract_mod(Version, SequenceNumber)
```

Table G.4 Specification of modification types

Appendix H

Generalised Mapping Representation Notation

This notation allows the definition of a mapping between two schemas in a pre-processed form that can be loaded by any implementation of a mapping system and used to determine the mappings that are required between entities in the schemas. As the mappings are tied to a particular version of a schema definition, a compacted form of the schema's entity definitions is also included in the definition.

The definitions below use a notation derived from the Prolog specification of the representation. Lower-case symbols are terminal tokens in the definition. Upper-case symbols represent variables which are to be specified for each record. The use of square brackets [] denotes a list of zero or more of the object defined between the brackets.

H.1 Schema

A schema definition has two parts: a definition of the schema and version being represented followed by the definition of all entities from the schema that are used in the mapping to be described, as shown in Table H.1.

```
view(ViewID, ViewName, ViewFileName, ViewVersion, ViewVersionType, ViewType)
entity(ClassName, ViewID, [Parent], [Attribute], [Method], Type, [CreateAttr], [Misc])
```

Table H.1 Specification of schema entities

The variables in these definitions are used for the following information: *ViewID* is the unique identifier of a particular schema, in the entity definition it can take the value temporary to describe a temporary entity; *ViewName* is the human readable name or description of a schema; *ViewFileName* points to the generic schema representation database of the schema; *ViewVersion*

represents the version of the schema in the version tree described in Appendix G; *ViewVersionType* describes the reason for the creation of the version being used; *ViewType* defines whether the schema is to be treated as *read_only*, *read_write* or *integrated* (see Section 5.3.1 for a definition of the meaning of these types); *ClassName* is the name of the entity; *Parent* is the name of a parent class of the current class; *Attribute* is an attribute of the entity, described as the attribute name and a set of facets describing the attribute type, units, constraints, etc.; *Method* is the interface definition of a method defined in the entity; *Type* specifies whether the class is abstract or not; *CreateAttr* specifies create parameters of an entity in the same form as an *Attribute* definition; *Misc* allows a set of extraneous class definitions which do not affect the mapping to be described in the entity definition (e.g., demons in Snart).

H.2 Mapping

The mapping definition is broken down into sections which represent the major components of VML maps as shown in Table H.2.

```
inter_view(ViewID, ViewID, Type, MaxMapNumber, MaxInvNum, MaxEquivNum, MaxInitNum)
inter_class(MapNumber, [ClassName], [ClassName], [InheritMapNumber], [InvariantNumber],
       [EquivalenceNumber], [InitialiserNumber])
invariant(InvariantNumber, Code, [ReferencedAttribute], [TypeFlag])
equivalence(EquivalenceNumber, Code, [ReferencedAttribute], [TypeFlag])
initialiser(InitialiserNumber, Code, [ReferencedAttribute], [TypeFlag])
```

Table H.2 Specification of atomic mapping components

The variables in these definitions are used for the following information: *ViewID* in the *inter_view* defines the ID of the left and right schema in a mapping; *Type* describes whether *partial* or *complete* mappings will be required (see Section 5.3.1); *MaxMapNumber* records the maximum *inter_class* ID that has been allocated; *MaxInvNum*, *MaxEquivNum*, and *MaxInitNum* record the maximum ID that has been allocated to invariants, equivalences and initialisers respectively; *MapNumber* is the unique identifier of an *inter_class* definition in this mapping; *ClassName* is the full reference of a class from either the left or right schemas in a mapping; *InheritMapNumber* is the ID of an *inter_class* definition which is inherited by this *inter_class*; *InvariantNumber*, *EquivalenceNumber*, and *InitialiserNumber* are the unique identifiers of individual invariants, equivalences, and initialisers; *Code* is the parse tree representation of the mapping which is being represented; *ReferencedAttribute* is the full reference of an entity, attribute or method from the *Code*; *TypeFlag* is a flag which indicates the type of mapping being represented (current values are: *list_ref* or *no_list_ref* depending upon whether there is a list reference in the equation; *pointer_equivalence* if the equation is purely between pointers to other objects or *calculation* otherwise).

H.3 Inverted Index

The inverted indices provide summary information about the complete mapping between the schemas. At the current time the information which is collated is for two purposes. One is for determining which inter_class definitions are purely class initialisers for a mapping systems initialisation (i.e., only reference classes from one schema). The other is to provide fast access from a given entity, attribute or method to the code segments which utilise that reference. Table H.3 shows the summary information structures in the generalised mapping representation.

```
class_initialisers([MapNumber])
class_maps(ClassName, ViewID, [MapNumber])
attribute_maps(AttributeName, ClassName, ViewID, [InvariantNumber], [EquivalenceNumber],
       [InitialiserNumber])
method_maps(MethodName, ClassName, ViewID, [InvariantNumber], [EquivalenceNumber],
       [InitialiserNumber])
```

Table H.3 Specification of inverted indices into mappings

The variables in these definitions are used for the following information: *MapNumber* is the ID of an *inter_class* mapping in this database; *ClassName* is the full reference of a class; *ViewID* is the ID of a schema in the mapping; *AttributeName* or *MethodName* is the full reference of a particular attribute or method referenced in a mapping; *InvariantNumber*, *EquivalenceNumber*, and *InitialiserNumber* are the IDs of invariants, equivalences, and initialisers which reference a particular attribute or method.

Glossary

Actor	A person who plays a set role in a project, for example an architect or structural engineer or project manager
A/E/C	Architecture, Engineering and Construction
ALF	Annual Loss Factor method, developed by BRANZ to encourage energy efficient building design
AP	Application Protocol, a name used in ISO-STEP for domain schema
ARPA	Advanced Research Projects Agency
Aspect model	Defines the various views of actors, design functions and project windows. Usually a schema definition which is a subset of the IDM, but with extra constraints specified on the model
ATLAS	Architecture, methodology and Tools for computer integrated LArge Scale engineering, an EU funded project
AVL	Adelson, Velskii and Landis tree, a partially balanced tree structure
BRANZ	Building Research Association of New Zealand
CAD	Computer Aided Draughting
CGE	Configurable Graphical Editor
CernoII	A model visualisation tool used and specialised in this thesis
CIME	Computer Integrated Manufacturing and Engineering
CIMsteel	Computer Integrated Manufacturing for constructional steelwork, an EU funded project
COMBI	Computer-integrated Object-oriented product Model for the Building Industry, an EU funded project
COMBINE	Computer Models for the Building Industry in Europe, an EU funded project
CombiNet	Project modelling methodology developed in this thesis
CORBA	Common Object Reference Brokering Architecture
CSCW	Computer Supported Collaborative Working
DCG	Definite Clause Grammar

DF	Design Function
DFD	Data Flow Diagram
DML	Design Modelling Language
DT	Design Tool
DTF	Design Tool Function
EDM-2	Engineering Data Model
EER	Extended Entity Relationship
Entity	The container which holds the attributes and relationships used to model a particular real life artifact
EPE	EXPRESS Programming Environment. A modelling environment for the EXPRESS and EXPRESS-G notations
ER	Entity Relationship
ESPRIT	The EU Information Technology Programme
EU	European Union, a major funding source for large projects
ExEx	Exchange Executive, workflow management tool developed in this thesis
EXPRESS	A schema specification language developed for ISO-STEP
EXPRESS-C	A proposed extension to EXPRESS with some mapping capability
EXPRESS-G	A graphical version of EXPRESS with greatly reduced functionality
EXPRESS-M	A mapping language proposed for ISO-STEP
EXPRESS-V	A mapping/view language proposed for ISO-STEP
FaceEditor	A material specification tool used in this thesis
GIS	Geographic Information System
GUI	Graphical User Interface
HVAC	Heating, Ventilating and Air Conditioning
IBDS	Integrated Building Design System
ID	Identifier, used in the context of objects
IDEF0	A function modelling method
IDEF1X	A data modelling method
IDEF3	Process flow and object state modelling method
IDL	Interface Description Language, associated with the CORBA standard
IDM	Integrated Data Model
IIBDS	Intelligent Integrated Building Design System
Instance	A set of data values which describe a particular building. The data values are structured as specified in the model definition for the particular tool or actor that requires the data
ISDE	Integrated Software Development Environment
ISO/TC	International Standards Organisation/Technical Committee
IT	Information Technology, synonymous with computers
KBS	Knowledge-Based System
KIF	Knowledge Interchange Format

Model	A conceptual model of a particular domain. In many cases it reflects the structures and attributes of individual tools or actors. A model which has values inserted to describe a particular building is called an instance
MSE	Modelling Support Environment
MViews	Multiple Views, a framework for developing ISDEs
MViewsER	MViews specialised to support ER modelling
NIAM	Nijssen Information Analysis Method
Object	A set of data values which describe a particular artefact. The data values are structured as specified in the entity definition
OO	Object Oriented
OOA/D	Object Oriented Analysis/Design
PDT	Product Data Technology, denotes the body of available tools, methodologies and standards in the area of integrated CAD systems, dealing with product data interchange, product models, data modelling, integrated software architectures, etc.
PlanEntry	A building plan specification tool used in this thesis
Project	The process of designing and constructing a building. A project starts with the initial ideas on a building and is finished at the time the building is completed
PW	Project Window. This is a window on a project, detailing some small part of the total project. A PW includes the actors responsible for individual tasks, the tools they utilise and the tasks they need to complete
RDBMS	Relational DataBase Management System
SANZ	Standards Association of New Zealand
Schema	A data model (or models) representing a distinct domain, or sub-domain, in the A/E/C area. Examples of a schema include models for the building skeleton, HVAC components, or structural steel components. A full model of a building could be thought of as a schema, though it is unlikely that a homogeneous model can be specified for a whole building over all life-cycles, and all actors. Each of these schema may comprise several, potentially duplicate, sub-schema, for example to describe materials, geometry, etc.
SDAI	Standard Data Access Interface, part of the ISO-STEP standard
Snart	An object-oriented language built on top of Prolog
SPE	Snart Programming Environment. A modelling environment for the Snart language
STAR	A Finnish research programme at VTT for construction process improvement and re-engineering
STEP	Standard for the Exchange of Product model data (ISO 10303)
SUMM	Semantic Unification Meta-Model

Superviews	A RDBMS integration technique
TES	Tool Encapsulation Specification
ToCEE	Towards a Concurrent Engineering Environment, an EU funded project
Transformr	A mapping specification language
VEGA	Virtual Enterprises using Groupware tools and distributed Architecture, an EU funded project
VISION-3D	A 3D modelling and visualisation tool used in this thesis
VML	View Mapping Language. A textual mapping specification language
VML-G	View Mapping Language - Graphical form. A subset of the VML textual language
VPE	VML Programming Environment. A modelling environment for the VML language

References

Ainsworth, M., Riddle, S. and Wallis, J.L. (1996) Formal validation of viewpoint specifications, Software Engineering Journal, 11(1), January, pp. 58-66.

Ambler, A. and Burnett, M. (1989) Influence of Visual Technology on the Evolution of Language Environments, IEEE Computer, 22, pp. 9-22.

Amor, R.W. (1991) ICAtect: Integrating Design Tools for Preliminary Architectural Design, MSc thesis, Victoria University of Wellington, Wellington, New Zealand.

Amor, R.W. (1993) QBE like queries in Snart, unpublished report, Computer Science Department, University of Auckland, Auckland, New Zealand.

Amor, R.W. (1994) A Mapping Language for Views, unpublished report, Computer Science Department, University of Auckland, Auckland, New Zealand.

Amor, R. and Clift, M. (1997) Documents as an Enabling Mechanism for Concurrent Engineering in the Construction Industry, accepted for presentation at the 1st international conference on Concurrent Engineering in Construction, CEC'97, London, UK, 3-4 July.

Amor, R.W. and Hosking, J.G. (1993) Multi-Disciplinary Views for Integrated and Concurrent Design, The Management of Information Technology for Construction, First International Conference, Mathur, K.S., Betts, M.P. and Tham, K.W. (eds), Selected (refereed) papers, Singapore, 17-20 August, pp. 255-268.

Amor, R. and Hosking, J. (1995) Mappings: The Glue in an Integrated System, The First European Conference on Product and Process Modelling in the Building Industry, Dresden, Germany, 5-7 October 1994, Scherer, R. (ed), Balkema, Rotterdam, pp. 117-124.

Amor, R. and TU Delft COMBINE Team (1993) A Set Theoretic Exchange Executive, COMBINE 2 report COMB2-93-34, 10pp.

Amor, R., Augenbroe, G., Hosking, J., Rombouts, W. and Grundy, J. (1995) Directions in Modelling Environments, Automation in Construction, 4, pp. 173-187.

Amor, R.W., Hosking, J.G., Mugridge, W.B., Hamer, J. and Williams, M. (1992) ThermalDesigner: an application of an object-oriented code conformance architecture, Vanier, D. and Thomas, R. (eds), Joint CIB Workshops on Computers and Information in Construction, International Council for Building Research Studies and Documentation (CIB), Publication 165, Montreal, Canada, 12-15 May, pp. 1-11.

ASTA (1996) PowerProject: Graphical Project Management Software, ASTA Development Corporation, Thame, Oxfordshire, UK.

Atkinson, M., Bailey, P., Chisolm, K., Cockshott, W., and Morrison, R. (1983) An Approach to Persistent Programming, Computer Journal, vol. 26, pp. 360-365.

ATLAS (1993) ATLAS: Architecture, methodology and Tools for computer integrated LArge Scale engineering, ESPRIT 7280, 37pp.

Augenbroe, G. (1993) COMBINE Final Report, CEC-DGXII, Brussels.

Augenbroe, G. (1994) Integrated Building Design Systems in the Context of Product Data Technology, ASCE Journal of Computing in Civil Engineering, 8(4), October, pp. 420-535.

Augenbroe, G. (1994b) Privileged communication.

Augenbroe, G. (1995a) COMBINE-2 Final Report, CEC-DGXII, Brussels.

Augenbroe, G. (1995b) An overview of the COMBINE project, The First European Conference on Product and Process Modelling in the Building Industry, Dresden, Germany, 5-7 October 1994, Scherer, R. (ed), Balkema, Rotterdam, pp. 547-554.

Augenbroe, G. and Laret, L. (1989) COMBINE pilot study report, CEC-JOULE Report, Brussels.

Bailey, I. (1994) EXPRESS-M Reference Manual, Product Data Representation and Exchange, ISO TC184/SC4/WG5 N51, January.

Bancilhon, F. and Spyratos, N. (1981) Update Semantics of Relational Views, ACM Transactions on Database Systems, 6(4), December, pp. 557-575.

Bassett, M.R., Bishop, R.C. and van der Werff, I.S. (1990) ALF Manual, Annual Loss Factor Design Manual, An aid to thermal design of buildings, Building Research Association of New Zealand, Judgeford, New Zealand.

Batini, C. and Lenzerini, M. (1984) A Methodology for Data Schema Integration in the Entity Relationship Model, IEEE Transactions on Software Engineering, 10(6), November, pp. 650-664.

Batini, C., Lenzerini, M. and Navathe, S.B. (1986) A Comparative Analysis of Methodologies for Database Schema Integration, ACM Computing Surveys, 18(4), December, pp. 323-364.

Bijnen, A. (1994) Operation Mapping or How to get the right data? Presented at 1st European Conference on Product and Process Modelling in the Building Industry (EC-PPM), Dresden, Germany, 5-7 October.

Bourke, P.D. (1989) VISION-3D User Manual, School of Architecture, University of Auckland, Auckland, New Zealand.

Bowen, J. and Bahler, D. (1991) Supporting Cooperation between Multiple Perspectives in a Constraint-based Approach to Concurrent Engineering, Technical report TR-91-17, North Carolina State University, USA.

Bowen, J. and Bahler, D. (1992) Negotiation in Concurrent Engineering, Technical report TR-92-10, North Carolina State University, USA.

Boyle, A. and Watson, A. (1993) STEP Tools Review: Phase 2, Computer-Aided Engineering Group, Department of Civil Engineering, University of Leeds, UK, February, 26pp.

Bright, M.W., Hurson, A.R. and Pakzad, S.H. (1992) A Taxonomy and Current Issues in Multidatabase Systems, IEEE Computer, 25(3), March, pp. 50-60.

Chen, P.P. (1976) The Entity-Relationship Model - Toward a Unified View of Data, ACM Transactions on Database Systems, 1(1), pp. 9-14.

CIMsteel (1995) Computer Integrated Manufacturing for constructional steelwork, Eureka project 130, Brussels.

Clarke, J.A., Rutherford, J.H. and MacRandal, D. (1989) An intelligent front-end for computer-aided building design, University of Strathclyde, Scotland.

Clark, S.N. (1992) Transformr: A Prototype STEP Exchange File Migration Tool, National PDES Testbed Report Series, NISTIR 4944, US Department of Commerce, National Institute of Standards and Technology, October, 14pp.

COMBI (1995) Computer-integrated Object-oriented product Model for the Building Industry, ESPRIT CIME, Brussels.

Curtis, B., Kellner, M.I. and Over, J. (1992) Process Modeling, Communications of the ACM, 35(9), September, pp. 75-90.

Dayal, U. and Bernstein, P. (1982) On the Correct Translation of Update Operations on Relational Views, ACM Transactions on Database Systems, 8(3), September, pp. 381-416.

Dubois, A.M. (1993) COMBINE IDM conceptual development task, COMBINE-Report, CSTB, Nice, France.

Eastman, C., Jeng, T-S., Assal, H., Cho, M. and Chase, S. (1995) EDM-2 Reference Manual, Center for Design and Computation, UCLA, Los Angeles, USA, 50pp.

Fenwick, S., Hosking, J.G. and Mugridge, W.B. (1994) Cerno-II: A program visualisation system, Department of Computer Science, University of Auckland, New Zealand, Report No 87, February.

Flynn, J. (1994) COMBINE Project Window 2 specification, Restricted COMBINE-Report, University College Galway, Ireland.

Fulton, J.A., Zimmerman, J., Eirich, P., Tyler, J., Burkhart, R., Lake, G.F., Law, M.H., Menzel, C., Speyer, B., Stumps, R. and Williams, A. (1992) Technical Report on the Semantic Unification Meta-Model (SUMM), Volume 1: Semantic Unification of Static Models, ISO TC184/SC4/WG3 N175.

Garlan, D. (1986) Views for Tools in Integrated Environments, Advanced Programming Environments - Lecture Notes in Computer Science No. 244, pp. 314-343.

General Electric (1985) Integrated Information Support System (IISS), Volume 5, Common Data Model Subsystem, Part 4, Information Modeling Manual, IDEF1 Extended, DTIC-A181952, December.

Genesereth, M.G. and Fikes, R.E. (1992) Knowledge Interchange Format Version 3, Reference Manual, Computer Science Department, Stanford University, USA.

Gielingh, W. (1988) General AEC Reference Model (GARM), ISO TC184/SC4, Document 3.2.2.1, October, 35pp.

Gielingh, W. and Suhm, A. (1992) IMPPACT Reference Model: An approach for integrated product and process modelling of discrete parts manufacturing, Springer Verlag.

Gogolla, M. (1994) An Extended Entity-Relationship Model: Fundamentals and Pragmatics, Springer-Verlag, 136pp.

Gosling, J. and McGilton, H. (1995) The Java Language Environment: A White Paper, Sun Microsystems, Inc. 2550 Garcia Avenue, Mountain View, California 94043-1100, USA.

Greening, R. and Edwards, M. (1995) ATLAS implementation scenario, The First European Conference on Product and Process Modelling in the Building Industry, Dresden, Germany, 5-7 October 1994, Scherer, R. (ed), Balkema, Rotterdam, pp. 467-472.

Grundy, J.C. (1993) Multiple Textual and Graphical Views for Interactive Software Development Environments, PhD thesis, University of Auckland, New Zealand.

Grundy, J. (1994) MViews User Manual, Department of Computer Science, University of Waikato, Hamilton, New Zealand, November.

Grundy, J.C. (1996) Serendipity: integrated environment support for process modelling, enactment and improvement, Working paper, Department of Computer Science, University of Waikato, Hamilton, New Zealand.

Grundy, J.C. and Hosking, J.G. (1993a) Integrated software development in SPE, Proc. 13th New Zealand Computer Society Conference, New Zealand Computer Society, August.

Grundy, J.C. and Hosking, J.G. (1993b) Constructing multi-view editing environments using MViews, Proc. 1993 IEEE Symposium on Visual Languages, IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 220-224.

Grundy, J.C. and Hosking, J.G. (1994) Constructing integrated software development environments with dependency graphs, Department of Computer Science, University of Waikato, Hamilton, New Zealand, Report No 94/4.

Grundy, J.C. and Venable, J. (1995) Providing Integrated Support for Multiple Development Notations, Proceedings of CAiSE'95, Finland, June, Lecture Notes in Computer Science No. 932, Springer-Verlag, pp. 255-268.

Grundy, J.C., Hosking, J.G., Fenwick, S. and Mugridge, W.B. (1993) Connecting the pieces: integrated development of object-oriented systems using multiple views, Information Engineering Report No 93/4, Information Engineering Section, Department of Electrical Engineering, Imperial College of Science, Technology and Medicine, London, November, 16pp.

Grundy, J.C., Hosking, J.G., Fenwick, S. and Mugridge, W.B. (1994) Chapter 11, Visual Object-Oriented Programming, M. Burnett, A. Goldberg, T. Lewis Eds, Manning/Prentice-Hall.

Grundy, J.C., Mugridge, W.B., Hosking, J.G. and Apperley, M.D. (1995) Coordinating, capturing and presenting work contexts in CSCW systems, in proceedings OZCHI'95, Wollongong, Australia, November.

Grundy, J.C., Hosking, J.G. and Mugridge, W.B. (1996) Low-level and High-level CSCW support in the Serendipity process modelling environment, in proceedings OZCHI'96, Hamilton, New Zealand, 24-27 November.

Hailpern, B. and Ossher, H. (1990) Extending Objects to Support Multiple Interfaces and Access Control, IEEE Transaction on Software Engineering, 16(11), November, pp. 1247-1257.

Hardwick, M. (1994) Towards Integrated Product Databases Using Views, Technical Report 94003, Rensselaer Polytechnic Institute, Troy, New York, USA.

Hardwick, M., Spooner, D., Kilty, M. and Jiang, Z. (1994) Mapping EXPRESS AIM's to ARM's Using Database Views: A Comparison of Three Approaches, Technical Report 94041, Rensselaer Polytechnic Institute, Troy, New York, USA.

Harrison, J.V. and Dietrich, S.W. (1994) Incremental View Maintenance, Department of Computer Science, University of Queensland, Brisbane, Australia, pp. 45-63.

Hosking J.G. and Mugridge W.B. (1994) A constraint based building model editor, Auckland UniServices Ltd, Ref. 4376.01, Auckland, New Zealand, June, 28pp.

Hosking, J., Mugridge, R. and Amor, R. (1995) An Integrated Building Design Environment, Auckland UniServices Ltd, Ref. 4376.02, Auckland, New Zealand, June, 16pp.

Hosking, J.G., Mugridge, W.B., and Blackmore, S. (1994) Objects and constraints: a constraint based approach to plan drawing, Mingins, C. and Meyer, B. (eds), Technology of object-oriented languages and systems, TOOLS 15, Prentice Hall, Sydney, pp. 9-19.

Huovila, P. and Seren, K-J. (1995) Customer-Oriented Design Methods, 10th International Conference on Engineering Design, 22-24 August, 6 pp.

IDEFine (1995) Design/IDEF User’s manual v3.5, IDEFine Ltd, 10 Salamanca, Wellington Park, Crowthorne RG45 6AP, UK.

IIC Consulting and Cimtech Limited (1996) 1996 Engineering Document Management and Product Data Management Guide, An Introduction to EDMS and PDM and the Suppliers of Products and Services, ISBN 0-900458-71-2.

ISO/TC184 (1992) Part 11: The EXPRESS Language Reference Manual in Industrial automation systems and integration - Product data representation and exchange, Draft International Standard, ISO-IEC, Geneva, Switzerland, ISO DIS 10303-11.

ISO/TC184 (1993) Part 1: Overview and fundamental principles in Industrial automation systems and integration - Product data representation and exchange, Draft International Standard, ISO-IEC, Geneva, Switzerland, ISO DIS 10303-1.

ISO/TC184 (1994) Part 21: Implementation methods: Clear text encoding of the exchange structure - Product data representation and exchange, Draft International Standard, ISO-IEC, Geneva, Switzerland, ISO DIS 10303-21.

ISO/TC184 (1995) Application Protocol 228, Building Services: HVAC, Working Draft v0.1, CSTB, Nice, France, 79pp.

ISO/TC184 (1996) Application Protocol 106, Building Construction Core Model, Working Draft, ISO-IEC, Geneva, Switzerland, ISO DIS 10303-106.

Jensen, K. (1990) Coloured Petri Nets: A High Level Language for System Design and Analysis, Advances in Petri Nets 1990, Rozenberg, G. (ed), Lecture Notes in Computer Science No. 483.

Katranuschkov, P., Scherer, R.J., Clift, M. and Amor, R. (1996) EU-ESPRIT IV, Project 20587, ToCEE - Deliverable J.1, Migration Perspectives, Public Report, EU/CEC, Directorate Generale III, Brussels.

Kay, A.C. (1977) Microelectronics and the Personal Computer, Scientific American, September, pp. 230-244.

KBSI (1995) ProSim: Automated Process Modeling for Windows, KBSI, College Station, Texas, USA, 37pp.

Khedro, T., Genesereth, M.R. and Teicholz, P.M. (1994) Concurrent Engineering Through Interoperable Software Agents, First conference on Concurrent Engineering: Research and Applications, Pittsburgh, USA.

Kim, W. and Seo, J. (1991) Classifying Schematic and Data Heterogeneity in Multidatabase Systems, IEEE Computer, 24(12), December, pp. 12-18.

Lamb, D.A. (1987) IDL: Sharing Intermediate Representations, ACM Transactions on Programming Languages and Systems, 9(3), July, pp. 297-318.

Lee, J. and Malone, T.W. (1990) Partially Shared Views: A Scheme for Communicating among Groups that Use Different Type Hierarchies, ACM Transactions on Information Systems, 8(1), January, pp. 1-26.

LPA (1995) LPA Prolog Technical Reference, Logic Programming Associates Ltd., Trinity Road, London, UK.

Luijten, B. (1992) A collection of PMShell papers, Report B1-92-087, TNO Building and Construction Research, Delft, The Netherlands.

Luiten, B. and Tolman F. (1992) Computer Aided DfC (Design for Construction) in the Building and Construction Industries, CIB W78, Computer and Building Standards Workshop, Montreal, Canada, 11-15 May, pp. 318-329.

Mayer, R.J. (1990) IDEF0 Function Modelling: A Reconstruction of the Original Air Force Report, Mayer, R.J. (ed), Knowledge Based Systems Inc., College Station, Texas, USA.

Mayer, R.J., Painter, M.K. and deWitte, P.S. (1994) IDEF Family of Methods for Concurrent Engineering and Business Re-engineering Applications, Knowledge Based Systems Inc., College Station, Texas, USA.

Mayer, R.J., Cullinane, T.P., deWitte, P.S., Knappenberger, W.B., Perakath, B. and Wells, M.S. (1992) Information Integration for Concurrent Engineering (IICE) IDEF3 Process Description Capture Method Report, Technical Report AL-TR-1992-0057, Armstrong Laboratory, College Station, Texas, USA, 136pp.

Meyers, S. (1991) Difficulties in Integrating Multiview Development Systems, IEEE Software, January, pp. 49-57.

Morrison, R., Brown, A., Carrick, R., Conner, R., and Dearle, A. (1988) On the Integration of Object-Oriented and Process-Oriented Computation in Persistent Environments, in: Advances in Object-Oriented Database Systems, Dittrich, K. (ed), Lecture Notes in Computer Science No. 334, Springer-Verlag, Ebernburg, West Germany, pp. 334-339.

Motro, A. (1987) Superviews: Virtual Integration of Multiple Databases, IEEE Transactions on Software Engineering, 13(7), July, pp. 785-798.

Mugridge, W. (1994) Snart: A Mixed-Paradigm Programming Language, Department of Computer Science University of Auckland, Auckland, NZ, Working Report.

Mugridge, W.B. and J.G. Hosking (1988) The development of an expert system for wall bracing design, Proceedings of NZES'88 The Third New Zealand Expert Systems Conference, Wellington, New Zealand, May, pp. 10-27.

Mugridge W.B. and Hosking J.G. (1995) Towards a lazy evolutionary common building model, Building and Environment, 30, (1), pp. 99-114.

Mugridge, W.B., Grundy, J.C., Hosking, J.G. and Amor, R. (1995) Snart94 Reference/User Manual, Department of Computer Science, University of Auckland, Auckland, New Zealand.

Mugridge, R., Hosking, J. and Amor, R. (1996) Adding a code conformance tool to an integrated building design environment, Auckland UniServices Ltd, Ref. 4376.04, May, 26pp.

Navathe, S., Elmasri, R. and Larson, J. (1986) Integrating User Views in Database Design, IEEE Computer, 19(1), January, pp. 50-62.

Nijssen, G.M. and Halpin, T.A. (1989) Conceptual Schema and Relational Database Design: A Fact Oriented Approach, Prentice-Hall, Englewood Cliffs, NJ, USA.

Otte, R., Patrick, P. and Roy, M. (1996) Understanding CORBA: the Common Object Request Broker Architecture, Prentice Hall Inc.

Pascoe, R.T. (1994) Construction of interfaces for the transfer of data between geographical information systems, PhD thesis, Department of Computer Science, University of Canterbury, New Zealand.

Papamichael, K.M. and Selkowitz, S.E. (1991) A Computer-Based Building Design Support Environment, proceedings of Building Systems Automation - Integration '91, Madison, Wisconsin, USA, 2-8 June, pp. 417-438.

PDIT (1993) FirstSTEP XG User’s Manual v1.0, Product Data Integration Technologies Inc, 3780 Kilroy Airport Way, Suite 430, Long Beach CA 90806, USA.

Pena-Mora, F., Sriram, D. and Logcher, R. (1993) SHARED-DRIMS: SHARED Design Recommendation-Intent Management System, MIT technical report, MIT, Massachusetts, USA.

Petri, C.A. (1976) Interpretation of Net Theory.

Popkin (1996) System Architect, Popkin Software & Systems Ltd., Leamington Spa, Warwickshire, UK.

Poyet, P., Dubois, A-M. and Delcambre, B. (1990) Artificial Intelligence Software Engineering in Building Engineering, Microcomputers in Civil Engineering, 5, pp. 167-205.

Poyet, P., Besse, G., Brisson, E., Debras, P., Zarli, A. and Monceyron, J. L. (1995) STEP Software Architectures for the Integration of Knowledge Based Systems, in proceedings of the CIB Workshop on Computers and Information in Construction, W78 and TC10, Modeling of Buildings Through their Life-cycle, CIB 180, Stanford University, Stanford, USA, 21-23 August, pp. 172-183.

Price, C. (1995) Guidelines for Implementing VML (view mapping language) Multi-schema Mapping Specifications in C++, BRANZ internal report, BRANZ, Porirua, New Zealand.

Qutaishat, M.A., Fiddian, N.J. and Gray, W.A. (1992) Association Merging in a Schema Meta-Integration System for Heterogeneous Object-Oriented Database Environment, University of Wales College of Cardiff, Cardiff, UK, pp. 209-226.

Ratcliff, M., Wang, C., Gautier, R.J. and Whittle, B.R. (1992) Dora - a structure oriented environment generator, Software Engineering Journal, 7(3), pp. 184-190.

Re*f*lex (1996) Re*f*lex: Building models to reflect reality, Re*f*lex Systems Headquarters, Kitsbury House, Kitsbury Road, Berkhamsted, Hertfordshire HP4 3EA, England.

Reiss, S.P. (1985) PECAN: Program Development Systems that Support Multiple Views, IEEE Transactions on Software Engineering, 11(3), March, pp. 267-285.

Sahlin, P., Bring, A. and Kolsaker, K. (1995) Future Trends of the Neutral Model Format (NMF), Building Simulation '95, Fourth International Conference Proceedings, Madison, Wisconsin, USA, 14-16 August, pp. 537-544.

SANZ (1977) Standards Association of New Zealand, NZS 4218P 1977: Minimum thermal insulation requirements for residential buildings, Standards Association of New Zealand, Wellington, New Zealand.

SANZ (1990) Standards Association of New Zealand, NZS 3604 1990: Code of practice for light timber frame buildings not requiring specific design, Standards Association of New Zealand, Wellington, New Zealand.

Scherer, R.J. (1995) EU-project COMBI: Objectives and overview, The First European Conference on Product and Process Modelling in the Building Industry, Dresden, Germany, 5-7 October 1994, Scherer, R. (ed), Balkema, Rotterdam, pp. 503-510.

Scitor (1995) Process Charter, Scitor Corporation, USA.

Staub, G., Nieva, A. and Schönefeld, F. (1994) PISA Information Modelling Language: EXPRESS-C, ISO TC184/SC4/WG5 working draft.

Stevens, W., Myers, G. and Constantine, L. (1974) Structured Design, IBM Systems Journal, 13(2).

Subrahmanian, E. and Westerberg, A. and Podnar, G. (1989) Towards a Shared Computational Environment for Engineering Design, Lecture Notes in Computer Science No. 492, pp. 200-228.

Swenson, K.D. (1993) A Visual Language to Describe Collaborative Work, Proceedings of the 1993 IEEE Symposium on Visual Languages, IEEE CS Press, pp. 298-303.

TES (1995) Tool Encapsulation Specification, Draft Standard, version 2.0.-2-103085, CAD Framework Initiative, Austin, Texas, USA.

TU Delft COMBINE Team and Amor, R. (1993) COMBINE Project Windows Modelling Approach, COMBINE 2 report COMB2-93-32, 26pp.

Ullman, D.U. (1982) Principles of Database Systems, Computer Science Press.

van der Lans, R.F. (1988) Introduction to SQL, Addison-Wesley Publishing Company.

van Horssen, J.J., Behage, B. and Mooij, M. (1994) Conversion, ESPRIT 7280 - ATLAS, Confidential report.

VEGA (1996) Virtual Enterprises using Groupware tools and distributed Architecture, VEGA project, ESPRIT, EP 20408, Brussels.

Venable, J.R. (1993) CoCoA: A Conceptual Data Modelling Approach for Complex Problem Domains, Ph.D. dissertation, Thomas J. Watson School of Engineering and Applied Science, State University of New York at Binghamton.

Venable, J.R. and Grundy, J.C. (1995) Integrating and Supporting Entity Relationship and Object Role Models, to appear in 14th International Object-oriented and Entity-Relationship Modelling Conference, Gold Coast (to be published in Lecture Notes in Computer Science).

Verhoef, M., Liebich, T. and Amor, R. (1995) A Multi-Paradigm Mapping Method Survey, Fischer, M.A., Law, K.H. and Luiten, B. (eds), CIB Workshop on Computers and Information in Construction, W78 and TC10, Modeling of Buildings Through their Life-cycle, Stanford University, Stanford, USA, 21-23 August, pp. 233-247.

Vogel, T. (1991) Configurable Graphical Editor: Users Guide, TNO Institute for Applied Computer Science, Delft, The Netherlands, 91-ITI-382, February.

Watson, A. and Crowley, A. (1995) CIMsteel integration standards, The First European Conference on Product and Process Modelling in the Building Industry, Dresden, Germany, 5-7 October 1994, Scherer, R. (ed), Balkema, Rotterdam, pp. 491-494.

Wen, J., Hardwick, M., Spooner, D.L., Schlenoff, C., Valois, J. and Bailey, I. (1996) EXPRESS-X Reference Manual, ISO TC184/SC4/WG5 Working Report, Rensselaer Polytechnic Institute, Troy, New York, USA.

Willems, P. (1988) A Meta-Topology for Product Modelling, CIB Meeting, Lund, Sweden, October, TNO-IBBC, PU-88-10-II.

Williams, M. (1990) Interfacing Building Design Tools with a Knowledge Base System, Honours Report, Victoria University of Wellington, New Zealand.

Wirth, N. (1977) What Can We Do about the Unnecessary Diversity of Notation for Syntactic Definitions?, Communications of the ACM, 20(11), November, pp. 822-823.

Wong, A., Sriram, D. and Logcher, R. (1992) SHARED: An Information Model for Cooperative Product Development, submitted to IEEE Computer.

Zarli, A. (1995) XP-RULE: A Language for the Representation of Knowledge, CSTB, Sophia Antipolis, France.