

# Using Multiple Representations Within a Viewpoint

Nigel James Stanger

A thesis submitted for the degree of  
Doctor of Philosophy  
at the University of Otago, Dunedin,  
New Zealand.

November 30, 1999



*Dedicated to the memory of my father.*



## Abstract

There are many different types of information to be considered when designing an information system, and a wide variety of modelling approaches and notations (or *representations*) have been developed to describe these different types of information. Some types of information are better expressed by some representations than others, so it is sensible to use multiple representations to describe a real-world phenomenon. Reconciling and integrating descriptions expressed using different representations is therefore an important part of the design process.

The objective of this research is to aid this reconciliation and integration within the context of information systems design. That is, to *facilitate the use of multiple modelling representations for describing a phenomenon*. To achieve this objective, the author has chosen an approach based upon translating descriptions of a phenomenon between different representations.

This thesis provides several important contributions in the area of information system design using multiple representations. Related work in the area is reviewed, and from this review is derived a terminology based on viewpoint-oriented methods that provides a consistent framework for the discussion of multiple representations.

Previous research into the use of multiple representations has focused on semantic data models. This is extended in this thesis to include diverse modelling representations such as functional dependencies and data flow modelling.

The process of translating between different representations is explored in depth, and several important issues identified. Translations are defined by a collection of rules that specify the mappings between constructs of representations. An abstract notation is developed for expressing these translations, and an extended version of Amor's (1997) View Mapping Language is defined for the purpose of building detailed rule specifications.

*Heuristics* and *enrichment* are proposed as two methods of improving the quality of translations. A measure for determining the relative quality of translations is developed to show that heuristics do indeed enhance translation quality.

In addition, a method is developed for using translations to highlight potential design inconsistencies by translating descriptions expressed using different representations into the same form and comparing them.



## Acknowledgements

The last four years have been a testing time, and this thesis could not have been completed without the help of a remarkable group of people.

First and foremost, I would like to thank my supervisor, Dr. Richard Pascoe. Without his vigilant supervision and constant ‘encouragement’ this document would probably not have been anywhere near as good as it is. I would also like to thank my second supervisor, Prof. Philip Sallis, for his continuing support and especially for granting me a semester’s leave from teaching in order to write most of this thesis. Without it this would have easily taken another year to complete.

I would like to thank to George Benwell for reading an earlier version of my ramblings and providing insightful comments and constructive criticism. I would like to thank Dr. Robert Amor for his help with VML, and Drs. John Grundy and John Hosking for their interest and encouragement. I would also like to thank my thesis examiners for their helpful comments.

A big hi to Kevin, Nicola, Mike, Judi, Roy, Matthew, Kaaren and Ruth for providing me with something at least vaguely resembling a social life during the last two years. Saturday night has been my only real ‘night off’ for a long time, and it’s great to be able to share that with friends. Thanks also go to Richard Kilgour and Brendon Sly for providing a source of music while I worked.

Donald Arseneau provided the macros to generate the  $\rightarrow$  and  $\leftrightarrow$  operators, and other useful  $\text{\TeX}$  and  $\text{\LaTeX}$  help; and Denis Girou provided answers to weird PSTricks questions.

Last, but not least, I would like to thank my mother and the other members of my family for their support and love. Thank you all.

Of course, the best part is that I no longer have to put up with people asking me “so is it finished yet?”.

This thesis was produced using  $\text{\OzTeX}$  on a Macintosh. Figures were produced using various  $\text{\LaTeX}$  packages, Microsoft PowerPoint and Excel 98, Claris MacDraw Pro and MicroFrontier’s Color-It!. Entity-relationship and data flow diagrams were produced using Visible Systems’ EasyCASE running under Windows NT.





# Contents

|  |              |
|--|--------------|
| <b>Abstract</b>  | <b>v</b>     |
| <b>Acknowledgements</b>  | <b>vii</b>   |
| <b>List of Tables</b>  | <b>xvii</b>  |
| <b>List of Figures</b>   | <b>xix</b>   |
| <b>List of Algorithms</b>  | <b>xxvii</b> |
| <b>Chapter 1 Introduction</b>                                    | <b>1</b>     |
| <b>Chapter 2 Related research</b>                                | <b>7</b>     |
| 2.1 Introduction . . . . .                                       | 7            |
| 2.2 Multiple representations for data modelling . . . . .        | 8            |
| 2.2.1 The multiple views approach . . . . .                      | 9            |
| 2.2.2 The multiple-data-model (MDM) approach . . . . .           | 12           |
| 2.2.3 The object-oriented rule-based (ORECOM) approach . . . . . | 13           |
| 2.3 Viewpoint-oriented methods . . . . .                         | 15           |
| 2.3.1 Origin of viewpoints . . . . .                             | 15           |
| 2.3.2 Viewpoint terminology . . . . .                            | 17           |
| 2.3.3 Viewpoint integration . . . . .                            | 20           |
| 2.4 Data translation methods . . . . .                           | 27           |
| 2.4.1 Translation quality . . . . .                              | 28           |
| 2.4.2 Translation performance . . . . .                          | 29           |
| 2.4.3 Interfacing strategies . . . . .                           | 30           |
| 2.4.4 Interface specification languages . . . . .                | 32           |
| 2.5 Summary . . . . .  | 35           |

|                  |  |           |
|------------------|--|-----------|
| <b>Chapter 3</b> | <b>Using multiple representations to describe a viewpoint</b>                          | <b>39</b> |
| 3.1              | Introduction . . . . .   | 39        |
| 3.2              | Extending the viewpoint framework . . . . .  | 42        |
| 3.2.1            | Representations . . . . .  | 42        |
| 3.2.2            | Descriptions . . . . .   | 44        |
| 3.2.3            | Constructs and elements . . . . .  | 46        |
| 3.3              | A notation to express representations, descriptions, constructs and elements . . . . . | 47        |
| 3.3.1            | Description and representation notation . . . . .                                      | 47        |
| 3.3.2            | Construct and element notation . . . . .   | 49        |
| 3.4              | Defining constructs within representations . . . . .                                   | 49        |
| 3.5              | Expressive power of representations . . . . .  | 55        |
| 3.6              | Maintaining consistency among descriptions . . . . .                                   | 58        |
| 3.7              | Summary . . . . .  | 64        |
| <br>             |  |           |
| <b>Chapter 4</b> | <b>Translating descriptions within a viewpoint</b>                                     | <b>67</b> |
| 4.1              | Introduction . . . . .   | 67        |
| 4.2              | Rules and heuristics . . . . .   | 68        |
| 4.2.1            | Specialisation of rules and heuristics . . . . .                                       | 71        |
| 4.3              | Properties of translations . . . . .   | 73        |
| 4.3.1            | Type . . . . .   | 74        |
| 4.3.2            | Completeness . . . . .   | 75        |
| 4.3.3            | Composition . . . . .  | 76        |
| 4.3.4            | Direction . . . . .  | 77        |
| 4.4              | Notations for specifying translations . . . . .  | 78        |
| 4.4.1            | Requirements for translation operators . . . . .                                       | 79        |
| 4.4.2            | A high-level notation for expressing translations . . . . .                            | 80        |
| 4.4.3            | Examples of translation decomposition . . . . .  | 84        |
| 4.5              | Highlighting potential viewpoint inconsistencies . . . . .                             | 86        |
| 4.6              | Improving translation quality . . . . .  | 91        |
| 4.7              | The translation process . . . . .  | 94        |
| 4.7.1            | Rule exclusion . . . . .   | 98        |

|   |   |            |
|---|---|------------|
| 4.8   | Summary . . . . .   | 103        |
| <b>Chapter 5 Definition of translations</b> |   | <b>107</b> |
| 5.1   | Introduction . . . . .  | 107        |
| 5.1.1                                       | Notation used in this chapter . . . . .   | 108        |
| 5.2   | Example viewpoints . . . . .  | 109        |
| 5.2.1                                       | The used cars viewpoint . . . . .   | 109        |
| 5.2.2                                       | The agricultural research institute viewpoint . . . . .   | 110        |
| 5.2.3                                       | The assessment marks viewpoint . . . . .  | 112        |
| 5.3   | $\mathfrak{R}_e(E-R, ERD_{Martin}) \rightleftharpoons \mathfrak{R}_r(Relational, SQL/92)$ . . . . . | 114        |
| 5.3.1                                       | Scheme-level rules . . . . .  | 116        |
| 5.3.2                                       | Additional scheme-level rules . . . . .   | 126        |
| 5.3.3                                       | Heuristics . . . . .  | 126        |
| 5.3.4                                       | Technique-level rules . . . . .   | 127        |
| 5.3.5                                       | Discussion . . . . .  | 129        |
| 5.4   | $\mathfrak{R}_f \rightarrow \mathfrak{R}_e / \mathfrak{R}_f \leftarrow \mathfrak{R}_e$ . . . . .    | 135        |
| 5.4.1                                       | Normalisation effect . . . . .  | 135        |
| 5.4.2                                       | Partial rules . . . . .   | 138        |
| 5.4.3                                       | Unidirectional and excluded rules . . . . .   | 138        |
| 5.4.4                                       | Observations . . . . .  | 139        |
| 5.5   | $\mathfrak{R}_e \rightleftharpoons \mathfrak{R}_d$ . . . . .  | 141        |
| 5.5.1                                       | Observations . . . . .  | 142        |
| 5.6   | Summary . . . . .   | 142        |
| <b>Chapter 6 Prototype implementation</b>   |   | <b>145</b> |
| 6.1   | Introduction . . . . .  | 145        |
| 6.2   | The implementation architecture of Swift . . . . .  | 146        |
| 6.2.1                                       | Implementation language(s) . . . . .  | 149        |
| 6.3   | The description modelling unit . . . . .  | 152        |
| 6.3.1                                       | Design issues . . . . .   | 152        |
| 6.3.2                                       | Implementation issues . . . . .   | 153        |
| 6.4   | The translation unit . . . . .  | 156        |
| 6.4.1                                       | Translation specification in Swift . . . . .  | 157        |

|  |  |            |
|--|--|------------|
| 6.4.2  | Rule evaluation in Swift . . . . .   | 159        |
| 6.4.3  | Implementation . . . . .   | 160        |
| 6.5  | The repository unit . . . . .  | 162        |
| 6.6  | Miscellaneous implementation issues . . . . .  | 170        |
| 6.6.1  | Logging of operations . . . . .  | 170        |
| 6.6.2  | Extensibility . . . . .  | 170        |
| 6.7  | Example of Swift in use . . . . .  | 171        |
| 6.7.1  | The effect of heuristics on translations . . . . .                                     | 175        |
| 6.8  | Summary . . . . .  | 175        |
| <b>Chapter 7 Translations using VML-S</b>              |  | <b>179</b> |
| 7.1  | Introduction . . . . .   | 179        |
| 7.2  | An overview of VML . . . . .   | 180        |
| 7.2.1  | Using VML to specify translations between representations . . . . .                    | 184        |
| 7.3  | Outstanding issues with VML . . . . .  | 184        |
| 7.3.1  | Unidirectional rules . . . . .   | 185        |
| 7.3.2  | Rule exclusion . . . . .   | 190        |
| 7.3.3  | Translating construct lists . . . . .  | 192        |
| 7.3.4  | Using the same construct multiple times . . . . .                                      | 193        |
| 7.4  | Converting the abstract notation into VML-S . . . . .                                  | 194        |
| 7.4.1  | Specification of $\mathfrak{R}_e \rightleftharpoons \mathfrak{R}_r$ rule S8 . . . . .  | 197        |
| 7.5  | Extensions to the translation process . . . . .  | 201        |
| 7.5.1  | Building subsumption/exclusion graphs . . . . .  | 205        |
| 7.5.2  | Modified algorithms for mapping list structures . . . . .                              | 207        |
| 7.6  | Summary . . . . .  | 210        |
| <b>Chapter 8 Measuring the quality of translations</b> |  | <b>213</b> |
| 8.1  | Introduction . . . . .   | 213        |
| 8.2  | Relative information capacity (Hull, 1986) . . . . .                                   | 214        |
| 8.2.1  | Building schema intension graphs (Miller, Ioannidis and Ramakrishnan, 1994b) . . . . . | 215        |
| 8.2.2  | Comparing schema intension graphs (Miller et al., 1994b) . . . . .                     | 218        |
| 8.3  | Categorising expressive overlap using schema intension graphs . . . . .                | 223        |

|                   |   |            |
|-------------------|---|------------|
| 8.3.1             | Example of determining expressive overlap . . . . .       | 227        |
| 8.3.2             | Analysis . . . . .  | 230        |
| 8.4               | Measuring the relative quality of translations . . . . .  | 232        |
| 8.4.1             | Initial analysis . . . . .                                | 234        |
| 8.4.2             | Improving the relative quality measurement . . . . .      | 236        |
| 8.5               | The effect of heuristics on translation quality . . . . . | 238        |
| 8.6               | Summary . . . . .   | 243        |
| <b>Chapter 9</b>  | <b>Evaluation of the proposed modelling approach</b>      | <b>245</b> |
| 9.1               | Introduction . . . . .                                    | 245        |
| 9.2               | Case study . . . . .                                      | 246        |
| 9.2.1             | Discussion . . . . .                                      | 251        |
| 9.3               | Novelty of the approach . . . . .                         | 252        |
| 9.3.1             | Survey of ‘conventional’ CASE tools . . . . .             | 253        |
| 9.3.2             | The MViews approach . . . . .                             | 255        |
| 9.3.3             | The MDM approach . . . . .                                | 258        |
| 9.3.4             | The ORECOM approach . . . . .                             | 261        |
| 9.3.5             | Discussion . . . . .                                      | 264        |
| 9.4               | Practicability of the approach . . . . .                  | 267        |
| 9.4.1             | Tractability of translations . . . . .                    | 268        |
| 9.4.2             | Complexity testing . . . . .                              | 273        |
| 9.5               | Summary . . . . .   | 278        |
| <b>Chapter 10</b> | <b>Further research</b>                                   | <b>279</b> |
| 10.1              | Introduction . . . . .                                    | 279        |
| 10.2              | Representation issues . . . . .                           | 280        |
| 10.3              | Quality improvement . . . . .                             | 282        |
| 10.4              | Implementation issues . . . . .                           | 286        |
| 10.5              | Schema generation from multiple descriptions . . . . .    | 289        |
| 10.6              | Consistency maintenance . . . . .                         | 291        |
| 10.7              | Outstanding issues with VML-S . . . . .                   | 294        |
| 10.8              | Other issues . . . . .                                    | 295        |
| 10.9              | Summary . . . . .   | 297        |

|   |            |
|---|------------|
| <b>Chapter 11 Conclusion</b>  | <b>299</b> |
| 11.1 Improving the depth and detail of a viewpoint . . . . .                        | 299        |
| 11.2 Using translations to facilitate the use of multiple representations . . . . . | 300        |
| 11.3 Terminology framework . . . . .  | 301        |
| 11.4 Translation specification . . . . .  | 302        |
| 11.5 Highlighting potential inconsistencies within a viewpoint . . . . .            | 302        |
| 11.6 Improving translation quality . . . . .  | 303        |
| 11.7 Novelty and practicability of the approach . . . . .                           | 305        |
| 11.7.1 Practicability . . . . .   | 306        |
| 11.8 Closing remarks . . . . .  | 307        |
| <b>References</b>   | <b>309</b> |
| <b>Glossary</b>   | <b>327</b> |
| <b>Appendix A Notations and terminology</b>   | <b>335</b> |
| A.1 Introduction . . . . .  | 335        |
| A.2 Martin/EasyCASE ERD notation . . . . .  | 335        |
| A.3 Gane & Sarson/EasyCASE DFD notation . . . . .                                   | 335        |
| A.4 Original translation notation . . . . .   | 335        |
| A.4.1 Description translations . . . . .  | 336        |
| A.4.2 Constructs . . . . .  | 337        |
| A.4.3 Construct translations . . . . .  | 338        |
| A.4.4 Heuristic construct translations . . . . .                                    | 339        |
| A.4.5 Composite and atomic translations . . . . .                                   | 339        |
| A.5 CASE tool survey . . . . .  | 340        |
| A.5.1 Visible Systems EasyCASE . . . . .  | 340        |
| A.5.2 Visible Systems Visible Analyst . . . . .                                     | 341        |
| A.5.3 Visible Systems EasyER/EasyOBJECT . . . . .                                   | 343        |
| A.5.4 Sybase Defit . . . . .  | 345        |

|                   |  |            |
|-------------------|--|------------|
| <b>Appendix B</b> | <b>Modifications to Smith’s Method</b>   | <b>347</b> |
| B.1               | Introduction . . . . .   | 347        |
| B.2               | Smith’s Method — an overview . . . . .   | 348        |
| B.2.1             | The notation . . . . .   | 348        |
| B.2.2             | Deriving a set of relations from an FDD . . . . .  | 349        |
| B.3               | The problem with deriving foreign keys . . . . .   | 352        |
| B.4               | The solution . . . . .   | 354        |
| B.5               | Example — university marks . . . . .   | 356        |
| <b>Appendix C</b> | <b>Example viewpoints</b>  | <b>359</b> |
| C.1               | Used cars viewpoint . . . . .  | 359        |
| C.2               | Agricultural research institute viewpoint . . . . .  | 364        |
| C.3               | Assessment marks viewpoint . . . . .   | 369        |
| <b>Appendix D</b> | <b>Technique and representation definitions</b>  | <b>375</b> |
| D.1               | Introduction . . . . .   | 375        |
| D.2               | Entity-relationship technique . . . . .  | 376        |
| D.2.1             | Martin ERD definition . . . . .  | 379        |
| D.3               | Functional dependency technique . . . . .  | 380        |
| D.3.1             | Smith FDD definition . . . . .   | 383        |
| D.4               | Relational technique . . . . .   | 384        |
| D.4.1             | SQL/92 definition . . . . .  | 386        |
| D.5               | Data flow modelling technique . . . . .  | 389        |
| D.5.1             | Gane & Sarson DFD definition . . . . .   | 391        |
| <b>Appendix E</b> | <b>Additional translations</b>   | <b>395</b> |
| E.1               | $\mathfrak{R}_f \rightarrow \mathfrak{R}_e / \mathfrak{R}_f \leftarrow \mathfrak{R}_e$ . . . . . | 395        |
| E.1.1             | Technique-level rules . . . . .  | 396        |
| E.1.2             | Scheme-level rules . . . . .   | 398        |
| E.1.3             | Heuristics . . . . .   | 408        |
| E.1.4             | Expressive overlap . . . . .   | 410        |
| E.1.5             | Relative quality . . . . .   | 410        |
| E.2               | $\mathfrak{R}_e \rightleftharpoons \mathfrak{R}_d$ . . . . .                                     | 414        |

|   |   |            |
|---|---|------------|
| E.2.1   | Technique-level rules . . . . .   | 414        |
| E.2.2   | Scheme-level rules . . . . .  | 414        |
| E.2.3   | Heuristics . . . . .  | 415        |
| E.2.4   | Expressive overlap . . . . .  | 416        |
| E.2.5   | Relative quality . . . . .  | 416        |
| E.3   | $\mathfrak{R}_e \rightleftharpoons \mathfrak{R}_r$ . . . . .  | 418        |
| E.3.1   | Expressive overlap . . . . .  | 418        |
| E.3.2   | Relative quality . . . . .  | 421        |
| <b>Appendix F VML-S syntax and specifications</b> |   | <b>423</b> |
| F.1   | VML-S BNF syntax definition . . . . .   | 423        |
| F.2   | Example of a complete VML-S translation . . . . .   | 427        |
| F.3   | Full VML-S translation specifications . . . . .   | 435        |
| F.3.1   | $\mathfrak{R}_e(E-R, ERD_{Martin}) \rightleftharpoons \mathfrak{R}_r(Relational, SQL/92)$ . . . . .                                       | 435        |
| F.3.2   | $\mathfrak{R}_f(FuncDep, FDD_{Smith}) \rightarrow \mathfrak{R}_e(E-R, ERD_{Martin}) / \mathfrak{R}_f \leftarrow \mathfrak{R}_e$ . . . . . | 440        |
| F.3.3   | $\mathfrak{R}_e(E-R, ERD_{Martin}) \rightleftharpoons \mathfrak{R}_d(DataFlow, DFD_{G\&S})$ . . . . .                                     | 448        |
| <b>Appendix G The Swift repository</b>            |   | <b>451</b> |
| G.1   | Glossary of representation tags . . . . .   | 451        |
| G.1.1   | $\mathfrak{R}_f(FuncDep, FDD_{Smith})$ . . . . .  | 451        |
| G.1.2   | $\mathfrak{R}_e(E-R, ERD_{Martin})$ . . . . .   | 452        |
| G.1.3   | $\mathfrak{R}_d(DataFlow, DFD_{G\&S})$ . . . . .  | 454        |
| G.1.4   | $\mathfrak{R}_r(Relational, SQL/92)$ . . . . .  | 455        |
| G.2   | Full repository schema . . . . .  | 456        |
| G.2.1   | repository.sql . . . . .  | 456        |
| G.2.2   | functions.sql . . . . .   | 459        |
| G.2.3   | staticdata.sql . . . . .  | 460        |
| <b>Appendix H Swift class hierarchy</b>           |   | <b>463</b> |



# List of Tables

|     |   |     |
|-----|---|-----|
| 2.1 | Approaches to resolving viewpoint conflicts . . . . .   | 25  |
| 3.1 | Examples of construct and element expressions . . . . .   | 50  |
| 3.2 | Construct properties of the entity-relationship technique . . . . .   | 54  |
| 3.3 | Construct properties of the representation $\mathfrak{R}_e(E-R, ERD_{Martin})$ . . . . .  | 55  |
| 3.4 | Summary of representation and description terminology . . . . .   | 65  |
| 3.5 | Summary of representation and description notation . . . . .  | 66  |
| 4.1 | Summary of representation, description and translation notation . . . . .   | 104 |
| 5.1 | Summary of translations . . . . .   | 108 |
| 5.2 | Correspondences between the properties of the MARTINREGULAREN-<br>TITY and SQL92TABLE constructs . . . . .  | 117 |
| 5.3 | Summary of $\mathfrak{R}_e(E-R, ERD_{Martin}) \rightleftharpoons \mathfrak{R}_r(Relational, SQL/92)$ rules . . . . .  | 130 |
| 5.4 | Summary of $\mathfrak{R}_f \rightarrow \mathfrak{R}_e/\mathfrak{R}_f \leftarrow \mathfrak{R}_e$ rules . . . . .   | 136 |
| 5.5 | Summary of $\mathfrak{R}_e \rightleftharpoons \mathfrak{R}_d$ rules . . . . .   | 141 |
| 6.1 | Features of Java, C++ and Tcl/Tk with respect to implementing Swift . . . . .   | 150 |
| 6.2 | Correspondence between repository entities and Swift classes . . . . .  | 165 |
| 6.3 | Repository database schema (PostgreSQL) . . . . .   | 168 |
| 7.1 | VML-S specification of $\mathfrak{R}_e \rightleftharpoons \mathfrak{R}_r$ rule S8 . . . . .   | 202 |
| 8.1 | Summary of SIG transformations . . . . .  | 219 |
| 8.2 | Relative quality measurements for the $\mathfrak{R}_e \rightleftharpoons \mathfrak{R}_d$ translation . . . . .  | 234 |
| 8.3 | Relative quality measurements for the $\mathfrak{R}_f \rightarrow \mathfrak{R}_e/\mathfrak{R}_f \leftarrow \mathfrak{R}_e$ translation<br>using the modified measurement method . . . . . | 238 |
| 8.4 | Relative quality measurements for the $\mathfrak{R}_f \rightarrow \mathfrak{R}_e/\mathfrak{R}_f \leftarrow \mathfrak{R}_e$ translation<br>(with heuristics) . . . . .                     | 241 |

|     |  |     |
|-----|--|-----|
| 9.1 | Attributes for $D_1(V_{conf}, DataFlow, DFD_{G\&S})$ . . . . .   | 247 |
| 9.2 | Summary of ‘conventional’ CASE tool features . . . . .   | 254 |
| 9.3 | Comparison of research approaches . . . . .  | 267 |
| 9.4 | Summary of test viewpoints . . . . .   | 274 |
|     |  |     |
| A.1 | Summary of the Martin/EasyCASE ERD notation . . . . .  | 336 |
| A.2 | Martin/EasyCASE notation for relationship cardinalities . . . . .  | 336 |
| A.3 | Summary of the Gane & Sarson/EasyCASE DFD notation . . . . .   | 337 |
| A.4 | Examples of constructs . . . . .   | 338 |
| A.5 | Representations supported by EasyCASE . . . . .  | 342 |
| A.6 | Representations supported by Visible Analyst . . . . .   | 343 |
| A.7 | Representations supported by EasyER/EasyOBJECT . . . . .   | 344 |
| A.8 | Representations supported by Deft . . . . .  | 345 |
| A.9 | Representations supported by MetaEdit . . . . .  | 346 |
|     |  |     |
| D.1 | Construct properties for the E-R technique . . . . .   | 378 |
| D.2 | Construct properties of $\mathfrak{R}_e(E-R, ERD_{Martin})$ . . . . .  | 381 |
| D.3 | Construct properties of the functional dependency technique . . . . .  | 383 |
| D.4 | Construct properties of $\mathfrak{R}_f(FuncDep, FDD_{Smith})$ . . . . .   | 385 |
| D.5 | Construct properties of the relational technique . . . . .   | 387 |
| D.6 | Construct properties of $\mathfrak{R}_r(Relational, SQL/92)$ . . . . .   | 389 |
| D.7 | Construct properties of the data flow modelling technique . . . . .  | 391 |
| D.8 | Construct properties of $\mathfrak{R}_d(DataFlow, DFD_{G\&S})$ . . . . .   | 393 |
|     |  |     |
| E.1 | Relative quality measurements for the $\mathfrak{R}_f \rightarrow \mathfrak{R}_e / \mathfrak{R}_f \leftarrow \mathfrak{R}_e$ translation . . . . . | 412 |
| E.2 | Relative quality measurements for the $\mathfrak{R}_e \rightleftharpoons \mathfrak{R}_d$ translation . . . . .                                     | 418 |
| E.3 | Relative quality measurements for the $\mathfrak{R}_e \rightleftharpoons \mathfrak{R}_r$ translation . . . . .                                     | 421 |
|     |  |     |
| F.1 | Initial element lists for the Sheep entity . . . . .   | 430 |
| F.2 | Filtered element combinations for the Sheep entity . . . . .   | 430 |
| F.3 | Generating element combinations for rule S12 . . . . .   | 431 |
| F.4 | Summary of mappings in the example translation . . . . .   | 434 |

# List of Figures

|      |   |    |
|------|---|----|
| 2.1  | Relationship between perspectives, viewpoints and representations . . .   | 18 |
| 2.2  | Possible states for a library book . . . . .  | 23 |
| 2.3  | Interfacing strategies . . . . .  | 30 |
| 3.1  | Extending the perspective/viewpoint/representation framework . . . .  | 43 |
| 3.2  | Multiple schemes within a technique . . . . .   | 43 |
| 3.3  | Four descriptions of the same viewpoint . . . . .   | 45 |
| 3.4  | Properties of a construct . . . . .   | 47 |
| 3.5  | Definition of the entity-relationship technique . . . . .   | 53 |
| 3.6  | Definition of the representation $\mathfrak{R}_e(E-R, ERD_{Martin})$ . . . . .  | 54 |
| 3.7  | Unique and shared constructs . . . . .  | 56 |
| 3.8  | Four categories of expressive overlap . . . . .   | 57 |
| 3.9  | Asymmetric expressive overlap . . . . .   | 57 |
| 3.10 | Sensitivity of a description to changes . . . . .   | 61 |
| 3.11 | Unexpected structural changes caused by a translation . . . . .   | 62 |
| 4.1  | Example of applying a heuristic . . . . .   | 70 |
| 4.2  | Specialisation of technique-level rules . . . . .   | 72 |
| 4.3  | A trivial scheme translation . . . . .  | 75 |
| 4.4  | A complex VML-G specification . . . . .   | 79 |
| 4.5  | Specialisation of translation operators . . . . .   | 83 |
| 4.6  | Translating a MARTINREGULARENTITY to a functional dependency di-<br>agram . . . . .   | 85 |
| 4.7  | Decomposing a translation . . . . .   | 85 |
| 4.8  | Consistency checking strategies using different representations . . . . .   | 87 |
| 4.9  | Two potentially inconsistent descriptions of a viewpoint . . . . .  | 89 |
| 4.10 | Two potentially inconsistent descriptions of a viewpoint after translat-<br>ing them into the same representation . . . . . | 90 |

|      |   |     |
|------|---|-----|
| 4.11 | The translation process . . . . .   | 95  |
| 4.12 | Translating an FDD description to an ERD description . . . . .  | 97  |
| 4.13 | Applying multiple rules to the same source structure . . . . .  | 99  |
| 4.14 | Results of applying multiple rules to the same source structure . . . . .   | 100 |
| 4.15 | 'Forward' subsumption/exclusion graph for the example translation . .   | 102 |
|      |   |     |
| 5.1  | Martin E-R description of the used cars viewpoint . . . . .   | 110 |
| 5.2  | SQL/92 description of the used cars viewpoint . . . . .   | 111 |
| 5.3  | Normalised Martin E-R description of the agricultural research institute<br>viewpoint . . . . .   | 112 |
| 5.4  | SQL/92 description of the agricultural research institute viewpoint . . .   | 113 |
| 5.5  | Normalised Martin E-R description of the assessment marks viewpoint .   | 114 |
| 5.6  | SQL/92 description of the assessment marks viewpoint . . . . .  | 115 |
| 5.7  | Translating a regular entity to and from SQL/92 . . . . .   | 117 |
| 5.8  | Translating a one-to-many relationship (mandatory-optional) to and from<br>SQL/92 . . . . .   | 119 |
| 5.9  | Translating a one-to-one relationship (mandatory-mandatory) to and from<br>SQL/92 . . . . .   | 120 |
| 5.10 | Translating a one-to-one relationship (optional-mandatory) to and from<br>SQL/92 . . . . .  | 121 |
| 5.11 | Translating a many-to-many relationship to and from SQL/92 . . . . .  | 122 |
| 5.12 | Translating a type hierarchy to SQL/92 . . . . .  | 123 |
| 5.13 | Translating a one-to-many relationship (optional-optional) to and from<br>SQL/92 . . . . .  | 125 |
| 5.14 | Translating a weak entity and dependent relationship to SQL/92 . . . .  | 125 |
| 5.15 | Translating a one-to-one relationship (optional-optional) to and from<br>SQL/92 . . . . .   | 127 |
| 5.16 | Loss of information in $\mathfrak{R}_e \leftrightarrow \mathfrak{R}_r$ . . . . .  | 131 |
| 5.17 | Subsumption/exclusion graphs for the $\mathfrak{R}_e \rightleftharpoons \mathfrak{R}_r$ translation . . . . .                             | 132 |
| 5.18 | SQL/92 constraints not generated by EasyCASE . . . . .  | 133 |
| 5.19 | Normalisation caused by the $\mathfrak{R}_f \rightarrow \mathfrak{R}_e / \mathfrak{R}_f \leftarrow \mathfrak{R}_e$ translation . . . . .  | 137 |
| 5.20 | Subsumption/exclusion graphs for the $\mathfrak{R}_f \rightarrow \mathfrak{R}_e / \mathfrak{R}_f \leftarrow \mathfrak{R}_e$ translation . | 139 |

|      |   |     |
|------|---|-----|
| 5.21 | Loss of information when translating from $\mathfrak{R}_e$ to $\mathfrak{R}_f$ . . . . .                      | 140 |
| 5.22 | Subsumption/exclusion graphs for the $\mathfrak{R}_e \rightleftharpoons \mathfrak{R}_d$ translation . . . . . | 141 |
| 6.1  | Swift's implementation architecture . . . . .   | 148 |
| 6.2  | The user interface to Swift . . . . .   | 154 |
| 6.3  | Java class hierarchy to implement $\mathfrak{R}_d(\text{DataFlow}, \text{DFD}_{G\&S})$ in Swift . . . . .     | 155 |
| 6.4  | The Translate menu . . . . .  | 161 |
| 6.5  | Enrichment during a translation . . . . .   | 162 |
| 6.6  | Rearranging the symbols of a description after a translation . . . . .  | 163 |
| 6.7  | Structure of the repository . . . . .   | 166 |
| 6.8  | Loading a viewpoint in Swift . . . . .  | 171 |
| 6.9  | Loading a description in Swift . . . . .  | 172 |
| 6.10 | The Translate menu for an FDD . . . . .   | 172 |
| 6.11 | The beginning and end of the translation process . . . . .  | 173 |
| 6.12 | The target description . . . . .  | 174 |
| 6.13 | Effect of heuristics on the $\mathfrak{R}_f \rightarrow \mathfrak{R}_e$ translation . . . . .                 | 175 |
| 6.14 | Effect of heuristics on the $\mathfrak{R}_e \rightarrow \mathfrak{R}_d$ translation . . . . .                 | 176 |
| 7.1  | Example of a VML mapping specification . . . . .  | 181 |
| 7.2  | BNF syntax of the VML-S <i>inter_class</i> definition . . . . .   | 185 |
| 7.3  | Unidirectional translations that cannot be fully specified using VML . . . . .                                | 187 |
| 7.4  | The <code>direction</code> clause . . . . .   | 188 |
| 7.5  | New directional equivalence operators . . . . .   | 189 |
| 7.7  | An example of rule exclusion . . . . .  | 190 |
| 7.8  | The <code>label</code> clause . . . . .   | 191 |
| 7.9  | The <code>excludes</code> clause . . . . .  | 192 |
| 7.10 | Aliases for constructs in the <i>inter_class</i> header . . . . .   | 195 |
| 7.11 | Translating many-to-many relationships ( $\mathfrak{R}_e \rightleftharpoons \mathfrak{R}_r$ ) . . . . .       | 198 |
| 7.12 | Amor's four-pass translation process . . . . .  | 203 |
| 7.13 | The modified translation process . . . . .  | 205 |
| 8.1  | E-R description of the used cars viewpoint and corresponding schema<br>intension graph . . . . .              | 217 |

|      |   |     |
|------|---|-----|
| 8.2  | Alternate E-R description of the used cars viewpoint and corresponding SIG . . . . .                                      | 221 |
| 8.3  | Transforming a subgraph of a SIG . . . . .  | 222 |
| 8.4  | SIGs for $\mathfrak{R}_e(E-R, ERD_{Martin})$ and $\mathfrak{R}_d(DataFlow, DFD_{G\&S})$ . . . . .                         | 228 |
| 8.5  | Transforming subgraphs of the $\mathfrak{R}_d$ SIG . . . . .  | 229 |
| 8.6  | Isomorphism between SIG subgraphs . . . . .   | 231 |
| 8.7  | Expressive overlap between $\mathfrak{R}_e$ and $\mathfrak{R}_d$ . . . . .  | 233 |
| 8.8  | Summary of initial relative translation quality measurements (without heuristics) . . . . .                               | 235 |
| 8.9  | Applying the modified relative quality measurement (ignoring heuristics)  | 237 |
| 8.10 | Summary of modified relative translation quality measurements . . . . .   | 239 |
| 8.11 | Applying the relative quality measurement (with heuristics) . . . . .   | 240 |
| 8.12 | Impact of heuristics on relative translation quality . . . . .  | 242 |
| 9.1  | DFD description $D_1(V_{conf}, DataFlow, DFD_{G\&S})$ for the process of assigning conference sessions to rooms . . . . . | 247 |
| 9.2  | ERD description $D_2(V_{conf}, E-R, ERD_{Martin})$ produced by translating description $D_1$ . . . . .                    | 248 |
| 9.3  | ERD description $D_2'$ produced by modifying $D_2$ . . . . .  | 249 |
| 9.4  | FDD description $D_3(V_{conf}, FuncDep, FDD_{Smith})$ . . . . .   | 250 |
| 9.5  | ERD description $D_4(V_{conf}, E-R, ERD_{Martin})$ produced by translating description $D_3$ . . . . .                    | 250 |
| 9.6  | ERD description $D_2''$ produced by correcting inconsistencies . . . . .  | 251 |
| 9.7  | The interconnectedness of elements in a description . . . . .   | 271 |
| 9.8  | Descriptions used for translation time testing . . . . .  | 272 |
| 9.9  | Results of Swift translation time testing . . . . .   | 276 |
| 10.1 | $\mathfrak{R}_r(Relational, SQL/92)$ definition using CoCoA . . . . .   | 282 |
| 10.2 | $\mathfrak{R}_r(Relational, SQL/92)$ as defined in this thesis . . . . .  | 283 |
| 10.3 | Schema generation from multiple representations . . . . .   | 289 |
| 10.4 | Progressive refinement of a generated schema . . . . .  | 290 |
| 10.5 | Effects of asynchronous translations . . . . .  | 293 |
| 10.6 | Two description fragments that are consistent but not equivalent . . . . .  | 295 |

|      |  |     |
|------|--|-----|
| B.1  | Example of a dependency-list statement . . . . .   | 348 |
| B.2  | Attributes and bubbles . . . . .   | 349 |
| B.3  | Single- and multivalued dependencies . . . . .   | 349 |
| B.4  | Multiple bubbles and domain flags . . . . .  | 350 |
| B.5  | Deriving a relation from a single-valued dependency . . . . .                                | 350 |
| B.6  | Deriving a relation from an end-key dependency . . . . .                                     | 351 |
| B.7  | Correcting an impracticable FDD . . . . .  | 352 |
| B.8  | Foreign keys that cannot be derived using the existing rules . . . . .                       | 353 |
| B.9  | Derivation of invalid foreign keys . . . . .   | 354 |
| B.10 | 'Target' attribute domain flag notation . . . . .  | 355 |
|      |  |     |
| C.1  | E-R description of the used cars viewpoint (unnormalised) . . . . .                          | 360 |
| C.2  | E-R description of the used cars viewpoint (normalised) . . . . .                            | 361 |
| C.3  | Functional dependency description of the used cars viewpoint . . . . .                       | 362 |
| C.4  | SQL/92 description of the used cars viewpoint . . . . .                                      | 363 |
| C.5  | Data flow description of the used cars viewpoint . . . . .                                   | 364 |
| C.6  | E-R description of the agricultural research institute viewpoint (unnormalised) . . . . .    | 365 |
| C.7  | E-R description of the agricultural research institute viewpoint (normalised) . . . . .      | 365 |
| C.8  | Functional dependency description of the agricultural research institute viewpoint . . . . . | 367 |
| C.9  | SQL/92 description of the agricultural research institute viewpoint . . . . .                | 368 |
| C.10 | Data flow description of the agricultural research institute viewpoint . . . . .             | 369 |
| C.11 | E-R description of the assessment marks viewpoint (unnormalised) . . . . .                   | 370 |
| C.12 | E-R description of the assessment marks viewpoint (normalised) . . . . .                     | 370 |
| C.13 | Functional dependency description of the assessment marks viewpoint . . . . .                | 372 |
| C.14 | Data flow description of the assessment marks viewpoint . . . . .                            | 372 |
| C.15 | SQL/92 description of the assessment marks viewpoint . . . . .                               | 373 |
|      |  |     |
| D.1  | Definition of the E-R technique . . . . .  | 377 |
| D.2  | Definition of the representation $\mathfrak{R}_e(E-R, ERD_{Martin})$ . . . . .               | 379 |
| D.3  | Definition of the functional dependency technique . . . . .                                  | 381 |

|      |  |     |
|------|--|-----|
| D.4  | Definition of the representation $\mathfrak{R}_f(\text{FuncDep}, \text{FDD}_{\text{Smith}})$ . . . . . | 384 |
| D.5  | Definition of the relational technique . . . . .   | 386 |
| D.6  | Definition of the representation $\mathfrak{R}_r(\text{Relational}, \text{SQL}/92)$ . . . . .          | 388 |
| D.7  | Definition of the data flow modelling technique . . . . .  | 390 |
| D.8  | Definition of the representation $\mathfrak{R}_d(\text{DataFlow}, \text{DFD}_{G\&S})$ . . . . .        | 392 |
| E.1  | Translating a single-valued dependency to and from a Martin ERD (rule S1) . . . . .                    | 398 |
| E.2  | Translating a multivalued dependency to and from a Martin ERD (rule S2) . . . . .                      | 399 |
| E.3  | Translating a single-key bubble between an FDD and an ERD (rule S3) . . . . .                          | 399 |
| E.4  | Translating an isolated bubble to and from a Martin ERD (rule S4) . . . . .                            | 400 |
| E.5  | Translating a weak entity to an FDD (rule S7) . . . . .  | 400 |
| E.6  | Translating a weak entity to an FDD (rule S8) . . . . .  | 401 |
| E.7  | Translating a weak entity to an FDD (rule S9) . . . . .  | 402 |
| E.8  | Translating a weak entity to an FDD (rule S10) . . . . .   | 402 |
| E.9  | Translating one-to-one relationships between an ERD and an FDD (rule S11) . . . . .                    | 403 |
| E.10 | Translating one-to-many relationships between an ERD and an FDD (rule S12) . . . . .                   | 404 |
| E.11 | Translating many-to-many relationships from an ERD to an FDD (rule S13) . . . . .                      | 405 |
| E.12 | Translating a domain flag from an FDD to an ERD (rule S14) . . . . .                                   | 406 |
| E.13 | Translating a domain flag from an FDD to an ERD (rule S15) . . . . .                                   | 406 |
| E.14 | Translating a contained single-key bubble from an FDD to an ERD (rule S16) . . . . .                   | 407 |
| E.15 | Translating a contained isolated bubble from an FDD to an ERD (rule S17)                               | 408 |
| E.16 | Translating a contained single-key bubble from an FDD to an ERD (rule S18) . . . . .                   | 408 |
| E.17 | Translating a contained isolated bubble from an FDD to an ERD (rule S19)                               | 409 |
| E.18 | Translating a circular multi-valued dependency from an FDD to an ERD (heuristic H1) . . . . .          | 409 |



|      |   |     |
|------|---|-----|
| E.19 | Deriving a type hierarchy from an FDD (heuristic H2) . . . . .  | 410 |
| E.20 | SIGs for $\mathfrak{R}_f(\text{FuncDep}, \text{FDD}_{\text{Smith}})$ and $\mathfrak{R}_e(E\text{-}R, \text{ERD}_{\text{Martin}})$ . . . . .       | 411 |
| E.21 | Transformed SIG for $\mathfrak{R}_f(\text{FuncDep}, \text{FDD}_{\text{Smith}})$ . . . . .   | 411 |
| E.22 | Expressive overlap for $\mathfrak{R}_e(E\text{-}R, \text{ERD}_{\text{Martin}})$ and $\mathfrak{R}_f(\text{FuncDep}, \text{FDD}_{\text{Smith}})$ . | 412 |
| E.23 | Relative quality analysis for $\mathfrak{R}_e \rightarrow \mathfrak{R}_f / \mathfrak{R}_e \leftarrow \mathfrak{R}_f$ . . . . .                    | 413 |
| E.24 | Translating an associative entity that represents an activity into a data<br>process and data store (heuristic H1) . . . . .                      | 415 |
| E.25 | Translating data flows into a relationship (heuristic H2) . . . . .   | 416 |
| E.26 | Relative quality analysis for $\mathfrak{R}_e \rightleftharpoons \mathfrak{R}_d$ . . . . .  | 417 |
| E.27 | SIGs for $\mathfrak{R}_e(E\text{-}R, \text{ERD}_{\text{Martin}})$ and $\mathfrak{R}_r(\text{Relational}, \text{SQL}/92)$ . . . . .                | 419 |
| E.28 | Transformed SIG for $\mathfrak{R}_r(\text{Relational}, \text{SQL}/92)$ . . . . .  | 419 |
| E.29 | Expressive overlap for $\mathfrak{R}_e(E\text{-}R, \text{ERD}_{\text{Martin}})$ and $\mathfrak{R}_r(\text{Relational}, \text{SQL}/92)$ .          | 420 |
| E.30 | Relative quality analysis for $\mathfrak{R}_e \rightleftharpoons \mathfrak{R}_r$ . . . . .  | 422 |
|      |   |     |
| F.1  | Complex internal structure of the Experiment entity . . . . .   | 429 |
| F.2  | Translating elements associated with the Sheep element . . . . .  | 430 |
| F.3  | Translating elements associated with the Paddock_detail element . . . . .   | 432 |
| F.4  | Further mappings in the example translation: I . . . . .  | 433 |
| F.5  | Further mappings in the example translation: II . . . . .   | 433 |
| F.6  | Final FDD description produced by the example translation . . . . .   | 434 |
|      |   |     |
| H.1  | Key for interpreting class diagrams . . . . .   | 464 |
| H.2  | Swift class hierarchy: swift.dd package . . . . .   | 464 |
| H.3  | Swift class hierarchy: swift.event package . . . . .  | 464 |
| H.4  | Swift class hierarchy: swift.model package . . . . .  | 464 |
| H.5  | Swift class hierarchy: swift.repn package . . . . .   | 465 |
| H.6  | Swift class hierarchy: swift.repn.erm_mrtc package . . . . .  | 465 |
| H.7  | Swift class hierarchy: swift.repn.fdepsmit package . . . . .  | 466 |
| H.8  | Swift class hierarchy: swift.repn.procgnsn package . . . . .  | 467 |
| H.9  | Swift class hierarchy: swift.trans packages . . . . .   | 467 |
| H.10 | Swift class hierarchy: swift.ui package . . . . .   | 468 |
| H.11 | Swift class hierarchy: swift.util package . . . . .   | 468 |



# List of Algorithms

|     |  |     |
|-----|--|-----|
| 7.1 | Modified translation algorithm . . . . .                           | 204 |
| 7.2 | Build a subsumption/exclusion graph for a translation . . . . .    | 206 |
| 7.3 | Determine whether one rule subsumes another . . . . .              | 206 |
| 7.4 | Determine initial element groups . . . . .                         | 209 |
| 7.5 | Generate element combinations for the <i>inter_class</i> . . . . . | 210 |
| 7.6 | Combine two lists of elements . . . . .                            | 210 |



# Chapter 1

## Introduction

There are many different types of information that should be considered when designing an information system, particularly when designing a system to solve complex business problems. A wide variety of modelling approaches and notations have been developed to model these different types of information, such as entity-relationship diagrams, functional dependencies, data flow diagrams, object diagrams, the relational model, state-transition diagrams, petri nets, formal methods and so on. Problems can arise when information is not included in a design that should be. For example, a typical approach to building an information system is to design the underlying data structures and semantics using entity-relationship diagrams, and implement these structures in a relational database management system. These structural models often form a basis for the user interface design, which is implemented atop the database using a rapid application development environment. Consider a systems analyst who designs and implements a system using this approach. Good design practices are followed throughout, yet the end-users find the finished application difficult to use. Some of the more commonly used data-entry forms have multiple states, but because the analyst did not think to include state information in the system design, the transitions between these states are not clear to users. Use of state-transition diagrams during the design phase could have resolved this problem before implementation.

While this is a purely theoretical example, it serves to illustrate an important point. Information systems are typically created to handle the data processing requirements of some real-world phenomenon. Such real-world phenomena may often be too complex to describe using a single modelling approach or *representation*. This is supported by the plethora of representations that currently exist, including those that model the structure of data (such as entity-relationship modelling and functional dependencies),

and those that model how data move around a system (such as data flow diagrams). This implies that in order to completely model a phenomenon, multiple descriptions of the phenomenon expressed using different representations are required.

Using multiple representations to describe a phenomenon is also important in other ways. If multiple developers are working on a project, each may prefer or be required to use a different representation to describe their particular part of the project (Atzeni and Torlone, 1993; Atzeni and Torlone, 1996a). Particular subproblems may also be better described using some representations than others (Easterbrook, 1991a, p. 56). Multiple representations are also important when integrating heterogeneous data sources to form a federated or distributed system (Atzeni and Torlone, 1996a), as each data source may potentially use a different logical data model (such as relational, network or object-oriented). Reconciling and integrating descriptions expressed using different representations is therefore an important part of the design process.

An important goal of this research is to aid this reconciliation and integration within the context of information system design. That is, to *facilitate the use of multiple modelling representations for describing a phenomenon*, with the purpose of building an information system to model that phenomenon. To achieve this goal, the author has chosen an approach based upon translating descriptions of a phenomenon between different representations. This goal and the approach taken provide three main threads to the thesis: using multiple representations to describe a phenomenon, translating between representations to facilitate this use, and measuring the efficacy of these translations.

The first thread of the thesis, using multiple representations to describe a phenomenon, brings together the work of other researchers in this area, and research into *viewpoint-oriented methods*. A well-documented problem in the area of information system design is that developers may build a design that models a phenomenon from only a single perspective, when it should really be modelled from multiple perspectives (Finkelstein et al., 1989; Easterbrook, 1991a). This can lead to important aspects of the phenomenon either not being modelled, or being modelled as exceptions instead of a fully-integrated component of the design (Finkelstein and Sommerville, 1996; Easterbrook, 1991a). Viewpoint-oriented methods, introduced in Chapter 2, were developed as a solution to this problem, and seek to model a phenomenon from several different *viewpoints* simultaneously. This is somewhat analogous to the approach

taken by the Soft Systems Methodology (Checkland, 1981). The multiple-viewpoint approach lessens the possibility of important design information being discarded or lost (Finkelstein and Sommerville, 1996; Easterbrook, 1991a; Easterbrook and Nuseibeh, 1996).

Each viewpoint will typically comprise one or more descriptions of a phenomenon, which may be expressed using the same representation, or different representations. It has already been noted that using multiple representations to describe a phenomenon is useful, but this is difficult if each viewpoint uses only a single representation. An obvious solution is to define multiple viewpoints, each of which uses a different representation, but this is somewhat cumbersome; using multiple representations within a viewpoint is a more flexible approach. It can be argued that using multiple representations within a single viewpoint is analogous to using multiple viewpoints to model a phenomenon, and can produce greater depth and detail than if only a single representation were used within the viewpoint (Finkelstein et al., 1989; Darke and Shanks, 1995a; Easterbrook, 1991a). This research will be focused on the use of multiple representations within a *single* viewpoint.

Research into the effective use of multiple representations to model real-world phenomena is scarce. Viewpoint researchers have been promoting the idea of using multiple representations to describe a single viewpoint for nearly ten years (Finkelstein et al., 1989), yet surprisingly little research has resulted. Outside the viewpoints field, the author is aware of only three other groups who have worked in this area (Atzeni and Torlone, 1993; Grundy, 1993; Su et al., 1992), whose work is briefly reviewed in Chapter 2. Independent research efforts have resulted in a disparate, and sometimes conflicting, collection of terms for similar concepts. Viewpoint concepts provide a useful framework within which to discuss the use of multiple representations, and a consistent terminology based on these concepts is defined by the author in Chapter 3.

The second thread of the thesis, using translations between multiple representations in order to facilitate their use, is the major thread of the thesis. Translating descriptions of a viewpoint between representations is a special case of the more general problem of data translation, which is reviewed in Chapter 2. The promotion of a more comprehensive design is important when designing information systems, and the use of multiple representations to describe a viewpoint should provide greater depth and

detail than if only a single representation were used. Providing the ability to translate descriptions from one representation to another should facilitate the use of multiple representations, as this allows the effective re-use of information stored in existing descriptions and provides a greater degree of integration between different representations. Translations can also provide a mechanism for keeping related descriptions consistent, as described in Chapter 3. In addition, translating descriptions expressed using different representations into the same representation can provide a mechanism for highlighting potential inconsistencies within a viewpoint, which is discussed in Chapter 4.

The definition of translations in this thesis follows the individual interfacing strategy. That is, translations are defined between representations in a pairwise fashion, rather than through some intermediate representation or interchange format. This approach provides greater flexibility and allows translations to be better tailored to the corresponding representations. It also provides the opportunity when defining a new translation to leverage the functionality of existing translations, which is discussed in Chapter 5. For example, instead of implementing normalisation in every translation that requires it, it could be implemented in a single translation that then provides normalisation ‘services’ to other translations.

A translation comprises a collection of *rules* that define mappings between the constructs of two representations. A translation is ‘executed’ by applying its rules to the components of some ‘source’ description; this process and the issues arising from it are discussed in Chapter 4. A variant of this translation process has been implemented in a simple prototype modelling environment, described in Chapter 6. Known as *Swift*, this environment was written in Java and supports translations between four representations: Martin entity-relationship diagrams, SQL/92, Smith functional dependency diagrams<sup>1</sup> and Gane & Sarson data flow diagrams.

The efficacy of a translation is measured by its *quality*, which is determined by how well it maps the constructs of one representation onto the constructs of another representation. An important goal of this research is therefore to improve the quality of translations, in order to reduce the amount of new material that must be provided by the designer. Two approaches to improving translation quality are described in Chap-

---

<sup>1</sup>This representation is described in Appendix B.



ter 4: the use of *heuristic rules*, which can make explicit semantics that are implicit in the source description, but can sometimes produce inconsistent results due to their heuristic nature; and the use of *enrichment* to provide additional information to the translation process that could not normally be obtained.

Translations must be specified using some notation in order to be of any use. A high-level notation is defined in Chapters 3 and 4 to provide a concise means of specifying representation, description and translation expressions, and is used in Chapter 5 and Appendix E to define three translations. This notation is not enough, however, to fully specify the details of a translation. Translations in the Swift prototype were defined in an ad hoc manner based on the rules described in Chapter 5. This is not a flexible approach; a more useful solution would be to use some form of translation specification language. Amor's (1997) *View Mapping Language* (VML) is a declarative language for specifying mappings between schemas. This language is extended in Chapter 7 to deal with the translation issues identified in Chapter 4.

The third thread of the thesis, measuring the efficacy of translations, brings together the first two threads, and provides verification that heuristics improve the quality of translations. The approach presented in Chapter 8 is based on the concepts of *relative information capacity* (Hull, 1986; Miller, 1994) and *schema intension graphs* (Miller et al., 1994b), which were originally designed to aid with schema integration. The relative information capacity of a representation is effectively the same as the *expressive power* of that representation, that is, it determines the limits of what may be expressed using the representation. The expressive powers of two representations may overlap in various ways, which determines how well constructs may be mapped from one representation to another (see Chapter 3). That is, the overlap in expressive power of two representations affects the quality of translations between those representations. Methods for determining the expressive overlap of representations and measuring the relative quality of translations between representations are defined by the author and applied in Chapter 8. The relative quality measure is used to show that the use of heuristics has a positive impact on translation quality.

The approach taken in this thesis is evaluated in Chapter 9. First, a case study using Swift is presented, showing how the approach presented here promotes some of the goals of the thesis. Second, the novelty of the approach is evaluated by comparing

it with similar approaches by Atzeni and Torlone (1993; 1995; 1997), Su et al. (1992) and Grundy (1993; 1994; 1995a; 1997). A survey of ‘conventional’ computer-aided software engineering (CASE) tools is also undertaken to show that although existing tools generally *support* the use of multiple representations, they typically do not *facilitate* this use. Finally, a general discussion of the tractability of the approach is presented, followed by a series of experiments to test the time complexity of translations in the Swift environment and show that the approach presented here is practicable.

Many issues are identified and discussed in this thesis, but some have been left as areas for future research, and are discussed in Chapter 10. The results of the research are summarised and conclusions are presented in Chapter 11.

In summary, the author has identified the following goals for this thesis:

1. Improve a viewpoint in terms of depth and detail by using multiple representations to describe the viewpoint.
2. Facilitate the use of multiple representations within a viewpoint by translating descriptions between representations.
3. Develop a consistent and unified terminology for discussing representations and translations between them.
4. Develop effective ways of specifying translations.
5. Use translations to enable the highlighting of potential inconsistencies between descriptions of a viewpoint.
6. Identify ways of improving translation quality.
7. Show that the approach presented here is novel and practicable.

# Chapter 2

## Related research

### 2.1 Introduction

As was discussed in the previous chapter, it is often useful (or necessary) to use multiple representations to describe a real-world phenomenon, for instance:

- A more complete description of a viewpoint may be built using multiple representations than if only a single representation were used.
- The use of multiple representations allows developers to use the representations that they are most familiar with.
- The methodology being followed may require developers to use particular representations for various aspects of the problem.
- Some representations are better suited to particular problems than others.
- Heterogeneous data sources to be integrated may be expressed using different representations.

Thus, the use of multiple representations to describe a phenomenon is an important issue. In this thesis, a phenomenon is modelled by a *viewpoint*, which can be informally defined as a particular interpretation of the phenomenon. An important goal to be addressed in this thesis is that of *facilitating the use of multiple modelling representations within a single viewpoint*. To achieve this goal, the author has chosen an approach based on translating between models expressed using different representations.

In the remainder of this chapter is provided a review of relevant research to support this approach. Facilitating the use of multiple representations is a fairly new area of

research, and there is as yet only a small amount of literature relating specifically to it, as identified by Atzeni and Torlone (1993, p. 350). Nevertheless, there are several related areas that are useful to the approach taken in this thesis. The idea of facilitating the use of multiple representations has been identified as a useful problem by several authors, in particular Grundy (1993), Atzeni and Torlone (1993) and Su et al. (1992). In Section 2.2, related work by these and other authors is reviewed.

The approach followed in this thesis is based on a framework derived from the area of *viewpoint-oriented methods*, which are discussed in Section 2.3. The concepts of *viewpoint* and *representation* are introduced, and the issues that arise from building and integrating multiple viewpoints of a phenomenon are discussed. Concepts relating to viewpoint integration will be used in Chapter 8 as a basis for a method of measuring the relative quality of translations.

Since the author is advocating an approach based on translations between models, the area of data translation is also important. In Section 2.4 the concepts of translation *quality* and *performance* are introduced, and possible *interfacing strategies* are discussed. Some means of specifying translations is also required, so specialised languages for specifying translations are also discussed.

## **2.2 Multiple representations for data modelling**

The idea of *using* multiple representations to model a phenomenon is not new, otherwise there would not be the plethora of different modelling representations in existence today (Tsichritzis and Lochovsky, 1982; Hull and King, 1987). Viewpoint researchers have also been suggesting the use of multiple representations to model viewpoints for nearly ten years (Finkelstein et al., 1989). *Facilitating* the use of these multiple representations is, however, a more recent idea.

Grundy (1993) developed an interest in facilitating the use of multiple representations, and took an approach based on maintaining consistency across multiple views of an underlying model. Work in this area is described in Section 2.2.1.

Atzeni and Torlone (1993) suggested the idea of translating between different representations as a means of facilitating the use of multiple representations. They proposed a formal model based on lattice theory which allowed them to express many different

data modelling approaches using the same representation. Their work is described in Section 2.2.2.

Su et al. (1992) approached the use of multiple representations from the point of view of integrating heterogeneous data sources in order to build federated and distributed databases. Their approach is similar in many respects to that taken by Atzeni and Torlone, but the key difference is that the model used by Su et al. is object-oriented rather than mathematically based. Their work is described in Section 2.2.3.

### **2.2.1 The multiple views approach**

The development of large information systems generally requires the services of many people working in different roles, such as designers, programmers and testers. Each of these people may have a different view of the system, and these different views may comprise both textual and graphical information. For example, programmers may view the system at the code level using a language such as C++, while designers may view the system using high-level structured diagrams. Meyers (1991, p. 50) notes that it can be very useful to simultaneously view a system in several different ways, especially when multiple people are working on the system. Some way is therefore needed of integrating these multiple views.

Reiss's (1985) PECAN environment was an early attempt to address some of the issues associated with providing multiple views of an information system. PECAN could display several views of a Pascal program (many of which were read-only), including a syntax-directed editor and several semantic views, such as expression trees and flow graphs. All of these views were derived from an underlying abstract syntax tree. That is, each view was effectively a different visualisation of the same underlying representation.

FIELD (Reiss, 1990a; Reiss, 1990b) addressed the problem of integrating several relatively distinct development tools, where each tool effectively constituted a different 'view' of the system being developed. There was no underlying representation as there was in PECAN; integration was achieved by passing messages between tools via a central message server. When a developer made a change in one tool, this change was selectively broadcast to other tools by the message server, and these tools then updated

their ‘views’ of the system appropriately. In a sense, FIELD could be said to ‘simulate’ multiple views of a system rather than directly supporting them.

Brown’s (1992) Zeus was a system for implementing algorithm animation, but could also be used to implement multi-view editing systems. In Zeus, an ‘algorithm’ maintained shared data structures that were displayed appropriately by each view. When a user altered a view, the view would generate a feedback event that was sent to the ‘algorithm’, which made the appropriate changes to the shared data structures. The ‘algorithm’ would then generate and send output events to each of its associated views, which would update themselves appropriately. As with PECAN, views in Zeus were different visualisations of the same underlying representation.

MultiView (Altmann et al., 1988) was a multi-view environment that supported multiple views of Modula-2 programs. As with PECAN, different views were derived from an underlying abstract syntax tree. Changes were propagated from view to view by a message-passing mechanism through a central program database (cf. FIELD and Zeus). The key innovations introduced in MultiView were the ability to edit multiple different views concurrently, and the parallel nature of the environment which facilitated multi-user access. Later work (Jacobs and Marlin, 1995) extended the notion of multiple views to modelling the software development process in addition to individual programs.

MViews is an object-oriented framework for implementing integrated software development environments (Grundy, 1993; Grundy and Hosking, 1993a; Grundy and Venable, 1995b; Grundy and Hosking, 1997). It incorporates many of the ideas from earlier environments and adds a robust mechanism for maintaining consistency across multiple graphical and textual views. Environments built using MViews allow developers to view an underlying model in several different ways at once, and to modify any of these views. Where possible, changes are automatically propagated across views to keep them consistent. For example, a developer might have both entity-relationship and SQL views of a model on screen. Changes made to the E-R view would (where possible) be propagated to the SQL view and vice versa.

In effect, MViews supports the implementation of environments that support multiple modelling representations, and the use of these representations is facilitated by using an automatic translation mechanism to propagate changes between related views.

Although it is not required, most of the environments built using MViews use a single integrated data model to store schemas (Grundy and Venable, 1995b).

MViews was originally implemented in an object-oriented variant of Prolog called Snart (Grundy, 1993), and a Java version (called *JViews*) has been constructed (Grundy et al., 1997b). Several environments have been built using the MViews framework, such as:

- The Snart Programming Environment, or SPE (Grundy and Hosking, 1993b) is an environment for visualising class structures in Snart programs.
- OOEER (Grundy and Venable, 1995b) combines object-oriented analysis and design (OOA/D) with extended entity-relationship (EER) modelling.
- NIAMER (Venable and Grundy, 1995) combines entity-relationship modelling and object-role modelling.
- EPE (Amor et al., 1995) supports the construction of EXPRESS specifications and associated EXPRESS-G diagrams.
- MViewsDP (Grundy et al., 1996) is a graphical user interface builder for dialog boxes that uses textual representations for specifying the dialog interface and validation rules.

The main focus of the MViews research is on the issues involved with building tools that make use of multiple representations, in particular (Grundy et al., 1996; Grundy and Hosking, 1996; Grundy, 1998; Hosking and Grundy, 1995; Hosking et al., 1995):

- architectural support for multiple modelling representations;
- maintaining consistency between multiple graphical and textual views;
- human interface issues, such as how to present inconsistencies to users, how to support their interaction with inconsistencies, and so on;
- collaborative multiple viewpoint system issues; and
- various applications — mostly software engineering, but also building/architectural design, user interfaces and process modelling.

Although translating views from one representation to another is obviously an important part of this, work with MViews has focused on the issue of maintaining consistency across multiple views (Grundy and Hosking, 1996). Related work has also been undertaken in the area of specifying translations (Amor, 1997), which will be discussed in Section 2.4.4.

## 2.2.2 The multiple-data-model (MDM) approach

While working on a larger project to define an integrated information system design and development environment, Atzeni and Torlone (1993) realised that in order to be more adaptable to the needs of different developers, such an environment must support multiple representations (referred to by them as ‘data models’ or simply ‘models’). The problem then became one of managing these multiple representations, that is, facilitating their use. The main goal of their research was to build a tool that allows schemas (referred to by them as ‘schemes’) to be translated from one representation to another. A tool called MDM (multiple-data-model) has been developed (Atzeni and Torlone, 1997) that provides this functionality.

Atzeni and Torlone (1993) argued that most data models use a limited set of constructs that fall into one of six types described by Hull and King (1987): lexical types, abstract types, aggregations, grouping constructs, functions and generalisations. With this in mind they defined a *metamodel* based on these construct types (or *metaconstructs*) that can be used to define any model that also uses these construct types. They have demonstrated this metamodel being used to define entity-relationship, relational and functional models (Atzeni and Torlone, 1993; Atzeni and Torlone, 1996a; Atzeni and Torlone, 1996c) and argue that their metamodel can be used to manage a wide range of models. This is done by using the metamodel to define a *supermodel* that subsumes all the models to be managed (Atzeni and Torlone, 1996a; Atzeni and Torlone, 1997). Valid schemas are then expressed using the appropriate subset of the supermodel. The supermodel is analogous in concept to the integrated model of MViews, although it is implemented differently.

Atzeni and Torlone (1996a) introduce the notions of *model* (representation) and *scheme* (schema) within a formal graph-theoretic framework. A model is defined by



a set of labelled trees, where the nodes and edges are elements of the metamodel and the labels represent the names of the constructs of the model (such as Entity or Relation). Similarly, a scheme is defined by a labelled directed acyclic graph where the nodes and edges are elements of the metamodel and the labels represent the concepts of the scheme (such as Staff and Sale). Schemes are said to be *allowed* in a particular model if they form an instance of that model.

Translations between models are derived automatically from a collection of basic predefined rules that are defined by a person or collection of people known as the *model engineer* (Atzeni and Torlone, 1993; Atzeni and Torlone, 1996a). These translations follow a pragmatic approach to the semantics of constructs: if two constructs from different models correspond to the same metaconstruct, they are assumed to have the same semantics (Atzeni and Torlone, 1996a). Translations can be shown to be correct and complete, and their quality, defined as how well they exploit the constructs of the target model, may also be measured (Atzeni and Torlone, 1995, Section 4).

The prototype MDM system comprises a collection of modules for manipulating and managing models, schemas and translations (Atzeni and Torlone, 1997). Models are defined using a menu-driven model definition language, which the model manager uses to generate a model-specific scheme definition language. This language is then used to define schemes. Translations from the supermodel to a particular model are automatically derived by the translation generator when the model is created; a translation from a model to the supermodel is not required as all models are subsets of the supermodel. If a new model is defined that is not subsumed by the supermodel, the model manager automatically generates a new supermodel.

### **2.2.3 The object-oriented rule-based (ORECOM) approach**

ORECOM (Object-oriented Rule-based Extensible COre Model) is an object-oriented model that has been used to support the integration of multiple heterogeneous data sources and/or schemas, and the exploration of translations between object-oriented and some of the richer semantic data models (Su et al., 1992; Su and Fang, 1993). A data model is expressed in ORECOM by decomposing its constructs and semantic constraints into collections of primitive ORECOM constructs and constraints. ORECOM

has three primitive constructs: *objects* and *classes*, which are very similar to the concepts of object and class in object-oriented programming; and *associations*, which are binary relationships between classes.

Semantic constraints of data models (such as cardinality and inheritance) are expressed in ORECOM by a collection of primitive constraints known as *micro-rules*. These are triggered by certain operations on a class, and capture those semantic constraints that are not captured by the structural constructs. Micro-rules may be combined to form *macros*, which model higher-level semantic constraints that occur across many different data models. ORECOM is a highly-extensible model as a consequence of its object-oriented basis. When a new data model introduces constructs and constraints that cannot be decomposed into ORECOM constructs and constraints, ORECOM may be extended by defining new primitive constructs and constraints.

An environment for performing schema translations has been built around ORECOM. Since all representations are defined using ORECOM primitives, the data model translation system can readily compare the constructs and constraints of the source and target representations. This comparison is used to build an equivalence matrix that identifies the closest matches between constructs in each representation. This matrix is then used by the schema translation system to translate schemas from one representation to the other.

The overall approach followed by ORECOM is similar to that taken by Atzeni and Torlone (1993) in that both use an abstract metamodel to define representations, and translations between representations are automatically derived from representation definitions. The major difference is that ORECOM is based on an object-oriented model rather than a mathematical one. Su and Fang (1993) have identified the following possible applications for their model:

- schema sharing;
- schema translation in the context of a heterogeneous DBMS;
- schema integration;
- schema verification and optimisation;

- semantics modification and extension before conversion, that is, extending the semantics of a schema before translating it to another representation; and
- helping users to learn new modelling representations, as all representations are decomposed into a standard form, making it easier to compare them.

## 2.3 Viewpoint-oriented methods

Viewpoint-oriented methods are an important element of the approach taken in this thesis, because they provide a useful framework within which to discuss the use of multiple representations to model a phenomenon. The concept of a viewpoint was first developed in the CORE method (Mullery, 1979), which was designed as a controlled method for expressing requirements for information systems. The CORE method and the rationale behind viewpoints are briefly described in Section 2.3.1.

A *viewpoint* can be thought of as a formalisation of the perceptions of a stakeholder group with respect to some real-world phenomenon that is being modelled. This and other associated terms are defined in Section 2.3.2.

An important part of any viewpoint-oriented approach is combining potentially conflicting viewpoints in a coherent manner, which is a process similar in many respects to schema integration. The issues surrounding viewpoint integration and some possible approaches for integrating viewpoints are discussed in Section 2.3.3. Techniques developed to help with the problem of viewpoint integration will be used in Chapter 8 as a basis for a method of measuring relative translation quality.

### 2.3.1 Origin of viewpoints

The first viewpoint-oriented method was the CORE (COntrolled Requirements Expression) method (Mullery, 1979). This method identified the need for “several expressions of requirement” during the requirements definition phase of the systems development life cycle. These expressions correspond to different viewpoints of the system, such as (Mullery, 1979, p. 127):

- “a *life cycle* view (e.g. production plans, information and money flows)

- an *environment* view (e.g. user interfaces, interfaces with other systems, physical operating conditions such as temperature, power supply, location)
- an *operator* view (e.g. interfaces between the system and its ‘servants’ — i.e. people whose jobs exist because the system exists)
- a *reliability* view (e.g. acceptable failure rates, fault tolerance measures required, recovery and back-up facilities required).”

The CORE method comprises the following activities:

1. **Propose relevant viewpoints:** In this activity, a small number of relevant viewpoints are determined (generally between two and five). If more than five viewpoints are found, then viewpoints are combined until there are no more than five viewpoints.
2. **Define relevant viewpoints:** In this activity, specific requirements are developed for each proposed viewpoint.
3. **Confirm relevant viewpoints:** This activity has two aims:
  - (a) **Combining viewpoints:** The viewpoints are reconciled and combined to produce a single viewpoint for the system; and
  - (b) **Defining reliability:** The reliability requirements of the combined viewpoint are determined.

The CORE method was derived from several years of practical experiment by Mullery, so the concepts involved are not based on any formal model. Mullery also assumes that viewpoints do not overlap (Easterbrook, 1991a, p. 54), and therefore they cannot conflict. This may not always be correct, however, as will be shown in Section 2.3.3 on page 22.

The basic notion of a viewpoint has changed little since the CORE method was introduced. Finkelstein et al. (1989) proposed a more formal definition of the concept, and also suggested that viewpoints need not be consistent with each other, that is, viewpoints may conflict. This can conceivably lead to situations where it is difficult, if not impossible, to reconcile differing viewpoints. Easterbrook (1991b; 1994) continued

this work by developing methods for resolving conflicts between viewpoints using computer-supported negotiation. Recent research has also focused on how viewpoints are represented (Darke and Shanks, 1995b), and particularly on the use of multiple representations to describe viewpoints.

The viewpoint terminology of Finkelstein et al. (1989), Easterbrook (1991a) and Darke and Shanks (1995b) will now be described.

### **2.3.2 Viewpoint terminology**

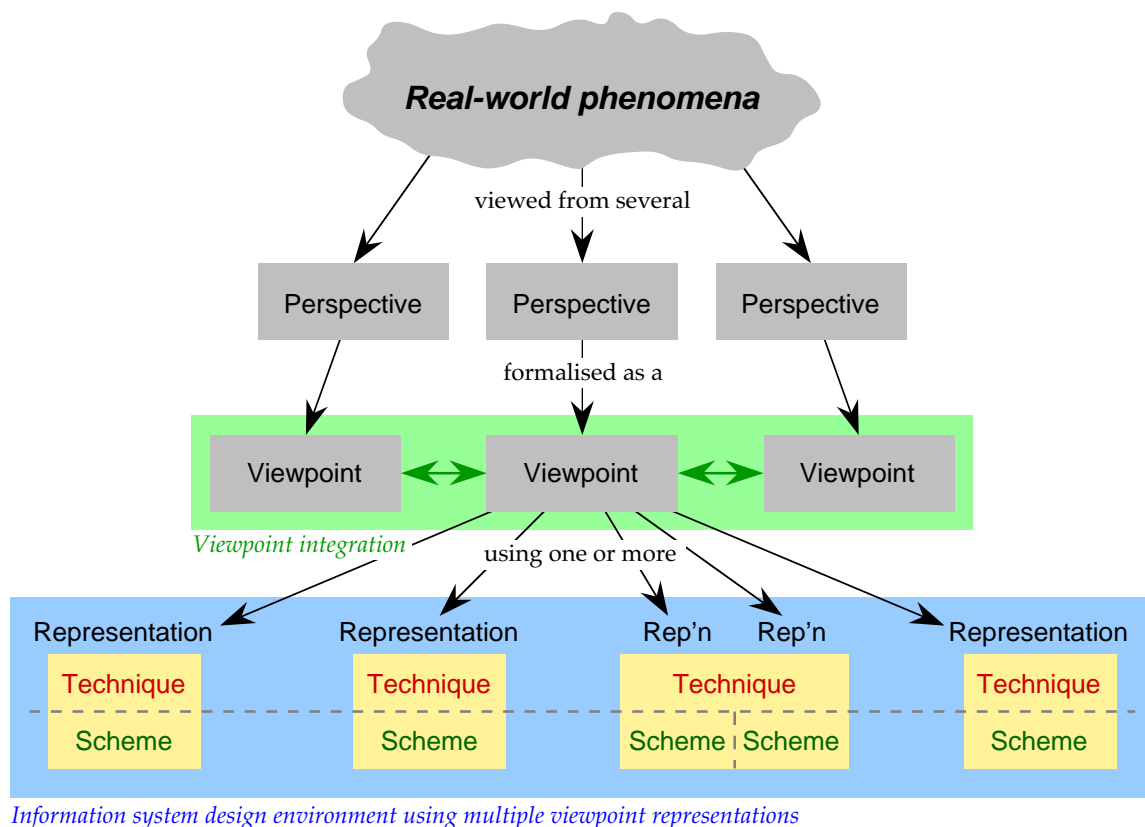
Viewpoint-oriented methods were originally developed to assist with requirements definition in a software engineering environment (Mullery, 1979), and subsequent research has followed a similar direction (Easterbrook, 1991a; Kotonya and Sommerville, 1996; Leite and Freeman, 1991; Nuseibeh et al., 1994). Recent work has examined the process of resolving inconsistencies between viewpoints in an evolving specification (Easterbrook and Nuseibeh, 1995; Easterbrook and Nuseibeh, 1996). The focus of this thesis, however, is on how to facilitate the use of multiple representations to describe a single viewpoint. This is an area that has only recently begun to receive attention (Darke and Shanks, 1995b).

In Figure 2.1 on the next page the relationships between the concepts of viewpoint-oriented methods are shown. This framework is derived by the author from the work of Finkelstein et al. (1989), Easterbrook (1991a) and Darke and Shanks (1995b).

In the previous discussion, the terms *viewpoint* and *representation* were used informally. The concepts of perspective, viewpoint and representation are now introduced.

#### **Perspectives and viewpoints**

Easterbrook (1991a, p. 54) defines a *perspective* as “a description of an area of knowledge which has internal consistency and an identifiable focus of attention”. During the requirements definition phase of systems analysis, developers may encounter many different perspectives on the problem being modelled. Perspectives may overlap and/or conflict with each other in various ways. Part of the process of information system development is determining how to deal with these multiple perspectives. This is an active area of research and has been discussed already by several authors (Kotonya



**Figure 2.1:** Relationship between perspectives, viewpoints and representations

and Sommerville, 1996; Leite and Freeman, 1991; Easterbrook et al., 1994).

Finkelstein et al. (1989, p. 43) describe a *viewpoint* as comprising the following parts:

- “a *style*, the representation scheme in which the ViewPoint [sic] expresses what it can see (examples of styles are data flow analysis, entity-relationship-attribute modelling, Petri nets, equational logic, and so on);
- a *domain* defines which part of the ‘world’ delineated in the style (given that the style defines a structured representation) can be seen by the ViewPoint (for example, a lift-control system would include domains such as user, lift and controller);
- a *specification*, the statements expressed in the ViewPoint’s style describing particular domains;
- a *work plan*, how and in what circumstances the contents of the specification can be changed; [and]
- a *work record*, an account of the current state of the development.”

Easterbrook (1991a, p. 54) simplifies Finkelstein et al.'s (1989) description by defining a viewpoint as “the formatted representation of a perspective”, and notes that a perspective is a “more abstract version of a viewpoint”. In effect, a viewpoint is the formalisation of a particular perspective, so there is a one-to-one correspondence between a viewpoint and the perspective it formalises, as illustrated in Figure 2.1.

Note that the concept of a viewpoint is different from that of a ‘view’ produced by a multi-view development environment (see Section 2.2.1 on page 9). The similarity of the terms has led to some confusion: the terms ‘viewpoint’ and ‘view’ have been used interchangeably in the past (Jacobs and Marlin, 1995, p. 586). The concept of a ‘view’, however, is more akin to the concept of a *description*, which is introduced in Chapter 3.

Darke and Shanks (1995b, p. 279) define two main types of viewpoint:

1. *user viewpoints* that capture “the perceptions and domain knowledge of a particular user group, reflecting the particular portion of the application domain relevant to that group”; and
2. *developer viewpoints* that capture “the perceptions, domain knowledge and modelling perspective relevant to a systems analyst or other developer responsible for producing some component of the requirements specification”.

Since a viewpoint is the formalisation of a perspective, some form of model is required to provide the formalised structure. The concept of a representation provides this.

## **Representations**

Darke and Shanks (1994, p. 4) note that viewpoints may be described using different representation *techniques*, within each of which there may be available a number of representation *schemes*. Neither Darke and Shanks (1994) nor Finkelstein et al. (1989) clearly define the terms ‘representation’, ‘technique’ or ‘scheme’; rather, they introduce each term by means of examples. This has led to some confusion in the use of this terminology. Darke and Shanks (1994, pp. 4–5) use the terms ‘representation’ and ‘representation technique’ interchangeably, while Finkelstein et al. (1989), as can be seen in their definition of a ‘style’, use the term ‘representation scheme’ in a similar way.

The intent of both Finkelstein et al. (1989) and Darke and Shanks (1994) appears to be that a *representation* should be thought of as a structured modelling approach that can be used to describe the content of a viewpoint. In order to clarify the confusion in terminology, the author has refined this informal definition and defined a *representation* as the combination of a particular technique and scheme to describe a viewpoint. This will be discussed further in Chapter 3.

Darke and Shanks (1995b, p. 280) group representations into three general categories:

- *informal* representations that form unstructured descriptions, often expressed using a natural language or simple diagrams;
- *semi-formal* representations that form structured descriptions, such as entity-relationship modelling or data flow diagrams; and
- *formal* representations that form structured descriptions, and include a set of operators for processing these descriptions, such as the relational model (Codd, 1970) and logic-based models (Gallaire and Minker, 1978; Date, 1995).

Unlike informal representations, which are often ill-defined, inconsistent and ambiguous, semi-formal and formal representations are well-defined, consistent and generally unambiguous. A key feature of formal representations that is lacking in semi-formal representations is the inclusion of operators which allow assertions to be made about the viewpoints being described; Greenspan et al. (1994) describe this as the ability to ‘reason’ about representations. User viewpoints are typically defined using informal representations, whereas developer viewpoints are typically defined using semi-formal or formal representations.

### **2.3.3 Viewpoint integration**

Viewpoint-oriented methods by definition produce a collection of differing viewpoints. Elements of these viewpoints may conflict with each other; in extreme cases, different stakeholder groups may have widely divergent perspectives on the phenomenon being modelled. To produce a useful information system, these differing viewpoints must be integrated in some way.



In the correct context, the viewpoint integration and schema integration processes are basically identical (Batini et al., 1986). This is to be expected, as both viewpoints and schemas are formalised means of describing real-world phenomena. The viewpoint and schema integration processes both suffer from similar problems with conflicting viewpoints or schemas. The connection between viewpoint integration and schema integration is briefly examined in this section, and some of the possible sources of conflict identified.

Viewpoint integration can be considered as a process of resolving the conflicts among viewpoints. The issues arising from this resolution process are also discussed in this section.

The correspondence between schema and viewpoint integration suggests that approaches and tools developed for schema integration will also be useful for viewpoint integration. One such approach, *relative information capacity* (Hull, 1986), will be used as the basis of a method for measuring relative translation quality in Chapter 8.

### **Viewpoints and schema integration**

Schema integration is the process of combining two or more schemas to form a single consistent schema. Batini et al. (1986, p. 324) noted that schema integration can occur in two contexts:

1. “*View integration* (in database design) produces a global conceptual description of a proposed database.
2. *Database integration* (in distributed database management) produces the global schema of a collection of databases. This global schema is a virtual view of all databases taken together in a distributed database environment.”

The second context is probably more familiar, due to the current popularity of integrating heterogeneous legacy databases. The former context, however, is more relevant to viewpoint integration, and indeed, Batini et al.’s (1986, pp. 326–327) description of view integration implies that viewpoint integration and view integration are the same activity. Both processes involve the integration of several different views of a system to form a consistent design, and both take place during similar phases of the development cycle (Batini et al., 1986, Figure 1).

Problems arise when attempting to integrate diverse schemas. There are many causes of diversity in schemas, such as (Batini et al., 1986, Section 1.2):

**Different perspectives** This corresponds directly to the idea of multiple perspectives outlined in Section 2.3.2.

**Equivalence among constructs of the model** The same concepts may be modelled in different ways in different schemas, for example, a type hierarchy of staff entities as opposed to a single staff entity with a *job\_code* attribute.

**Incompatible design specifications** There may be design errors in one or more of the schemas to be integrated.

In addition to these causes of diversity, there is also the issue of identifying common concepts in each source schema and determining how they are related to each other. The kind of problems that can arise include synonyms (similar concepts with different names) and homonyms (different concepts with similar names). There is also the issue of inter-schema relationships. Suppose one schema defines a *person* entity, while another defines an *employee* entity. It may be that the *employee* entity should be a subset of the *person* entity, but this relationship obviously cannot exist in either of the source schemas alone; rather it is an *inter*-schema relationship.

All these problems can cause conflicts between viewpoints. Since view integration and viewpoint integration are basically the same activity, it should be possible to apply similar methodologies, such as those of Kahn (1979), Batini and Lenzerini (1984) and Navathe and Gadgil (1982). Recent work in viewpoint integration, however, has focused more on the conflict resolution process.

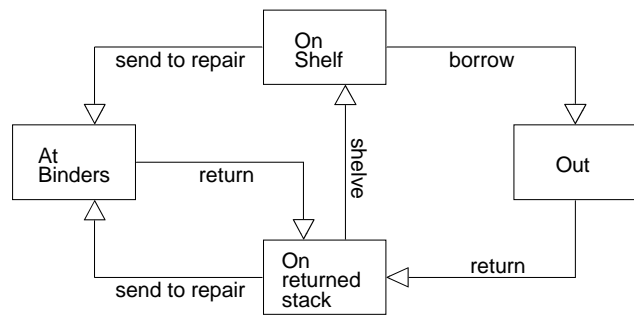
### **Conflict resolution**

In general, inconsistency arises when two parts of a specification or design do not obey some relationship that should, in theory, hold between them (Easterbrook and Nuseibeh, 1996). Easterbrook (1991a, p. 95) gave the following example of two librarians who are asked to describe the possible states of a book:

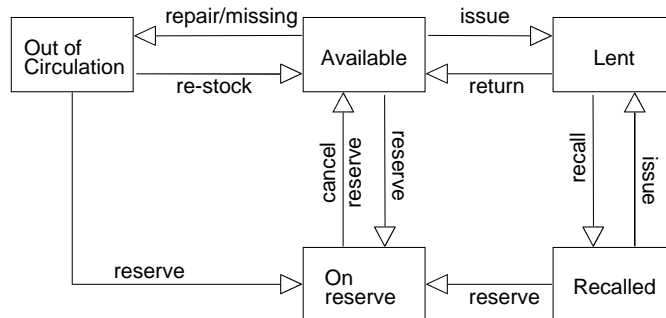
Two possible state transition diagrams for the books of the library are given [see Figure 2.2], which may have been elicited from two different librarians. One gives

a description based roughly on a book's physical whereabouts, whereas the other gives a description based on how a book can be accessed. While it may appear that there are a number of correspondences between the two descriptions, these are not as simple as they seem. For example, the concepts ON SHELF and AVAILABLE are similar, except that the latter includes books waiting to be shelved: it assumes that librarians are able to locate unshelved books for loan. OUT and LENT are also similar, except that the former includes books being used within the library, while the latter only includes such books if they are from the reserve collection. To make things worse, both could have used the same terminology.

The elicited viewpoints are both equally valid interpretations of the same problem, but are inconsistent with each other, that is, a *conflict* exists between the two viewpoints.



(a)



(b)

**Figure 2.2:** Possible states for a library book: (a) from the perspective of physical whereabouts; (b) from the perspective of accessibility of a book (Easterbrook, 1991a, Figure 6.3)

Conflict resolution is an important part of viewpoint-oriented methods. Historically, viewpoints have not been recognised as first-class citizens in the systems devel-

opment process (Finkelstein and Sommerville, 1996). Instead, developers have typically chosen a single 'correct' viewpoint and effectively forced the various other viewpoints to fit within the framework of the chosen viewpoint (Easterbrook, 1991a, pp. 18–19, 34). Inconsistencies which cannot be made to fit are assumed to be anomalies and either discarded or, more likely, dealt with as 'special cases' during the implementation phase. This approach effectively sacrifices diversity of perspectives for the sake of consistency and coherence.

Easterbrook (1991a, Section 2.2.4) notes, however, that suppressing inconsistency in this manner can cause problems, for example, a systems analyst may impose her own preconceived solution on the problem, even though it may not be the best. If there is an inconsistency between two viewpoints, then there is presumably some valid reason for this inconsistency (assuming that it is not a genuine error); this is an important aspect of the phenomenon being modelled and should therefore be documented. Easterbrook and Nuseibeh (1996) note that inconsistency between viewpoints can often imply that there is some information missing that the analyst still needs to extract, suggesting that these areas of inconsistency are precisely those that systems analysts should spend the most time on. If inconsistencies are ignored during the requirements definition and design phases, they are likely to cause problems during the implementation phase.

Viewpoint-oriented methods consider viewpoints as distinct entities within the development process. Viewpoints may be manipulated in various ways, and relationships may be defined between them. Note, however, that viewpoint-oriented methods do not sacrifice consistency and coherence for the sake of perspective diversity. Rather, a balance must be struck between the two. The overlaps and conflicts among viewpoints must be explicitly resolved in some way to result in a useful system. Finkelstein and Sommerville (1996, p. 2) argue that the consistency of the final result need not be complete, as long as it is sufficient to meet the goals of the development effort. Conflicts may be dealt with by either integrating the viewpoints in a single operation, or resolving conflicts as they arise on a case-by-case basis. Superficially, this is similar to the approach taken in conventional methods, but the two key differences are:

1. conflict resolution is carried out during the requirements definition and design phases, as opposed to during the implementation phase; and

- multiple viewpoints are allowed to coexist, as opposed to enforcing a single ‘correct’ viewpoint.

The previous discussion implies the existence of two main approaches to conflict resolution, although it would probably be more accurate to say that there is a spectrum of conflict resolution approaches, with these two approaches at the extremes. These two approaches, which are summarised in Table 2.1, are:

- choosing a single ‘correct’ viewpoint and suppressing conflicts; and
- allowing multiple viewpoints to coexist and resolving conflicts between them.

**Table 2.1:** Approaches to resolving viewpoint conflicts

|  | <b>Single ‘correct’ viewpoint</b>         | <b>Multiple conflicting viewpoints</b>   |
|--|---|--|
| <b>Paradigm</b>                                | Objectivist                               | Subjectivist                             |
| <b>Phase of systems development life cycle</b> | Implementation                            | Requirements definition and design       |
| <b>Source of conflict</b>                      | Inconsistency with ‘discarded’ viewpoints | Inconsistency between defined viewpoints |
| <b>Resolution method</b>                       | ‘Special case’ handling                   | Viewpoint conflict resolution            |
| <b>Documentation of resolution</b>             | Sometimes                                 | Yes                                      |

The first approach is an example of the *objectivist* paradigm (Klein and Hirschheim, 1987) of data modelling, which effectively states that there is always some kind of underlying ‘truth’ to be found when modelling reality. In other words, a single, ‘correct’ solution can always be found. Conversely, the second approach is an example of the *subjectivist* paradigm (Klein and Hirschheim, 1987), which states that there is no underlying ‘truth’, only interpretations of reality, constructed according to the viewpoint holder’s particular social and organisational context (Darke and Shanks, 1995a, p. 2).

As noted above, a typical approach to inconsistency during the requirements definition and design phases has been to apply an objectivist approach and avoid the

inconsistency by either ignoring it completely, or by settling on a single, consistent viewpoint and treating the inconsistency as a special case during the implementation phase. There are notable exceptions to this, such as the Object-Oriented Software Engineering (OOSE) approach<sup>1</sup> of Jacobson et al. (1992), in which *use cases* model the differing viewpoints of various *actors*.

Easterbrook (1991a, Section 3.3.1) refers to the avoidance of inconsistency as ‘the single viewpoint bias’ and notes that it has been criticised by several authors. If the conflict is caused by a genuine misunderstanding or error, then resolving the conflict is simply a matter of correcting the error. There may, however, often be several different, but equally ‘correct’, solutions to any but the most trivial of data modelling problems. Easterbrook’s (1991a) library example shown in Figure 2.2 on page 23 illustrates this. The process of generating a single ‘correct’ viewpoint may result in much useful information being discarded, especially when different viewpoints conflict, as there is usually some valid reason for such conflicts. If the conflict is a result of differing perspectives, then an objectivist approach can result in two major problems:

1. a potential reduction in the efficacy of the final system; and
2. there may be no way of determining at a later date how a conflict was resolved, as there is often no documentation of the resolution.

The first problem arises because the analyst may be ignoring real and important aspects of the phenomenon being modelled. Suppose an analyst elicits three non-overlapping viewpoints of a problem. If only one of these is chosen as the ‘correct’ viewpoint, it will make the final system virtually useless for dealing with the other viewpoints. As a result, users would either have to drop their use of the alternate viewpoints, or not use the system at all.

The second problem (that of no documentation of the conflict resolution) will typically not manifest itself until the system requires maintenance, although it could arise during the specification process if the specification goes through considerable revision. If a conflict is dealt with using an objectivist approach, then it is possible that no record will be kept of the fact that there even was a conflict, let alone how it was resolved

---

<sup>1</sup>OOSE later merged with the OMT and Booch object-oriented approaches to form the Unified Modelling Language or UML (Rational Software Corporation, 1997).

(Easterbrook, 1991a, pp. 91–92). When the problem is revisited at a later date, and the conflict is rediscovered, the maintainers may not be able to determine how the conflict was originally resolved, thus wasting effort in resolving the conflict for a second time. If the original conflict was implemented as a special case, there may be comments in the application source code documenting this, but this is by no means guaranteed.

Easterbrook (1991b) and Easterbrook and Nuseibeh (1995; 1996) have examined these problems in depth, and Easterbrook (1991a) has developed a framework and tools for negotiated resolution of conflicts among viewpoints. Other researchers have approached the problem of conflict resolution from a formal basis. In particular, Miller (1994) has explored the problem of resolving conflicts among schemas using the concept of relative information capacity, which will be discussed further in Chapter 8.

## 2.4 Data translation methods

The approach taken by this research is to facilitate the use of multiple representations within a viewpoint by translating models from one representation to another. Relevant aspects of the field of data translation must therefore be reviewed.

Data translation is an important part of everyday computing activities. Data are translated from one data format to another by an *interface* (Pascoe and Penny, 1995) that translates elements of the source format into elements of the target format.

Several issues arise when translating data from one form to another, for example (Pascoe and Penny, 1990; Gawkowski and Mamrak, 1992):

- Data formats may use different encoding schemes (for example, ASCII versus EBCDIC). This issue is not particularly relevant to translating between representations, and will not be discussed further.
- Translating from one format to another may result in a loss of information if the target format is not as expressive as the source format. This determines the *quality* of that translation, and is discussed in Section 2.4.1.
- An interface between two specific formats may not exist, requiring either a new interface to be generated, or the translation to be made via some other format(s),

according to some *interfacing strategy*. This determines the *performance* of a translation, and is discussed in Section 2.4.2. Interfacing strategies are discussed in Section 2.4.3.

- Many interfaces are generated manually in an ad hoc fashion, when they could be generated automatically, thus reducing the potential for coding errors. This issue may be alleviated by the use of *interface specification languages*, which are discussed in Section 2.4.4.

### 2.4.1 Translation quality

When translating from one format to another, information may be lost. The *quality* of a translation is determined by how well it deals with this information loss (Pascoe and Penny, 1990). In practical terms this refers to how completely a translation maps items and structures from the source format to the target format. It is important to note that loss of information during a translation is often unavoidable, as the target format may not be able to express the same set of information as the source format. For example, when translating from an entity-relationship diagram (ERD) to a functional dependency diagram, it is likely that the entity names cannot be translated to anything meaningful, as functional dependencies have no concept of 'entity'. This loss of information is not the fault of the translation; rather it is because of a mismatch in the expressive powers of the source and target formats. The impact of this mismatch on facilitating the use of multiple representations is discussed further in Chapter 3.

For a translation to be of the highest quality possible, any potential loss of information must be minimised, that is, a translation must translate *all* of the information that can conceivably be translated, not just that which is expedient or convenient. As the amount of information translated decreases below this maximum, so does the quality of the translation. For example, when translating from an ERD to an SQL schema, it is possible to automatically generate SQL foreign keys from the ERD by examining the optionality and cardinality of the relationships between entities (see Section 5.3 on page 114). Many CASE tools, however, do not do this; instead they require developers to explicitly define all foreign keys in the ERD. If these foreign keys are not defined, then they are not generated in the resultant SQL, despite the fact that it is possible to



do so. The ERD to SQL translations used by many CASE tools are therefore of lower quality than they could be.

Translation quality is also affected by the number of interfaces required to translate from the source format to the target format. Each interface may potentially result in a loss of information, so the total loss of information will tend to increase with the number of interfaces (Pascoe and Penny, 1990, pp. 152–153). It is therefore important to minimise the total number of interfaces required to perform a translation between two formats. This number is determined to a large extent by the interfacing strategy used.

One of the goals of this research is to improve the quality of translations, which is achieved in two ways in this thesis. Translations in this thesis are defined by a collection of rules. A special type of rule, known as a *heuristic*, can improve the quality of a translation by translating information that would otherwise not be translated. Heuristics are introduced in Section 4.2 on page 68, and their effect on translation quality is discussed in Chapter 8. It is also possible to improve the quality of a translation by adding extra information or metadata to the source, thus providing additional hints to the translation. Missing information could also be acquired from the user during the translation. This process is known as *enrichment*, and is discussed further in Section 4.6 on page 91.

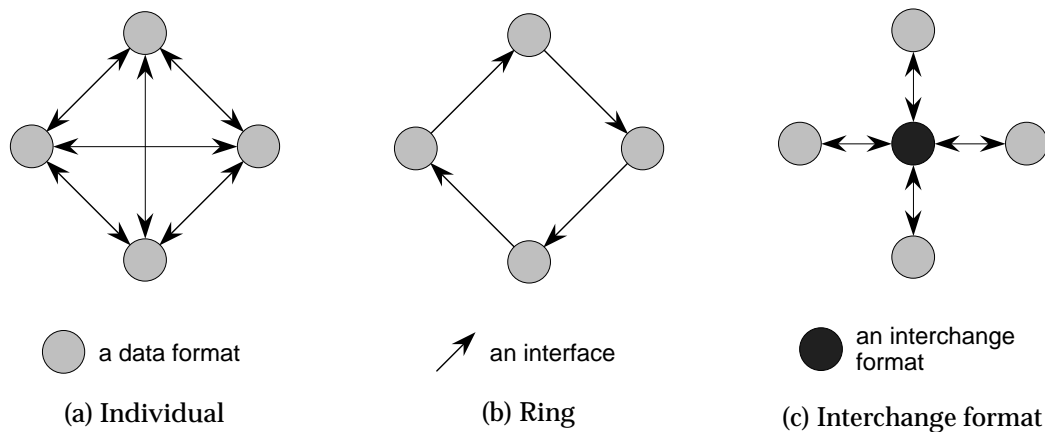
### **2.4.2 Translation performance**

The *performance* of a translation is determined by the number of distinct interfaces required to translate from one format to another (Pascoe and Penny, 1990). For example, suppose there is no interface between formats  $P$  and  $R$ , but there are interfaces between  $P$  and  $Q$ , and between  $Q$  and  $R$ . It is thus possible to translate from  $P$  to  $R$  by translating data first into format  $Q$ , then into format  $R$ . This will usually be less efficient than translating directly from  $P$  to  $R$ , and may also affect the quality of the translation, as noted above. Performance is directly affected by the interfacing strategy used, as this determines the number of distinct interfaces that are required between each pair of formats.

### 2.4.3 Interfacing strategies

Fosnight and van Roessel (1985) identify three possible interfacing strategies (illustrated in Figure 2.3) for performing translations:

1. The *individual* interfacing strategy, where separate interfaces are constructed between each pair of formats. This is also known as the *pairwise technique* (Mamrak and Barnes, 1994) or the *direct translation approach* (Su et al., 1992).
2. The *ring* interfacing strategy, where individual formats in a group are connected in series.
3. The *interchange format* interfacing strategy, where translations are performed via an intermediate format. This is also known as the *intermediate-form technique* (Mamrak and Barnes, 1994) or the *indirect translation approach* (Su et al., 1992).



**Figure 2.3:** Interfacing strategies

Pascoe and Penny (1990) show that the individual interfacing strategy is the best of the three with respect to the quality and performance of the translation. The individual strategy also has the advantage that an environment based on it is easily extensible. Adding a new format does not require changes to any other format, although it does require the definition of at least two interfaces in order to be useful (one ‘to’ the format and one ‘from’). This is, however, the individual strategy’s major disadvantage: to provide a complete set of translations, a potentially large number of interfaces must

be constructed. For  $n$  formats,  $n(n - 1)/2$  interfaces must be constructed to implement all possible translations.

The ring interfacing strategy is the worst with respect to performance (in the worst case,  $n - 1$  translations are required), and as a result, also the worst with respect to quality. It does however share the advantage of being able to construct easily extensible translation environments — as with the individual strategy, changes are not required to other formats, and only two additional interfaces need to be defined for each new format. The number of distinct interfaces required is better than that of the individual interfacing strategy — for  $n$  formats, at most  $n$  distinct interfaces are required.

The interchange format interfacing strategy provides a reasonable compromise — performance is good (exactly two interfaces per format), and at most  $2n$  distinct interfaces are required. The main argument against this strategy is the effect it can have on the quality of the translations. Su et al. (1992, p. 7) state that “the expressive power of the intermediate model has to be at least as strong as the sum of all the data models to be handled by the translation system”. Defining an interchange format in terms of the formats currently in use may result in the interchange format becoming a ‘moving target’ — as new formats are added, the interchange format may often need to be updated to handle them (Pascoe and Penny, 1990). This can lead to a proliferation of potentially incompatible versions. It may also be difficult to combine different formats into a coherent interchange format because of incompatibilities and differing levels of abstraction across formats (Su et al., 1992).

Consider the situation that exists with SQL. Originally designed as a language for accessing and defining relational databases, SQL has evolved into a standard language for building, manipulating and communicating among relational database management systems (Date and Darwen, 1993, p. 7). Despite early efforts at standardisation (ISO-IEC, 1987; ISO-IEC, 1989), SQL splintered into a disparate collection of incompatible dialects as database vendors attempted to differentiate their products (the CASE tool EasyCASE supports 31 distinct dialects of SQL, for instance). The efficacy of SQL as a standard interchange format for relational databases was thus diluted. Attempts have been made to remedy this situation with newer versions of the SQL standard, such as SQL/92 (ISO-IEC, 1992a) and the forthcoming SQL3, by incorporating many of the previously proprietary features developed by vendors. This has ameliorated the

incompatibility problem, but has produced a highly complex standard that is difficult for vendors to understand and implement (Date and Darwen, 1993, p. vii).

Complexity can also be a common feature of interchange formats, as they need to handle the many different constructs and conventions that other formats provide (Su et al., 1992, p. 7). The Standard Generalised Markup Language (SGML) for structuring documents is a powerful, extensible language that is capable of handling any document structuring convention that can be described in a coherent manner, but as a result, it is extremely complex, making it very difficult and time-consuming to develop SGML tools (Tittel, 1998).

Another example of this complexity is the CASE Data Interchange Format (CDIF) defined by the Electronic Industries Association (CDIF Technical Committee, 1994a; Ernst, 1997). This is an interchange format designed to allow CASE tools to exchange data. It consists of an extensible meta-model (CDIF Technical Committee, 1994b) that is partitioned into several *subject areas*, such as data modelling, data flow analysis and project management. As with SGML, this model is extremely complex, and becomes even more so as new subject areas are added, especially when those subject areas overlap in some way. For instance, both the data modelling (CDIF Technical Committee, 1996a) and data flow model (CDIF Technical Committee, 1995b) subject areas include the concept of an *attribute*. The approach taken by the EIA to solve this problem is to make all elements of a subject area subclasses of elements in the 'foundation' subject area (CDIF Technical Committee, 1994c).

Because of the issues with the interchange format and ring interfacing strategies outlined above, the individual interfacing strategy has been chosen for this research. This will typically provide higher quality and higher performance translations, albeit at the expense of requiring a greater number of interfaces. It may, however, be possible to reduce somewhat the number of interfaces without compromising translation quality; this has been left as an area for future research and will be discussed in Chapter 10.

#### **2.4.4 Interface specification languages**

In the past, the process of creating new interfaces has often been an ad hoc process in which programmers must manually code each individual interface (Amor, 1997,

p. 64). This approach to interface implementation has the advantages of using standard programming languages and providing direct control over the functionality of the interface. It does, however, have the following distinct disadvantages:

- the efficacy of an interface can be reduced by coding errors;
- interfaces can be difficult to maintain;
- different interfaces may not be specified in a consistent manner (for example, different languages, coding styles or paradigms);
- interfaces may be coded to take advantage of specific machine or operating system functionality, thus reducing portability;
- interface implementations may be removed from the semantics of the problem domain (Amor, 1997, p. 63); and
- it may be harder to involve domain experts in the specification process (Amor, 1997, p. 63).

A possible solution to several of these problems is to use a high-level interface specification language to define interfaces (also known as a *mapping specification language*). Interfaces may be specified in a consistent manner using a single language, and these specifications may then be used to generate (that is, implement) translators for particular situations. This solves the problems of specification consistency and portability, and should help reduce the effect of coding errors and improve maintainability by reducing the amount of code required. Interface specifications will be focused more on the semantics of the problem domain and less on the implementation details, which makes it easier to include domain experts in the specification process (Amor, 1997, p. 63).

Amor (1997, pp. 67–69) has proposed the following list of requirements for an ideal interface specification language:

**Language level:** The language should use a high-level notation.

**Language notation and modelling environment:** Both a textual and a graphical notation should be provided, which are supported by an appropriate modelling tool.

**Language style:** A declarative notation is likely to provide the highest level of specification, but some mappings may also require a procedural notation.

**Bidirectionality:** Interfaces may be bidirectional or unidirectional, that is, they may be executed either in both 'directions', or in one 'direction' only. Any specification language should therefore support both types.

**Conditional mapping:** Different mappings may apply based on certain conditions, so any language must allow the specification of constraints on the applicability of a given mapping.

**Aggregation:** Some means of aggregating information to go from a very detailed format to a less detailed format must be provided.

**Relationship handling:** The ability to manipulate relationships in a schema is needed.

**Initialisers:** These provide the ability to initialise new objects that are created by the interface, and also provide a means of making explicit some assumptions that are implicitly specified in a schema.

**Unit handling:** The ability to convert values between different unit types is required, as different schemas may use different units or magnitudes of units.

**Type handling:** Different formats may use different data types or structures to express the same information. Some means of mapping between these types is therefore required.

Amor (1997, Section 4.2) reviewed several interface specification languages and mapping methods with respect to these requirements, including three variants of the EXPRESS language (ISO-IEC, 1992b), Transformr (Clark, 1992), EDM-2 (Eastman et al., 1995), KIF (Genesereth and Fikes, 1992), Superviews (Motro, 1987) and relational views, and found that all of them violate one or more of the requirements. These deficiencies in existing interface specification languages led him to develop the *View Mapping Language* (VML). VML is a high-level declarative language for specifying mappings between schemas (Amor, 1997, Chapter 5). It was designed to address many of the shortcomings with other languages and meets all of the requirements listed above. VML is

based on an object-oriented derivative of Prolog known as Snart (Grundy, 1993), thus inheriting the declarative nature of Prolog, while also providing the advantages of an object-oriented programming language, such as complex type handling and inheritance.

VML specifications define the mappings between classes of objects in the source and target schemas. Mappings may have constraints associated with them (known in VML as *invariants*) that limit the applicability of a particular mapping. A mapping between classes comprises a collection of *equivalences* that define the mappings between the individual attributes of the objects. VML also has many other powerful features (Amor, 1997):

- VML allows the specification of arbitrary mappings. Most mappings may be specified declaratively, but a procedural syntax is provided for more complex mappings.
- A graphical notation is defined to allow expression of mappings at a high level of abstraction.
- VML is a declarative language, so mappings may be applied in arbitrary order.
- Mappings between schemas may be unidirectional (that is, either schema A  $\rightarrow$  schema B, or schema B  $\rightarrow$  schema A) or bidirectional (that is, schema A  $\leftrightarrow$  schema B).
- Mappings may either create a new schema, or incrementally update an existing schema.

An extended version of VML will be used in Chapter 5 to specify the details of translations between different representations.

## 2.5 Summary

In this chapter have been reviewed three areas of research that are related to the objective of facilitating the use of multiple modelling representations. These areas are: the use of multiple representations for data modelling, viewpoint-oriented methods and data translation.

In Section 2.2 three research projects were discussed that deal with the use of multiple representations for data modelling in different ways. The first approach, developed by Grundy, is derived from work on multi-view editing systems, and centres on the concept of maintaining consistency among multiple graphical and textual views of an underlying model. Modelling representations are defined within the framework of an integrated data model, and consistency among views is maintained by propagating changes from one view to another via the integrated model. This approach has been implemented in an application development framework called MViews.

The second approach, developed by Atzeni and Torlone, is based on a formal graph theoretic framework in which modelling representations are defined using an abstract metamodel comprising six generic constructs. Translations between representations are achieved by deriving translations from a collection of predefined 'standard' translations. This approach has been implemented in a tool known as MDM.

The third approach, developed by Su et al., is based on an object-oriented rule-based meta-model called ORECOM that is used to define representations. Translations between representations are automatically derived by comparing representations to determine the mappings between constructs. These three approaches were only briefly reviewed in this chapter. They will be examined in more detail in Chapter 9, and compared with the approach taken in this thesis.

In Section 2.3 the concepts and issues of viewpoint-oriented methods were discussed. The concepts of *viewpoint*, *representation*, *technique* and *scheme* were introduced. Viewpoint integration was identified as an important element of viewpoint-oriented methods. The process of resolving conflicts between overlapping viewpoints was discussed, and a comparison was drawn between viewpoint integration and schema integration. A tool developed to aid the schema integration process, relative information capacity (Hull, 1986; Miller, 1994), will be used in Chapter 8 as the basis for a method of measuring the relative quality of translations.

The approach taken by this thesis to the goal of facilitating the use of multiple representations is to perform translations between the instances of these representations. This is related to the more general problem of data translation, and hence a discussion of the issues associated with data translation was presented in Section 2.4. The concepts of translation *quality* and *performance* were introduced and discussed. Both of



these are affected by the *interfacing strategy* used, so three major interfacing strategies (individual, ring and interchange format) were also introduced and the advantages and disadvantages of each strategy discussed. The individual interfacing strategy was chosen for use in this research. Interface specification languages were introduced as a tool for helping with the problem of defining interfaces. The requirements of an ideal interface specification language were described, and a language that meets these requirements, the *View Mapping Language* (VML), was introduced. An extended version of VML will be defined in Chapter 7 for the purpose of specifying translations between representations.

This completes the background review for this research. The concepts introduced in this chapter will now be extended and enhanced. The use of multiple representations has an impact on many of the concepts introduced in this chapter, and is discussed further in the next chapter.



# Chapter 3

## Using multiple representations to describe a viewpoint

### 3.1 Introduction

It was stated in Section 2.3.2 on page 19 that viewpoints may be described using any of a large number of different representations. There are several arguments in favour of using *multiple* representations to describe a viewpoint, and several authors have proposed this approach (Finkelstein et al., 1989; Darke and Shanks, 1995a; Easterbrook, 1991a). Darke and Shanks (1995a) in particular argue for the use of multiple representations within a *single* viewpoint. The use of multiple representations to describe a single viewpoint provides three important advantages:

- a more thorough understanding of a viewpoint;
- a means of highlighting potential inconsistencies within a viewpoint; and
- the opportunity for designers to describe a viewpoint using the most appropriate tools.

Multiple representations allow the description of a viewpoint in many different ways. It can be argued that no single representation will be adequate to fully describe a viewpoint (Darke and Shanks, 1995a, p. 4), and indeed, the current plethora of modelling approaches suggests that a single representation is inadequate to fully describe even a *single* viewpoint, except in trivial cases. It is therefore reasonable to expect that using multiple representations will result in a more complete description of a viewpoint, which in turn implies that a more thorough understanding of a viewpoint could be built using multiple representations than if just a single representation was used.

Environments that facilitate the use of multiple representations must deal with the issue of maintaining consistency across descriptions. This is particularly important within a single viewpoint, as a viewpoint is by definition internally consistent (see Section 2.3.2 on page 17). Two approaches to consistency maintenance and the issues arising from these approaches are discussed in Section 3.6. It is also important to be able to identify and correct inconsistencies that may arise between different descriptions of a viewpoint. Potential inconsistencies between descriptions may be highlighted by translating the descriptions so that they are expressed using the same representation. Suppose a developer describes a viewpoint using an entity-relationship diagram (a semi-formal technique), from which he derives a collection of relations. If the developer describes the same viewpoint using a functional dependency diagram, he can use approaches such as Smith's Method (Smith, 1985) to generate a second collection of normalised relations. The second collection can then be compared against the first collection for obvious discrepancies. This provides a useful aid for evaluating the consistency of a viewpoint and will be discussed further in Section 4.5 on page 86.

The third advantage of using multiple representations is that designers are able to choose the representation that is most appropriate to the part of the problem they are working on at the time. They could even choose to use the representation that they best understand or prefer (Atzeni and Torlone, 1996a); for example, designers who naturally think in graphical terms might be more comfortable with a graphical style of representation, such as entity-relationship modelling, whereas non-graphical thinkers may prefer a formal representation, such as the relational model (Batra and Srinivasan, 1992; Batra and Antony, 1994). Easterbrook (1991a, p. 56) also points out that different representations "are more suited to particular areas of knowledge, in the same way that different programming languages suit particular problems", while Atzeni and Torlone (1993, p. 349) note that different representations may be useful at different levels of abstraction.

The use of multiple representations has already been identified as a useful area of research (Atzeni and Torlone, 1996a; Grundy, 1993, see also Section 2.2), and the use of multiple representations within a single viewpoint was suggested as early as 1989 by Finkelstein et al. It is therefore interesting to note that despite the potential advantages outlined above and the encouragement of several authors, Darke and Shanks (1996,

p. 101) found in a review of twelve different viewpoint development approaches that only two supported multiple representations to describe a single viewpoint. These two approaches were the Soft Systems methodology (Checkland, 1981) and Scenario Analysis (Hsia et al., 1994), both of which are user viewpoint approaches rather than developer viewpoint approaches.

Consequently, this research will follow the approach of using multiple representations to describe a single developer viewpoint. The focus will be on developer viewpoints described using formal and semi-formal representations. Informal representations are not included because of their unstructured and often ill-defined nature, which makes them inherently more difficult to translate in an automated manner than the structured and well-defined formal and semi-formal representations. Since user viewpoints are often described using informal representations, they have also been excluded. The inclusion of informal representations is an interesting line of future research that will be discussed further in Chapter 10.

The viewpoint framework introduced in Chapter 2 provides a useful context for discussing representations and translations between them, but some confusion over the definition of the term ‘representation’ was identified. This confusion is addressed in Section 3.2, and the basic framework is extended with the concepts of *descriptions*, *constructs* of representations and *elements* of descriptions. A notation for expressing these items is described in Section 3.3. This notation provides a concise and consistent means of expressing representations, descriptions, and particularly constructs and elements, which would be cumbersome to express using natural language.

A representation comprises a collection of constructs that have certain properties and are associated in various ways. Thus, in order to define a representation, one must define its constructs and the associations between them. An approach to defining the constructs of a representation, the properties of those constructs and the associations between constructs is described in Section 3.4.

The definition of a representation and its constructs determines what may be expressed using that representation; this is termed here the *expressive power* of a representation, which is discussed in Section 3.5. The expressive powers of two representations will typically overlap in some way, which has an effect on the quality of translations between those representations.

## 3.2 Extending the viewpoint framework

### 3.2.1 Representations

In Section 2.3.2 on page 19, some confusion was identified over the use of the terms ‘representation’, ‘technique’ and ‘scheme’, and an informal definition of the concept of a representation was presented. This confusion will now be addressed, and all three terms formally defined (see also Stanger and Pascoe, 1997a).

Informally, a data modelling representation can be thought of as comprising two main parts:

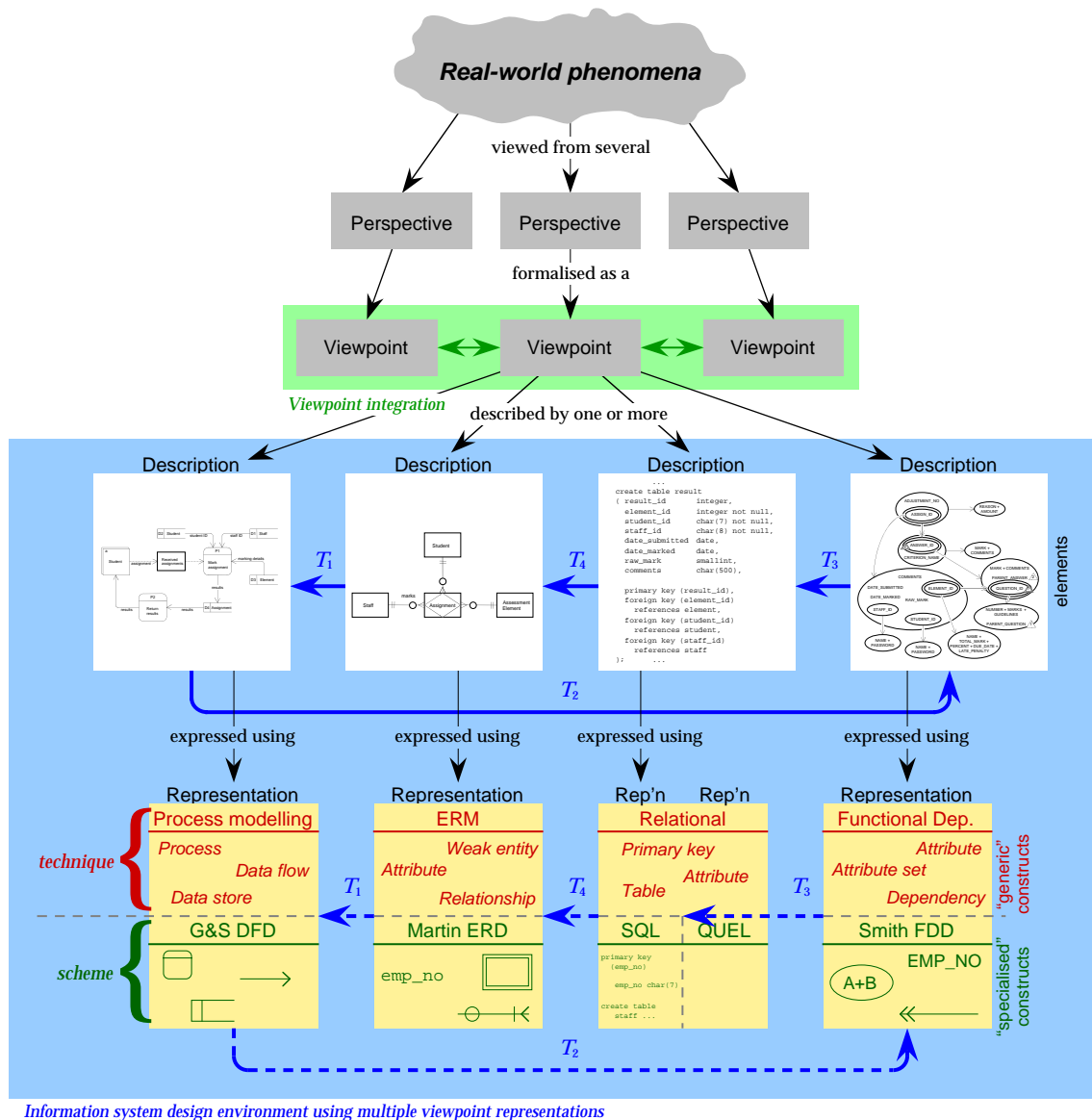
1. a *generic* part that specifies the generic constructs that may be used to describe a viewpoint, such as entities, relations, and so on; which then determines
2. a *specialised* part that specifies the constructs peculiar to the representation, along with their visual appearance or notation, such as boxes for entities, lines for relationships, and so on.

Finkelstein et al.’s (1989) use of the term ‘style’ does not clearly distinguish between these two parts; conversely, the ‘techniques’ and ‘schemes’ of Darke and Shanks (1994) seem to match these two parts quite well. It is therefore proposed to use the term *technique* to refer to the generic part of a representation, and the term *scheme* to refer to the specialised part of a representation. As noted by Darke and Shanks (1994), each technique may have a number of schemes associated with it. In practical terms, a technique can be thought of as a modelling ‘approach’, such as the entity-relationship approach or the relational model, and a scheme can be thought of as a particular ‘notation’ within that approach, such as a particular entity-relationship notation or relational calculus.

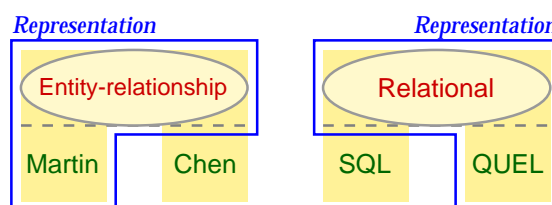
A *representation* can thus be formally defined as the combination of a particular technique and a particular scheme, as shown in Figure 3.1, and is analogous to Atzeni and Torlone’s (1993) concept of a ‘model’. In general, a technique may have one or more associated schemes, but each combination of a technique and a scheme forms a distinct representation. For example, the relational model is a technique, with *SQL* and *QUEL* being two possible schemes, but the combinations  $(\textit{Relational}, \textit{SQL})^1$  and  $(\textit{Relational}, \textit{QUEL})$  form two distinct representations, as shown in Figure 3.2. Similarly,

---

<sup>1</sup>In practice, the many dialects of SQL will form many different representations. This has been ignored here in the interests of clarity.



**Figure 3.1:** Extending the perspective/viewpoint/representation framework [after Stanger and Pascoe (1997a)]



**Figure 3.2:** Multiple schemes within a technique

the entity-relationship approach (*E-R*) is a technique, with  $ERD_{Martin}$  and  $ERD_{Chen}$  as two possible schemes. The combinations  $(E-R, ERD_{Martin})$  and  $(E-R, ERD_{Chen})$  again form two distinct representations.

It is expected that a technique will not attempt to specify all the possible concepts for all possible schemes within that technique. Such an approach would suffer from the same problems as interchange formats, in that the technique could potentially need to be modified every time a new scheme was added. Rather, a technique defines the ‘base’ model, which is then specialised and extended by schemes to form a representation. This implies that a scheme may provide constructs to a representation that have no analogue in the technique. For example, the relational technique (Codd, 1970) does not include constraints, but they are an important feature of the relational scheme SQL/92. Similarly, type hierarchies are not part of the E-R technique (Chen, 1976; Chen, 1977), but they do appear in some E-R schemes.

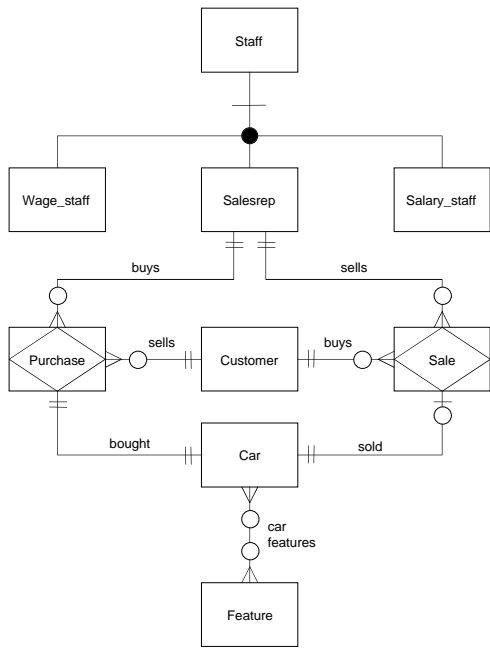
### 3.2.2 Descriptions

Representations are an abstract concept, so they must be instantiated in some way in order to describe the content of a viewpoint. Such an instantiation is analogous to Atzeni and Torlone’s (1995, p. 8) concept of an ‘allowed scheme’, that is, a schema that is valid within a particular model. Another way of viewing the instantiation of a representation is as the set of ‘statements’ that describe a viewpoint or some subset thereof. Finkelstein et al. (1989, p. 43; see also Section 2.3.2 on page 17) refer to this as a *specification* or *description*; Easterbrook (1991a, p. 54) also refers to this concept as a description. The author has adopted the term ‘description’ as it emphasises the idea that they are used to *describe* a viewpoint.

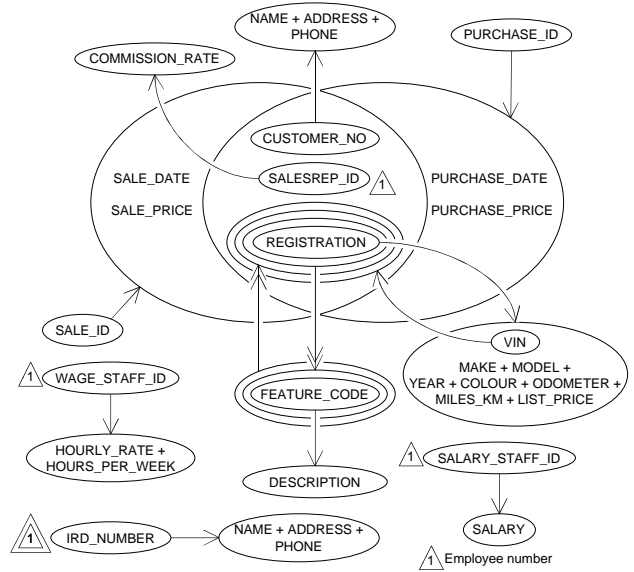
A viewpoint is thus specified by a set of *descriptions*, each expressed using some representation, as shown in Figure 3.1. Each description may describe either the whole viewpoint or some subset of the viewpoint; this is analogous to the concept of a ‘view’ in multi-view editing environments (see Section 2.2.1 on page 9). In Figure 3.3 is shown a developer viewpoint of a simplified used car dealership, specified by the union of four descriptions (the  $D(V, T, S)$  notation is described in Section 3.3):

- (a) an entity-relationship description expressed using Martin E-R diagram notation (see Appendix A; Martin, 1990; Evergreen Software Tools, 1995b);





(a)  $D_1(V_{cars}, E-R, ERD_{Martin})$



(b)  $D_2(V_{cars}, FuncDep, FDD_{Smith})$

```

...
create table car
( registration char(6),
  vin          char(20) not null unique,
  make         char(20),
  model        char(20),
  year         smallint,
  colour       char(20),
  odometer     integer,
  miles_km     char(1),
  list_price   integer,
  purchase_id  char(6) not null unique,
  sale_id      char(6) unique,

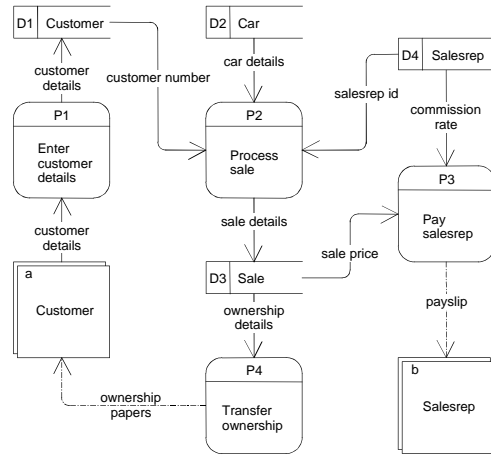
  primary key (registration),
  foreign key (purchase_id)
    references purchase,
  foreign key (sale_id)
    references sale
);

create table purchase
( purchase_id char(6),
  purchase_date date,
  purchase_price integer,
  customer_no char(6) not null,
  salesrep_id char(7) not null,
  registration char(6) not null unique,

  primary key (purchase_id),
  foreign key (customer_no)
    references customer,
  foreign key (salesrep_id)
    references salesrep,
  foreign key (registration)
    references car
);
...

```

(c)  $D_3(V_{cars}, Relational, SQL/92)$



(d)  $D_4(V_{cars}, DataFlow, DFD_{G\&S})$

**Figure 3.3:** Four descriptions of the same viewpoint

- (b) a functional dependency description expressed using Smith functional dependency diagram notation (see Appendix B; Smith, 1985);
- (c) a relational description expressed using SQL/92 (Date and Darwen, 1993); and
- (d) a data flow description expressed using Gane & Sarson data flow diagram notation (see Appendix A; Gane and Sarson, 1979; Evergreen Software Tools, 1995b).

Similarly, a user viewpoint might be specified by the union of a natural language description and a collection of diagrammatic descriptions. Descriptions may be distinct from each other, or they may overlap, in a way analogous to the way viewpoints overlap. Such redundancy can be useful in exposing conflicts, as noted by Easterbrook (1991a, p. 56).

While a translation specifies a mapping from the constructs of one representation to those of another, it is not the representations that are translated; rather it is the descriptions expressed using those representations that are translated.

### 3.2.3 Constructs and elements

Every representation comprises a collection of *constructs*, which are analogous to Atzeni and Torlone's (1993, p. 350) concept of a construct. These may be divided into constructs associated with the technique (*technique-level constructs*) and constructs associated with the scheme (*scheme-level constructs*), as shown in Figure 3.1. The nature of a construct is defined by its associations with other constructs, and its *properties*, such as name, domain or cardinality. For instance, as illustrated in Figure 3.4, a data store in a data flow diagram might have the properties *name* (the name of the data store), *label* and *fields* (a list of data fields in the data store). The *flows* property specifies an association between the data store construct and a list of data flow constructs.

In the same way that a description is an instantiation of a representation, an *element* is an instantiation of a construct; elements are combined to build descriptions. Examples of constructs include entities, processes and attributes; elements corresponding to these constructs could be Staff, Generate invoice and address. The translation of a description from one representation to another can be decomposed into a collection of translations of elements or groups of elements (Atzeni and Torlone, 1995, p. 3).

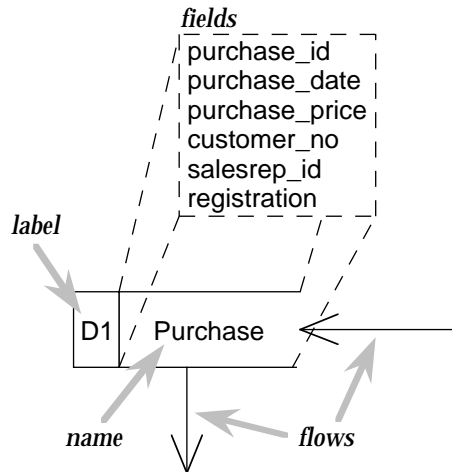


Figure 3.4: Properties of a construct

### 3.3 A notation to express representations, descriptions, constructs and elements

It can be cumbersome to discuss aspects of representations and descriptions using natural language, for example, ‘the Staff regular entity element of the description  $D_1$  (expressed using Martin entity-relationship notation) of the used cars viewpoint’. A concise notation for expressing representations, descriptions, constructs of representations and elements of descriptions is therefore defined in this section. This notation is modelled in part on the data transfer notation of Pascoe and Penny (1995). Using the notation, the statement above would be expressed as:

$$D_1(V_{cars}, E-R, ERD_{Martin}) [staff : \text{MARTINREGULARENTITY}].$$

#### 3.3.1 Description and representation notation

The notation  $D(V, T, S)$  denotes that description  $D$  of viewpoint  $V$  is expressed using constructs of technique  $T$  and scheme  $S$  (this may be abbreviated to  $D$  when  $V$ ,  $T$  and  $S$  are clear). Thus,  $D_1(V_p, E-R, ERD_{Martin})$  denotes a description  $D_1$  of the viewpoint  $V_p$  that is expressed using constructs of the entity-relationship technique ( $E-R$ ) and the Martin ERD scheme ( $ERD_{Martin}$ ). A viewpoint  $V$  may be specified using  $n$  possibly overlapping descriptions  $D_i$ :

$$V = \bigcup_{i=1}^n D_i$$

The notation  $\mathfrak{R}(T, S)$  denotes a representation  $\mathfrak{R}$  that comprises a collection of constructs defined by the combination of technique  $T$  and scheme  $S$  (this may be abbreviated to  $\mathfrak{R}$  when  $T$  and  $S$  are clear). Thus,  $\mathfrak{R}_e(E-R, ERD_{Martin})$  denotes the representation  $\mathfrak{R}_e$  formed by combining the constructs of the entity-relationship technique ( $E-R$ ) with the Martin ERD scheme ( $ERD_{Martin}$ )<sup>2</sup>. This notation is similar to that used by Finkelstein et al. (1989) to describe viewpoint styles, but focuses on the technique and scheme used rather than individual constructs within a representation.

The combination of technique  $T$  and scheme  $S$  forms the representation  $\mathfrak{R}(T, S)$ , so it is also possible to denote the description  $D(V, T, S)$  by  $D(V, \mathfrak{R}(T, S))$ , or simply  $D(V, \mathfrak{R})$ . Thus, the notations  $D_1(V_p, E-R, ERD_{Martin})$ ,  $D_1(V_p, \mathfrak{R}_e(E-R, ERD_{Martin}))$  and  $D_1(V_p, \mathfrak{R}_e)$  are equivalent. The first form is preferred in this thesis as it clearly distinguishes between the technique and scheme, which will prove useful in Chapter 4.

Representations may differ in both the technique and scheme used, or they may share the same technique and differ only in the scheme. Thus, two descriptions  $D_1$  and  $D_2$  of viewpoint  $V$  that are expressed using representations having different schemes  $S_i$  and  $S_j$  are denoted by  $D_1(V, T, S_i)$  and  $D_2(V, T, S_j)$  respectively. Similarly, two descriptions  $D_3$  and  $D_4$  of viewpoint  $V$  that are expressed using representations having different techniques ( $T_k, T_l$ ) and schemes ( $S_m, S_n$ ) are denoted by  $D_3(V, T_k, S_m)$  and  $D_4(V, T_l, S_n)$  respectively.

Consider a viewpoint  $V_q$  that has three descriptions  $D_1, D_2$  and  $D_3$ .  $D_1$  is expressed using the entity-relationship technique and the Martin ERD scheme, and is denoted by  $D_1(V_q, E-R, ERD_{Martin})$ .  $D_2$  is expressed using the functional dependency technique and the Smith functional dependency diagram (FDD) scheme<sup>3</sup>, and is denoted by  $D_2(V_q, FuncDep, FDD_{Smith})$ .  $D_1$  and  $D_2$  differ in both the technique and the scheme used.  $D_3$  is expressed using the entity-relationship technique and the Chen scheme (Chen, 1976), and is denoted by  $D_3(V_q, E-R, ERD_{Chen})$ .  $D_3$  differs from  $D_1$  only in the scheme used.

If the viewpoint, technique or scheme are unspecified, they may be omitted from the notation. Thus, the notation  $\mathfrak{R}_r(Relational, )$  denotes any relational representation, and  $D_1(, FuncDep, FDD_{Smith})$  denotes a Smith FDD in an unspecified viewpoint.

---

<sup>2</sup>Martin notation (Martin, 1990) is described in Appendix A.

<sup>3</sup>The Smith FDD notation is described in Appendix B.

As noted in the introduction to this chapter, this research follows the approach of using multiple representations to describe a *single* developer viewpoint. In other words, given any description  $D_i(V, T_j, S_k)$ ,  $V$  remains constant for all values of  $i$ ,  $j$  and  $k$ . This prevents straying into the areas of viewpoint integration and conflict resolution, which are beyond the scope of this thesis.

### 3.3.2 Construct and element notation

Constructs are the fundamental components of a representation, whereas elements are the fundamental components of a description. Given a representation  $\mathfrak{R}(T, S)$ , a construct  $\text{CON}$  of  $\mathfrak{R}$  is denoted by  $\mathfrak{R}(T, S) [\text{CON}]$ , or, if  $T$  and  $S$  are clear, simply  $\mathfrak{R} [\text{CON}]$ . Much of the time,  $\mathfrak{R}$  will also be clear from the context, allowing the  $\mathfrak{R} []$  notation to also be omitted, leaving just  $\text{CON}$ . The name of the construct itself is denoted by SMALL CAPS; construct names for four representations are defined in Appendix D.

The construct  $\text{CON}$  can be thought of as analogous to the concept of a relational domain in that it specifies a pool of possible ‘values’ from which an element  $e$  may be drawn. The notation  $e : \text{CON}$  is used here to denote that  $e$  is a member of the set of all possible elements corresponding to the construct  $\text{CON}$ . This use of the ‘:’ notation is similar to both domain calculus (Date, 1995, p. 204) and Z (Brien and Nicholls, 1992, p. 6), where it is interpreted as meaning ‘ $e$  is a member of the set  $\text{CON}$ ’.

Now consider a description  $D(V, T, S)$  (or  $D(V, \mathfrak{R}(T, S))$ ). An element  $e$  (instantiated from construct  $\mathfrak{R} [\text{CON}]$ ) of  $D$  is denoted by  $D(V, T, S) [e : \text{CON}]$ , or, if  $T$  and  $S$  are clear, simply  $D [e : \text{CON}]$ . The construct may also be omitted if it is clear from the context, that is,  $D [e]$ . The representation  $\mathfrak{R}$  is omitted from the construct  $\text{CON}$  because  $\mathfrak{R}$  is implied by  $T$  and  $S$  in the description and would therefore be redundant.

Some examples of construct and element expressions are given in Table 3.1 on the next page. Both types of expression may specify a list, as illustrated by the last two examples.

## 3.4 Defining constructs within representations

There has as yet been no discussion of how to define the properties of constructs within a representation, or the associations between these constructs. One approach is to use

**Table 3.1: Examples of construct and element expressions**

|  |   |
|--|---|
| $\mathfrak{R}_e(E-R, ERD_{Martin}) [ERENTITYTYPE]$               | denotes the generic entity construct of the E-R/Martin representation $\mathfrak{R}_e$  |
| $D_1(V, FuncDep, FDD_{Smith}) [s : SSSINGLEVALUED]$              | denotes a single-valued dependency element in the Smith notation functional dependency description $D_1$                              |
| $D_2(V, Relational, SQL/92) [c_1, \dots, c_n : SQL92COLUMN]$     | denotes a collection of column elements in the SQL/92 description $D_2$   |
| $\mathfrak{R}_d(DataFlow, DFD_{G\&S}) [DFDATASTORE, DFDATAFLOW]$ | denotes the data store construct and the data flow construct of the data flow modelling/Gane & Sarson representation $\mathfrak{R}_d$ |

an abstract metamodel to define representations, in which constructs are decomposed into collections of simpler, primitive constructs. This is the approach taken by Atzeni and Torlone (1993), who use a metamodel comprising only six abstract metaconstructs (such as lexical types and functions), which they claim can be used to define the constructs of most representations. Su and Fang (1993) use an extensible object-oriented core model that comprises three structural constructs (objects, classes and associations), and a collection of rules for defining the behaviour of constructs.

A similar approach is taken by the CASE Data Interchange Format (CDIF), which defines a generic, object-oriented metamodel (CDIF Technical Committee, 1994c; CDIF Technical Committee, 1995a) from which specific representations are specialised (for example, CDIF Technical Committee, 1996a; CDIF Technical Committee, 1995b; CDIF Technical Committee, 1996b). Unfortunately, the CDIF standard rather confusingly applies the term ‘meta-model’ to mean a representation and the term ‘model’ to mean a description (CDIF Technical Committee, 1994a, Figure 4). Compare this with Atzeni and Torlone (1993), who use the term ‘model’ to mean a representation and the term ‘scheme’ to mean a description. The MDM concept of a metamodel therefore does not correspond to the CDIF concept of a meta-model; rather it corresponds to the CDIF concept of a ‘meta-meta-model’. A correspondence between the various terminology used by different groups and the terminology used in this thesis can be found in Table 3.4 on page 65.

The object-oriented approach to defining representations used by the CDIF stan-

dard and Su and Fang (1993) provides a useful and easily extensible means of defining representations and their constructs. The CDIF standard defines representations using a variant of the entity-relationship approach that supports object-oriented concepts (CDIF Technical Committee, 1996a). It was initially intended to use the unmodified CDIF meta-models as representation definitions in this thesis, but several issues were encountered, such as:

- The data modelling meta-model has an unfortunate relational bias, for example, “a relational ‘Table’ is also represented by *Entity*” (CDIF Technical Committee, 1996a, p. 17). That is, CDIF entities are assumed to be normalised, which may limit their efficacy for modelling purposes.
- Participation of entities in relationships seems inadequate — the way that the meta-model is structured seems to imply that entities may only participate in a single relationship (CDIF Technical Committee, 1996a, p. 71).
- The CDIF meta-models do not clearly reflect the technique/scheme division identified earlier.

The approach proposed here is to define the representations and the properties of their constructs in an object-oriented manner, using a combination of an entity-relationship diagram to define the associations between constructs, and a data dictionary style table to define the properties of constructs. This provides the opportunity to define representations using standard CASE tools, and also allows the definition of translations between representations using mapping specification languages like Amor’s (1997) View Mapping Language (VML). This approach is similar to that taken by Venable’s (1993) meta-modelling language CoCoA (from COmplex COvering Aggregation), which uses an extended entity-relationship approach to support the modelling of complex problem domains (Grundy and Venable, 1996). Unfortunately, this language was discovered too late to be of use in this research, but may be useful for future work (see Chapter 10).

Techniques are independent of schemes, which means that it is possible to define techniques in isolation. It is not, however, possible to define schemes separately, because schemes are dependent on their technique. A scheme is a specialisation of its

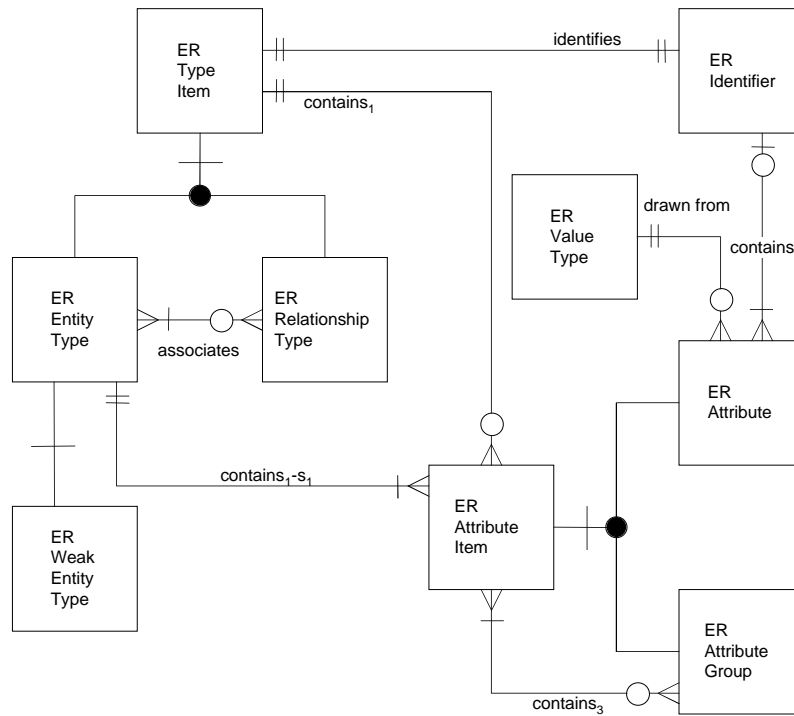
technique, similar to the way that CDIF subject areas are specialisations of the generic CDIF metamodel. Any attempt to build a separate scheme definition would need to include the constructs of the technique to provide the correct context, thus turning the scheme definition into a definition of the entire representation.

The definition of the E-R technique and the representation  $\mathfrak{R}_e(E-R, ERD_{Martin})$  will be summarised here. Complete definitions of the techniques and representations used in this thesis may be found in Appendix D. The definitions of techniques have been derived as much as possible from the ‘canonical’ definition of the approach, with some minor alterations.

The definition of the E-R technique is shown in Figure 3.5 and comprises the ‘core’ constructs `ERENTITYTYPE`, `ERWEAKENTITYTYPE`, `ERRELATIONSHIPTYPE`, `ERIDENTIFIER`, `ERATTRIBUTE` and `ERVALUETYPE`, plus some extra constructs to simplify some aspects of the definition. The key points to note about this definition are:

- The `ERTYPEITEM` construct is a generalisation of the `ERENTITYTYPE` and `ERRELATIONSHIPTYPE` constructs, and is provided for convenience.
- The `ERATTRIBUTEGROUP` construct has been introduced to support composite attributes. Composite attributes are not explicitly mentioned in Chen’s (1977) original definition of the E-R approach, but neither are they specifically ruled out. The `ERATTRIBUTEITEM` construct is a generalisation of both this construct and the `ERATTRIBUTE` construct.
- The `ERRELATIONSHIPTYPE` construct denotes an association between a collection of `ERENTITYTYPE` constructs, and may have attributes. A link between entities is implied by the existence of an `ERRELATIONSHIPTYPE` construct between those entities. It is thus not necessary to explicitly include ‘linking’ attributes in the entities in order to define the link.
- The `ERVALUETYPE` construct is analogous to the concept of a domain.
- Type hierarchies are not included in the technique because Chen’s (1977) original definition does not include them.
- Repeating groups are modelled by the boolean repeating property of the `ERATTRIBUTEITEM` construct.





**Figure 3.5:** Definition of the entity-relationship technique

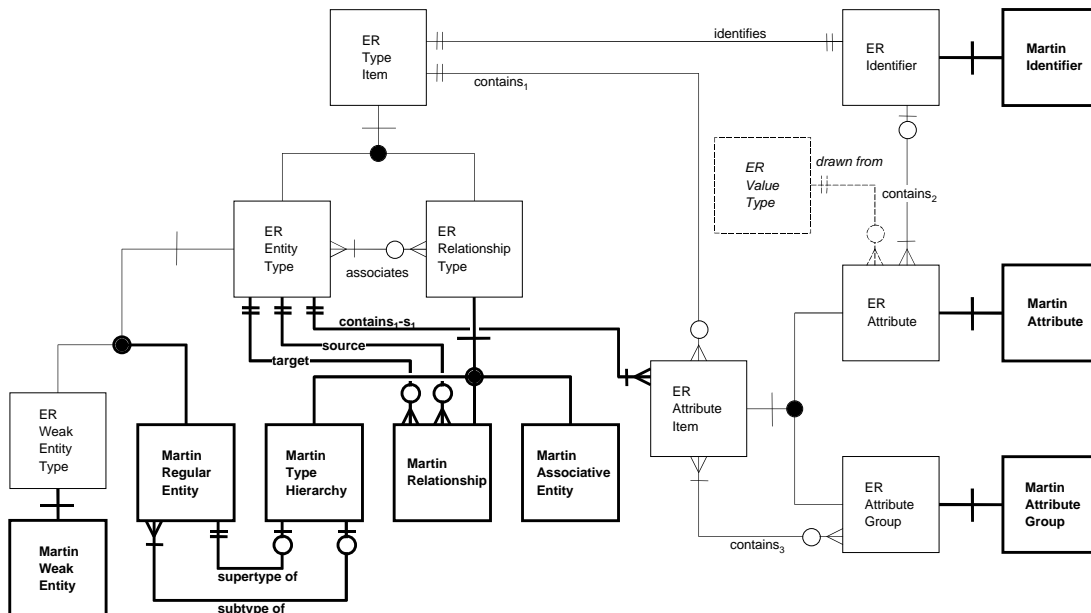
The properties of the E-R technique’s constructs are summarised in Table 3.2 on the next page; a complete description of their meanings may be found in Appendix D. Each construct has a collection of properties that define the nature of the construct. Each property has a nominal ‘data type’, which may be a simple scalar type such as string or integer, an enumerated type, a construct or a list of any of these. The keyword **specialises** is used here to indicate that a construct is a specialisation of some other construct and inherits all the properties of the construct that it specialises. Specialised constructs may also add new properties that are peculiar to that construct.

The definition of the representation  $\mathfrak{R}_e(E-R, ERD_{Martin})$ , shown in Figure 3.6 on the following page, extends the E-R technique definition with the following:

- The `MARTINREGULARENTITY` and `MARTINWEAKENTITY` constructs are further specialisations of the `ERENTITYTYPE` and `ERWEAKENTITYTYPE` constructs respectively. `MARTINWEAKENTITY` elements may be ‘embedded’ within other entity types, to allow for composite attributes that have relationships attached to them. This is analogous to the concept of a ‘repeating group’ within an entity.
- `MARTINRELATIONSHIP` and `MARTINASSOCIATIVEENTITY` are further specialisa-

**Table 3.2:** Construct properties of the entity-relationship technique

|   |   |
|---|---|
| <b>ERTYPEITEM</b><br>name string<br>attributes list(ERATTRIBUTEITEM)<br>identifier ERIDENTIFIER   | <b>dp</b> integer<br>attributes list(ERATTRIBUTE)   |
| <b>ERENTITYTYPE</b><br><b>specialises</b> ERTYPEITEM<br>relationships list(ERRELATIONSHIPTYPE)  | <b>ERATTRIBUTEITEM</b><br>name string<br>containingItem ERTYPEITEM<br>attributeGroups list(ERATTRIBUTEGROUP)<br>repeating boolean |
| <b>ERWEAKENTITYTYPE</b><br><b>specialises</b> ERENTITYTYPE<br>dependentVia ERRELATIONSHIPTYPE   | <b>ERATTRIBUTEGROUP</b><br><b>specialises</b> ERATTRIBUTEITEM<br>attributeItems list(ERATTRIBUTEITEM)                             |
| <b>ERRELATIONSHIPTYPE</b><br><b>specialises</b> ERTYPEITEM<br>entities list(ERENTITYTYPE)<br>cardinalities list(integer)<br>existence_dependent boolean<br>id_dependent boolean | <b>ERATTRIBUTE</b><br><b>specialises</b> ERATTRIBUTEITEM<br>valueType ERVALUETYPE<br>identifier ERIDENTIFIER                      |
| <b>ERVALUETYPE</b><br>name string<br>datatype enumerator<br>size integer  | <b>ERIDENTIFIER</b><br>name string<br>identifiedItem ERTYPEITEM<br>partial boolean<br>attributes list(ERATTRIBUTE)                |



**Figure 3.6:** Definition of the representation  $\mathfrak{R}_e(E-R, ERD_{Martin})$

tions of the ERRELATIONSHIPTYPE construct. A MARTINRELATIONSHIP construct differs from other specialisations of ERRELATIONSHIPTYPE in that it has no attributes, and allows only binary associations. The MARTINASSOCIATIVE-ENTITY construct corresponds more closely to the original definition.

- MARTINATTRIBUTE and MARTINATTRIBUTEGROUP are specialisations of ERATTRIBUTE and ERATTRIBUTEGROUP, respectively.
- MARTINIDENTIFIER is a specialisation of ERIDENTIFIER.
- The ERVALUETYPE construct is not used in  $\mathfrak{R}_e$  and is therefore not specialised.
- MARTINTYPEHIERARCHY is a specialisation of the ERRELATIONSHIPTYPE construct and adds support for type hierarchies. It represents a generalisation association between EENTITYTYPE constructs.

The specialised constructs introduced by the Martin ERD scheme are highlighted in bold in Figure 3.6, and their properties are summarised in Table 3.3.

**Table 3.3:** Construct properties of the representation  $\mathfrak{R}_e(E-R, ERD_{Martin})$

|  |  |
|--|--|
| MARTINREGULARENTITY<br><b>specialises</b> ERENTITYTYPE<br>typeHierarchy MARTINTYPEHIERARCHY<br>embeddedEntities list(MARTINWEAKENTITY) | srcOpt integer<br>dstCard integer<br>dstOpt integer  |
| MARTINWEAKENTITY<br><b>specialises</b> ERWEAKENTITYTYPE<br>embedded boolean<br>embeddedEntities list(MARTINWEAKENTITY)                 | MARTINTYPEHIERARCHY<br><b>specialises</b> MARTINRELATIONSHIPTYPE<br>supertype MARTINREGULARENTITY<br>subtypes list(MARTINREGULARENTITY)<br>exclusive boolean |
| MARTINASSOCIATIVEENTITY<br><b>specialises</b> ERRELATIONSHIPTYPE<br>embeddedEntities list(MARTINWEAKENTITY)                            | MARTINATTRIBUTE<br><b>specialises</b> ERATTRIBUTE  |
| MARTINRELATIONSHIP<br><b>specialises</b> ERRELATIONSHIPTYPE<br>source ERTYPEITEM<br>target ERTYPEITEM<br>srcCard integer               | MARTINATTRIBUTEGROUP<br><b>specialises</b> ERATTRIBUTEGROUP  |
|  | MARTINIDENTIFIER<br><b>specialises</b> ERIDENTIFIER  |

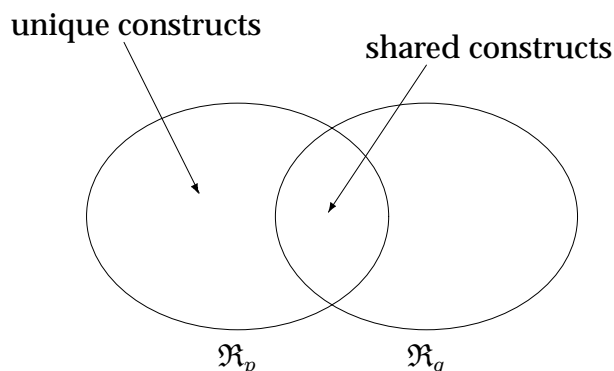
### 3.5 Expressive power of representations

Every representation has boundaries to what may be expressed using the constructs of that representation. These boundaries define what is referred to here as the *expressive power* of a representation. The expressive power of a representation covers the

constructs of a representation and any semantic constraints that may exist between constructs. Representations may overlap in terms of expressive power, that is, some subset of the constructs of one representation can be mapped to constructs of another representation. This concept is referred to here as the *expressive overlap* between two representations. The concept of expressive overlap results in two distinct categories of representation constructs:

- *unique* constructs that are peculiar to a particular representation; and
- *shared* constructs that are shared across two or more representations.

The relationship between these two categories of constructs is shown in Figure 3.7 for two representations  $\mathfrak{R}_p$  and  $\mathfrak{R}_q$ . The ovals denote the expressive power of the representations  $\mathfrak{R}_p$  and  $\mathfrak{R}_q$ .

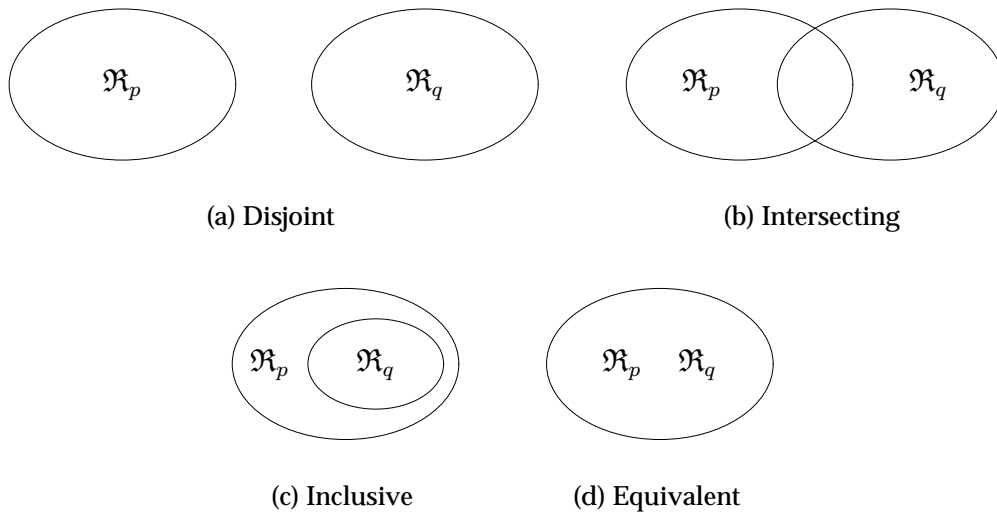


**Figure 3.7:** Unique and shared constructs

The nature of the expressive overlap between a particular pair of representations  $\mathfrak{R}_p(T_i, S_j)$  and  $\mathfrak{R}_q(T_k, S_l)$  can fall into one of four categories, as illustrated in Figure 3.8 and described below. The category of expressive overlap between two representations determines the theoretical best mapping of constructs from the source representation to constructs of the target representation. The aim is to make the ‘implemented’ quality of translations as close to this theoretical maximum as possible.

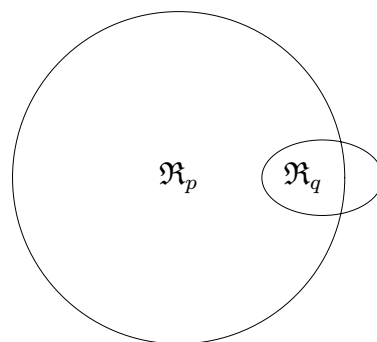
The four categories of expressive overlap are:

**Disjoint** Both  $\mathfrak{R}_p$  and  $\mathfrak{R}_q$  have unique constructs and there are no shared constructs, as shown in Figure 3.8(a). That is, there is no expressive overlap between  $\mathfrak{R}_p$  and  $\mathfrak{R}_q$ , and translating descriptions between them is therefore typically impossible.



**Figure 3.8:** Four categories of expressive overlap

**Intersecting** Both  $\mathfrak{R}_p$  and  $\mathfrak{R}_q$  have some unique constructs and there are some shared constructs, as shown in Figure 3.8(b). That is, there is a partial expressive overlap between  $\mathfrak{R}_p$  and  $\mathfrak{R}_q$ , and it is therefore possible to partially translate descriptions from  $\mathfrak{R}_p$  to  $\mathfrak{R}_q$ , and vice versa. The degree of expressive overlap may, of course, vary considerably. If the expressive powers of  $\mathfrak{R}_p$  and  $\mathfrak{R}_q$  have greatly different ‘capacities’, as shown in Figure 3.9, then the translation of a description from  $\mathfrak{R}_q$  to  $\mathfrak{R}_p$  will generally be more complete than the translation of a description from  $\mathfrak{R}_p$  to  $\mathfrak{R}_q$ . This is because a relatively greater proportion of the constructs of  $\mathfrak{R}_q$  may participate in the translation.



**Figure 3.9:** Asymmetric expressive overlap

**Inclusive**  $\mathfrak{R}_p$  has some unique constructs and some shared constructs, but  $\mathfrak{R}_q$  has only shared constructs, or vice versa, as shown in Figure 3.8(c). That is, one represen-

tation is contained entirely within the other, or more formally, one representation is a *proper subset* (Borowski and Borwein, 1989) of the other ( $\mathfrak{R}_q \subset \mathfrak{R}_p$  in Figure 3.8(c)). That is, anything that can be expressed using constructs of  $\mathfrak{R}_q$  may also be expressed using constructs of  $\mathfrak{R}_p$ , but not vice versa. It is therefore possible to translate descriptions completely from  $\mathfrak{R}_q$  to  $\mathfrak{R}_p$ , but only partially from  $\mathfrak{R}_p$  to  $\mathfrak{R}_q$ .

**Equivalent** Neither  $\mathfrak{R}_p$  nor  $\mathfrak{R}_q$  have unique constructs — all constructs are shared, as shown in Figure 3.8(d). This means that anything that can be expressed using constructs of  $\mathfrak{R}_p$  can also be expressed using constructs of  $\mathfrak{R}_q$ , and vice versa. It is therefore possible to translate descriptions completely in either direction.

The expressive overlap of a pair of representations thus determines the maximum amount of information that may be translated from one representation to the other, which effectively determines the maximum quality of any translation between those representations. The intersecting and inclusive categories are of the most interest, because of the differences in expressive power. Translating a description from a more expressive representation to a less expressive one can result in a ‘loss’ of information, whereas the reverse translation can result in the need to ‘gain’ information during the translation in order to build a sensible target description. The extent of this loss or gain can be affected by how the elements of the source description are translated, which will be discussed further in Chapter 4.

## 3.6 Maintaining consistency among descriptions

An important issue that arises from the use of multiple representations is the need to maintain consistency among the descriptions expressed using these representations. Suppose that a developer describes a viewpoint using a functional dependency diagram (FDD), then translates that into an entity-relationship diagram (ERD). After examining the ERD, the developer may identify some additional entities and alter the ERD appropriately. If the two descriptions are to remain consistent with each other, these changes should be propagated back to the original FDD.

Two approaches to maintaining consistency among descriptions are discussed in this section: the *synchronous* approach, where associated descriptions are kept con-

sistent at all times; and the *asynchronous* approach, where associated descriptions are made consistent with each other only when a translation is initiated.

In the synchronous approach, changes made to one description are automatically propagated to all associated descriptions as appropriate. For example, as attributes and candidate keys are added to the ERD, the FDD will be concurrently updated with appropriate attributes and dependencies. This approach is exemplified by the MViews framework (Grundy, 1993; Grundy and Hosking, 1993a; Grundy and Venable, 1995b; Grundy and Hosking, 1997), which is an object-oriented framework for implementing integrated software development environments that support multiple consistent graphical and textual views. Change propagation is generally synchronous (although this is not required) and facilitated by means of an update record mechanism (Grundy, 1993, Section 5.3.3).

The major advantage of the synchronous approach is that descriptions remain as consistent as possible at all times. Sometimes, however, it may not be possible to propagate a change (Grundy, 1993, p.69). MViews addresses this issue by expressing the relevant update record in human-readable form; this is used to alert the user to the need to modify the affected description(s).

A disadvantage of the synchronous approach is that translations are 'active' at all times. That is, every time a change is made to a description, a translation must be activated to propagate the changes to associated descriptions. These translations may in turn trigger other translations, which could result in considerable background processing that can degrade the overall performance of a modelling environment. Another issue is that a developer may sometimes deliberately wish to build a description that is inconsistent with other descriptions, perhaps to explore an alternative design path. This could be difficult under a purely synchronous approach.

In the asynchronous approach, a series of changes are made to one description, then all of these changes are propagated at once to the associated descriptions. For example, the developer adds several new entities to the ERD, then initiates a translation to propagate these changes to the FDD. An advantage of the asynchronous approach is that the amount of background processing required to maintain consistency is reduced. This is because translations are only activated when needed, thus localising translation processing to well-defined periods.

The main issue with the asynchronous approach is that descriptions are not always consistent with each other. Until a translation is applied, it is more likely that descriptions will be inconsistent with each other, albeit temporarily. In this sense, the asynchronous approach is similar to the transaction mechanism used by database management systems (DBMSs). Transaction boundaries define consistent states of the database, but during the course of a transaction the database may be in an inconsistent state (Date, 1995, p. 379). If the asynchronous approach is treated in this manner, the issue of description inconsistency can be ameliorated. The asynchronous approach also allows developers to deliberately build inconsistent descriptions.

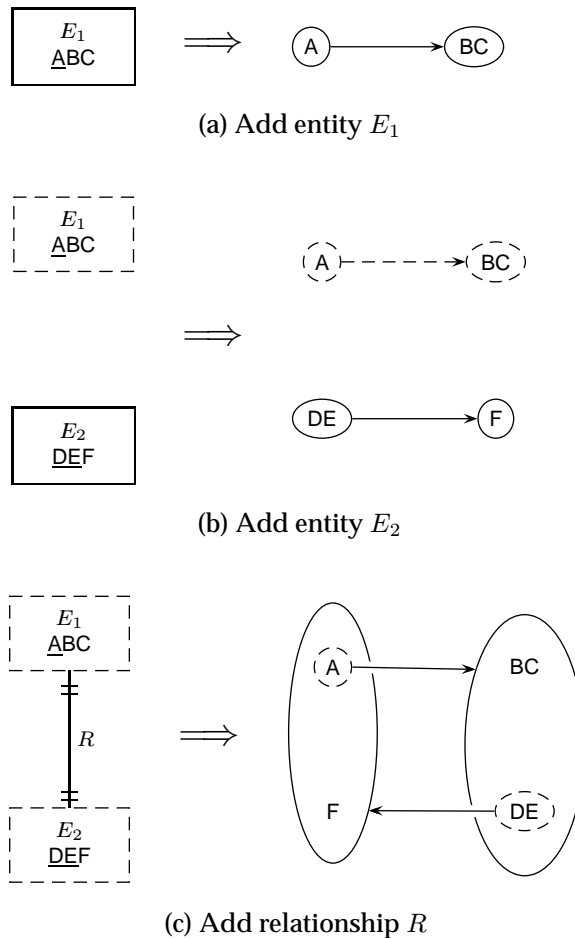
The choice between synchronous versus asynchronous can be influenced to some extent by the interfacing strategy used. The increased processing required to manage a synchronous approach can be ameliorated by following the interchange format interfacing strategy, as all information in a viewpoint can be stored using a single representation. This is the approach taken by most MViews tools (Grundy and Venable, 1995b): descriptions are stored using an integrated data model, and views of this model are 'rendered' in various representations by the environment.

The individual interfacing strategy is more suited to an asynchronous approach, as the information stored in each description will usually be stored using different representations. This is the approach followed by the author, and will be discussed further in Chapter 6.

Translating descriptions between multiple representations can create some interesting difficulties for maintaining consistency among descriptions, regardless of the approach used. Consider the situation illustrated in Figure 3.10, where a developer is building a simple ERD containing two entities connected by a relationship. This ERD is associated with a functional dependency diagram (FDD) that must be kept consistent. Adding the first entity produces the FDD structure shown in Figure 3.10(a). Adding the second entity produces an additional FDD structure, shown in Figure 3.10(b). Adding the relationship between the two entities, however, requires a complete rearrangement of the existing FDD structures, as shown in Figure 3.10(c).

In this example, defining the two entities and the relationship between them forms a single cognitive unit of work (again, this is similar to the concept of a DBMS transaction). It is unlikely that the boundaries of cognitive units could be determined in an

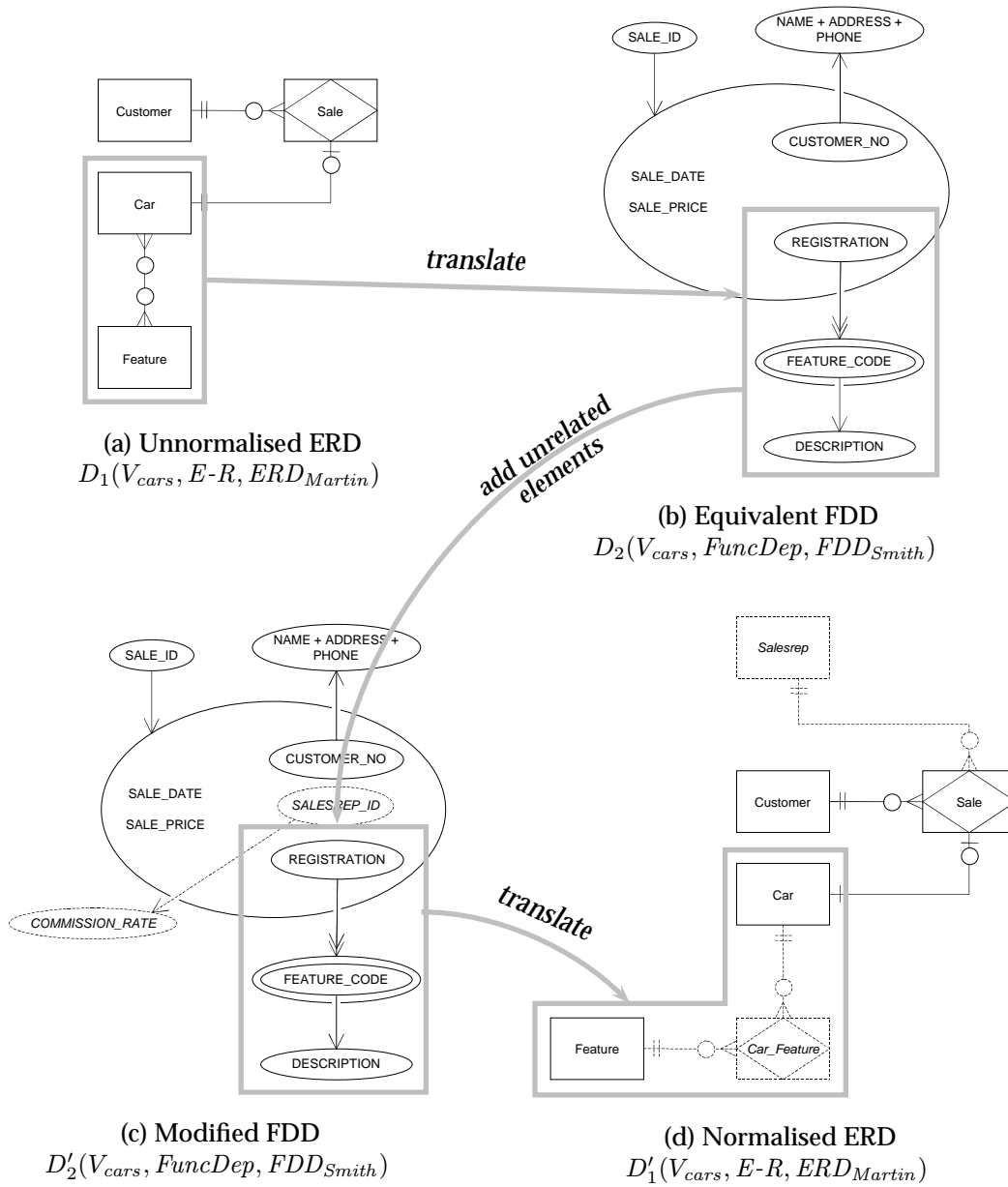




**Figure 3.10:** Sensitivity of a description to changes

automated fashion, however, as these boundaries are dependent on the order in which the developer defines objects. Because of this lack of determinability, a synchronous environment would be more affected by this issue. Asynchronous environments also suffer from this issue, but the effect is likely to be lessened because the developer knows the boundaries of the cognitive units, and can thus activate translations at appropriate points. An alternative solution would be to provide some form of explicit transaction management, similar to that used by Amor (1997) in the mapping system for his View Mapping Language (see Section 2.4.4 on page 32).

A more serious issue arises when a description translation produces unexpected changes. Consider the situation illustrated in Figure 3.11 on the following page. A developer builds the unnormalised ERD description shown in Figure 3.11(a) and translates this to the FDD description shown in Figure 3.11(b). The developer then adds



**Figure 3.11:** Unexpected structural changes caused by a translation

some elements to the FDD that are unrelated to the Car and Feature entities, as shown in Figure 3.11(c). The problem arises when these changes are propagated back to the ERD, as shown in Figure 3.11(d). The FDD to ERD translation produces a normalised ERD with a different structure to the original ERD, and may not be what the developer expected or intended. Again, this issue may occur with both approaches.

An important part of maintaining consistency between descriptions of a viewpoint is the ability to identify correspondences between the constructs of different representations. The better this correspondence, the easier it will be to meaningfully compare and model interchangeably in the different representations. This correspondence can be specified by mapping the constructs of one representation onto those of another. As will be discussed in the next chapter, such a mapping defines a *translation* between the two representations.

The degree of correspondence between the constructs of two different representations can be characterised by the expressive overlap of those representations (see Section 3.5 on page 55). The degree of expressive overlap between two representations determines the amount of information that may be translated from one representation to the other, and will therefore also determine the ease of maintaining consistency between descriptions expressed using those representations. Thus, maintaining consistency among descriptions will typically be impossible if the corresponding representations have a disjoint expressive overlap. As the degree of expressive overlap increases, so too will the ease of consistency maintenance.

The expressive overlap between two representations effectively determines the ‘maximum quality’ of any translation between those representations. Translation quality is therefore an important factor in the efficacy of any consistency maintenance scheme; this is another argument in favour of the individual interfacing strategy, as this strategy can provide higher quality translations (Pascoe and Penny, 1990). It is also important to identify ways of improving translation quality, in order to produce better correspondences between representations. Methods for improving translation quality will be discussed in Chapter 4 and the efficacy of these methods will be discussed in Chapter 8.

Maintaining consistency among descriptions should be an important element of any modelling environment that supports the use of multiple representations, but

it is not part of the translation process per se. Rather the opposite situation holds, that is, the translation process forms an important part of the consistency maintenance mechanism. The major emphasis of this research is on facilitating the use of multiple representations using a translation-based approach, not maintaining consistency among descriptions, which has already been well-researched by other authors (Grundy and Hosking, 1994; Grundy et al., 1996; Grundy and Hosking, 1996; Hosking and Grundy, 1995). The prototype modelling environment developed for this research (see Chapter 6) does not implement a consistency maintenance mechanism because it was developed mainly as a tool for exploring the issues arising from the translation process, rather than for use by end-users. The only consistency maintenance in the prototype is the basic asynchronous approach implied by executing a translation. That is, when a description is translated from one representation to another, the new description will be consistent with the original.

Consistency maintenance cannot be completely disregarded, of course, and some possible solutions to the issues outlined above are suggested in Chapter 10.

### 3.7 Summary

In this chapter, the implications of using multiple representations to describe a viewpoint were discussed, in particular the impact on the viewpoint discussion from Section 2.3 and the data translation discussion from Section 2.4.

The concepts of *representation*, *technique* and *scheme* were clarified and formally defined, and the viewpoint framework introduced in Chapter 2 was extended with the concepts of *construct*, *description* and *element*. The terminology introduced in this chapter is summarised in Table 3.4, which also indicates the correspondence between the terms used in this thesis and those used by other authors. A notation for expressing these concepts was also defined, and is summarised in Table 3.5 on page 66.

An entity-relationship based method for defining the structure of representations and the properties of their constructs was proposed. The concept of the *expressive power* of a representation was introduced and discussed, and it was shown how the expressive overlap between two representations falls into one of four categories, which determines the maximum quality of translations between those representations.

**Table 3.4: Summary of representation and description terminology**

| Term used in this thesis               | Meaning   | Example                    | Corresponding term used by:     |                |                |               |                     | CDIF                             |                  |       |
|--|---|----------------------------|---------------------------------|----------------|----------------|---------------|---------------------|----------------------------------|------------------|-------|
|  |   |                            | Finkelstein                     | Easterbrook    | Darke & Shanks | Grundy et al. | Atzeni & Torlone    |                                  | Su et al.        |       |
| perspective                            | A description of a real-world phenomenon that has internal consistency and an identifiable focus. | -                          | -                               | perspective    | perspective    | -             | -                   | -                                | -                |       |
| viewpoint                              | The formatted description of a perspective.   | -                          | ViewPoint                       | viewpoint      | viewpoint      | -             | -                   | -                                | -                |       |
| technique                              | A collection of abstract constructs that form a modelling 'method'.                               | relational model           | } style                         |                | technique      | -             | } model             |                                  | } meta-model     |       |
| scheme                                 | A collection of concrete constructs that form a modelling 'notation'.                             | SQL/92                     | } style                         |                | scheme         | -             | } model             |                                  | } data model     |       |
| representation                         | The combination of a particular technique and scheme.   | relational model + SQL/92  |                                 | representation | representation |               |                     |                                  |                  |       |
| description                            | An instantiation of a representation.   | SQL/92 schema              | specification                   | description    | -              | view          | scheme              | schema                           | model            | model |
| construct                              | The basic unit of a representation.   | a relation                 | -                               | -              | -              | -             | construct           | class <sup>a</sup>               | meta-entity      |       |
| element                                | An instantiation of a construct within a particular description.                                  | Staff table                | -                               | -              | -              | component     | varies <sup>b</sup> | object                           | entity           |       |
| representation definition <sup>c</sup> | An abstract representation used to define other representations.                                  | MDM metamodel <sup>d</sup> | ViewPoint template <sup>e</sup> | -              | -              | meta-model    | metamodel           | core model <sup>e</sup>          | meta-meta-model  |       |
| -                                      | A construct of a metamodel.   | aggregation <sup>d</sup>   | -                               | -              | -              | concept       | metaconstruct       | primitive construct <sup>f</sup> | meta-meta-entity |       |

**Notes on Table 3.4:**

- <sup>a</sup> Also 'construct'.
- <sup>b</sup> Terms used include 'component', 'element' and 'concept'.
- <sup>c</sup> This is not quite the same as a metamodel, but it is similar in intent.
- <sup>d</sup> See Atzeni and Torlone (1996a).
- <sup>e</sup> Also 'intermediate model'.
- <sup>f</sup> Also includes 'micro-rule' and 'macro', albeit with a somewhat different meaning. '-' indicates that a term is not used by that author.

**Table 3.5:** Summary of representation and description notation

| Notation   | Associated term                 | Definition  |
|--|---------------------------------|---|
| $V$  | A viewpoint                     | A formatted expression of a description of a real-world phenomenon.   |
| $T$  | A technique                     | A collection of generic constructs that form a modelling ‘method’, for example, the relational model or entity-relationship approach.           |
| $S$  | A scheme                        | A collection of specialised constructs that form a modelling ‘notation’, for example, SQL/92 or Martin ERD notation.                            |
| $\mathfrak{R}(T, S)$ or $\mathfrak{R}$   | A representation                | Representation $\mathfrak{R}$ comprises constructs defined by the combination of technique $T$ and scheme $S$ .                                 |
| $D(V, T, S)$ or $D$  | A description                   | Description $D$ of viewpoint $V$ is expressed using constructs of technique $T$ and scheme $S$ .  |
| $\mathfrak{R}(T, S) [\text{CON}]$ ,<br>$\mathfrak{R} [\text{CON}]$ , or $\text{CON}$ | A construct of a representation | $\text{CON}$ specifies a construct of representation $\mathfrak{R}(T, S)$ .   |
| $D(V, T, S) [e : \text{CON}]$ ,<br>$D [e : \text{CON}]$ , or $D [e]$                 | An element of a description     | $e$ specifies an element (instantiated from construct $\text{CON}$ ) of description $D(V, T, S)$ .  |
| $\mathfrak{R}_b \preceq \mathfrak{R}_a$  | Expressive power                | The expressive power of representation $\mathfrak{R}_b$ is inclusive of the expressive power of representation $\mathfrak{R}_a$ . <sup>a</sup>  |
| $\mathfrak{R}_a \equiv \mathfrak{R}_b$   | Expressive power                | The expressive power of representation $\mathfrak{R}_b$ is equivalent to the expressive power of representation $\mathfrak{R}_a$ . <sup>a</sup> |

**Notes on Table 3.5:**<sup>a</sup> Defined in Chapter 8.

The problem of maintaining consistency between descriptions was also discussed, and it was shown that the ease of maintaining consistency across descriptions is determined by the expressive overlap of the corresponding representations. The correspondences between constructs of two different representations effectively define a translation between those representations, so translation quality is an important factor affecting the ease of consistency maintenance.

In this chapter has been discussed the impact of the use of multiple representations on the translation process, but the translation process itself has not been explored. This will be done in the next chapter.

# Chapter 4

## Translating descriptions within a viewpoint

### 4.1 Introduction

An important goal of this research is to facilitate the use of multiple representations for modelling a single viewpoint, and the approach taken here is to perform translations between descriptions within a single viewpoint. This approach is similar to that proposed by Atzeni and Torlone (1995), in that a translation comprises a collection of *rules* that specify how the constructs of one representation map onto the constructs of another. The concept of a rule is discussed in Section 4.2, and the concept of a *heuristic rule* is proposed as a means of translating additional information that is not normally translated. Translations also have several properties, which are discussed in Section 4.3.

Translations may be specified in many ways, including natural language, abstract notations such as predicate logic and specification languages such as Amor's (1997) View Mapping Language (VML). An abstract notation for specifying translations of descriptions and elements is defined in Section 4.4. This notation, while useful for the high-level specification of translations, does not however allow the detailed specification of rules. The issue of how to specify the details of rules will be discussed further in Chapters 6 and 7.

Another goal of this thesis is to show that the translation of descriptions between representations can be used as a means of highlighting potential inconsistencies within a viewpoint. Descriptions expressed using different representations may be translated into the same representation, then compared for obvious discrepancies. This process and its implications are discussed in Section 4.5.

Identifying ways of improving translation quality is also a goal of the thesis. Heuristics are one method for improving translation quality; another is *enrichment*, which is discussed in Section 4.6. (A method for comparing the relative quality of translations will be defined in Chapter 8).

## 4.2 Rules and heuristics

A representation comprises a collection of constructs that are instantiated to form the elements of descriptions. The translation of a description from a source representation  $\mathfrak{R}_s$  to a target representation  $\mathfrak{R}_t$  can therefore be decomposed into a collection of translations of the elements that make up the description. The translation of an element (or group of elements) of one description to an element (or group of elements) of a second description is defined by a *rule* that specifies a mapping from a collection of constructs of  $\mathfrak{R}_s$  to some collection of constructs of  $\mathfrak{R}_t$  (Atzeni and Torlone, 1995; Atzeni and Torlone, 1996a). This mapping may have constraints attached to it specifying pre- and post-conditions that elements translated by the rule must meet. Pre-conditions are used to ensure that a rule is only applied to appropriate collections of source elements. Thus, if a collection of elements does not meet the pre-conditions of a rule, the rule cannot be applied to those elements. Post-conditions are used to enforce the semantics of the target representation. That is, they are used to ensure that any groups of elements generated by a rule make sense in the context of the target representation. Amor (1997) refers to both pre- and post-conditions as *invariants*, as a pre-condition when a rule is applied in one direction ('left-to-right') can become a post-condition when the rule is applied in the opposite direction ('right-to-left'). Amor's term will be adopted here to avoid confusion.

The application of a rule will always result in a target structure that is semantically consistent with the source description, but the differences in expressive overlap noted in the previous chapter place limits on the efficacy of rules, as the degree of expressive overlap determines the how well constructs may be mapped from one representation to another, and hence the extent of the set of rules. If  $\mathfrak{R}_s$  and  $\mathfrak{R}_t$  do not have equivalent or inclusive expressive powers, it is unlikely that the set of rules will define a complete translation from the source to the target. This is because some constructs of



$\mathfrak{R}_s$  may not be directly mappable onto constructs of  $\mathfrak{R}_t$ , or vice-versa. This mismatch can sometimes be ameliorated by the use of *heuristic rules*, or simply *heuristics*, which are used in much the same way as rules, except that the application of a heuristic will *generally* result in a target structure that is semantically consistent with the source, but not always. Heuristics can be thought of as ‘rules of thumb’ that usually produce the correct result, but not always.

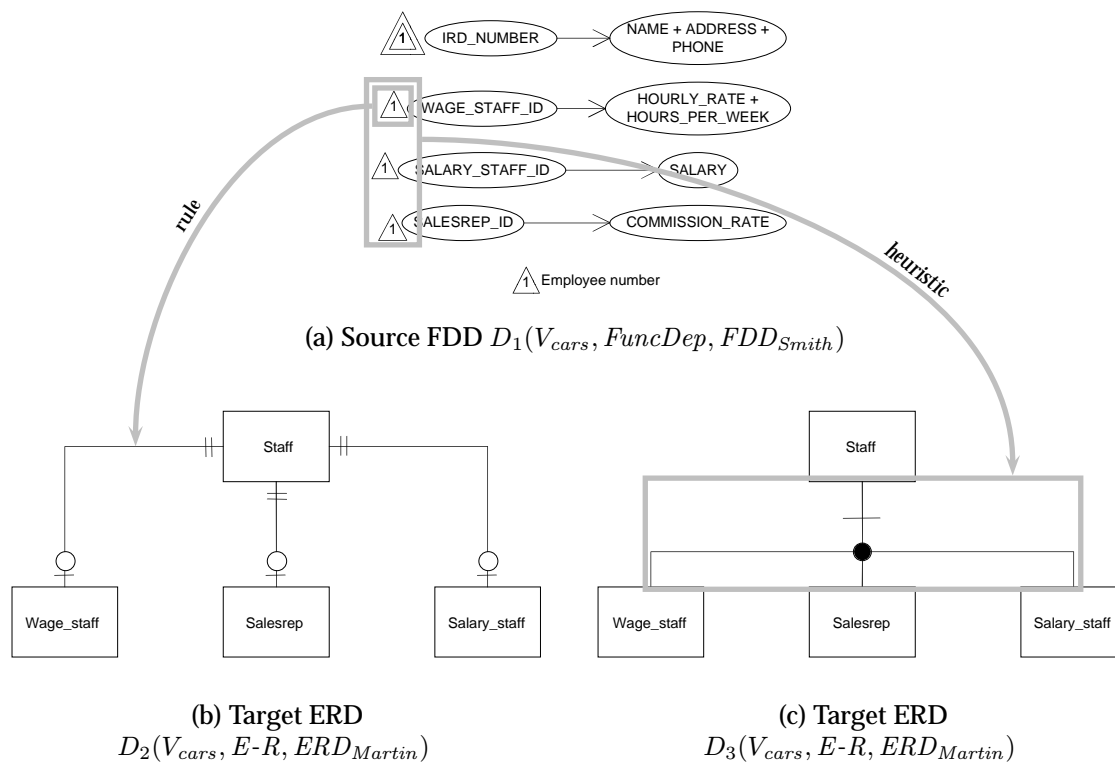
Rules generally only translate the structure or syntax of a description. Heuristics also translate structure, but can also affect the semantics of the encompassing viewpoint, as they can make explicit semantics that may be implicit in the original description. Consider the functional dependency description  $D_1(V_{cars}, FuncDep, FDD_{Smith})$ <sup>1</sup> shown in Figure 4.1(a), which has a collection of domain flag elements that all reference the same attribute element (IRD\_NUMBER). Now suppose there is a rule specifying that a domain flag maps to a relationship. The description  $D_2(V_{cars}, E-R, ERD_{Martin})$  that results from applying this rule is shown in Figure 4.1(b). Examining  $D_2$ , it becomes apparent that the Salary\_staff, Wage\_staff and Salesrep entities are in fact subtypes of the Staff entity. A rule cannot be defined to this effect, however, because it may not always be correct.

Instead, a heuristic could be defined that specifies a mapping from a collection of domain flags to a type hierarchy. The likelihood of this being the correct interpretation increases as the number of domain flags referencing the attribute increases. A reasonable assumption is that if three or more domain flags reference the same attribute, they should be mapped to a type hierarchy. The result of applying this heuristic is shown in Figure 4.1(c). Compare  $D_2$  and  $D_3$  — without the use of a heuristic, the type hierarchy implicit in  $D_1$  and  $D_2$  would not be drawn out in  $D_3$ . This heuristic will, of course, not always apply, but it will usually produce the correct result.

In effect, the application of heuristics can cause an increase in the explicit semantic content of a viewpoint, by drawing out semantics that are implicit in the source description. Thus, in the example shown in Figure 4.1 on the next page, the explicit semantic content of the viewpoint is increased by making explicit the type hierarchy implicit in the source FDD. Taken in isolation, a translation may result in a target description with less semantic content than the source, but the semantic content of the

---

<sup>1</sup>Extracted from the used cars viewpoint defined in Appendix C.



**Figure 4.1:** Example of applying a heuristic

encompassing viewpoint will never decrease, as the ‘lost’ content is still present in the source description.

It could be argued that altering the viewpoint’s semantics in this manner effectively creates a new viewpoint, thus contradicting the focus of this thesis on translations within a single viewpoint. It should be noted, however, that a viewpoint is internally consistent (Easterbrook, 1991a, p. 54), so a new viewpoint is only created if a translation generates a description that is in some way inconsistent with the rest of the viewpoint (this will be discussed further in Section 4.5).

In summary, the translation of a description from one representation to another can be decomposed into a collection of translations of elements. These element translations are defined by a collection of rules and heuristics, which specify mappings between collections of constructs in the source and target representations. Rules always produce semantically consistent results, whereas heuristics can sometimes produce semantically inconsistent results.

In order to fully specify a translation, some form of notation is required. Before this

can be defined, however, the properties of translations must be examined in order to determine the types of operators required (see Section 4.3).

### 4.2.1 Specialisation of rules and heuristics

The rules and heuristics of a translation may be divided into two categories in the same way as the constructs of a representation: those that define a mapping between technique-level constructs only (*technique-level* rules/heuristics), and those that define a mapping between scheme-level constructs only (*scheme-level* rules/heuristics). This reinforces the distinction between the technique and scheme components of representations, remembering that it is the scheme of a representation that distinguishes it from other representations that share the same technique. Scheme-level rules and heuristics are therefore unique to a particular translation, whereas technique-level rules may be shared across translations in the same way that a technique may be shared across representations.

Technique-level rules thus specify the ‘generic’ translation between two representations at the technique level, which can then be specialised to specify a translation at the scheme level. Technique-level rules that will be used in a specific translation must be specialised for that translation in the same way that technique-level constructs are specialised for a specific representation. This allows a scheme-level rule to introduce additional mappings that do not exist in the corresponding technique-level rule.

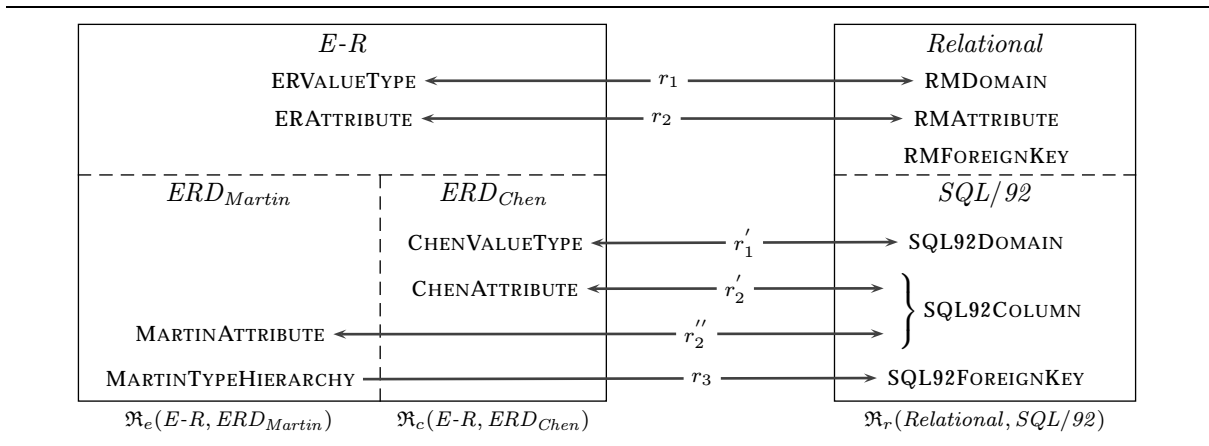
Note that the existence of a technique-level rule does not imply that it will be specialised in all translations involving that technique, in the same way that a representation may not specialise all of its technique-level constructs. For example, suppose the E-R technique was extended with an ERCANDIDATEKEY construct. Any description translation from a relational representation to an E-R representation could then include a technique-level rule that maps RMALTERNATEKEY constructs to ERCANDIDATEKEY constructs<sup>2</sup>. The translation from  $\mathfrak{R}_r(\textit{Relational}, \textit{SQL}/92)$  to  $\mathfrak{R}_e(\textit{E-R}, \textit{ERD}_{\textit{Martin}})$  would not, however, specialise this rule because  $\mathfrak{R}_r$  does not specialise the RMALTERNATEKEY construct. In addition, not all scheme-level rules are specialisations of technique-level rules, as schemes can introduce constructs into a representation that do not exist

---

<sup>2</sup>These constructs are defined in Appendix D.

in the technique. For example, the scheme of representation  $\mathfrak{R}_f(\text{FuncDep}, \text{FDD}_{\text{Smith}})$  introduces the `SSDOMAINFLAG` construct, which has no analogue in the functional dependency technique (see Figures D.3 and D.4 in Appendix D). Any translation specification will thus comprise a collection of generic technique-level rules, specialised scheme-level rules and non-specialised scheme-level rules.

Fragments of the three representation  $\mathfrak{R}_e(E-R, \text{ERD}_{\text{Martin}})$ ,  $\mathfrak{R}_c(E-R, \text{ERD}_{\text{Chen}})$  and  $\mathfrak{R}_r(\text{Relational}, \text{SQL}/92)$  are shown in Figure 4.2. There are two possible technique translations between these representations, and the technique-level rules  $r_1$  and  $r_2$  are shared across both translations. The translation between  $\mathfrak{R}_c$  and  $\mathfrak{R}_r$  specialises both  $r_1$  and  $r_2$  (to scheme-level rules  $r'_1$  and  $r'_2$  respectively). The translation between  $\mathfrak{R}_e$  and  $\mathfrak{R}_r$  does not use  $r_1$  at all, because  $\mathfrak{R}_e$  does not specialise the `ERVALUE TYPE` construct. This translation does, however, specialise  $r_2$  (to scheme-level rule  $r''_2$ ) and also introduces a new, non-specialised scheme-level rule  $r_3$ .



**Figure 4.2:** Specialisation of technique-level rules

At the start of this section, it was stated that scheme-level rules define mappings between scheme-level constructs only. In practice, this can sometimes lead to an explosion in the number of rules required. Consider a set of rules for translating `MARTIN-RELATIONSHIP` constructs into `SQL92FOREIGNKEY` constructs. Taking into account the cardinality and optionality of the relationship, there are six cases to be accounted for: optional one to optional one; optional one to mandatory one; mandatory one to mandatory one; optional one to many (optionality makes no difference for the ‘many’ side of a relationship — see Chapter 5); mandatory one to many; and many to many.

If rules for these situations were specified in the obvious manner, there would need

to be at least thirty-six of them (six basic situations, two entities per situation and three types of entity: weak, regular and associative), many of which would be almost identical. The six basic situations could be defined as technique-level rules, but they would still need to be specialised for the specific cases, so this would not solve the problem. Instead, an object-oriented programming technique can be borrowed to reduce the number of rules required. A programmer working with an object-oriented language such as C++ might define the object class `Staff`, which has the subclasses `Waged`, `Salesrep` and `Salaried`. While the instances of each subclass are of distinct types, they may also be treated as instances of the `Staff` class.

This approach also applies to the scheme-level constructs of a representation, as they are defined as subclasses of the technique-level constructs. Thus, in the example above, the number of rules could be reduced dramatically by using technique-level constructs at appropriate points in the rule definitions. For example, instead of defining three otherwise identical rules for `MARTINREGULAREntity`, `MARTINASSOCIATIVEEntity` and `MARTINWEAKEntity` constructs, a single rule could be defined that refers to the `ERTYPEITEM` construct, which is the common superclass of all three. This is interpreted as meaning any subclass of the `ERTYPEITEM` construct may be used in its place (limited, of course, to those within the representation in question). That is, a `MARTINREGULAREntity`, `MARTINWEAKEntity` or `MARTINASSOCIATIVEEntity` may be used wherever `ERTYPEITEM` appears in a rule. Applying this approach to the example above reduces the number of rules required to just six, which is the minimum possible for this example.

### 4.3 Properties of translations

In the context of this research, the goal of a translation is to translate a description or element(s) from a source representation  $\mathfrak{R}_s$  to a target representation  $\mathfrak{R}_t$ . For convenience, the translation of a description from one representation to another is referred to here as a *description translation*, and the translation of an element or group of elements from one representation to another is referred to as an *element translation* (this can be thought of as the ‘instantiation’ of a rule or heuristic). Any description translation will comprise a collection of element translations.

The author has identified four major properties that are shared by description and element translations:

1. the *type* of the translation (Section 4.3.1);
2. the *completeness* of the translation (Section 4.3.2);
3. the *composition* of the translation (Section 4.3.3); and
4. the *direction* of the translation (Section 4.3.4).

These properties are not mutually exclusive, and it is expected that translations will have several, if not all, of these properties.

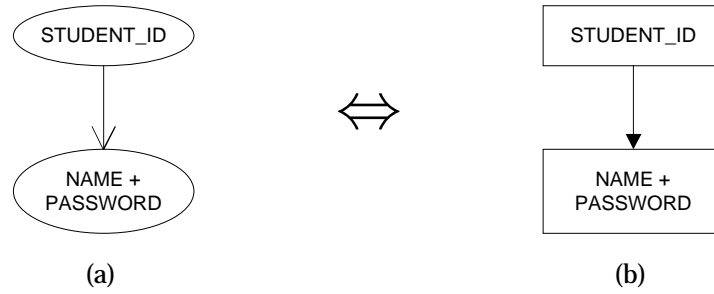
### 4.3.1 Type

How a translation is applied depends on whether it is a description translation or an element translation. Element translations fall into two types: those specified by rules and those specified by heuristics. Rules and heuristics have already been discussed in Section 4.2, so no further discussion is required here.

Description translations also fall into two types: those that change both the technique and the scheme, and those that change just the scheme. These two types are termed *technique translations* and *scheme translations* respectively. Thus, a translation from  $\mathfrak{R}_c(E-R, ERD_{Chen})$  to  $\mathfrak{R}_r(Relational, SQL/92)$  is a technique translation, whereas a translation from  $\mathfrak{R}_c(E-R, ERD_{Chen})$  to  $\mathfrak{R}_e(E-R, ERD_{Martin})$  is a scheme translation.

A scheme translation is *trivial* if the source and target representations have identical expressive powers and identical constructs, but the constructs of one representation are presented differently from those of the other representation (that is, only the notations differ). An example of a trivial scheme translation is shown in Figure 4.3, which merely changes the appearance of attribute collections in a functional dependency diagram. Technique translations are never trivial; if two techniques have identical expressive powers and identical constructs, then they are the same technique.

On first inspection, it might appear that all scheme translations are trivial. This is not always true, however, as some schemes do not fully adhere to their associated technique. For instance, SQL does not fully adhere to the definition of the relational model



**Figure 4.3:** A trivial scheme translation between: (a) a Smith-style FDD using bubbles ( $\mathfrak{R}_f(\text{FuncDep}, \text{FDD}_{\text{Smith}})$ ); (b) a Date-style FDD using boxes ( $\mathfrak{R}_g(\text{FuncDep}, \text{FDD}_{\text{Date}})$ )

(Codd, 1988a; Codd, 1988b; Date, 1990b; Date, 1990a; Date and Darwen, 1993). This can cause the representation definition to have different associations between constructs than the technique definition. In the relational technique, for example, all attributes *must* be drawn from a domain, whereas in the SQL/92 scheme, attribute domains are optional (see Figure D.6 on page 388).

In addition, some schemes add new constructs that are not present in the technique, and may therefore be able to express more information than other schemes for the same technique. For example, the Gane & Sarson data flow diagram representation  $\mathfrak{R}_d(\text{DataFlow}, \text{DFD}_{\text{G\&S}})$  can express resource flows (that is, flows of physical resources instead of data), whereas the Yourdon DFD representation  $\mathfrak{R}_y(\text{DataFlow}, \text{DFD}_{\text{Yourdon}})$  cannot. Although scheme translations cannot be disregarded, this research will focus on technique translations, as they contribute more to the goal of improving the depth and detail of a viewpoint.

### 4.3.2 Completeness

The completeness of a description translation is determined by the expressive overlap of the source and target representations (see Section 3.5). A translation is *complete* if it is possible to map the entire expressive power of the source representation into the target representation. Conversely, a translation is *partial* if only part of the source representation’s expressive power can be mapped. Thus, as noted in Section 3.5 on page 55, translations between representations of equivalent expressive power will always be complete, whereas translations between representations with intersecting ex-

pressive powers will always be partial. If the expressive power of one representation is inclusive of the other, then translations from the ‘containing’ representation to the ‘contained’ will be partial, and translations from the ‘contained’ to the ‘containing’ representation will be complete.

Element translations may also be categorised as complete or partial, depending on what proportion of the properties of an element are translated. An informal way of determining the completeness of an element translation is to translate a set of elements from the source representation ( $\mathfrak{R}_s$ ) to the target representation ( $\mathfrak{R}_t$ ) and back again (that is,  $e_s \Rightarrow e_t$  followed by  $e_t \Rightarrow e'_s$ ), and then check whether the new set of elements ( $e'_s$ ) is identical to, or a superset of, the original set ( $e_s$ ). If so, then the rule is complete when translating from  $\mathfrak{R}_s$  to  $\mathfrak{R}_t$ , as enough information was translated to reconstruct the original set of elements.

### 4.3.3 Composition

Consider a description translation that translates from the representation  $\mathfrak{R}_d(\text{DataFlow}, \text{DFD}_{G\&S})$  to the representation  $\mathfrak{R}_r(\text{Relational}, \text{SQL}/92)$  by first translating descriptions to  $\mathfrak{R}_e(\text{E-R}, \text{ERD}_{\text{Martin}})$  and then translating from  $\mathfrak{R}_e$  to  $\mathfrak{R}_r$ . This description translation can be decomposed into two smaller description translations, the first from  $\mathfrak{R}_d$  to  $\mathfrak{R}_e$  and the second from  $\mathfrak{R}_e$  to  $\mathfrak{R}_r$ .

This property is not unique to description translations. Consider the translation of a regular entity element (representation  $\mathfrak{R}_e(\text{E-R}, \text{ERD}_{\text{Martin}})$ ) and its associated attributes and identifiers into appropriate functional dependency elements (representation  $\mathfrak{R}_f(\text{FuncDep}, \text{FDD}_{\text{Smith}})$ ). This element translation can be decomposed into three smaller element translations, one that translates the entity itself, one that translates individual attributes and one that translates individual identifiers.

A translation that can be decomposed in this manner is referred to as a *composite* translation. An *atomic* translation is one that cannot be decomposed in this manner. Specifically, a description translation is composite if it may be decomposed into a collection of description translations, and an element translation is composite if it may be decomposed into a collection of element translations. Note that individual representations and constructs in a translation are considered indivisible in this definition. For



example, a `MARTINIDENTIFIER` construct comprises a collection of `MARTINATTRIBUTE` constructs, but this does not mean that an element translation involving a `MARTINIDENTIFIER` can be decomposed into one or more rules involving `MARTINATTRIBUTE` constructs.

The examples given above will be revisited in Section 4.4.3 on page 84, following the definition of the translation notation.

### 4.3.4 Direction

Consider a translation that may only be applied in a single direction, either from ‘left to right’ or ‘right to left’. Such a translation is termed a *unidirectional* translation. Conversely, a translation that may be applied in either direction is termed *bidirectional*. When a rule is applied in a particular direction, the constructs on the side of the rule that corresponds to the source representation are referred to as the *source constructs* of the rule, and the constructs on the other side are referred to as the *target constructs*. If the rule is applied in the opposite direction, the source and target constructs will thus be exchanged.

Any translation may be either unidirectional or bidirectional, with the exception of heuristics, which are always unidirectional. This is because the result of applying a heuristic is not always guaranteed to be semantically consistent, so allowing them to be applied in the opposite direction would not be appropriate. Any ‘new’ information that might be generated by a heuristic will presumably be translated by a rule (if it is possible to translate it at all) when translating in the opposite direction.

Unidirectional description translations will typically be uncommon in practice — if it is possible to map constructs in one direction, then it is reasonable to assume that it will be possible to map the same constructs in the opposite direction, but this may vary depending on the approach taken. For example, all translations in Atzeni and Torlone’s (1996b, slides 1-29–1-30) MDM environment are unidirectional because every representation is a subset of the ‘supermodel’ (that is, the expressive power of the supermodel is inclusive of the expressive powers of all the other representations). Translations from each representation to the supermodel are not required, as every description is by definition an instance of both its ‘native’ representation and the supermodel.

Unidirectional element translations will often arise when translating between intersecting or inclusive representations. This is because there may be information which is possible to derive in one direction, but not in the other. This is particularly relevant to inclusive representations when translating from the ‘containing’ representation to the ‘contained’ representation, as there is more scope for information ‘loss’. Conflicts between rules may also result in unidirectional rules, as will be discussed in Section 4.7.1.

## 4.4 Notations for specifying translations

Representations and their constructs are defined in this thesis using an E-R based model, as described in Section 3.4 on page 49. This means that a representation definition is effectively a schema, so translation specification languages designed for specifying mappings between schemas may also be used to specify mappings between representations. The translation specification language VML was developed in response to perceived deficiencies in existing translation specification languages (see Section 2.4.4 on page 32) and provides many features that are useful for specifying translations between representations, such as declarative specification and bidirectionality. The use of VML for specifying the details of translations will be discussed further in Chapters 6 and 7.

VML specifications are quite detailed, and it can sometimes be difficult to obtain a high-level understanding of what a particular mapping actually does. The original VML specification defines a graphical notation (VML-G) for displaying VML specifications at a higher level (Amor, 1997), but for complicated mappings even this can become difficult to read due to the large numbers of connections among the various parts of the mapping (see Figure 4.4). In addition, VML-G cannot specify any of the translation properties identified in the previous section.

It was therefore decided to develop an abstract notation for expressing translations. The translation properties identified in Section 4.3 can be used as a basis for determining the required translation operators, which are discussed in Section 4.4.1. The abstract notation for expressing translations is developed around these requirements and described in Section 4.4.2.

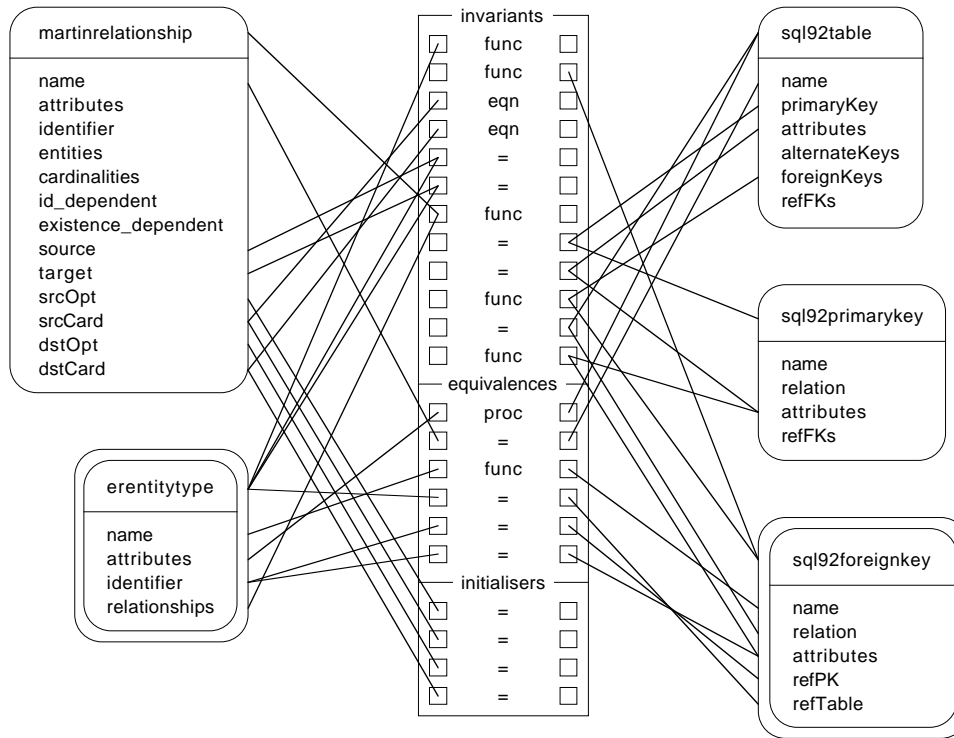


Figure 4.4: A complex VML-G specification

#### 4.4.1 Requirements for translation operators

The type of a translation effectively specifies how the translation is carried out, that is, scheme versus technique translations for descriptions, and rules versus heuristics for elements. The type of description translations can be determined simply by noting whether the technique and/or scheme are different in the source and target descriptions. Consider the three representations  $\mathfrak{R}_e(E-R, ERD_{Martin})$ ,  $\mathfrak{R}_d(DataFlow, DFD_{G\&S})$  and  $\mathfrak{R}_c(E-R, ERD_{Chen})$ . Any description translation from  $\mathfrak{R}_e$  to  $\mathfrak{R}_d$  is a technique translation, as both the technique and scheme change. Conversely, any description translation from  $\mathfrak{R}_e$  to  $\mathfrak{R}_c$  is a scheme translation, as only the scheme changes. Since this is obvious merely by inspecting the descriptions or representations, there is no need to introduce a special operator to distinguish between technique and scheme translations.

There is, however, no equivalent way of determining the type of element translations; there is no consistent change in notation that may be examined, as there is with description translations. For example, translating an E-R entity into a data store in a data flow diagram is achieved by use of a rule, while translating a collection of domain flags in a Smith FDD to a type hierarchy in a Martin ERD is achieved by use of

a heuristic. There is no way to tell which is which merely by examining the constructs involved. Since heuristics may sometimes produce semantically inconsistent results, it is important to be able to identify them, so a notation will need to provide some means of distinguishing rules and heuristics.

The expressive overlap between two representations is a major determinant of the quality of any translations between those representations, so it would be useful to be able to distinguish between complete and partial translations.

Although the distinction between composite and atomic translations is not explored in depth in this thesis, it may be useful to know which translations can be decomposed further (an example of decomposition may be found in Section 4.4.3).

As noted above, many representations are intersecting or inclusive, so both unidirectional and bidirectional element translations will be reasonably common. Some form of notational support is thus required.

#### **4.4.2 A high-level notation for expressing translations**

Early in this research, before the author encountered languages such as VML, an attempt was made to define a declarative notation for fully expressing both description and element translations. This original notation is described in its entirety in Appendix A and briefly summarised here. The notation had a rewrite rule-like syntax and allowed the specification of both description and element translations. Invariants could be attached to element translations to specify constraints that elements had to satisfy. Although this notation initially appeared to work well, further investigation revealed several problems:

- The original notation was *too* abstract for the intended purpose — while useful for expressing description translations, the full expression of element translations was problematic. Many constraints were difficult to express, and there was no way to express procedural translations or user-defined functions.
- The original notation was concise for simple translations, but rapidly became unwieldy and difficult to understand when specifying complex rules. For example, the translation of a `MARTINIDENTIFIER` element to an `SQL92UNIQUE` element was denoted by:

$$\mathfrak{S} \left[ \begin{array}{l} C_k : \text{MARTINIDENTIFIER} \\ \left( \begin{array}{l} \exists \mathfrak{S} [E_k : \text{MARTINENTITY}], \\ \mathfrak{S} \{a_k, \dots, a_l\} : \text{MARTINATTRIBUTE} \end{array} \right) \left( \begin{array}{l} \mathfrak{S} [C_k] \in \mathfrak{S} [E_k], \\ \{ \mathfrak{S} [a_k], \dots, \mathfrak{S} [a_l] \} \in \mathfrak{S} [C_k] \end{array} \right) \end{array} \right] \rightarrow \mathfrak{T} \left[ \begin{array}{l} A_k : \text{SQL92UNIQUE} \\ \left( \begin{array}{l} \exists \mathfrak{T} [T_k : \text{SQL92TABLE}], \\ \mathfrak{T} \{c_k, \dots, c_l\} : \text{SQL92CCOLUMN} \end{array} \right) \left( \begin{array}{l} \mathfrak{T} [A_k] \in \mathfrak{T} [T_k], \\ \{ \mathfrak{T} [c_k], \dots, \mathfrak{T} [c_l] \} \in \mathfrak{T} [A_k], \\ \forall \mathfrak{T} \{c_m, \dots, c_n\} : \text{SQL92COLUMN}, \bigcup_{i=k}^l c_i \neq \bigcup_{j=m}^n c_j \end{array} \right) \end{array} \right]$$

- There were insufficient differences between the proposed notation and existing approaches such as VML to warrant further development.

Despite these problems, the original notation did provide a convenient shorthand for expressing both description and element translations in an abstract way. It was therefore decided to remove those elements of the notation that dealt with the detailed specification of element translations, such as invariants. The construct/element notation was simplified (see Section 3.3 on page 47) and the translation operators were retained; these operators will now be defined.

### Translation expressions

The generic translation operator is denoted by the symbol  $\rightarrow$  and can be used to denote both description and element translations. The translation of a specific source description  $D_s(V, T_i, S_j)$  into a specific target description  $D_t(V, T_k, S_l)$  is denoted by:

$$D_s(V, T_i, S_j) \rightarrow D_t(V, T_k, S_l).$$

For example, the translation of the SQL/92 description  $D_1$  into a QUEL description  $D_2$  is denoted by:

$$D_1(V, \text{Relational}, \text{SQL}/92) \rightarrow D_2(V, \text{Relational}, \text{QUEL}),$$

The translation of a set of elements  $\{e_m, \dots, e_n\}$  of  $D_s$  into a corresponding set of elements  $\{e_p, \dots, e_q\}$  of  $D_t$  is denoted by:

$$D_s(V, T_i, S_j) [e_m, \dots, e_n : \text{CON}_s] \rightarrow D_t(V, T_k, S_l) [e_p, \dots, e_q : \text{CON}_t]$$

where  $\text{CON}_s$  and  $\text{CON}_t$  are constructs of  $\mathfrak{R}_s(T_i, S_j)$  and  $\mathfrak{R}_t(T_k, S_l)$  respectively. Note that the element sets may include elements drawn from more than one construct —

this was omitted from the above expression for clarity. Thus, the translation of the MARTINREGULAREntity element *student* of Martin ERD  $D_3$  into a corresponding set of elements of Smith FDD  $D_4$  is denoted by:

$$D_3(V, E-R, ERD_{Martin}) [student : MARTINREGULAREntity] \rightarrow \\ D_4(V, FuncDep, FDD_{Smith}) [s\_src : SSSINGLEKEYBUBBLE, \\ s\_dst : SSTARGETBUBBLE, \\ s\_dep : SSSINGLEVALUED].$$

Such expressions refer only to specific descriptions and elements, however. Rules and heuristics should not be concerned with specific descriptions or elements; rather they should be defined in terms of the representations and constructs involved. To this end, a translation of an unspecified description from a source representation  $\mathfrak{R}_s(T_i, S_j)$  to a target representation  $\mathfrak{R}_t(T_k, S_l)$  is denoted by:

$$\mathfrak{R}_s(T_i, S_j) \rightarrow \mathfrak{R}_t(T_k, S_l).$$

Similarly, the translation of an unspecified element from  $\mathfrak{R}_s$  to  $\mathfrak{R}_t$  is denoted by:

$$\mathfrak{R}_s(T_i, S_j) [CON_s] \rightarrow \mathfrak{R}_t(T_k, S_l) [CON_t]$$

where  $CON_s$  and  $CON_t$  are constructs of  $\mathfrak{R}_s$  and  $\mathfrak{R}_t$  respectively (once again, multiple constructs of different types are allowed).

Thus, the translation of any SQL/92 description into a QUEL description is denoted by:

$$\mathfrak{R}_r(Relational, SQL/92) \rightarrow \mathfrak{R}_q(Relational, QUEL).$$

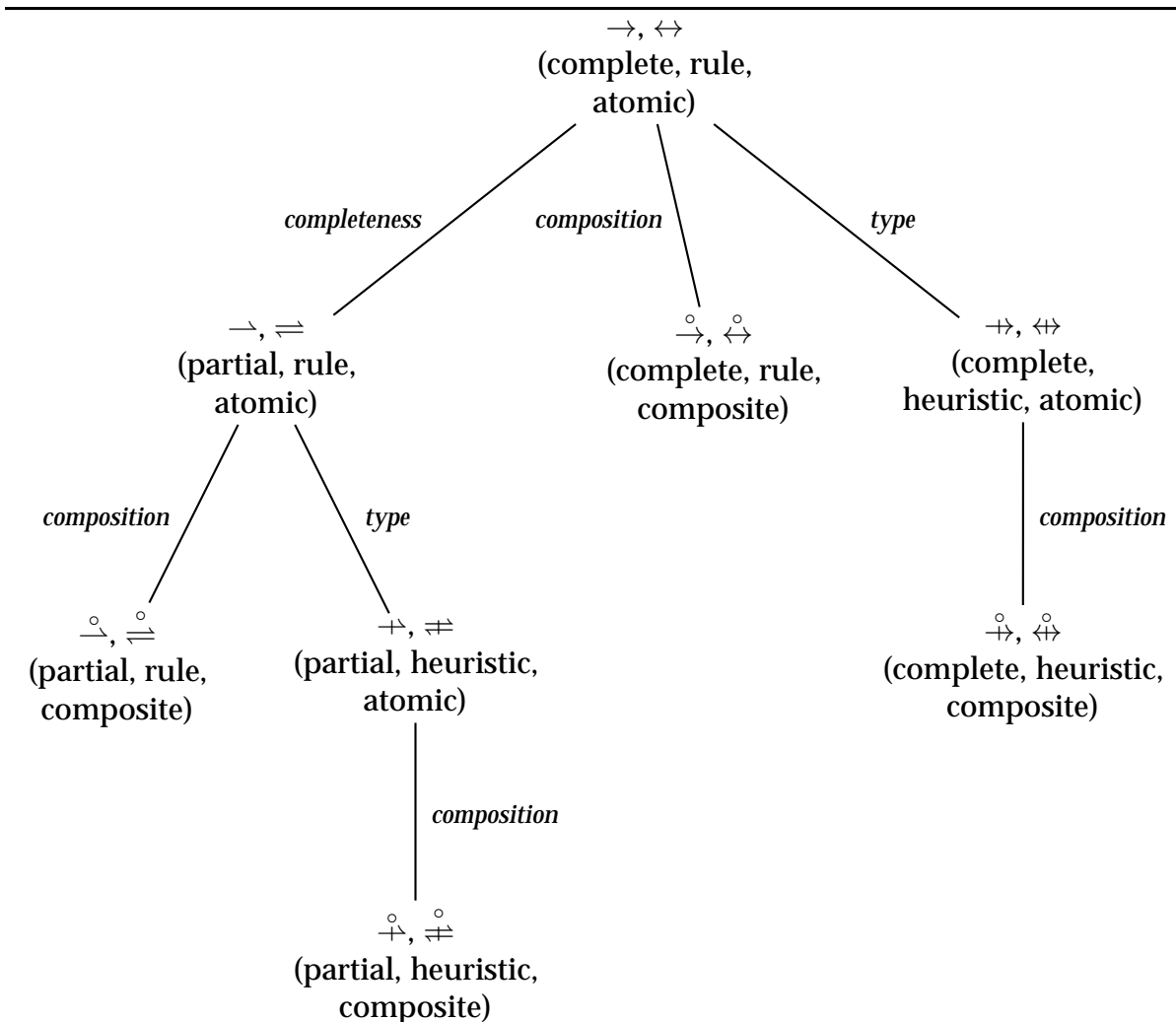
and the translation of any MARTINREGULAREntity element of a Martin ERD into corresponding elements of a Smith FDD is denoted by:

$$\mathfrak{R}_e(E-R, ERD_{Martin}) [MARTINREGULAREntity] \rightarrow \\ \mathfrak{R}_f(FuncDep, FDD_{Smith}) [SSSINGLEKEYBUBBLE, SSTARGETBUBBLE, \\ SSSINGLEVALUED].$$

Multiple occurrences of the same construct are distinguished by subscripts, for example,  $FDAATTRIBUTE_1$ ,  $FDAATTRIBUTE_2$ ,  $FDAATTRIBUTE_3$ , and so on.

## Translation operators

The generic translation operator  $\rightarrow$  may be specialised according to the properties identified in Section 4.3 (see Figure 4.5). Unless otherwise specified, a translation is assumed to be complete, defined by a rule (for element translations) and atomic. A partial translation is denoted by the operator  $\dashrightarrow$ ; an element translation that is defined by a heuristic is denoted by  $\mapsto$  (complete) or  $\dashmapsto$  (partial); and a composite translation is denoted by  $\overset{\circ}{\rightarrow}$  (complete) or  $\overset{\circ}{\dashrightarrow}$  (partial). The direction of a translation is indicated in the obvious way, that is,  $\rightarrow$  and  $\dashrightarrow$  denote ‘forward’ unidirectional translations,  $\leftarrow$  and  $\dashleftarrow$  denote ‘reverse’ unidirectional translations, and  $\leftrightarrow$  and  $\rightleftharpoons$  denote bidirectional complete and partial translations respectively.



**Figure 4.5:** Specialisation of translation operators

The  $\rightarrow$  operator thus denotes a complete, atomic, unidirectional description translation, or a complete, atomic, rule-based, unidirectional element translation, depending on the operands. Additional variations may be achieved by combining the operators in an intuitive manner:  $\overset{\circ}{\rightarrow}$  (complete) and  $\overset{\circ}{\dashrightarrow}$  (partial) denote composite, heuristic, unidirectional translations, while  $\overset{\circ}{\leftrightarrow}$  (complete) and  $\overset{\circ}{\rightleftarrows}$  (partial) denote composite, bidirectional translations.

### 4.4.3 Examples of translation decomposition

In Section 4.3.3 on page 76 were presented two examples of composite translations. These examples will now be revisited using the translation notation defined above. The first example was a composite description translation from the representation  $\mathfrak{R}_d(\text{DataFlow}, \text{DFD}_{G\&S})$  to the representation  $\mathfrak{R}_r(\text{Relational}, \text{SQL}/92)$  via the representation  $\mathfrak{R}_e(\text{E-R}, \text{ERD}_{\text{Martin}})$ . This translation is denoted:

$$\mathfrak{R}_d(\text{FuncDep}, \text{DFD}_{G\&S}) \overset{\circ}{\rightarrow} \mathfrak{R}_r(\text{Relational}, \text{SQL}/92),$$

and can be decomposed into the following two atomic description translations:

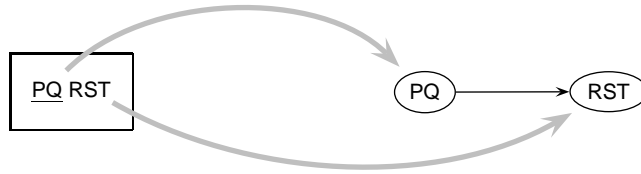
$$\begin{aligned} \mathfrak{R}_d(\text{FuncDep}, \text{DFD}_{G\&S}) &\rightarrow \mathfrak{R}_e(\text{E-R}, \text{ERD}_{\text{Martin}}), \text{ and} \\ \mathfrak{R}_e(\text{E-R}, \text{ERD}_{\text{Martin}}) &\rightarrow \mathfrak{R}_r(\text{Relational}, \text{SQL}/92). \end{aligned}$$

The decomposition of element translations can be more complex. The second example presented in Section 4.3.3 on page 76 was the translation of a regular entity element (MARTINREGULARENTITY) and its associated attributes and identifiers (MARTINATTRIBUTE and MARTINIDENTIFIER respectively) into equivalent functional dependency elements (in this case, an SSSINGLEKEYBUBBLE and an SSTARGETBUBBLE connected by an SSSINGLEVALUED, and a collection of SSATTRIBUTE elements). This is illustrated in Figure 4.6.

This translation could be denoted by:

$$\begin{aligned} \mathfrak{R}_e[\text{MARTINREGULARENTITY}, \text{MARTINIDENTIFIER}, \\ \text{MARTINATTRIBUTE}_1, \dots, \text{MARTINATTRIBUTE}_n] \overset{\circ}{\rightarrow} \\ \mathfrak{R}_f[\text{SSSINGLEKEYBUBBLE}, \text{SSTARGETBUBBLE}, \\ \text{SSATTRIBUTE}_1, \dots, \text{SSATTRIBUTE}_n, \text{SSSINGLEVALUED}] \end{aligned} \quad (\text{D1})$$





**Figure 4.6:** Translating a MARTINREGULAREntity to a functional dependency diagram

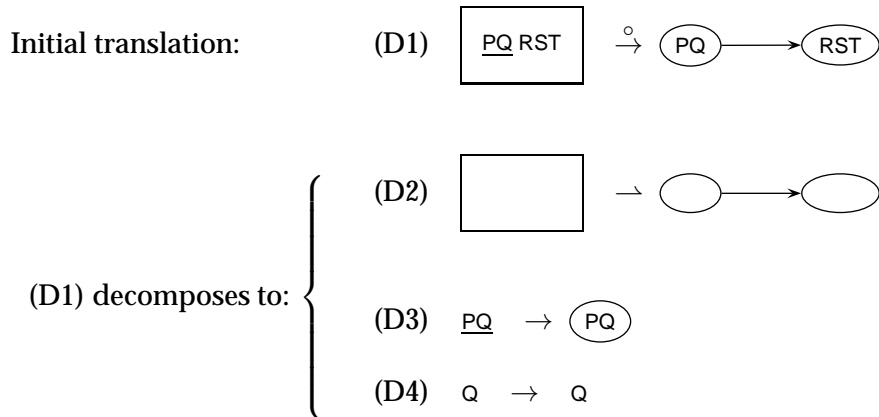
which could be decomposed into the following three atomic translations:

$$\mathfrak{R}_e [\text{MARTINREGULAREntity}] \rightarrow \mathfrak{R}_f [\text{SSSINGLEKEYBUBBLE}, \text{SSTARGETBUBBLE}, \text{SSSINGLEVALUED}], \quad (\text{D2})$$

$$\mathfrak{R}_e [\text{MARTINIDENTIFIER}] \rightarrow \mathfrak{R}_f [\text{SSSINGLEKEYBUBBLE}] \text{ and} \quad (\text{D3})$$

$$\mathfrak{R}_e [\text{MARTINATTRIBUTE}] \rightarrow \mathfrak{R}_f [\text{SSATTRIBUTE}] \quad (\text{D4})$$

This decomposition is illustrated in Figure 4.7.



**Figure 4.7:** Decomposing a translation

Translation D2 is of particular interest, as it is atomic despite having multiple constructs on the right-hand side of the translation. This is because the translation would not make sense without all three constructs on the right-hand side. That is, the three constructs on the right-hand side form an indivisible unit. Translations D3 and D4 are atomic because individual constructs are considered indivisible for the purposes of decomposition, as noted in Section 4.3.3 on page 76.

## 4.5 Highlighting potential viewpoint inconsistencies

Suppose a developer uses a single representation to build a description of a viewpoint. How can they verify the consistency of this viewpoint? In the past, a design would often be developed and implemented, and only then discovered to be inadequate (Brooks, 1975). One solution to this problem is to adopt a prototyping methodology (Sallis et al., 1995), which, although potentially time-consuming, can lessen the discontinuity between the original requirements and the final system.

If the same developer used an environment that facilitated the use of multiple representations, they could use different descriptions to help in evaluating the consistency of viewpoints. This is supported by Easterbrook (1991a, p. 56), who suggests that multiple descriptions expressed using different representations can “help reduce misunderstandings and discover conflicts.” It can be difficult, however, to compare descriptions expressed using different representations. Translations between representations can help with the process of conflict discovery by allowing descriptions to be compared in a more uniform manner. For example, consider an environment that supports the three representations  $\mathfrak{R}_e(E-R, ERD_{Martin})$ ,  $\mathfrak{R}_f(FuncDep, FDD_{Smith})$  and  $\mathfrak{R}_r(Relational, SQL/92)$ , and suppose that two descriptions  $D_1(V, E-R, ERD_{Martin})$  and  $D_2(V, FuncDep, FDD_{Smith})$  are independently created. As shown in Figure 4.8(a), the consistency of the combination could be evaluated by translating  $D_1$  and  $D_2$  into the relational descriptions  $D_3$  and  $D_4$  respectively, and comparing  $D_3$  with  $D_4$ . That is, given:

$$D_1(V, E-R, ERD_{Martin}) \rightarrow D_3(V, Relational, SQL/92) \text{ and}$$

$$D_2(V, FuncDep, FDD_{Smith}) \rightarrow D_4(V, Relational, SQL/92),$$

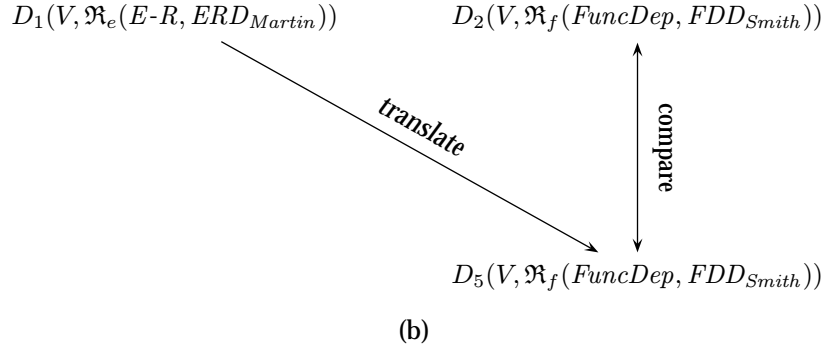
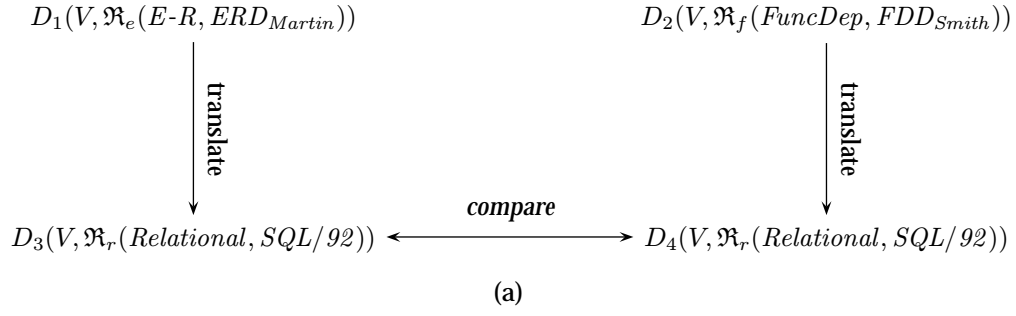
it becomes possible to meaningfully compare  $D_3$  with  $D_4$  to see if they are consistent with each other.

Alternatively, as shown in Figure 4.8(b),  $D_1$  could be translated into the functional dependency description  $D_5$ , which could then be compared with  $D_2$ . That is, given:

$$D_1(V, E-R, ERD_{Martin}) \rightarrow D_5(V, FuncDep, FDD_{Smith}) \text{ and}$$

$$D_2(V, FuncDep, FDD_{Smith}),$$

it becomes possible to meaningfully compare  $D_5$  with  $D_2$ .




---

**Figure 4.8:** Consistency checking strategies using different representations

If, in either of the examples above, the resulting descriptions are not consistent, then there is a *potential* viewpoint inconsistency between the two original descriptions, that is, either  $D_1$  or  $D_2$  may be inconsistent with the definition of the viewpoint  $V$ . If this proves to be an actual inconsistency within the viewpoint, it must be addressed by altering either  $D_1$  or  $D_2$ . A viewpoint is internally consistent (Easterbrook, 1991a), so descriptions within a viewpoint should be consistent with each other. Alternatively,  $D_1$  or  $D_2$  may actually describe a new viewpoint  $V_2$ , that is, either  $D_1(V_2, E-R, ERD_{Martin})$  or  $D_2(V_2, FuncDep, FDD_{Smith})$ . Such a situation could arise for many reasons, such as a developer wishing to explore a divergent design path, but situations involving multiple viewpoints are beyond the scope of this research and will not be discussed further.

Note that although two descriptions might be consistent, this does not necessarily imply that they are *equivalent*. There are two main causes of this lack of equivalence:

1. one of the descriptions does not describe exactly the same parts of the viewpoint as the other; and/or
2. the corresponding representations ( $\mathfrak{R}_e$  and  $\mathfrak{R}_f$  in the example above) are either

intersecting or inclusive, that is,  $\mathfrak{R}_e$  can represent some information that  $\mathfrak{R}_f$  cannot, or vice versa.

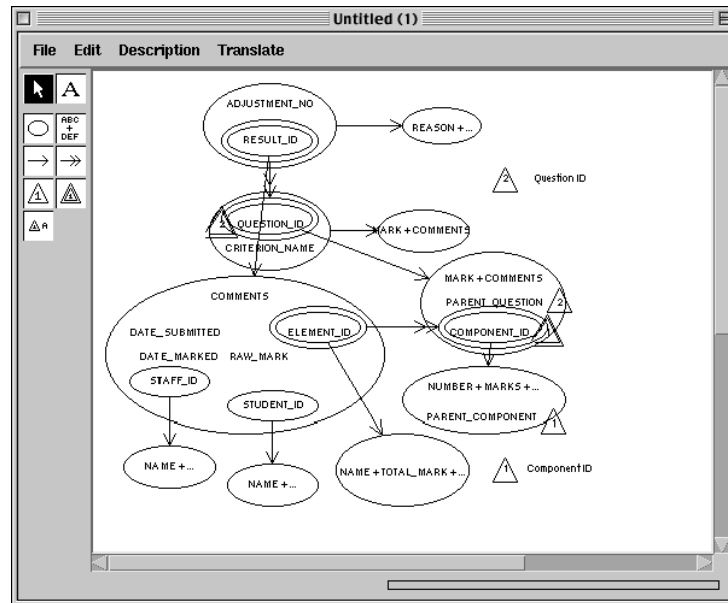
For example, suppose that two developers are building a viewpoint for an assessment marks recording system (this example is a slight variation on the assessment marks viewpoint described in Appendix C), and suppose that they independently define two descriptions of the viewpoint. A basic part of the viewpoint is the recording of students' marks for assessment elements. The first developer assumes that students' marks for an assessment element are stored separately from the data for the assessment element, whereas the second assumes that the marks are stored with the assessment element data. The first developer then builds description  $D_1(V_{marks}, FuncDep, FDD_{Smith})$ , and the second developer builds description  $D_2(V_{marks}, DataFlow, DFD_{G\&S})$ , both of which are shown in Figure 4.9.

The first developer suspects that  $D_2$  is not consistent with  $D_1$ . To test this suspicion, she decides to translate both descriptions into entity-relationship diagrams using the representation  $\mathfrak{R}_e(E-R, ERD_{Martin})$ . The resultant descriptions ( $D_3$  and  $D_4$ ) are shown in Figure 4.10 on page 90.

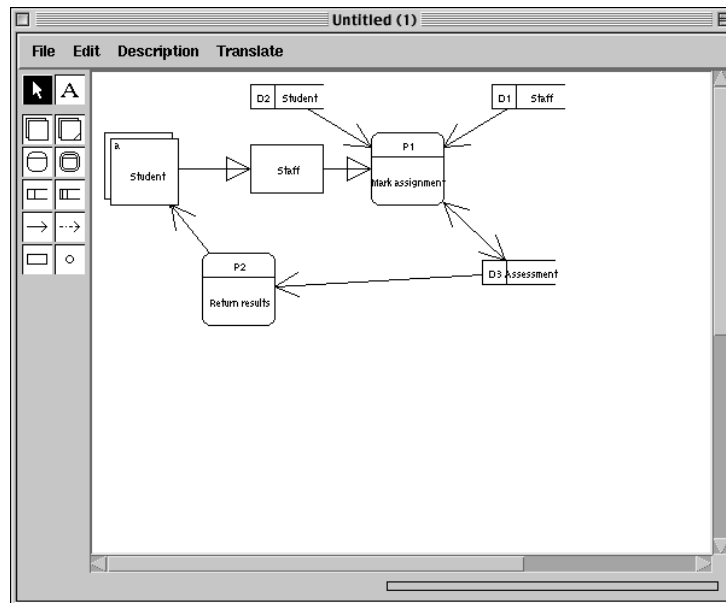
$D_2$  describes the marking process. Students submit assessment elements as assignments, which are marked by staff members. It would therefore be reasonable to expect  $D_4$  to have entities corresponding to the Staff, Student, Assessment and Assignment entities of  $D_3$ .  $D_3$  is normalised because it was derived from a collection of functional dependencies, whereas  $D_4$  may not be normalised. Despite this, there is an obvious discrepancy between the two descriptions:  $D_4$  has no entity corresponding to the Assignment entity of  $D_3$ <sup>3</sup>. This is a consequence of the second developer's assumption that results are stored with the assessment element, as illustrated by the data flows between the Mark assignment process and the Assessment data store in Figure 4.9(b). As noted above, both interpretations could be valid, which means the developers are actually dealing with two viewpoints,  $V_{marks_1}$  and  $V_{marks_2}$ . Regardless, the use of translations has provided an opportunity to highlight a potential inconsistency between the two descriptions. Another example of the use of translations to highlight potential inconsistencies will be presented in Chapter 9.

---

<sup>3</sup>This discrepancy would be more obvious if  $D_3$  were compared with the unnormalised marks ERD in Figure C.11 on page 370.

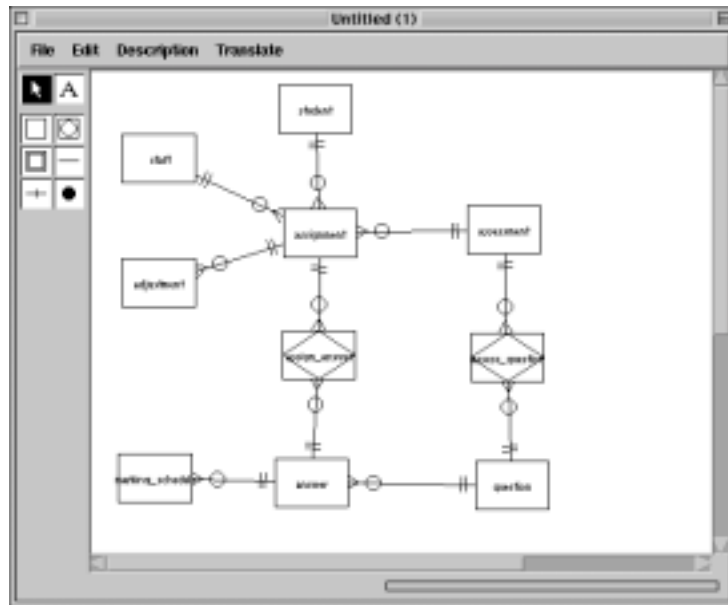


(a)  $D_1(V_{marks}, FuncDep, FDD_{Smith})$

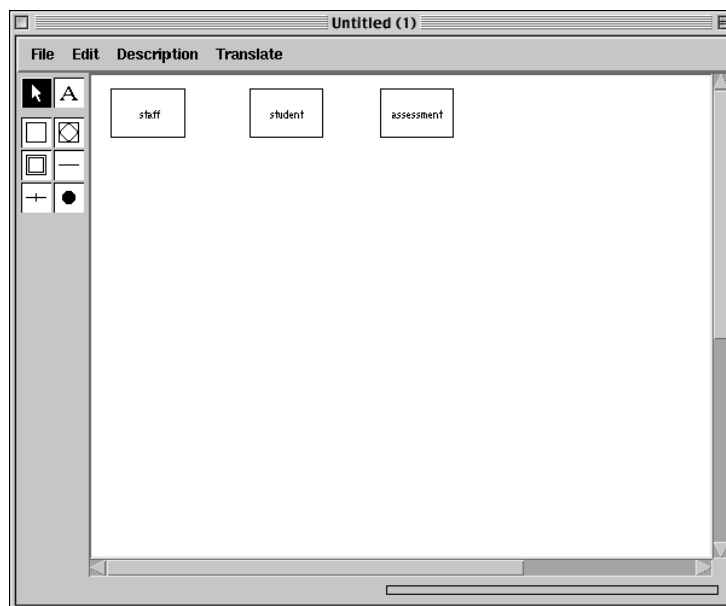


(b)  $D_2(V_{marks}, DataFlow, DFD_{G\&S})$

**Figure 4.9:** Two potentially inconsistent descriptions of a viewpoint



(a)  $D_3(V_{marks}, E-R, ERD_{Martin})$



(b)  $D_3(V_{marks}, E-R, ERD_{Martin})$

**Figure 4.10:** Two potentially inconsistent descriptions of a viewpoint after translating them into the same representation

While the use of translations can provide a means of highlighting potential inconsistencies within a viewpoint, it is difficult to determine how well this process could be automated. The problem of automatically determining the consistency of two descriptions is by no means trivial. As the above example shows, even though the two descriptions are expressed using the same representation, they may still be quite different, depending on the translations used to generate them. In the example above, one of the descriptions is normalised while the other may not be, which complicates the process of determining consistency. It seems likely that the use of translations will be limited to providing a platform for highlighting potential inconsistencies to the designer, who must then determine the actual nature of the inconsistencies from their contextual and semantic knowledge of the phenomenon being modelled.

The efficacy of any comparison also depends to a large extent on the target representation chosen for comparison. If  $D_2$  above was translated directly to an FDD description  $D_5(V_{marks}, FuncDep, FDD_{Smith})$ , it might be easier to compare  $D_1$  with  $D_5$  than to compare  $D_3$  with  $D_4$ .

In summary, translations between representations within a viewpoint can be used as a support mechanism for highlighting potential inconsistencies among different descriptions of the viewpoint, although it is unclear how easy this process would be to automate.

## 4.6 Improving translation quality

In an environment that facilitates the use of multiple representations, information loss may not be as great a concern as it is in more typical data translation problems. This is because the information that is ‘lost’ during a description translation is still held in the source description. All descriptions will presumably be held in the same repository, so it may sometimes be possible to ‘restore’ information that is ‘lost’ during a translation when the translation is applied in the reverse direction. For instance, if the data flow description  $D_1(V, DataFlow, DFD_{G\&S})$  is translated into the functional dependency description  $D_2(V, FuncDep, FDD_{Smith})$ , the names of data stores may not be translated because of differences in the expressive powers of the two representations. The data store names still exist in  $D_1$ , however, so if  $D_2$  is at some later stage translated

back to  $D_1$  (that is, used to incrementally update  $D_1$ ), or to a new DFD  $D_3$ , then it is in principle possible to extract the data store names from the original  $D_1$ . This would require some means of tracking the translation ‘history’ of descriptions.

A less obvious concern that arises when performing description translations, however, is that it may be necessary to ‘gain’ information that does not already exist when performing a translation. For example, when translating from a functional dependency description to an entity-relationship description, the names of the entities that are created must be generated or acquired in some way. This sort of situation would most commonly occur with inclusive representations when translating a description from the ‘contained’ representation to the ‘containing’ representation, but could also occur with intersecting representations.

Information ‘gain’ is a particular problem with technique translations, as the expressive powers of two representations with different techniques are more likely to be divergent than the expressive powers of two representations that share the same technique. The problem will therefore be less severe in scheme translations, as the technique does not change.

The problem is determining what information can and cannot be generated automatically during a translation, and why. Information that cannot be automatically generated and that is essential to build a syntactically correct target description (such as the names of tables in SQL) must be acquired somehow, otherwise it will not be possible to complete the translation. Although semantically important, information that is not essential to build a syntactically correct target description (such as the names of data flows in a DFD) may be ignored, although it would obviously be preferable to include it. One way to improve the quality of a translation would therefore be to acquire information from the user in some way. This process of acquiring information is termed *enrichment*, and may occur before, during and after a translation.

Enrichment performed before a translation (‘pre-enrichment’) involves pre-populating the viewpoint with information that will be useful to the target description, and is analogous to Su and Fang’s (1993) notion of modifying and extending the semantics of a schema before translating it. For example, before translating from an FDD to an ERD, entities with appropriate names but no attributes could be created; these could then be filled in by the translation. The example viewpoints used in this thesis



(see Appendix C) are all pre-enriched to some extent; for instance, all dependencies in the functional dependency descriptions have names associated with them that may be used to automatically derive entity or table names.

Enrichment performed during a translation requires the user to at least supply any information that is essential to properly build the target description, and that has not already been supplied. Using the same example as above, if the entity names were not defined prior to the translation, then the user would be requested to enter suitable names for each entity as it was generated by the translation. Non-essential information could also be included at this stage.

Enrichment after a translation ('post-enrichment') involves adding any remaining missing information that was not automatically generated during the translation or entered prior to the translation. In effect, the user is refining the resultant description.

Another way to improve the quality of a translation is the application of heuristics. Heuristics allow the *automatic* translation of more information than would normally be possible using rules, by making explicit semantics that are implicit in the source description, as noted in Section 4.2. Adding heuristics can reduce the amount of enrichment that is required. This produces an improvement in translation quality, as less information is lost or needs to be gained during the translation.

An important issue with the use of heuristics is that they may sometimes produce semantically inconsistent results, which could result in an invalid viewpoint. This issue will typically arise when heuristics are applied without regard to the context in which they are being applied. That is, semantically inconsistent results will be produced in cases where the heuristic was not appropriate and should not have been applied. The applicability of a particular heuristic, however, will typically be difficult (if not impossible) to determine in an automatic manner. Fortunately, the enrichment process provides a practical solution to this issue. Rather than applying heuristics regardless of context, the user can be notified that a heuristic is about to be applied and what the result of applying the heuristic will be. The user can then use their contextual and semantic knowledge of the viewpoint to determine whether this is a correct result, and thus choose whether or not the heuristic should be applied.

Heuristics can increase the explicit semantic content of a viewpoint, so ensuring that heuristics are explicitly activated by the user provides the benefit of making the

user aware of the semantic changes that are occurring during the translation. This process may also help identify possibilities that the user may not otherwise have considered, thus promoting a more complete understanding by the user of the phenomenon being modelled.

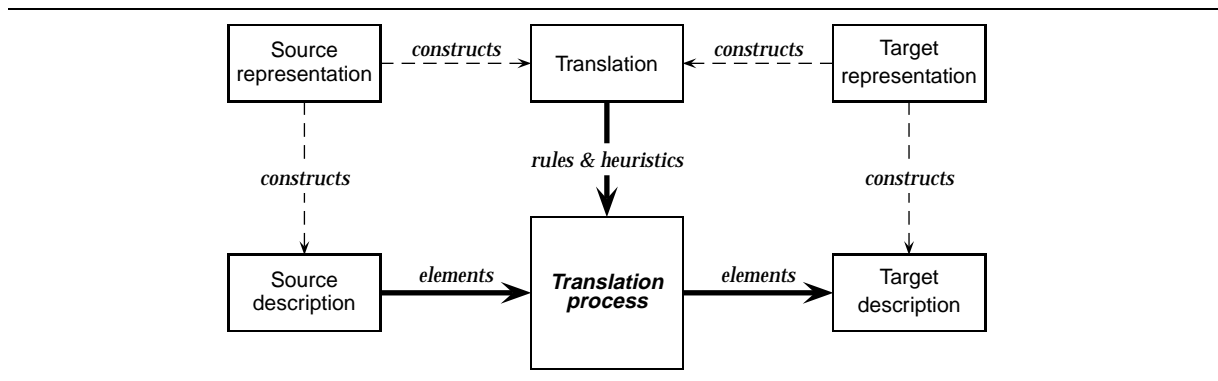
An interesting discovery was made while defining the representations used in this thesis: it was found that a small change to a representation definition can result in a marked increase in the quality of translations to and from that representation. For example, the functional dependency diagram (FDD) notation used in this thesis is a minor variant of Smith's (1985) bubble notation. Smith's original notation includes the concept of a *domain flag* for tagging attributes that are drawn from the same domain. Smith's method for deriving relations from an FDD makes use of this domain flag information to derive foreign keys. Smith's original notation has been augmented by the author (see Appendix B) to indicate that one of the tagged attributes is the 'referenced' attribute. This information is not particularly important in the FDD itself, as all that is required is to indicate that all the tagged attributes share the same domain. This small piece of extra information is, however, vitally important when performing translations to the relational model, as it is otherwise impossible to automatically determine which attribute should be the one to be referenced by foreign keys.

Thus, a small change to a representation can result in a noticeable improvement in the quality of some translations to and from that technique. Any such changes should, however, be very small and should only be used when the benefits of the change are large relative to the extent of the change. This will ensure that representations do not deviate too much from their canonical definitions.

## 4.7 The translation process

It was noted in Section 4.2 that a description translation comprises a collection of element translations, which are defined using rules and heuristics. There has, however, been no discussion of how these rules and heuristics are actually used during a translation. In this section, the translation process and the issues arising from it will be discussed. Note that the following discussion, while referring mainly to rules, is equally applicable to heuristics unless otherwise specified.

The set of rules could be thought of as analogous to a ‘program’, and the evaluation of these rules in order to perform a translation is analogous to the ‘execution’ of the program. The translation process takes as input the elements of the source description and the rules of a translation, and produces as output the elements of the target description, as illustrated in Figure 4.11.



**Figure 4.11:** The translation process

There are two obvious strategies for evaluating a set of rules and heuristics against a source description:

1. For each rule, find all collections of elements that match the source constructs of the rule, then apply the rule to each such collection; or
2. For each element, determine all rules that may apply to that element, then generate all possible element collections that match each rule. Delete all rules that cannot be matched, and apply the rest.

There may be other evaluation strategies, but they are not crucial to this thesis and will be left as an area for future research. The VML mapping system follows the second strategy, and is adopted in this thesis, as an extended version of VML will be defined in Chapter 7 for the purpose of specifying translations.

The VML mapping system iterates through all the elements of a description, and for each element, identifies all rules whose source constructs include the element’s construct. For each rule identified in this manner, possible combinations of elements matching the source constructs are generated, using the original element as a placeholder in the rule. If no such combinations can be generated for a rule, it is removed from consideration. Next, each generated combination of elements is tested against the

rule's invariants. Any combinations failing to satisfy the invariants are removed from consideration. The rule is applied to any combinations that remain at the end of this process.

For example, consider a modelling environment that supports the two representations  $\mathfrak{R}_f(\text{FuncDep}, \text{FDD}_{\text{Smith}})$  and  $\mathfrak{R}_e(\text{E-R}, \text{ERD}_{\text{Martin}})$ , and the description translation  $\mathfrak{R}_f \leftrightarrow \mathfrak{R}_e$ . Suppose this translation is specified by the following rules (ignoring completeness and composition for the sake of clarity):

$$\mathfrak{R}_f[\text{SSSINGLEKEYBUBBLE}] \leftrightarrow \mathfrak{R}_e[\text{MARTINIDENTIFIER}] \quad (\text{R1})$$

$$\begin{aligned} \mathfrak{R}_f[\text{SSSINGLEKEYBUBBLE}, \text{SSSINGLEVALUED}, \text{SSTARGETBUBBLE}] \leftrightarrow \\ \mathfrak{R}_e[\text{MARTINREGULARENTITY}] \end{aligned} \quad (\text{R2})$$

$$\begin{aligned} \mathfrak{R}_f[\text{SSMULTIKEYBUBBLE}, \text{SSMULTIVALUED}, \text{SENDKEYBUBBLE}] \leftrightarrow \\ \mathfrak{R}_e[\text{MARTINREGULARENTITY}] \end{aligned} \quad (\text{R3})$$

$$\begin{aligned} \mathfrak{R}_f[\text{SSSINGLEKEYBUBBLE}_1, \text{SSSINGLEKEYBUBBLE}_2, \\ \text{SSSINGLEVALUED}_1, \text{SSSINGLEVALUED}_2, \\ \text{SSTARGETBUBBLE}_1, \text{SSTARGETBUBBLE}_2] \leftrightarrow \\ \mathfrak{R}_e[\text{MARTINRELATIONSHIP}, \text{MARTINREGULARENTITY}_1, \\ \text{MARTINREGULARENTITY}_2] \end{aligned} \quad (\text{R4})$$

The following invariants are associated with these rules to ensure that the translation produces sensible results:

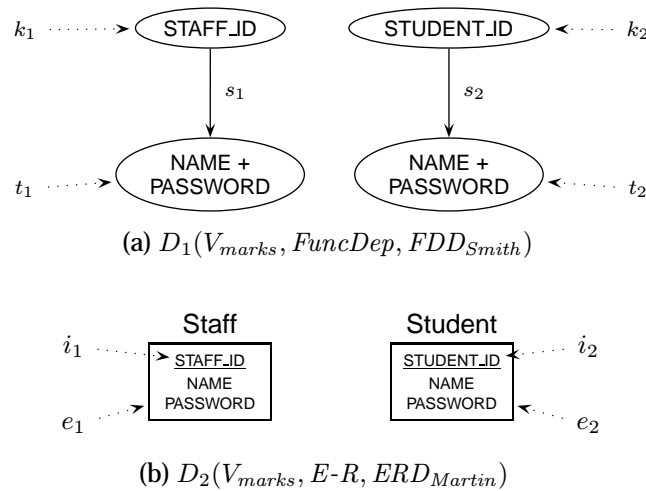
**Rule R2:** the SSSINGLEVALUED element must point from the SSSINGLEKEYBUBBLE element to the SSTARGETBUBBLE element.

**Rule R3:** the SSMULTIVALUED element must point from the SSMULTIKEYBUBBLE element to the SENDKEYBUBBLE element.

**Rule R4:** SSSINGLEVALUED<sub>1</sub> must point from SSSINGLEKEYBUBBLE<sub>1</sub> to SSTARGETBUBBLE<sub>1</sub>; SSSINGLEVALUED<sub>2</sub> must point from SSSINGLEKEYBUBBLE<sub>2</sub> to SSTARGETBUBBLE<sub>2</sub>; SSTARGETBUBBLE<sub>1</sub> must contain SSSINGLEKEYBUBBLE<sub>2</sub>; and the

MARTINRELATIONSHIP element must connect MARTINREGULARENTITY<sub>1</sub> and MARTINREGULARENTITY<sub>2</sub>.

Now suppose that a developer wishes to translate the functional dependency description  $D_1$  shown in Figure 4.12(a) into an E-R description (this example is extracted from the university marks viewpoint; see Appendix C).



**Figure 4.12:** Translating an FDD description to an ERD description

Consider the SSSINGLEKEYBUBBLE element  $k_1$ . Rule R3 is the only rule that does not have the SSSINGLEKEYBUBBLE construct in its source constructs, so the set of potentially applicable rules for element  $k_1$  is {R1, R2, R4}. Using  $k_1$  as a placeholder in the source constructs of rule R1, it can be seen that the only possible combination of elements that can be generated for this rule is  $\{k_1\}$ . Since the rule has no invariants, it can be applied to this combination of elements.

The possible combinations of elements for rule R2 are  $\{k_1, s_1, t_1\}$ ,  $\{k_1, s_1, t_2\}$ ,  $\{k_1, s_2, t_1\}$  and  $\{k_1, s_2, t_2\}$ . The invariant for rule R2 eliminates the latter three combinations, leaving only  $\{k_1, s_1, t_1\}$ , to which the rule is applied.

The possible combinations of elements for rule R4 are  $\{k_1, k_2, s_1, s_2, t_1, t_2\}$ ,  $\{k_1, k_2, s_1, s_2, t_2, t_1\}$ ,  $\{k_1, k_2, s_2, s_1, t_1, t_2\}$ ,  $\{k_2, k_1, s_1, s_2, t_1, t_2\}$ ,  $\{k_1, k_2, s_2, s_1, t_2, t_1\}$ ,  $\{k_2, k_1, s_1, s_2, t_2, t_1\}$ ,  $\{k_2, k_1, s_2, s_1, t_1, t_2\}$  and  $\{k_2, k_1, s_2, s_1, t_2, t_1\}$ . None of these combinations satisfy the invariants for rule R4, so they are all eliminated, leaving nothing to which the rule may be applied.

Thus, rule R1 is applied to element  $k_1$ , resulting in the creation of the MARTINIDENTIFIER element  $i_1$  in the target description, as shown in Figure 4.12(b), and rule R2 is applied to the combination of elements  $\{k_1, s_1, t_1\}$ , resulting in the creation of the MARTINREGULARENTITY element  $e_1$  in the target description. A similar process is carried out for the remaining elements of  $D_1$ , resulting in the E-R description shown in Figure 4.12(b). Elements that have already been created in the target description are re-used as appropriate.

This process only works as long as there are no conflicts among the rules. Sometimes it is possible for the use of one rule to exclude the use of another, which will now be discussed.

### 4.7.1 Rule exclusion

One rule is said to *exclude* the use of another if applying both rules would lead to an incorrect result. This can happen in two ways:

1. the result is syntactically incorrect, that is, an impossible target structure is produced; or
2. the result is semantically incorrect, because overlapping groups of elements in the source description may be translated in slightly different ways by different rules.

Note that the second case is different from what happens with heuristics — a heuristic may produce a semantically incorrect result, but this is a result of applying the heuristic *alone*, and is expected. The inconsistency in case 2 above is a direct consequence of applying multiple rules and/or heuristics *in combination*, and is unexpected.

The first case can arise when two or more rules map the same collection of source constructs to mutually exclusive collections of target constructs. Returning to the example in the previous section, suppose the following rule is added to the existing translation:

$$\mathfrak{R}_f[\text{SSSINGLEKEYBUBBLE}, \text{SSSINGLEVALUED}, \text{SSTARGETBUBBLE}] \leftrightarrow \mathfrak{R}_e[\text{MARTINASSOCIATIVEENTITY}], \quad (\text{R5})$$

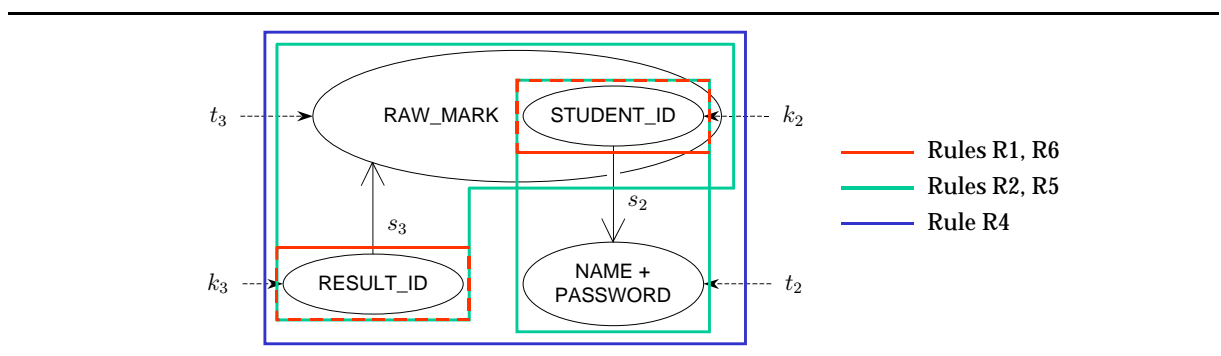
and suppose that this rule has invariants identical to those of rule R2. The addition of this rule causes a conflict: rule R2 maps the collection of source constructs  $\{\text{SSSINGLEKEYBUBBLE}, \text{SSSINGLEVALUED}, \text{SSTARGETBUBBLE}\}$  onto the target construct  $\text{MARTINREGULARENTITY}$ , whereas rule R5 maps the same collection of source constructs onto the target construct  $\text{MARTINASSOCIATIVEENTITY}$ . It is impossible for an element to be both a  $\text{MARTINREGULARENTITY}$  and a  $\text{MARTINASSOCIATIVEENTITY}$  simultaneously, in the same way that it is impossible for variables in a strongly-typed programming language to have more than one declared data type. The result of applying both rules is thus syntactically impossible in the target representation. (In practice, the rule that is applied second will override the result of the first rule, but this does not solve the problem, as the order of rule evaluation cannot be guaranteed.)

Conversely, consider the new rule:

$$\mathfrak{R}_f[\text{SSSINGLEKEYBUBBLE}] \leftrightarrow \mathfrak{R}_e[\text{MARTINATTRIBUTE}_1 \dots, \text{MARTINATTRIBUTE}_n], \quad (\text{R6})$$

which has no invariants. Despite having source constructs identical to rule R1, R6 does not conflict with R1, as the results of applying each rule are independent of each other. It is thus possible to apply both rules to an appropriate collection of source elements.

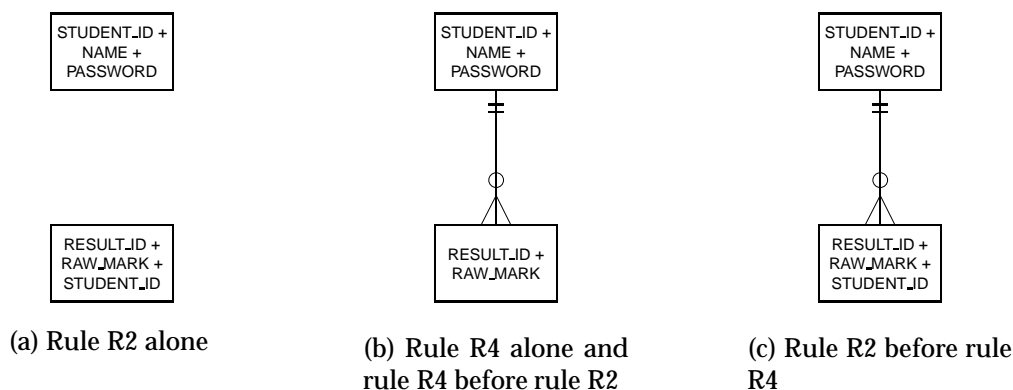
The second case can arise when two or more rules map the same collection of source elements to different, *non*-mutually exclusive collections of target elements. Consider the description  $D_2$  shown in Figure 4.13. As can be seen from the figure, rules R1, R2, R4, R5 and R6 may all be applied to different parts of the description.



**Figure 4.13:** Applying multiple rules to the same source structure

Suppose that rules R2 and R4 are applied in isolation. This results in two different E-R structures, as shown in Figures 4.14(a) and 4.14(b) on the next page. Both of these structures are semantically consistent with the original description when considered

in isolation, but the semantics of the first are different from those of the second, as the bottom entity has slightly different attributes in each ERD. The order in which the rules are applied also has an impact on the result — applying rule R4 before rule R2 will also result in the structure shown in Figure 4.14(b), whereas applying rule R2 before rule R4 will result in the structure shown in Figure 4.14(c). The structure that best corresponds to the semantics of original description is that shown in Figure 4.14(b), so in this example, only rule R4 should be applied. In effect, rule R4 should be checked before rule R2, that is, there is an implied ordering to rule evaluation in this situation.



**Figure 4.14:** Results of applying multiple rules to the same source structure

In both cases 1 and 2 described at the beginning of this section, it is impossible to determine which rule(s) should be applied when presented with a matching set of elements. In the absence of any additional information, the only solution is to choose one of the rules as the ‘default’, and ensure that the remaining rules cannot be applied in the same direction (if the excluded rules were already unidirectional, this will effectively delete the excluded rules). While this is appropriate for the first case, it may not always be appropriate for the second. A better approach would therefore be to explicitly specify the exclusions among rules. This would make it possible to filter a set of potentially applicable rules by removing from consideration all rules that are excluded by others.

The problem thus becomes one of determining which rules may potentially conflict when applied in a particular direction. This can be done by finding those rules whose source constructs are subsets of the source constructs of other rules. A rule  $r_i$  is said to be *subsumed* by another rule  $r_s$  in a particular direction  $d$  if:



- the source constructs of  $r_i$  are a subset ( $\subseteq$ ) of the source constructs of  $r_s$ ; and
- both rules may be applied in the direction  $d$ ; and
- the invariants for both rules are not contradictory.

Consider rules R1 and R2 above. The source constructs of rule R2 are {SSSINGLEKEYBUBBLE, SSSINGLEVALUED, SSTARGETBUBBLE}, and the source constructs of rule R1 are {SSSINGLEKEYBUBBLE}, which are obviously a subset of the source constructs of rule R2. Both rules are bidirectional, and there are no contradictory invariants, so rule R1 is subsumed by rule R2 and there is a potential for the two to conflict. Examining the rules, however, shows that the results of applying the rules are independent, so no exclusion is required.

Now consider rules R2 and R4. The source constructs of R4 are {SSSINGLEKEYBUBBLE<sub>1</sub>, SSSINGLEKEYBUBBLE<sub>2</sub>, SSSINGLEVALUED<sub>1</sub>, SSSINGLEVALUED<sub>2</sub>, SSTARGETBUBBLE<sub>1</sub>, SSTARGETBUBBLE<sub>2</sub>}, which form a superset of the source constructs of rule R2. Both rules are bidirectional and there are no contradictory invariants, so rule R2 is subsumed by rule R4. Applying both rules can produce the semantically incorrect structure shown in Figure 4.14(c), so rule R4 should exclude the use of rule R2.

It is possible to express rule subsumption by means of a directed graph  $G = (\mathcal{V}, \mathcal{E})$  with no loops<sup>4</sup>, where  $\mathcal{V}$  is a set of labelled vertices representing rules, and  $\mathcal{E}$  is a set of directed edges representing rule subsumption. Each vertex  $v_i$  corresponds to a rule  $r_i$ . An edge  $e_{ij} : v_i \rightarrow v_j$  indicates that rule  $r_i$  subsumes rule  $r_j$ . The edges of this graph may then be manually annotated to indicate which rules exclude others, producing what is termed here a *subsumption/exclusion graph* for a translation. The annotation  $\rho$  on an edge indicates rule exclusion, thus an edge  $e_{kl} : v_k \xrightarrow{\rho} v_l$  indicates that rule  $r_k$  subsumes and excludes the use of rule  $r_l$ . Each translation will have two subsumption/exclusion graphs, one for the forward direction ( $\vec{G}$ ) and one for the reverse ( $\overleftarrow{G}$ ). Subsumption/exclusion graphs for the translations defined in this thesis may be found in Chapter 5.

In Figure 4.15 on the following page is shown the subsumption/exclusion graph  $\vec{G}$  for the forward direction of the translation used in the examples above. The numbers in the vertices correspond to rule numbers. Note the pair of opposing exclusion edges

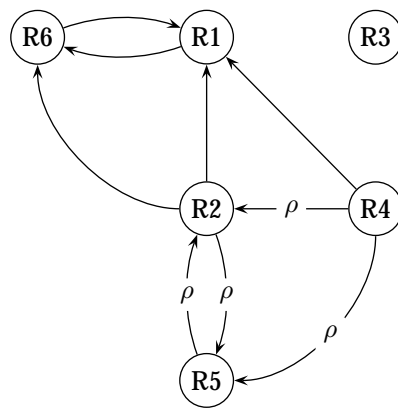
---

<sup>4</sup>A loop is an edge that starts and ends at the same vertex (Diestel, 1997, p. 24).

between R2 and R5. This makes it impossible to determine in which order these two rules should be evaluated, and is an indication that both rules should not be used in the forward direction. It would therefore be sensible to make rule R5 unidirectional in the reverse direction, that is:

$$\mathfrak{R}_f[\text{SSINGLEKEYBUBBLE}, \text{SSINGLEVALUED}, \text{SSTARGETBUBBLE}] \leftarrow \mathfrak{R}_e[\text{MARTINASSOCIATIVEENTITY}] \quad (\text{R5})$$

This would remove rule R5 and its associated edges from the graph, because unidirectional rules only appear in the graph appropriate to their direction.



**Figure 4.15:** ‘Forward’ subsumption/exclusion graph for the example translation

Rule subsumption may be determined automatically from a set of rules, and an algorithm for building subsumption/exclusion graphs is presented in Section 7.5.1. Rule exclusion cannot be determined automatically and must therefore be explicitly specified by translations. Once subsumption/exclusion graphs have been built, the translation process can use this information to more effectively filter the set of rules that may apply to a particular source structure, and also to determine the order in which rules should be evaluated. In particular, rules that are not subsumed by others should be evaluated first.

In summary, the issue of rule exclusion arises because it is possible for multiple rules to be applied to similar groups of source elements. When the source constructs of one rule subsume the source constructs of another, there is potential for the rules to conflict. If the rules do not conflict, then no special treatment is required, but if the rules *do* conflict, the ‘subsuming’ rule should be evaluated before the subsumed rule. If

the subsuming rule can be applied to the collection of source elements under consideration, then any conflicting subsumed rules should be excluded from further evaluation against those elements. The subsumption/exclusion graphs for a translation provide a means of encoding this information so that the mapping system can determine the correct order in which to evaluate rules against source elements. Reducing the amount of overlap among rules would obviously ameliorate the rule exclusion problem; ideally, each rule would translate only a single source element into a single target element, but in practice the complexity of the mappings between source and target elements precludes this. The FDD to ERD translation (which will be fully defined in Section 5.4 on page 135) is a particularly good example of this complexity.

## 4.8 Summary

In this chapter, the issues arising from the process of translating between different viewpoint representations were examined. In particular, it was shown that the translation of a description from one representation to another may be defined in terms of a collection of *rules* and *heuristics*, which specify mappings between the constructs of the source and target representations. Rules are guaranteed to produce a result that is semantically consistent with the source description, whereas heuristics may sometimes produce a semantically inconsistent result. Heuristics may also affect the semantics of the viewpoint by making explicit semantics that are implicit in the source description. Rules may be categorised as technique- or scheme-level rules, in the same way that constructs may be categorised as either technique- or scheme-level constructs. Technique-level rules define the generic translation between two techniques, and scheme-level rules specialise this to produce a translation between a specific pair of representations.

A high-level abstract notation was defined for specifying description and element translations, and is summarised in Table 4.1 on the following page (the representation and description notation defined in Chapter 3 is repeated here for completeness). The operators of this notation were derived from four properties of translations: *type* (technique/scheme for descriptions, rule/heuristic for elements), *completeness* (partial or complete), *composition* (atomic or composite) and *direction* (unidirectional or bidirec-

**Table 4.1: Summary of representation, description and translation notation**

| Notation   | Associated term                    | Definition  |
|--|------------------------------------|---|
| $V$  | A viewpoint .....                  | A formatted expression of a description of a real-world phenomenon. <sup>a</sup>  |
| $T$  | A technique .....                  | A collection of generic constructs that form a modelling 'method', for example, the relational model or entity-relationship approach. <sup>a</sup>                                  |
| $S$  | A scheme .....                     | A collection of specialised constructs that form a modelling 'notation', for example, SQL/92 or Martin ERD notation. <sup>a</sup>   |
| $\mathfrak{R}(T, S)$ or $\mathfrak{R}$   | A representation .....             | Representation $\mathfrak{R}$ comprises constructs defined by the combination of technique $T$ and scheme $S$ . <sup>a</sup>  |
| $D(V, T, S)$ or $D$  | A description .....                | Description $D$ of viewpoint $V$ is expressed using constructs of technique $T$ and scheme $S$ . <sup>a</sup>   |
| $\mathfrak{R}(T, S) [\text{CON}]$ ,<br>$\mathfrak{R} [\text{CON}]$ , or $\text{CON}$ | A construct of a representation    | $\text{CON}$ specifies a construct of representation $\mathfrak{R}(T, S)$ . <sup>a</sup>  |
| $D(V, T, S) [e : \text{CON}]$ ,<br>$D [e : \text{CON}]$ , or $D [e]$                 | An element of a description ..     | $e$ specifies an element (instantiated from construct $\text{CON}$ ) of description $D(V, T, S)$ . <sup>a</sup>   |
| $\mathfrak{R}_b \preceq \mathfrak{R}_a$  | Expressive power (inclusive) ..... | The expressive power of representation $\mathfrak{R}_b$ is inclusive of (dominates) the expressive power of representation $\mathfrak{R}_a$ . <sup>c</sup>                          |
| $\mathfrak{R}_a \equiv \mathfrak{R}_b$   | Expressive power (equivalent) ...  | The expressive power of representation $\mathfrak{R}_b$ is equivalent to the expressive power of representation $\mathfrak{R}_a$ . <sup>c</sup>                                     |
| $\rightarrow$  | A translation .....                | The translation of an entire description from one representation to another; or the translation of an entire element from one representation to another using a rule. <sup>b</sup>  |
| $\mapsto$  | A partial translation .....        | The translation of part of a description from one representation to another; or the translation of part of an element from one representation to another using a rule. <sup>b</sup> |
| $\mapsto, \mapsto$   | A heuristic translation .....      | The complete or partial translation of an element from one representation to another using a heuristic. <sup>b</sup>  |
| $\overset{\circ}{\mapsto}, \overset{\circ}{\mapsto}$                                 | A composite translation .....      | A translation of a description or element from one representation to another that can be decomposed into a collection of 'sub-translations'.  |
| $\mapsto$  | A refinement .....                 | The combination of several input descriptions to produce a single output description during the schema generation process. <sup>d</sup>   |

**Notes on Table 4.1:**

- <sup>a</sup> Defined in Section 3.3 on page 47.
- <sup>b</sup> The translation cannot be decomposed into sub-translations.
- <sup>c</sup> Defined in Chapter 8.
- <sup>d</sup> Discussed in Section 10.5 on page 289.

tional). This notation is useful for expressing translations in a concise manner; more detailed specification of translations will use a modified variant of VML, discussed in Chapter 7.

Also discussed was a potentially beneficial side-effect of the translation process: translating descriptions that are expressed using different representations makes it easier to highlight potential inconsistencies between the descriptions. The quality of translations will obviously have an impact on this process. Heuristics are one method of improving the quality of a translation (this improvement will be shown in Chapter 8); another method is *enrichment*, which provides the opportunity to elicit additional information from the user before, during and after a translation. Enrichment also provides a practical solution to the problem of heuristics generating semantically inconsistent results.

The translation process itself was also described, and the issue of rule exclusion was raised. The concept of rule subsumption was defined as a means of detecting rules that may potentially conflict with each other, and the subsumption/exclusion graph was introduced as a method of encoding rule subsumption and exclusion information.

In the next chapter, the abstract translation notation is used to define the rules for the  $\mathfrak{A}_e(E-R, ERD_{Martin}) \rightleftharpoons \mathfrak{A}_r(Relational, SQL/92)$  translation. Issues arising from these rules and those for two other translations will also be discussed.



# Chapter 5

## Definition of translations

### 5.1 Introduction

In this chapter the three technique translations:

- $\mathfrak{R}_e(E-R, ERD_{Martin}) \rightleftharpoons \mathfrak{R}_r(Relational, SQL/92)$ ,
- $\mathfrak{R}_f(FuncDep, FDD_{Smith}) \rightarrow \mathfrak{R}_e(E-R, ERD_{Martin}) / \mathfrak{R}_f \leftarrow \mathfrak{R}_e$  and
- $\mathfrak{R}_e(E-R, ERD_{Martin}) \rightleftharpoons \mathfrak{R}_d(DataFlow, DFD_{G\&S})$

are examined and a set of rules and heuristics defined for performing the first translation. The remaining two translations are presented here in summary form only, but full definitions of these translations may be found in Appendix E.

The first translation ( $\mathfrak{R}_e \rightleftharpoons \mathfrak{R}_r$ ) was chosen as an example of a well-known and understood translation between two representations with a relatively large expressive overlap. It is interesting to note, however, that despite being a well-understood problem, analysis of the rules presented here reveals aspects of the translation that are often not included in tools that implement this translation (see Section 5.3.5 for details).

The second translation ( $\mathfrak{R}_f \rightarrow \mathfrak{R}_e / \mathfrak{R}_f \leftarrow \mathfrak{R}_e$ ) was chosen as an example of a less familiar but still reasonably conventional translation between two representations with a moderate to high expressive overlap. Functional dependencies are usually used to derive relational structures, but using them to generate E-R structures is equally valid.

The third translation ( $\mathfrak{R}_e \rightleftharpoons \mathfrak{R}_d$ ) was chosen as an example of an unusual translation between two representations with a relatively small expressive overlap. Because of this small expressive overlap it was expected that there would be very little information that could be translated between the two representations.

The translations are summarised in Table 5.1. The table lists the translation, the type of each representation (formal, semi-formal or informal) and the estimated expressive overlap between the two representations. The completeness of the translation in each direction is indicated by the translation operators in the first column.

**Table 5.1:** Summary of translations

| Translation   | Representations  | Expressive overlap  |
|---|--|---------------------|
| $\mathfrak{R}_e(E-R, ERD_{Martin}) \rightleftharpoons \mathfrak{R}_r(Relational, SQL/92)$   | $\mathfrak{R}_e$ (semi-formal)<br>$\mathfrak{R}_r$ (formal)      | high                |
| $\mathfrak{R}_f(FuncDep, FDD_{Smith}) \rightarrow \mathfrak{R}_e(E-R, ERD_{Martin})$<br>$\mathfrak{R}_f(FuncDep, FDD_{Smith}) \leftarrow \mathfrak{R}_e(E-R, ERD_{Martin})$ | $\mathfrak{R}_f$ (formal)<br>$\mathfrak{R}_e$ (semi-formal)      | moderate<br>to high |
| $\mathfrak{R}_e(E-R, ERD_{Martin}) \rightleftharpoons \mathfrak{R}_d(DataFlow, DFD_{G\&S})$   | $\mathfrak{R}_e$ (semi-formal)<br>$\mathfrak{R}_d$ (semi-formal) | low                 |

The rules for the  $\mathfrak{R}_e \rightleftharpoons \mathfrak{R}_r$  translation are defined in the context of three example viewpoints, which are briefly described in Section 5.2. Full details of these viewpoints may be found in Appendix C. The  $\mathfrak{R}_e \rightleftharpoons \mathfrak{R}_r$  translation is defined in detail in Section 5.3, followed by summaries of the  $\mathfrak{R}_f \rightarrow \mathfrak{R}_e / \mathfrak{R}_f \leftarrow \mathfrak{R}_e$  and  $\mathfrak{R}_e \rightleftharpoons \mathfrak{R}_d$  translations in Sections 5.4 and 5.5 respectively.

### 5.1.1 Notation used in this chapter

Translation and rule definitions use the abstract notation defined in Chapters 3 and 4. SMALL CAPS denote constructs of representations, which are defined in detail in Appendix D.

Many of the rule definitions are illustrated using diagrams. Some representations allow the definition of unique identifiers for ‘objects’, for example, primary keys in relational representations. Where appropriate, the notation AB denotes that the attributes A and B are components of the unique identifier of some object.

Rules within a translation are numbered in the order they are presented. The rule numbers for scheme-level rules are prefixed by the letter ‘S’ (S1, S2, ...), those for technique-level rules by the letter ‘T’ (T1, T2, ...), and those for heuristics by the letter ‘H’ (H1, H2, ...).



## 5.2 Example viewpoints

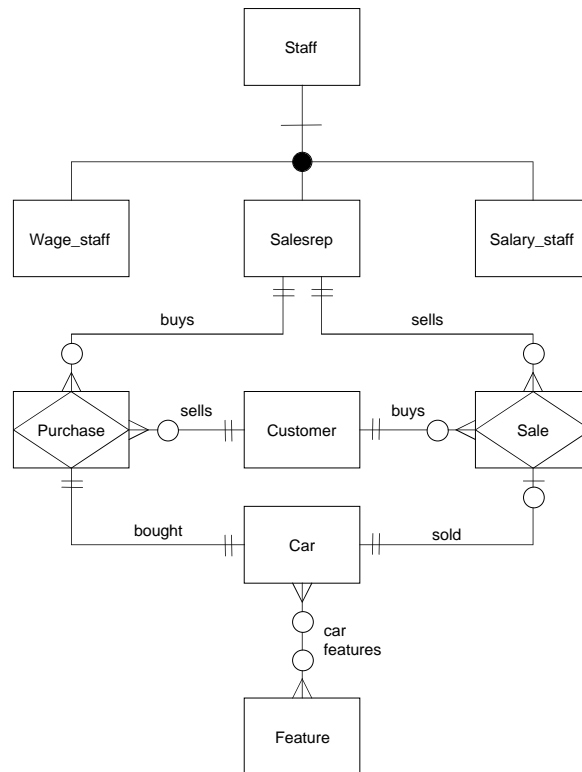
Three unrelated viewpoints are used in the examples in this chapter and elsewhere in the thesis: a viewpoint of a simple used cars dealership, a viewpoint of an agricultural research institute and a viewpoint of a course assessment marks database. These viewpoints are drawn from examples developed for teaching over the course of several years, and are thus well-defined and understood. Since only the  $\mathfrak{R}_e \Rightarrow \mathfrak{R}_r$  translation is fully detailed in this chapter, only Martin E-R and SQL/92 descriptions are presented here. Other descriptions and full details of the viewpoints may be found in Appendix C.

### 5.2.1 The used cars viewpoint

The first viewpoint to be examined ( $V_{cars}$ ) is of a small used car dealership. A full explanation of this viewpoint may be found in Section C.1 on page 359. The main points may be summarised as follows:

- The business purchases cars from and sells cars to customers. Purchases and sales are always for a single car and are dealt with exclusively by sales representatives.
- The business needs to know which sales representative was involved in a sale or purchase for the purposes of recording commissions.
- Sales representatives are paid commissions, while other staff are paid salary or wages.
- A list of non-standard features is kept for each car (such as air conditioning or alloy wheels); some cars have several features, some have none.

A Martin E-R description ( $D_1$ ) of the viewpoint is shown in Figure 5.1 on the following page and an SQL/92 description ( $D_2$ ) is shown in Figure 5.2 on page 111. Note in Figure 5.1 that the various types of staff have been modelled as subtypes of the Staff entity.



**Figure 5.1:** Martin E-R description  $D_1(V_{cars}, E-R, ERD_{Martin})$  of the used cars viewpoint

## 5.2.2 The agricultural research institute viewpoint

The second viewpoint to be examined ( $V_{agri}$ ) is of an agricultural research institute. A full explanation of this viewpoint may be found in Section C.2 on page 364. The main points may be summarised as follows:

- The institute carries out research under contract to clients; each contract is for a single client.
- A contract comprises a series of experiments. These experiments are to evaluate the effect of a collection of fertilisers on the growth rate of various breeds of sheep grazing on various grass types. Each experiment is run by a single scientist.
- Each experiment takes place in several paddocks, each of which is sown with a particular grass type and treated with a particular fertiliser (or no fertiliser as a control).
- A flock of sheep of the same breed is placed in each paddock.

---

```

create table staff
( ird_number char(7),
  name      char(80),
  address   char(80),
  phone     char(12),

  primary key (ird_number)
);

create table wage_staff
( wage_staff_id char(7),
  hourly_rate   smallint,
  hours_per_week integer(2),

  primary key (wage_staff_id),
  foreign key (wage_staff_id)
    references staff (ird_number)
);

create table salary_staff
( salary_staff_id char(7),
  salary          smallint,

  primary key (salary_staff_id),
  foreign key (salary_staff_id)
    references staff (ird_number)
);

create table salesrep
( salesrep_id   char(7),
  commission_rate integer(2),

  primary key (salesrep_id),
  foreign key (salesrep_id)
    references staff (ird_number)
);

create table customer
( customer_no char(6),
  name        char(80),
  address     char(80),
  phone       char(12),

  primary key (customer_no)
);

create table car
( registration char(6),
  vin          char(20) not null unique,
  make         char(20),
  model        char(20),
  year         smallint,
  colour       char(20),
  odometer     integer,
  miles_km     char(1),
  list_price   integer,
  purchase_id  char(6) not null unique,
  sale_id      char(6) unique,

  primary key (registration),
  foreign key (purchase_id)
    references purchase,
  foreign key (sale_id)
    references sale
);

create table feature
( feature_code char(6),
  description   char(80),

  primary key (feature_code)
);

create table car_feature
( feature_code char(6),
  registration  char(6),

  primary key (feature_code, registration),
  foreign key (feature_code)
    references feature,
  foreign key (registration)
    references car
);

create table purchase
( purchase_id   char(6),
  purchase_date date,
  purchase_price integer,
  customer_no   char(6) not null,
  salesrep_id   char(7) not null,
  registration  char(6) not null unique,

  primary key (purchase_id),
  foreign key (customer_no)
    references customer,
  foreign key (salesrep_id)
    references salesrep,
  foreign key (registration)
    references car
);

create table sale
( sale_id       char(6),
  sale_date     date,
  sale_price    integer,
  customer_no   char(6) not null,
  salesrep_id   char(7) not null,
  registration  char(6) not null unique,

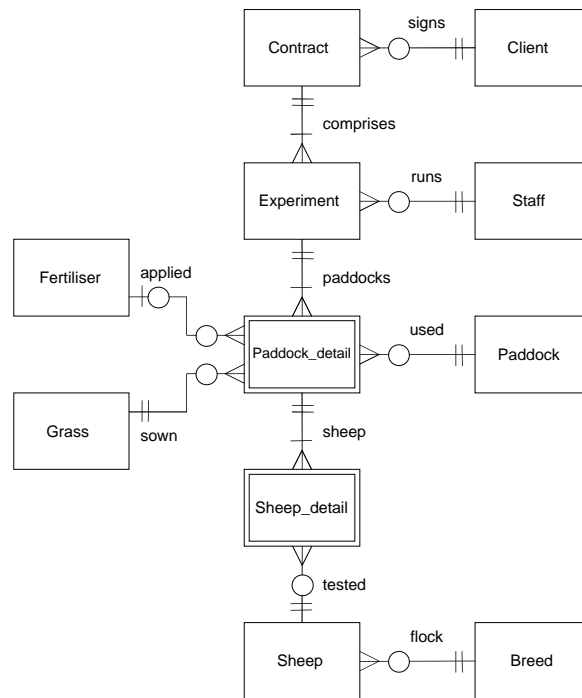
  primary key (sale_id),
  foreign key (customer_no)
    references customer,
  foreign key (salesrep_id)
    references salesrep,
  foreign key (registration)
    references car
);

```

---

**Figure 5.2:** SQL/92 description  $D_2(V_{cars}, Relational, SQL/92)$  of the used cars view-point

A normalised Martin E-R description ( $D_3$ ) of this viewpoint is shown in Figure 5.3, and an SQL/92 description ( $D_4$ ) of the viewpoint is shown in Figure 5.4. Note in Figure 5.3 the assumption that details of the sheep associated with an experiment are dependent on the paddock details.



**Figure 5.3:** Normalised Martin E-R description  $D_3(V_{agri}, E-R, ERD_{Martin})$  of the agricultural research institute viewpoint

### 5.2.3 The assessment marks viewpoint

The third viewpoint to be examined ( $V_{marks}$ ) is of a teacher recording assessment marks for a course. A full explanation of this viewpoint may be found in Section C.3 on page 369. The main points may be summarised as follows:

- The course assessment package comprises several assessment elements; students complete and submit each as an assignment.
- Each assessment element comprises several questions, and each question in turn may comprise several sub-questions. Similarly, an assignment comprises several answers which in turn may comprise several sub-answers.

---

```

create table staff
( staff_id char(10),
  name      char(20),
  address   char(20),
  gender    char(1),
  dob       date,
  title     char(15),
  salary    decimal(6,2),

  primary key (staff_id)
);

create table client
( client_id char(10),
  name      char(20),
  address   char(20),
  category  char(1),

  primary key (client_id)
);

create table fertiliser
( fertiliser_id char(10),
  name          char(20),
  supplier      char(20),

  primary key (fertiliser_id)
);

create table grass
( grass_id char(10),
  name     char(20),

  primary key (grass_id)
);

create table paddock
( paddock_id char(10),
  address    char(20),
  area       smallint,
  moisture   smallint,
  sunshine   smallint,

  primary key (paddock_id)
);

create table breed
( breed_name char(20),
  details    char(20),
  flock_size smallint,

  primary key (breed_name)
);

create table sheep
( sheep_id char(10),
  breed_name char(20) not null,
  dob       date,
  gender    char(1),
  health    char(1),

  primary key (sheep_id),
  foreign key(breed_name)
    references breed
);

create table contract
( contract_id char(10),
  client_id   char(10) not null,
  details     char(20),
  fee         decimal(8,2),
  gst         decimal(8,2),
  sign_date   date not null,
  finish_date date,

  primary key (contract_id),
  foreign key (client_id)
    references client
);

create table experiment
( experiment_id char(10),
  contract_id   char(10) not null,
  staff_id      char(10) not null,
  finish_date   date,
  start_date    date,

  primary key (experiment_id),
  foreign key (contract_id)
    references contract,
  foreign key(staff_id)
    references staff
);

create table paddock_detail
( experiment_id char(10),
  paddock_id    char(10) not null,
  fertiliser_id char(10),
  grass_id      char(10) not null,

  primary key (experiment_id, paddock_id),
  foreign key (experiment_id)
    references experiment,
  foreign key (paddock_id)
    references paddock,
  foreign key (fertiliser_id)
    references fertiliser,
  foreign key (grass_id)
    references grass
);

create table sheep_detail
( experiment_id char(10),
  paddock_id    char(10),
  sheep_id      char(10),
  finish_weight smallint,
  start_weight  smallint,

  primary key (experiment_id, paddock_id,
              sheep_id),
  foreign key (experiment_id)
    references experiment(experiment_id),
  foreign key (sheep_id)
    references sheep,
  foreign key (experiment_id, paddock_id)
    references paddock_detail
);

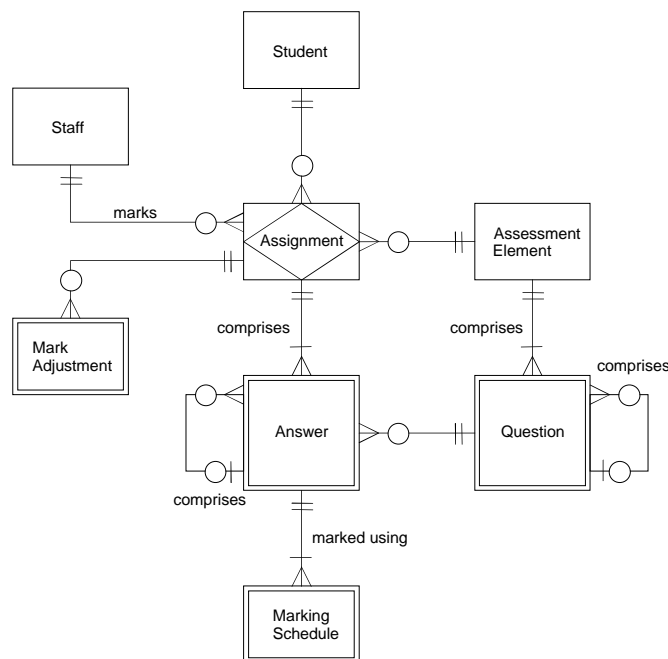
```

---

**Figure 5.4:** SQL/92 description  $D_4(V_{agri}, Relational, SQL/92)$  of the agricultural research institute viewpoint

- An assignment is marked by a single staff member and an overall mark awarded. This mark may be broken down into marks for each individual answer.
- The mark for each answer is determined according to a marking schedule that specifies several criteria and a mark allocation for each criterion.
- Marks may be adjusted at a later date for reasons such as illness or technical difficulties.

A normalised Martin E-R description ( $D_5$ ) of the viewpoint is shown in Figure 5.5, and an SQL/92 description ( $D_6$ ) of the viewpoint is shown in Figure 5.6. Note that the Question and Answer weak entities have been given their own unique entity identifiers rather than a partial identifier in order to facilitate questions having sub-questions and answers having sub-answers.



**Figure 5.5:** Normalised Martin E-R description  $D_5(V_{marks}, E-R, ERD_{Martin})$  of the assessment marks viewpoint

### 5.3 $\mathfrak{R}_e(E-R, ERD_{Martin}) \rightleftharpoons \mathfrak{R}_r(Relational, SQL/92)$

In this translation descriptions are translated between the entity-relationship approach expressed using Martin ERD notation and the relational model expressed using ANSI

---

```

create table staff
( staff_id char(8),
  name      char(80),
  password  char(20),

  primary key (staff_id)
);

create table student
( student_id char(7),
  name       char(80),
  password   char(20),

  primary key (student_id)
);

create table element
( element_id integer,
  name       char(80),
  total_mark smallint,
  percent    smallint,
  due_date   date,
  late_penalty smallint,

  primary key (element_id)
);

create table question
( question_id integer,
  element_id  integer not null,
  number      char(5),
  marks       smallint,
  guidelines  char(500),
  parent_question integer,

  primary key (question_id),
  foreign key (element_id)
    references element,
  foreign key (parent_question)
    references question
);

create table assignment
( assign_id integer,
  element_id integer not null,
  student_id char(7) not null,
  staff_id   char(8) not null,
  date_submitted date,
  date_marked date,
  raw_mark   smallint,
  comments   char(500),

  primary key (assign_id),
  foreign key (element_id)
    references element,
  foreign key (student_id)
    references student,
  foreign key (staff_id)
    references staff
);

create table adjustment
( assign_id integer,
  adjustment_no integer,
  reason     char(200),
  amount     smallint,

  primary key (assign_id,
              adjustment_no),
  foreign key (assign_id)
    references assignment
);

create table answer
( answer_id integer,
  assign_id integer not null,
  question_id integer not null,
  mark      smallint,
  comments  char(500),
  parent_answer integer,

  primary key (answer_id),
  foreign key (assign_id)
    references assignment,
  foreign key (question_id)
    references question,
  foreign key (parent_answer)
    references answer
);

create table marking_schedule
( answer_id integer,
  criterion_name char(30),
  mark         smallint,
  comments     char(500),

  primary key (answer_id,
              criterion_name),
  foreign key (answer_id)
    references answer
);

```

---

**Figure 5.6:** SQL/92 description  $D_6(V_{marks}, \textit{Relational}, \textit{SQL}/92)$  of the assessment marks viewpoint

SQL/92. The translation is partial in both directions because some of the rules are partial in each direction. That is:

$$\mathfrak{R}_e(E-R, ERD_{Martin}) \rightleftharpoons \mathfrak{R}_r(Relational, SQL/92)$$

In the forward direction,  $\mathfrak{R}_e \rightarrow \mathfrak{R}_r$  is effectively SQL schema generation from an ERD and is a common operation performed by many CASE tools. The reverse translation  $\mathfrak{R}_r \rightarrow \mathfrak{R}_e$  corresponds to reverse engineering of an SQL schema into an ERD, and is illustrated by the arrow labelled  $T_4$  in Figure 3.1 on page 43. When translating in the forward direction, the input ERD must be normalised (although many-to-many relationships are allowed), for reasons that will be discussed in Section 5.3.5 on page 129.

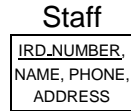
Scheme-level rules for this translation are defined in Section 5.3.1, in the context of the example viewpoints described above. Rules that are applicable to the used cars viewpoint are considered first, followed by additional rules applicable to the agricultural and assessment marks viewpoints. Other scheme-level rules that are not applicable to any of the example viewpoints are defined in Section 5.3.2, followed by heuristics in Section 5.3.3. Generalisations of the rules are then identified in Section 5.3.4 to produce a list of technique-level rules for the translation. The translation is summarised and some interesting features of it are discussed in Section 5.3.5.

### 5.3.1 Scheme-level rules

The two descriptions shown in Figure 5.1 on page 110 and Figure 5.2 on page 111 will now be compared to identify possible rules. The first correspondence is between a regular entity and a table, as shown in Figure 5.7. The Staff regular entity translates directly to the `staff` table and vice versa. The correspondences between the properties of the appropriate representation constructs (`MARTINREGULARENTITY` and `SQL92TABLE` respectively) are shown in Table 5.2. Note that the `alternateKeys` property of `SQL92TABLE` has no corresponding property in `MARTINREGULARENTITY`, and the `typeHierarchy` property of `MARTINREGULARENTITY` has no corresponding property in `SQL92TABLE`. Consequently, this rule is partial in both directions. That is:

$$\mathfrak{R}_e[\text{MARTINREGULARENTITY}] \rightleftharpoons \mathfrak{R}_r[\text{SQL92TABLE}] \quad (\text{S1})$$





```

create table staff
( ird_number char(7),
  name char(80),
  address char(80),
  phone char(12),

  primary key (ird_number)
);
  
```

**Figure 5.7:** Translating a regular entity to and from SQL/92 (rule S1)

**Table 5.2:** Correspondences between the properties of the MARTINREGULARENTITY and SQL92TABLE constructs

| MARTINREGULARENTITY |   | SQL92TABLE                      |
|---------------------|---|---------------------------------|
| name                | ↔ | name                            |
| attributes          | ↔ | attributes <sup>a</sup>         |
| identifier          | ↔ | primaryKey                      |
| relationships       | ⇒ | refFKs/foreignKeys <sup>b</sup> |
| –                   |   | alternateKeys                   |
| typeHierarchy       |   | –                               |

**Notes on Table 5.2:**

<sup>a</sup> The attributes property is inherited from the RMRELATION construct and contains the columns of the table.

<sup>b</sup> This is not an exact correspondence, but is close enough for the purposes of this comparison.

Definitions of the properties of representation constructs may be found in Appendix D.

Rule S1 may also be extended to associative entities such as the Purchase entity. In the forward direction, an associative entity translates to an SQL/92 table in a manner identical to a regular entity. In the reverse direction, however, it is impossible to determine whether an SQL/92 table corresponds to a regular entity or an associative entity, as the representation  $\mathfrak{R}_r$  cannot express this information. This leads to a case 1 conflict as described in Section 4.7.1 on page 98: two rules with the same source constructs but different incompatible target constructs. To avoid this conflict, the rule for associative entities is unidirectional in the forward direction. That is:

$$\mathfrak{R}_e [\text{MARTINASSOCIATIVEENTITY}] \rightarrow \mathfrak{R}_r [\text{SQL92TABLE}] \quad (\text{S2})$$

Two further correspondences may be identified from rule S1. First, an attribute of an entity translates directly to an SQL/92 column and vice versa, as shown in Figure 5.7. The repeating and attributeGroups properties of the MARTINATTRIBUTE construct have no corresponding properties in SQL92COLUMN, so this rule is partial in the forward direction. In the reverse direction, there is no MARTINATTRIBUTE property

corresponding to the `SQL92COLUMN` constraints property, so the rule is also partial in the reverse direction. That is:

$$\mathfrak{R}_e [\text{MARTINATTRIBUTE}] \rightleftharpoons \mathfrak{R}_r [\text{SQL92COLUMN}] \quad (\text{S3})$$

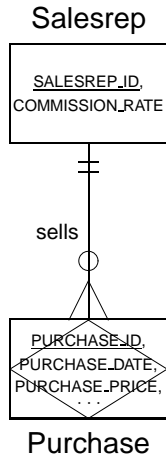
Second, the entity identifier of an entity translates to the primary key of an SQL/92 table and vice versa, as illustrated by the `IRD_NUMBER` attribute in Figure 5.7. This rule is complete in the forward direction, but partial in the reverse direction, as there is no `MARTINIDENTIFIER` property corresponding to the `SQL92PRIMARYKEY` refFKs property. That is:

$$\begin{aligned} \mathfrak{R}_e [\text{MARTINIDENTIFIER}] &\rightarrow \mathfrak{R}_r [\text{SQL92PRIMARYKEY}] \\ \mathfrak{R}_e [\text{MARTINIDENTIFIER}] &\leftarrow \mathfrak{R}_r [\text{SQL92PRIMARYKEY}] \end{aligned} \quad (\text{S4})$$

The next elements to consider in the used cars viewpoint are the relationships between entities. The rules for these vary depending on the optionality and cardinality of the relationships, as will be seen below. Consider the sells one-to-many relationship shown on the left of Figure 5.8. A relationship between two entities of any sort will typically translate to a foreign key linking the two corresponding tables and vice versa. For a one-to-many relationship, the foreign key must appear in the table corresponding to the ‘many’ entity, and reference the table corresponding to the ‘one’ entity, as illustrated by the foreign key to `salesrep` in the `purchase` table on the right of Figure 5.8. Placing the foreign key in the other table would result in an unnormalised structure, which is impossible in SQL/92.

Foreign keys are designed to support referential integrity, which states that a foreign key must either refer to a valid primary key value in the referenced table, or it must be null (Codd, 1990, p. 23). The ‘one’ end of the sells relationship is mandatory, which means that the foreign key cannot be null. Thus, a `not null` constraint is placed on the columns of the foreign key, as illustrated by the `salesrep_id` column in Figure 5.8.

Nothing has been said about the optionality of the ‘many’ end of the relationship. Examining the SQL/92 code in Figure 5.8, it can be seen that optionality of the ‘one’ end of the relationship is expressed in SQL/92 by the existence (or not) of a `not null` constraint on the columns of the foreign key in the ‘many’ table. There is no foreign



```

create table salesrep
( salesrep_id char(7),
  commission_rate integer(2),

  primary key (salesrep_id),
  foreign key (salesrep_id) references staff
);

create table purchase
( purchase_id char(6),
  purchase_date date,
  purchase_price integer,
  customer_no char(6) not null,
  salesrep_id char(7) not null,
  registration char(6) not null unique,

  primary key (purchase_id),
  foreign key (customer_no) references customer,
  foreign key (salesrep_id) references salesrep
  foreign key (registration) references car
);

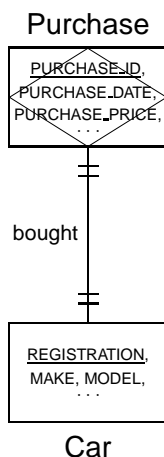
```

**Figure 5.8:** Translating a one-to-many relationship (mandatory-optional) to and from SQL/92 (rule S5)

key in the ‘one’ table, so there is nothing with which to express the optionality of the ‘many’ end. That is, the optionality of the ‘many’ end of the relationship is lost in the translation, and the rule is consequently partial in the forward direction. In the reverse direction, however, the rule is complete. That is:

$$\begin{aligned}
\mathfrak{R}_e[\text{MARTINRELATIONSHIP}, \text{ERTYPEITEM}_1, \text{ERTYPEITEM}_2] &\rightarrow \\
&\mathfrak{R}_r[\text{SQL92FOREIGNKEY}, \text{SQL92NOTNULL}] \\
&\hspace{10em} (\text{S5}) \\
\mathfrak{R}_e[\text{MARTINRELATIONSHIP}, \text{ERTYPEITEM}_1, \text{ERTYPEITEM}_2] &\leftarrow \\
&\mathfrak{R}_r[\text{SQL92FOREIGNKEY}, \text{SQL92NOTNULL}]
\end{aligned}$$

Now consider the bought one-to-one relationship between Purchase and Car. For this relationship, the corresponding foreign key could be placed in either table, or indeed, foreign keys could be placed in both tables. Placing foreign keys in both tables provides greater flexibility, and is the approach adopted here, as shown in Figure 5.9 on the next page. The relationship is mandatory at both ends, so both foreign keys have `not null` constraints on their columns. In addition, the cardinality of both ends of the relationship is ‘one’, which means that there is a one-to-one mapping between the values of the foreign keys and the values of the primary keys that they reference. The values of the primary keys are unique by definition, so the values of the foreign keys must also be unique. This is represented by `unique` constraints on the columns of both foreign keys in Figure 5.9.



```

create table purchase
( purchase_id char(6),
  purchase_date date,
  purchase_price integer,
  customer_no char(6) not null,
  salesrep_id char(7) not null,
  registration char(6) not null unique,

  primary key (purchase_id),
  foreign key (customer_no) references customer,
  foreign key (salesrep_id) references salesrep
  foreign key (registration) references car
);

create table car
( registration char(6),
  vin char(20) not null unique,
  :
  :
  purchase_id char(6) not null unique,
  sale_id char(6) unique,

  primary key (registration),
  foreign key (purchase_id) references purchase,
  foreign key (sale_id) references sale
);

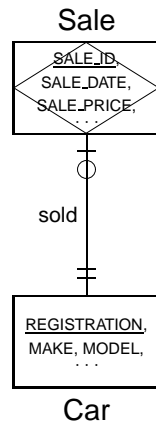
```

**Figure 5.9:** Translating a one-to-one relationship (mandatory-mandatory) to and from SQL/92 (rule S6)

The advantage of placing foreign keys in both tables is that it allows the optionality of both ends of the relationship to be translated, that is, there is no loss of optionality information as there was with rule S5. As a result, the rule is complete in both directions. That is:

$$\mathfrak{R}_e [\text{MARTINRELATIONSHIP}, \text{ERTYPEITEM}_1, \text{ERTYPEITEM}_2] \leftrightarrow \mathfrak{R}_r [\text{SQL92FOREIGNKEY}_1, \text{SQL92FOREIGNKEY}_2, \text{SQL92UNIQUE}_1, \text{SQL92UNIQUE}_2, \text{SQL92NOTNULL}_1, \text{SQL92NOTNULL}_2] \quad (\text{S6})$$

The sold relationship between Sale and Car is almost identical to that between Purchase and Car, except that the Sale end of the relationship is optional rather than mandatory. This means that there is no `not null` constraint on the foreign key in the `car` table, as shown in Figure 5.10. Foreign keys are still included in both tables, which could result in nulls being stored in the `car` foreign key. The advantage of retaining optionality information should outweigh any disadvantages of allowing nulls in the foreign key. As with rule S6, this rule is complete in both directions. That is:



```

create table sale
( sale_id char(6),
  sale_date date,
  sale_price integer,
  sale_no char(6) not null,
  salesrep_id char(7) not null,
  registration char(6) not null unique,

  primary key (purchase_id),
  foreign key (customer_no) references customer,
  foreign key (salesrep_id) references salesrep
  foreign key (registration) references car
);

create table car
( registration char(6),
  vin char(20) not null unique,
  : ,
  purchase_id char(6) not null unique,
  sale_id char(6) unique,

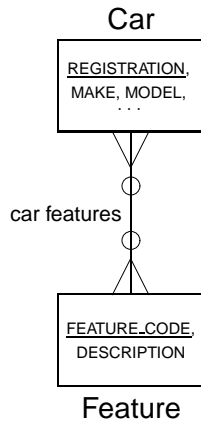
  primary key (registration),
  foreign key (purchase_id) references purchase,
  foreign key (sale_id) references sale
);

```

**Figure 5.10:** Translating a one-to-one relationship (optional-mandatory) to and from SQL/92 (rule S7)

$$\mathfrak{R}_e [\text{MARTINRELATIONSHIP}, \text{ERTYPEITEM}_1, \text{ERTYPEITEM}_2] \leftrightarrow \mathfrak{R}_r [\text{SQL92FOREIGNKEY}_1, \text{SQL92FOREIGNKEY}_2, \text{SQL92UNIQUE}_1, \text{SQL92UNIQUE}_2, \text{SQL92NOTNULL}] \quad (\text{S7})$$

It was stated in the introduction to this translation that the source ERD must be normalised, but that many-to-many relationships are allowed. This is not a contradiction, because  $\mathfrak{R}_e(E-R, ERD_{Martin})$  defines the links between entities using relationships only. That is, explicit attributes to ‘implement’ the relationship are not required, as they often are in CASE tools; rather, these attributes are implied by the relationship. Many-to-many relationships like car features between Car and Feature cannot be translated to SQL/92 in the same way as the relationships discussed above, however, as this would result in the generation of multi-valued (repeating) columns in the car and feature tables. Instead, an intermediate table must be generated to link the car and feature tables, as shown in Figure 5.11 on the following page. This table (car\_feature) contains only the attributes from the primary keys of car and feature, which are concatenated to form the primary key. In addition, the attributes from car form a foreign key to car and the attributes from feature form a foreign key to feature.



```

create table car
( registration char(6),
  vin char(20) not null unique,
  :
  purchase_id char(6) not null unique,
  sale_id char(6) unique,

  primary key (registration),
  foreign key (purchase_id) references purchase,
  foreign key (sale_id) references sale
);

create table car_feature
( registration char(6),
  feature_code char(6),

  primary key (registration, feature_code),
  foreign key (registration) references car,
  foreign key (feature_code) references feature
);

create table feature
( feature_code char(6),
  description char(80),

  primary key (feature_code)
);
  
```

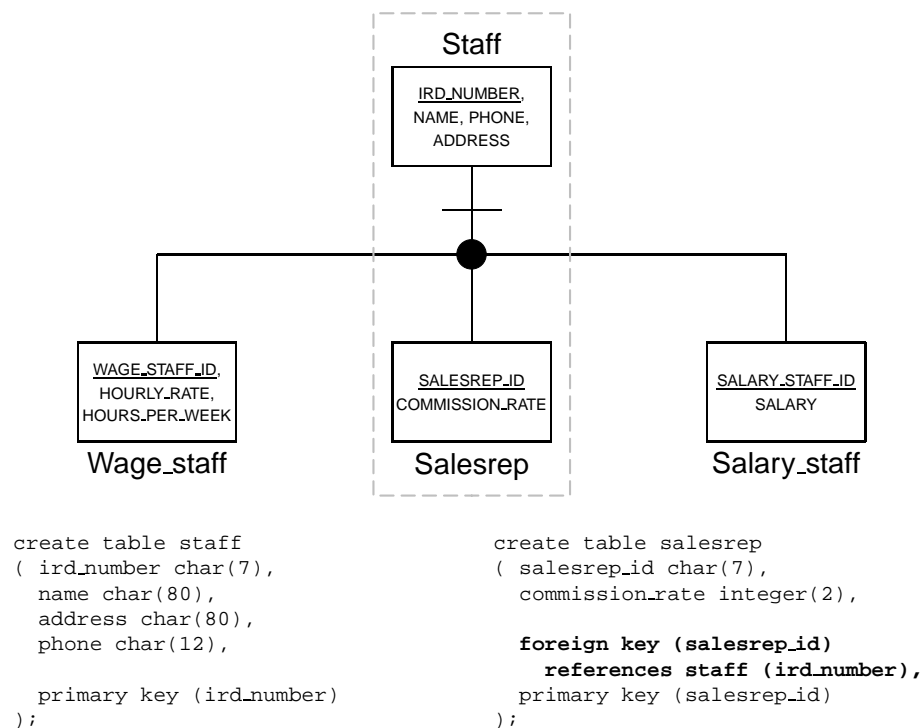
**Figure 5.11:** Translating a many-to-many relationship to and from SQL/92 (rule S8)

Rule S5 for one-to-many relationships loses the optionality of the ‘many’ end of the relationship. It follows from this that the translation of a many-to-many relationship loses the optionality of *both* ends of the relationship, as the many-to-many relationship effectively becomes two one-to-many relationships. The rule is therefore partial in the forward direction, but complete in the reverse direction. That is:

$$\begin{aligned}
 \mathfrak{R}_e[\text{MARTINRELATIONSHIP}, \text{ERTYPEITEM}_1, \text{ERTYPEITEM}_2] &\rightarrow \\
 &\mathfrak{R}_r[\text{SQL92TABLE}, \text{SQL92PRIMARYKEY}, \\
 &\quad \text{SQL92FOREIGNKEY}_1, \text{SQL92FOREIGNKEY}_2] \quad (\text{S8}) \\
 \mathfrak{R}_e[\text{MARTINRELATIONSHIP}, \text{ERTYPEITEM}_1, \text{ERTYPEITEM}_2] &\leftarrow \\
 &\mathfrak{R}_r[\text{SQL92TABLE}, \text{SQL92PRIMARYKEY}, \\
 &\quad \text{SQL92FOREIGNKEY}_1, \text{SQL92FOREIGNKEY}_2]
 \end{aligned}$$

The only remaining structure in description  $D_1$  that has not been considered is the type hierarchy associating the Staff entity (the supertype) and the entities Wage\_staff, Salesrep and Salary\_staff (the subtypes). This can be treated as if there were one-to-one relationships between supertype entity (mandatory) and each of the subtypes (optional). This is similar to rule S7, except that a foreign key is not placed in the `staff`

table. Rather, foreign keys are only placed in the tables corresponding to the subtypes, as shown in Figure 5.12 for the `salesrep` table.



**Figure 5.12:** Translating a type hierarchy to SQL/92 (rule S9)

In this example, the subtypes and the supertype share the same identifier, albeit with different names. This means that no `unique` or `not null` constraints are required on the foreign key attributes of `salesrep`, because these are also the primary key attributes, which are unique and not null by definition. In general, however, there is no guarantee that the supertype and the subtypes will share the same identifier, so the rule for this translation should therefore generate `unique` and `not null` constraints regardless. This may sometimes lead to redundant constraints in the resultant SQL code.

The only property of the `MARTINTYPEHIERARCHY` construct that cannot be translated to SQL is the exclusivity of the type hierarchy, that is, whether a given instance of an entity may fall into multiple subtypes or only one subtype. SQL has no concept of generalisation/specialisation, and hence no concept of mutually exclusive subtypes. Consequently, this rule is partial in the forward direction, but it is complete in the

reverse direction. That is:

$$\begin{aligned}
\mathfrak{R}_e[\text{MARTINTYPEHIERARCHY}, \text{MARTINREGULARENTITY}_p, \\
\text{MARTINREGULARENTITY}_c] \rightarrow \mathfrak{R}_r[\text{SQL92FOREIGNKEY}, \text{SQL92UNIQUE}, \\
\text{SQL92NOTNULL}] \tag{S9} \\
\mathfrak{R}_e[\text{MARTINTYPEHIERARCHY}, \text{MARTINREGULARENTITY}_p, \\
\text{MARTINREGULARENTITY}_c] \leftarrow \mathfrak{R}_r[\text{SQL92FOREIGNKEY}, \text{SQL92UNIQUE}, \\
\text{SQL92NOTNULL}]
\end{aligned}$$

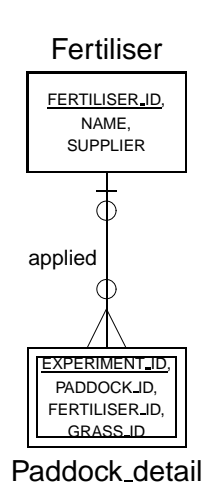
The rules that have been defined thus far are adequate to translate the structures appearing in descriptions  $D_1$  and  $D_2$  of the used cars viewpoint, but there are other structures that have not yet been considered. There are two additional E-R structures in description  $D_3(V_{agri}, E-R, ERD_{Martin})$  (see Figure 5.3 on page 112) that do not appear in description  $D_1$ . The first of these is the comprises one-to-many mandatory-mandatory relationship between the Contract and Experiment entities. Since the optionality of the ‘many’ end of a one-to-many relationship is lost when translated to SQL, this is effectively identical to the situation for rule S5 on page 119. That is, rule S5 will serve for both one-to-many mandatory-mandatory and mandatory-optional relationships.

The second additional structure to be considered in description  $D_3$  is the applied one-to-many optional-optional relationship between the Fertiliser and Paddock\_detail entities. As with rule S5, this translates to a foreign key in the `paddock_detail` table. The result is, however, slightly different: since the ‘one’ end of the relationship is optional, the columns of the foreign key are allowed to be null, so there is no `not null` constraint on them, as shown in Figure 5.13. Otherwise, this rule is identical to rule S5: it is partial in the forward direction and complete in the reverse direction. That is:

$$\begin{aligned}
\mathfrak{R}_e[\text{MARTINRELATIONSHIP}, \text{ERTYPEITEM}_1, \text{ERTYPEITEM}_2] \rightarrow \\
\mathfrak{R}_r[\text{SQL92FOREIGNKEY}] \tag{S10} \\
\mathfrak{R}_e[\text{MARTINRELATIONSHIP}, \text{ERTYPEITEM}_1, \text{ERTYPEITEM}_2] \leftarrow \\
\mathfrak{R}_r[\text{SQL92FOREIGNKEY}]
\end{aligned}$$

Only one additional structure is introduced by  $D_5(V_{marks}, E-R, ERD_{Martin})$ : weak entities that are dependent on some other ‘parent’ entity. One example of this is the relationship between the Assignment associative entity and the Mark adjustment weak





```

create table fertiliser
( fertiliser_id char(10),
  name char(20),
  supplier char(20),

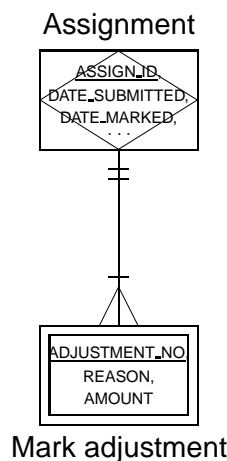
  primary key (fertiliser_id)
);

create table paddock_detail
( experiment_id char(10),
  paddock_id char(10) not null,
  fertiliser_id char(10),
  grass_id char(10) not null,

  primary key (experiment_id, paddock_id),
  foreign key (experiment_id) references experiment,
  foreign key (paddock_id) references paddock,
  foreign key (fertiliser_id) references fertiliser,
  foreign key (grass_id) references grass
);
  
```

**Figure 5.13:** Translating a one-to-many relationship (optional-optional) to and from SQL/92 (rule S10)

entity illustrated in Figure 5.14. The weak entity Mark adjustment translates directly to the table adjustment. Note, however, that a weak entity will typically not have a unique identifier of its own, although it may have a partial identifier, such as the attribute ADJUSTMENT\_NO in the entity Mark adjustment. The primary key of the adjustment table is thus formed by concatenating the identifying attributes of the Mark adjustment weak entity with the entity identifier of the entity it depends upon (Assignment), as illustrated in Figure 5.14. This may produce a non-minimal primary key if the weak entity does have a unique identifier.



```

create table assignment
( assign_id integer,
  element_id integer not null,
  : ,
  raw_mark smallint,
  comments char(500),

  primary key (assign_id),
  foreign key (element_id) references element,
  foreign key (student_id) references student,
  foreign key (staff_id) references staff
);

create table adjustment
( assign_id integer,
  adjustment_no integer,
  reason char(200),
  amount smallint,

  primary key (assign_id, adjustment_no),
  foreign key (assign_id) references assignment
);
  
```

**Figure 5.14:** Translating a weak entity and dependent relationship to SQL/92 (rule S11)

This rule is partial in the forward direction for similar reasons to rules S1 and S2. The reverse direction is more complex, however. Although the right-hand side of this rule is different from every other rule defined to this point (and is in fact is different from every rule in the translation), in practice it is impossible in the reverse direction to distinguish this rule from rule S1. Every SQL/92 table *must* have a unique primary key, regardless of the type of entity that the table corresponds to, which means that an SQL table could either be translated to a weak entity by this rule or to a regular entity by rule S1. This is a case 1 conflict as discussed in Section 4.7.1 on page 98. Consequently, this rule is unidirectional in the forward direction. That is:

$$\mathfrak{R}_e [\text{MARTINWEAKENTITY}, \text{ERENTITYTYPE}, \text{MARTINRELATIONSHIP}] \rightarrow \mathfrak{R}_r [\text{SQL92TABLE}, \text{SQL92PRIMARYKEY}] \quad (\text{S11})$$

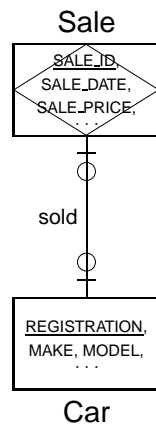
### 5.3.2 Additional scheme-level rules

In addition to the rules defined above, there is one rule that is not applicable to the descriptions in any of the three viewpoints. Consider a one-to-one relationship that is optional at both ends. This is similar to rule S7, except that there is no `not null` constraint in either of the generated SQL tables. A modification to the example for rule S7 is shown in Figure 5.15. This rule is complete in both directions. That is:

$$\mathfrak{R}_e [\text{MARTINRELATIONSHIP}, \text{ERTYPEITEM}_1, \text{ERTYPEITEM}_2] \leftrightarrow \mathfrak{R}_r [\text{SQL92FOREIGNKEY}_1, \text{SQL92FOREIGNKEY}_2, \text{SQL92UNIQUE}_1, \text{SQL92UNIQUE}_2] \quad (\text{S12})$$

### 5.3.3 Heuristics

The author has at present only identified one heuristic for this translation. The relational model has the notion of an *alternate key*, which is an alternative unique identifier for a relation. Although SQL/92 does not explicitly use this construct, alternate keys may be simulated in SQL/92 using `unique` constraints. It is possible to translate such a constraint to an E-R entity identifier, although this may not always produce a semantically correct result, as not all `unique` constraints may represent alternate keys. This



```

create table sale
( sale_id char(6),
  sale_date date,
  sale_price integer,
  sale_no char(6) not null,
  salesrep_id char(7) not null,
  registration char(6) unique,

  primary key (purchase_id),
  foreign key (customer_no) references customer,
  foreign key (salesrep_id) references salesrep
  foreign key (registration) references car
);

create table car
( registration char(6),
  vin char(20) not null unique,
  : ,
  purchase_id char(6) not null unique,
  sale_id char(6) unique,

  primary key (registration),
  foreign key (purchase_id) references purchase,
  foreign key (sale_id) references sale
);

```

**Figure 5.15:** Translating a one-to-one relationship (optional-optional) to and from SQL/92 (rule S12)

is to be expected for a heuristic, however. For example, the `vin` column of the `car` table has a `unique` constraint attached to it, so it can be translated to an entity identifier for the `Car` entity. This happens to be a correct result, as `vin` is in fact an alternate key for `car`.

The use of this heuristic can result in entities that have multiple entity identifiers. It is unclear from Chen’s (1977) definition of the E-R approach as to whether this is illegal — while not explicitly disallowed, the implication is that each entity has only a single entity identifier. The heuristic is complete. That is:

$$\mathcal{R}_e [\text{MARTINIDENTIFIER}] \leftrightarrow \mathcal{R}_r [\text{SQL92UNIQUE}] \quad (\text{H1})$$

### 5.3.4 Technique-level rules

The scheme-level rules defined above will now be examined to determine whether they may be generalised into technique-level rules, and two technique-level rules that are not specialised by this translation will also be defined. Note that a technique by definition has no notation associated with it, so no examples of these rules are given.

Of the rules defined above, S1–S4, S8, S10 and S11 can potentially be generalised, as they do not include any scheme-specific constructs. Rules S8 and S10 are not con-

sidered here, however, because the definition of the E-R technique allows for relationships with degree greater than two, whereas all relationships in  $\mathfrak{R}_e$  are binary only. The technique-level rules would thus have to deal with n-ary relationships in order to be of any use. Since no other E-R representations are currently being used by the author, technique-level rules for n-ary relationships have not been defined in order to reduce the complexity of the translation. It is expected that appropriate technique-level rules will be defined at a later date.

The MARTINREGULARENTITY construct generalises to the EREntityType construct and the SQL92TABLE construct generalises to the RMRELATION construct, so rule S1 may be generalised to:

$$\mathfrak{R}_e [\text{ERENTITYTYPE}] \rightleftharpoons \mathfrak{R}_r [\text{RMRELATION}] \quad (\text{T1})$$

Note that the EREntityType element in this rule should not be a weak entity type, as these are dealt with by rule T5 below.

Associative entities (construct MARTINASSOCIATIVEENTITY) are specialisations of the ERRELATIONSHIPType construct rather than the EREntityType construct. In effect, an associative entity is a relationship type that has attributes. Rule S2 may therefore be generalised to:

$$\mathfrak{R}_e [\text{ERRELATIONSHIPType}] \rightarrow \mathfrak{R}_r [\text{RMRELATION}] \quad (\text{T2})$$

The MARTINATTRIBUTE construct generalises to ERATTRIBUTE, and SQL92COLUMN generalises to RMATTRIBUTE, so rule S3 may be generalised to:

$$\mathfrak{R}_e [\text{ERATTRIBUTE}] \rightleftharpoons \mathfrak{R}_r [\text{RMATTRIBUTE}] \quad (\text{T3})$$

The MARTINIDENTIFIER construct generalises to ERIDENTIFIER, and SQL92PRIMARYKEY generalises to RMPRIMARYKEY, so rule S4 may be generalised to:

$$\begin{aligned} \mathfrak{R}_e [\text{ERIDENTIFIER}] &\rightarrow \mathfrak{R}_r [\text{RMPRIMARYKEY}] \\ \mathfrak{R}_e [\text{ERIDENTIFIER}] &\leftarrow \mathfrak{R}_r [\text{RMPRIMARYKEY}] \end{aligned} \quad (\text{T4})$$

The MARTINWEAKENTITY construct generalises to ERWEAKENTITYType, and the MARTINRELATIONSHIP construct generalises to ERRELATIONSHIPType, so rule S11 may be generalised to:

$$\begin{aligned} \mathfrak{R}_e [\text{ERWEAKENTITYType, EREntityType, ERRELATIONSHIPType}] &\rightarrow \\ &\mathfrak{R}_r [\text{RMRELATION, RMPRIMARYKEY}] \end{aligned} \quad (\text{T5})$$

An additional technique-level rule may be derived from heuristic H1, which was designed to translate alternate keys simulated by `unique` constraints. The relational technique explicitly includes the `RMALTERNATEKEY` construct, so it is possible to define a technique-level rule that performs the appropriate translation. This is not a generalisation of heuristic H1, however, because the heuristic refers to a construct (`SQL92-UNIQUE`) that does not exist in the technique. This rule is unidirectional and partial in the reverse direction, to avoid a case 1 conflict with rule T4 in the forward direction. That is:

$$\mathfrak{R}_e [\text{ERIDENTIFIER}] \dashv \mathfrak{R}_r [\text{RMALTERNATEKEY}] \quad (\text{T6})$$

Finally, an additional technique-level rule may be identified directly from the E-R and relational techniques themselves. The E-R technique includes the `ERVALUE-TYPE` construct, which is not specialised by  $\mathfrak{R}_e(E-R, ERD_{Martin})$ . The E-R concept of a value type corresponds directly to the relational concept of a domain (construct `RM-DOMAIN`). This rule is complete in both directions. That is:

$$\mathfrak{R}_e [\text{ERVALUE-TYPE}] \leftrightarrow \mathfrak{R}_r [\text{RMDOMAIN}] \quad (\text{T7})$$

### 5.3.5 Discussion

The rules and heuristics for this translation are summarised in Table 5.3 on the following page. The **Spec.** column of this table indicates which technique-level rule a scheme-level rule specialises, if any. Rules that refer to relationships use an abbreviated notation to signify cardinality and optionality. The notation 0:1–1:N indicates that the relationship is optional-one (0:1) to mandatory-many (1:N). An asterisk (“\*”) is used to denote unspecified optionality or cardinality, for example, the notation \*:M–1:1 denotes an unspecified-many (\*:M) to mandatory-one (1:1) relationship.

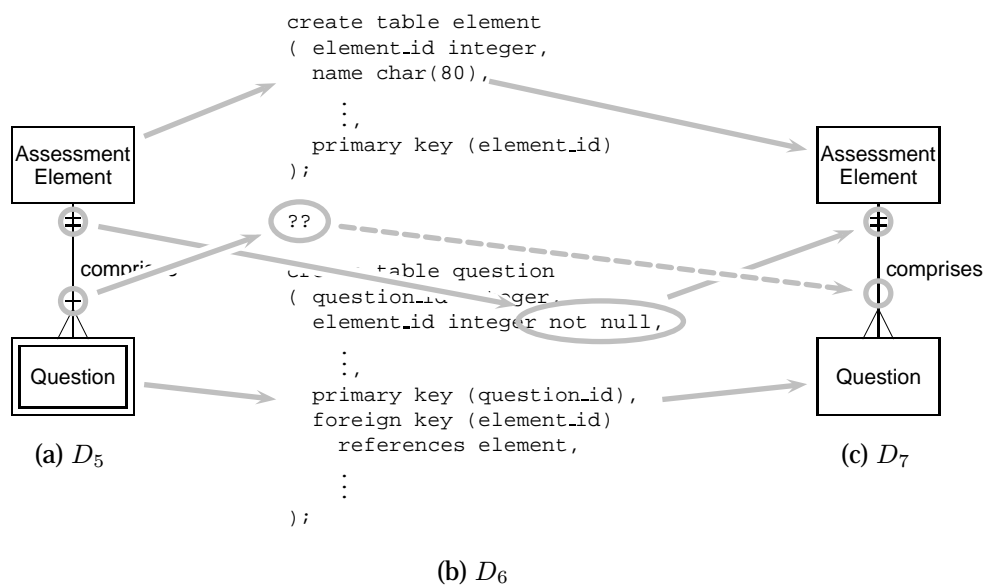
#### Information loss during the translation

As noted under the description of rule S5 on page 119, when a one-to-many relationship is translated to equivalent SQL/92 constructs, the optionality of the ‘many’ side of the relationship is lost because there is no construct in SQL/92 that can be used to express this information. With a many-to-many relationship, the optionality of *both* ends

**Table 5.3:** Summary of  $\mathfrak{R}_e(E-R, ERD_{Martin}) \rightleftharpoons \mathfrak{R}_r(Relational, SQL/92)$  rules

| Rule | Spec. | $\leftrightarrow$        | $\mathfrak{R}_e$ construct(s)   | $\mathfrak{R}_r$ construct(s)                      |
|------|-------|--------------------------|---|--|
| T1   | -     | $\rightleftharpoons$     | non-weak, normalised EREntityType   | RMRELATION   |
| T2   | -     | $\rightarrow$            | normalised ERRELATIONSHIPType with attributes   | RMRELATION   |
| T3   | -     | $\rightleftharpoons$     | non-repeating ERATTRIBUTE   | RMATTRIBUTE  |
| T4   | -     | $\rightarrow/\leftarrow$ | non-partial ERIDENTIFIER  | RMPRIMARYKEY                                       |
| T5   | -     | $\rightarrow$            | normalised ERWEAKENTITYType + normalised EREntityType + normalised ERRELATIONSHIPType | RMRELATION + RMPRIMARYKEY                          |
| T6   | -     | $\leftarrow$             | non-partial ERIDENTIFIER  | RMALTERNATEKEY                                     |
| T7   | -     | $\leftrightarrow$        | ERVALUETYPE   | RMDOMAIN   |
| S1   | T1    | $\rightleftharpoons$     | normalised MARTINREGULAREntity  | SQL92TABLE   |
| S2   | T2    | $\rightarrow$            | normalised MARTINASSOCIATIVEEntity  | SQL92TABLE   |
| S3   | T3    | $\rightleftharpoons$     | MARTINATTRIBUTE   | SQL92COLUMN  |
| S4   | T4    | $\rightarrow/\leftarrow$ | non-partial MARTINIDENTIFIER  | SQL92PRIMARYKEY                                    |
| S5   | -     | $\rightarrow/\leftarrow$ | 1:1-*:N MARTINRELATIONSHIP + 2 normalised ERTypeItem                                  | SQL92FOREIGNKEY + SQL92NOTNULL                     |
| S6   | -     | $\leftrightarrow$        | 1:1-1:1 MARTINRELATIONSHIP + 2 normalised ERTypeItem                                  | 2 SQL92FOREIGNKEY + 2 SQL92UNIQUE + 2 SQL92NOTNULL |
| S7   | -     | $\leftrightarrow$        | 0:1-1:1 MARTINRELATIONSHIP + 2 normalised ERTypeItem                                  | 2 SQL92FOREIGNKEY + 2 SQL92UNIQUE + SQL92NOTNULL   |
| S8   | -     | $\rightarrow/\leftarrow$ | *:M-*:N MARTINRELATIONSHIP + 2 normalised ERTypeItem                                  | SQL92TABLE + SQL92PRIMARYKEY + 2 SQL92FOREIGNKEY   |
| S9   | -     | $\rightarrow/\leftarrow$ | MARTINTYPEHIERARCHY + 2 MARTINREGULAREntity   | SQL92FOREIGNKEY + SQL92UNIQUE + SQL92NOTNULL       |
| S10  | -     | $\rightarrow/\leftarrow$ | 0:1-*:N MARTINRELATIONSHIP + 2 normalised ERTypeItem                                  | SQL92FOREIGNKEY                                    |
| S11  | T5    | $\rightarrow$            | normalised MARTINWEAKEntity + normalised EREntityType + 1:1-*:N MARTINRELATIONSHIP    | SQL92TABLE + SQL92PRIMARYKEY                       |
| S12  | -     | $\leftrightarrow$        | 0:1-0:1 MARTINRELATIONSHIP + 2 normalised ERTypeItem                                  | 2 SQL92FOREIGNKEY + 2 SQL92UNIQUE                  |
| H1   | -     | $\leftarrow$             | non-partial MARTINIDENTIFIER  | SQL92UNIQUE  |

of the relationship is lost. This loss of information is illustrated in Figure 5.16 for the assessment marks viewpoint; in this figure, description  $D_5(V_{marks}, E-R, ERD_{Martin})$  is being translated into description  $D_6(V_{marks}, Relational, SQL/92)$ , and then back to description  $D_7(V_{marks}, E-R, ERD_{Martin})$ . The example starts in Figure 5.16(a) with a one-to-many mandatory-mandatory relationship between the Assessment Element and Question entities. Note also that the Question entity is changed from a weak entity to a regular entity, due to the inability of SQL/92 to express this information.

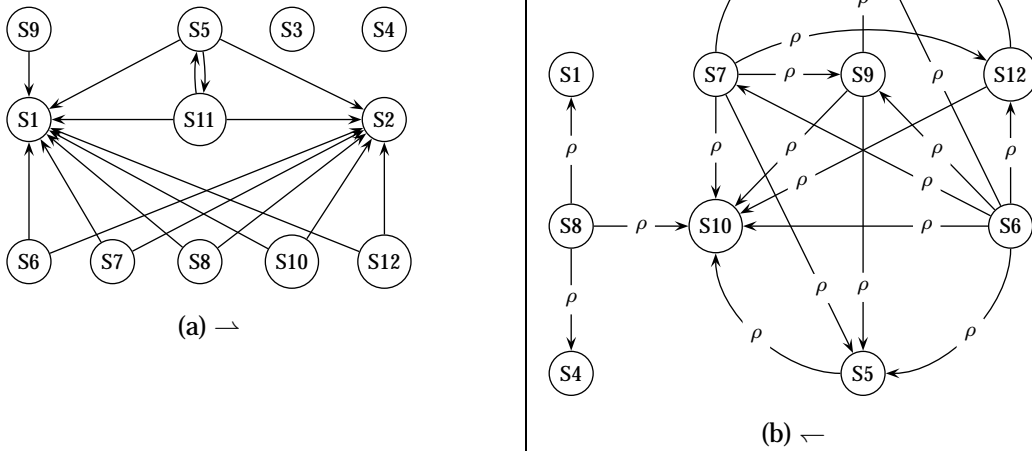


**Figure 5.16:** Loss of information when translating between  $\mathfrak{R}_e$  and  $\mathfrak{R}_r$  (assessment marks viewpoint)

### Unidirectional and excluded rules

The subsumption/exclusion graphs for this translation are shown in Figure 5.17 on the following page. Several rules in this translation have been defined as unidirectional, as they fall into the first case described in Section 4.7.1 on page 98. That is, rules that would produce syntactically incorrect results if they were applied together. For example, rules S2 and S11 have been defined as unidirectional in the forward direction to prevent conflicts with rule S1 in the reverse direction. Translating a table to a regular entity is the most sensible default, which is why rule S1 was chosen to be bidirectional.

There are also a large number of exclusions in the reverse direction, as shown in Figure 5.17(b), for rules that fall into the second case described in Section 4.7.1 on page 98.



**Figure 5.17:** Subsumption/exclusion graphs for the  $\mathfrak{R}_e \Rightarrow \mathfrak{R}_r$  translation

That is, rules that would generate semantically incorrect descriptions if they were applied together. It is interesting to note that there are no exclusions in the forward direction, as the rules are relatively independent of each other in this direction. The large number of exclusions in the reverse direction is presumably a direct consequence of the large collection of rules that deal with translating foreign keys.

## Observations

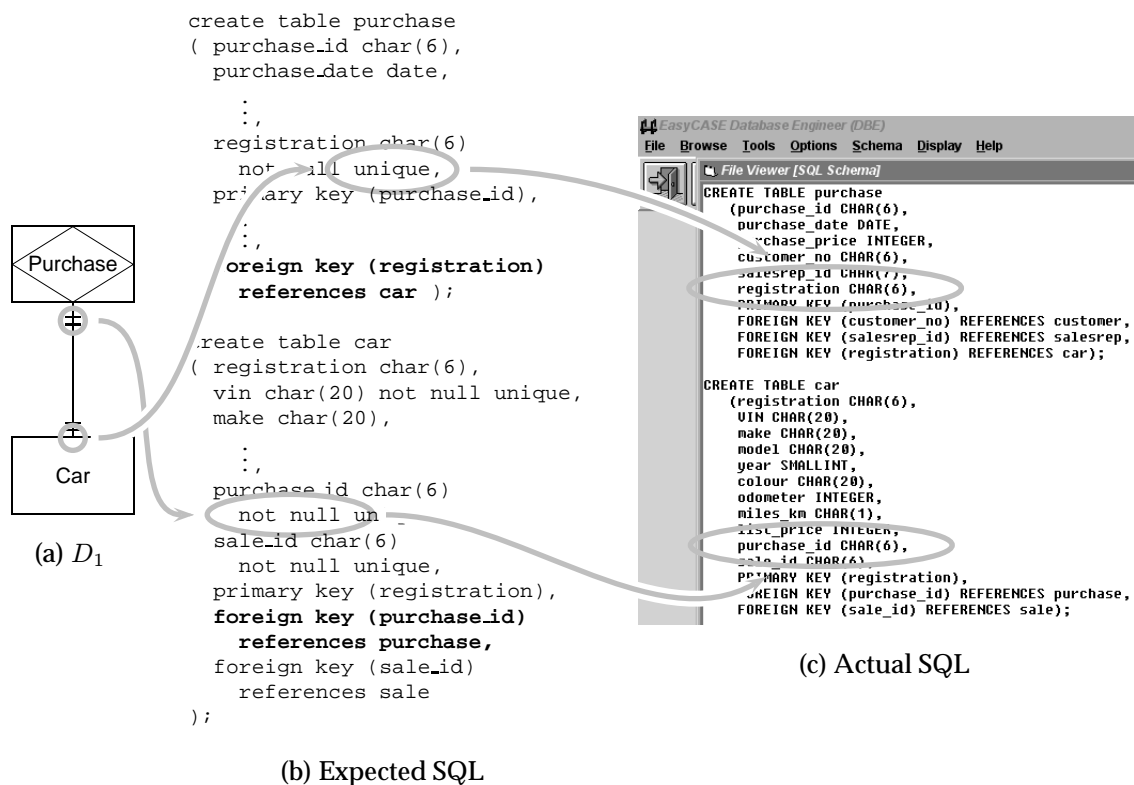
Two interesting points can be identified from the rules for this translation:

1. A one-to-anything relationship between two entities implies a foreign key in the table corresponding to the ‘anything’ entity. If the ‘one’ end of the relationship is optional, this foreign key is allowed to be null; conversely, if the ‘one’ end of the relationship is mandatory, then the foreign key must not be null. This is implemented in SQL/92 by placing a `not null` constraint on the foreign key columns. This is an important aspect of referential integrity that can easily be overlooked and it is interesting to note that none of the CASE tools surveyed in Chapter 9 appear to implement this.
2. If the relationship described in point 1 is one-to-one, then the foreign key should also be unique. This is expressed in SQL/92 by placing a `unique` constraint



on the foreign key columns. Again, none of the surveyed CASE tools appear to implement this.

As shown in Figure 5.18, the CASE tool EasyCASE does not deal with either of these points when SQL is generated from the ERD for the used cars viewpoint. The original ERD structure (from  $D_1$ ) is shown in Figure 5.18(a). The expected SQL/92 code from applying the rules defined above is shown in Figure 5.18(b), and the corresponding SQL/92 code generated by EasyCASE is shown in Figure 5.18(c).



**Figure 5.18:** SQL/92 constraints not generated by EasyCASE (used cars viewpoint)

It was stated in the introduction to this translation (see page 114) that the forward translation assumes the input ERD is normalised (that is, no repeating groups or multi-valued attributes). The main reason for this assumption is to reduce the complexity of the translation. SQL/92 assumes a normalised structure, so the translation of an unnormalised entity will obviously require some kind of normalisation process. The definition of  $\mathfrak{R}_e(E-R, ERD_{Martin})$  in Appendix D allows for the possibility of unnormalised entities, but in order to translate such structures additional rules would be required.

All of these new rules would be unidirectional, as there is no way to determine whether a table corresponds to an entity or a repeating group within an entity.

In addition, the E-R technique as defined in Appendix D does not explicitly include foreign keys; rather it is assumed that foreign key links are implied by the relationships between entities. This assumption fails in the face of unnormalised ERDs, however, as there is no way to determine whether a given relationship is associated with the entity as a whole, or with a repeating group embedded within that entity (as occurs the agricultural viewpoint). This is remedied here by allowing weak entities to be embedded within other entities; relationships may then be attached to these entities as appropriate. The exclusion of explicit foreign keys from the E-R technique definition means that there is no need for foreign key attributes in entities, so many-to-many relationships do not result in multi-valued attributes. That is, the ERD effectively remains normalised, even though many-to-many relationships are allowed. This also means that it is possible to ‘reconstruct’ many-to-many relationships when translating in the reverse direction, as illustrated by rule S8.

Another reason for restricting this translation to ‘normalised’ ERDs is that normalisation also takes place during the translation of an ERD to an FDD. Assuming that both the translations  $\mathfrak{R}_e(E-R, ERD_{Martin}) \rightleftharpoons \mathfrak{R}_r(Relational, SQL/92)$  and  $\mathfrak{R}_f(FuncDep, FDD_{Smith}) \rightarrow \mathfrak{R}_e(E-R, ERD_{Martin})/\mathfrak{R}_f \leftarrow \mathfrak{R}_e$  are available, including normalisation in the  $\mathfrak{R}_e \rightleftharpoons \mathfrak{R}_r$  translation is an unnecessary duplication of effort, as an ERD can be normalised by a composite translation ‘through’  $\mathfrak{R}_f$ . The resultant normalised ERD can then be translated into SQL if desired. The normalisation step could produce a new ERD, or it could modify the original ERD. It is also likely that the normalisation performed by the  $\mathfrak{R}_f \rightarrow \mathfrak{R}_e/\mathfrak{R}_f \leftarrow \mathfrak{R}_e$  translation will be more effective than that performed by the  $\mathfrak{R}_e \rightleftharpoons \mathfrak{R}_r$  translation, as the former explicitly manipulates functional dependencies, which form the theoretical basis for normalisation, whereas the latter does not. Leveraging existing translations in this way can lead to the following benefits:

- unnecessary duplication of effort is reduced;
- translation size and complexity can be reduced by not including unnecessary functionality; and

- the quality of individual translations may be improved, partly because of the reduction of duplication, and partly because of the use of more specialised translations.

## 5.4 $\mathfrak{R}_f \rightarrow \mathfrak{R}_e/\mathfrak{R}_f \leftarrow \mathfrak{R}_e$

In this translation, descriptions are translated between functional dependencies expressed using a functional dependency diagram in Smith notation and the entity-relationship approach expressed using Martin notation. This translation is complete in the forward direction and partial in the reverse direction. That is:

$$\mathfrak{R}_f(\text{FuncDep}, \text{FDD}_{\text{Smith}}) \rightarrow \mathfrak{R}_e(\text{E-R}, \text{ERD}_{\text{Martin}}) \text{ and}$$

$$\mathfrak{R}_f(\text{FuncDep}, \text{FDD}_{\text{Smith}}) \leftarrow \mathfrak{R}_e(\text{E-R}, \text{ERD}_{\text{Martin}}).$$

In the forward direction, the input to this translation is a Smith functional dependency diagram with no dependency chains, that is, each bubble has only a single dependency attached to it (see Appendix B). The output of the translation is a normalised ERD in at least fourth normal form. There are no restrictions in the reverse direction.

The rules for this translation are not presented here to conserve space — they may be found in Appendix E, and are summarised in Table 5.4 on the next page.

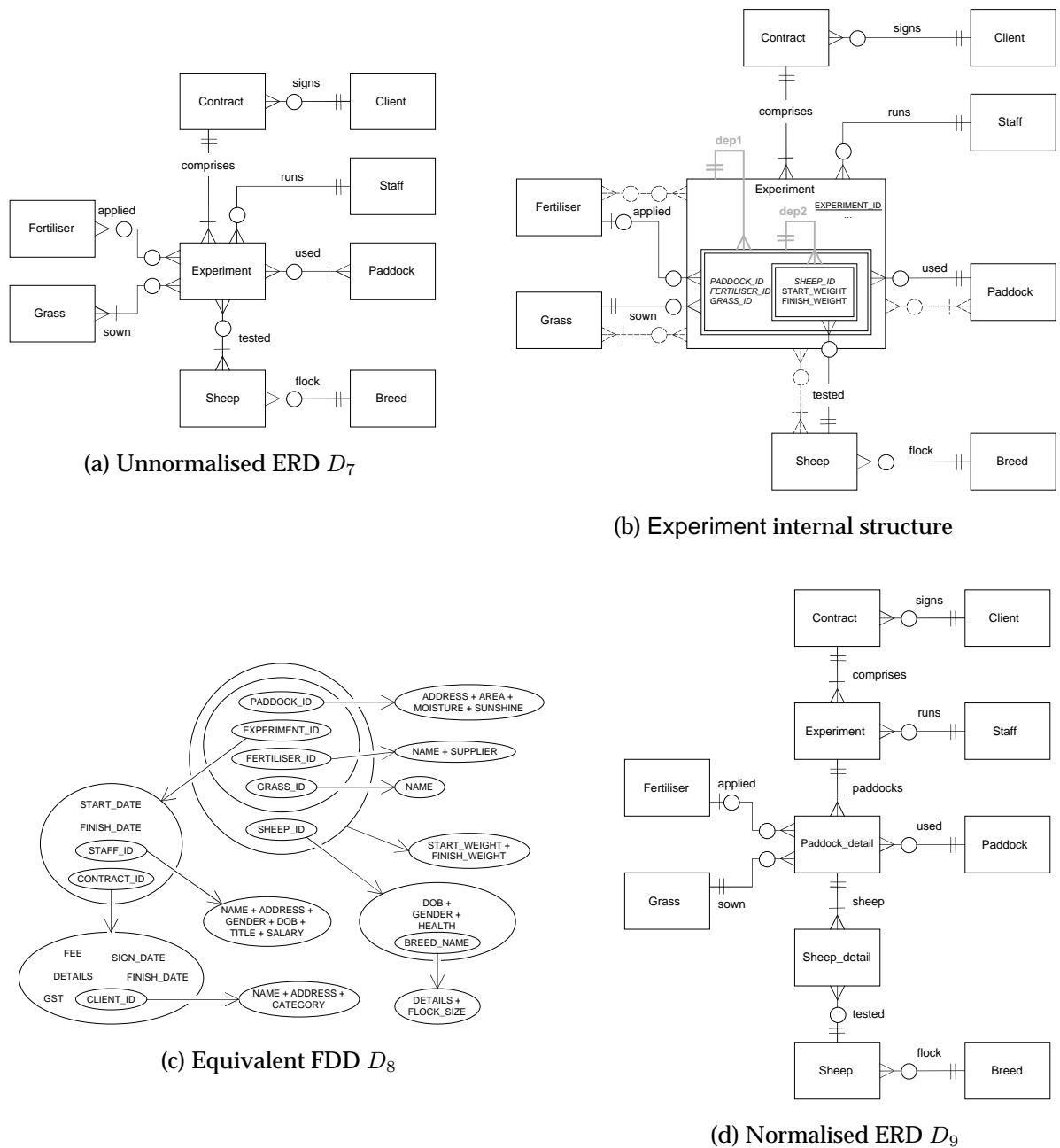
### 5.4.1 Normalisation effect

If an ERD is translated to an FDD, and the resulting FDD is translated back to an ERD, the original ERD is effectively normalised, as shown in Figure 5.19 on page 137 for the agricultural viewpoint. In this example, the unnormalised description  $D_7(V_{\text{agri}}, \text{E-R}, \text{ERD}_{\text{Martin}})$  of the agricultural viewpoint is translated to description  $D_8(V_{\text{agri}}, \text{FuncDep}, \text{FDD}_{\text{Smith}})$ , and back to the description  $D_9(V_{\text{agri}}, \text{E-R}, \text{ERD}_{\text{Martin}})$ .

The Experiment entity has a complex internal structure comprising two nested repeating groups, as shown in Figure 5.19(b) (the dashed lines indicate the unnormalised relationships). This internal structure is translated into an appropriate FDD ( $D_8$ ), shown in Figure 5.19(c). When this FDD is translated back to an ERD ( $D_9$ ), the result is the normalised ERD shown in Figure 5.19(d). The resulting ERD will be in at least

**Table 5.4: Summary of  $\mathfrak{R}_f \rightarrow \mathfrak{R}_e / \mathfrak{R}_f \leftarrow \mathfrak{R}_e$  rules**

| Rule | Spec. | $\leftrightarrow$        | $\mathfrak{R}_f$ construct(s)   | $\mathfrak{R}_e$ construct(s)   |
|------|-------|--------------------------|---|---|
| T1   | -     | $\rightarrow/\leftarrow$ | FDFUNCTIONALSOURCE + FDFUNCTIONAL + FDFUNCTIONALTARGET  | ERENTITYTYPE (not all key)  |
| T2   | -     | $\rightarrow/\leftarrow$ | FDMULTISOURCE + FDMULTIVALUED + FDMULTITARGET   | ERENTITYTYPE (all key, 2 attributes) + non-partial ERIDENTIFIER   |
| T3   | -     | $\rightarrow/\leftarrow$ | FDATEATTRIBUTE  | ERATTRIBUTE   |
| T4   | -     | $\rightarrow/\leftarrow$ | 2 FDFUNCTIONALSOURCE + 2 FDFUNCTIONAL + 2 FDFUNCTIONALTARGET  | 2 ERENTITYTYPE + ERRELATIONSHIPTYPE   |
| T5   | -     | $\leftarrow$             | 2 FDFUNCTIONALSOURCE + 2 FDFUNCTIONAL + 2 FDFUNCTIONALTARGET 2 FDMULTISOURCE + 2 FDMULTIVALUED + 2 FDMULTITARGET    | 2 ERENTITYTYPE + many-to-many ERRELATIONSHIPTYPE  |
| T6   | -     | $\rightarrow/\leftarrow$ | FDFUNCTIONALSOURCE  | non-partial ERIDENTIFIER  |
| T7   | -     | $\leftarrow$             | FDFUNCTIONALSOURCE + FDFUNCTIONAL + FDFUNCTIONALTARGET  | ERWEAKENTITYTYPE + dependent ERRELATIONSHIPTYPE + ERENTITYTYPE  |
| S1   | T1    | $\rightarrow/\leftarrow$ | SSSINGLEKEYBUBBLE + SSTARGETBUBBLE + SSSINGLEVALUED   | MARTINREGULARENTITY (not all key)   |
| S2   | T2    | $\rightarrow/\leftarrow$ | SSMULTIKEYBUBBLE + SSENDKEYBUBBLE + SSMULTIVALUED   | MARTINREGULARENTITY (all key, 2 attributes) + non-partial MARTINIDENTIFIER  |
| S3   | T6    | $\rightarrow/\leftarrow$ | SSSINGLEKEYBUBBLE   | non-partial MARTINIDENTIFIER  |
| S4   | -     | $\rightarrow/\leftarrow$ | SSISOLATEDBUBBLE  | MARTINREGULARENTITY (all key, $\neq$ 2 attributes)  |
| S5   | T3    | $\rightarrow/\leftarrow$ | SSATTRIBUTE   | MARTINATTRIBUTE   |
| S6   | -     | $\leftarrow$             | SSSINGLEKEYBUBBLE + SSTARGETBUBBLE + SSSINGLEVALUED   | MARTINASSOCIATIVEENTITY   |
| S7   | T7    | $\leftarrow$             | SSSINGLEKEYBUBBLE + SSTARGETBUBBLE + SSSINGLEVALUED   | MARTINWEAKENTITY (attributes, partial key) + 1:1-*:N dependent MARTINRELATIONSHIP + ERTYPEITEM                    |
| S8   | T7    | $\leftarrow$             | SSSINGLEKEYBUBBLE + SSTARGETBUBBLE + SSSINGLEVALUED   | MARTINWEAKENTITY (attributes, partial key, $n$ relationships) + 1:1-*:N dependent MARTINRELATIONSHIP + ERTYPEITEM |
| S9   | T7    | $\leftarrow$             | SSSINGLEKEYBUBBLE + SSTARGETBUBBLE + SSSINGLEVALUED   | MARTINWEAKENTITY (attributes, $n$ relationships) + 1:1-*:N dependent MARTINRELATIONSHIP + ERTYPEITEM              |
| S10  | T7    | $\leftarrow$             | SSISOLATEDBUBBLE  | MARTINWEAKENTITY (no attributes, $n$ relationships) + 1:1-*:N dependent MARTINRELATIONSHIP + ERTYPEITEM           |
| S11  | T4    | $\rightarrow/\leftarrow$ | 2 SSSINGLEKEYBUBBLE + 2 SSTARGETBUBBLE + 2 SSSINGLEVALUED (configuration 1)   | 2 non-weak ERTYPEITEM + *:1-*:1 MARTINRELATIONSHIP  |
| S12  | T4    | $\rightarrow/\leftarrow$ | 2 SSSINGLEKEYBUBBLE + 2 SSTARGETBUBBLE + 2 SSSINGLEVALUED (configuration 2)   | 2 non-weak ERTYPEITEM + *:1-*:N MARTINRELATIONSHIP  |
| S13  | T5    | $\leftarrow$             | 2 SSSINGLEKEYBUBBLE + 2 SSSINGLEVALUED + 2 SSTARGETBUBBLE + 2 SSMULTIKEYBUBBLE + 2 SSMULTIVALUED + 2 SSENDKEYBUBBLE | 2 non-weak ERTYPEITEM + *:M-*:N MARTINRELATIONSHIP  |
| S14  | -     | $\rightarrow$            | non-isolated FDATEATTRIBUTESET + SSDOMAINFLAG + SSSINGLEKEYBUBBLE + 2 SSATTRIBUTE                                   | *:1-*:N MARTINRELATIONSHIP  |
| S15  | -     | $\rightarrow$            | SSISOLATEDBUBBLE + SSDOMAINFLAG + SSSINGLEKEYBUBBLE + 2 SSATTRIBUTE   | *:1-*:N MARTINRELATIONSHIP  |
| S16  | -     | $\rightarrow$            | non-isolated FDATEATTRIBUTESET contains SSSINGLEKEYBUBBLE   | 2 ERTYPEITEM + *:1-*:N MARTINRELATIONSHIP   |
| S17  | -     | $\rightarrow$            | non-isolated FDATEATTRIBUTESET contains SSISOLATEDBUBBLE  | 2 ERTYPEITEM + *:1-*:N MARTINRELATIONSHIP   |
| S18  | -     | $\rightarrow$            | SSISOLATEDBUBBLE contains SSSINGLEKEYBUBBLE   | 2 ERTYPEITEM + *:1-*:N MARTINRELATIONSHIP   |
| S19  | -     | $\rightarrow$            | SSISOLATEDBUBBLE contains SSISOLATEDBUBBLE  | 2 ERTYPEITEM + *:1-*:N MARTINRELATIONSHIP   |
| H1   | -     | $\rightarrow$            | 2 SSMULTIKEYBUBBLE + 2 SSMULTIVALUED + 2 SSENDKEYBUBBLE   | MARTINASSOCIATIVEENTITY + MARTINIDENTIFIER  |
| H2   | -     | $\rightarrow$            | $n$ FDATEATTRIBUTESET (single-key or isolated) + SSDOMAINFLAG + SSSINGLEKEYBUBBLE + $n$ SSATTRIBUTE ( $n > 2$ )     | MARTINTYPEHIERARCHY   |



**Figure 5.19:** Normalisation caused by the  $\mathfrak{R}_f \rightarrow \mathfrak{R}_e/\mathfrak{R}_f \leftarrow \mathfrak{R}_e$  translation (agricultural viewpoint)

fourth normal form, as the rules for this translation (see Section E.1.2 on page 398) are based on Smith's method (Smith, 1985), which claims to produce fully normalised relations from a set of functional dependencies. Higher normal forms may be possible, depending on the structure of the FDD.

### 5.4.2 Partial rules

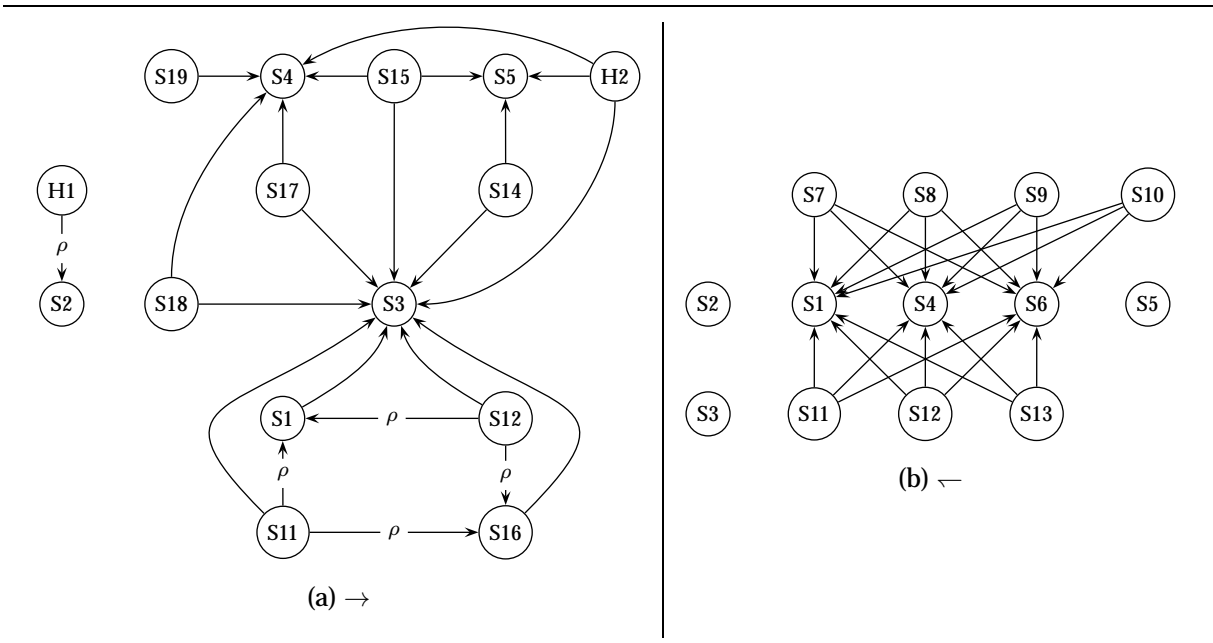
Rules S1, S2 and S6 are all partial in the reverse direction because the name of the entity cannot be translated. In addition, rule S2 may only be applied in the reverse direction when the number of attributes in the regular entity is two. If there are more than two attributes, it is impossible to determine how they should be split to form a multi-key bubble and an end-key bubble (although this could perhaps be implemented by a heuristic).

Rule S3 is partial in the reverse direction because the entity identified by an entity identifier element is not translated. Rule S4 is partial in the reverse direction because the relationships associated with the regular entity are not translated. Rule S5 is partial in the reverse direction because the data type information of an E-R attribute cannot be expressed in  $\mathfrak{R}_f$ . Rules S7–S10 are all partial because the name of the weak entity cannot be translated. Finally, rules S11–S13 are partial in the reverse direction as the entity names and optionalities of the relationship are not translated.

### 5.4.3 Unidirectional and excluded rules

The subsumption/exclusion graphs for both directions of this translation are shown in Figure 5.20. Rule S6 is unidirectional because it is at best very difficult to determine whether a given FDD construct translates to an associative entity. Rules S7–S10 could possibly be made bidirectional, but this would remove the normalisation effect, which is one of the more interesting features of this translation. The rules would also have to exclude each other in the forward direction. Rule S13 is unidirectional because the reverse translation is already dealt with by a combination of other rules (case 2 conflict). There are only five exclusions in the forward direction, and none in the reverse direction, as shown in Figure 5.20.

While not an exclusion per se, the implementations of rules S11–S13 should ignore



**Figure 5.20:** Subsumption/exclusion graphs for the  $\mathfrak{R}_f \rightarrow \mathfrak{R}_e / \mathfrak{R}_f \leftarrow \mathfrak{R}_e$  translation

relationships that are terminated by weak entities because these are already dealt with by rules S7–S10. This restriction prevents the generation of redundant structures, and can be dealt with by defining appropriate invariants.

#### 5.4.4 Observations

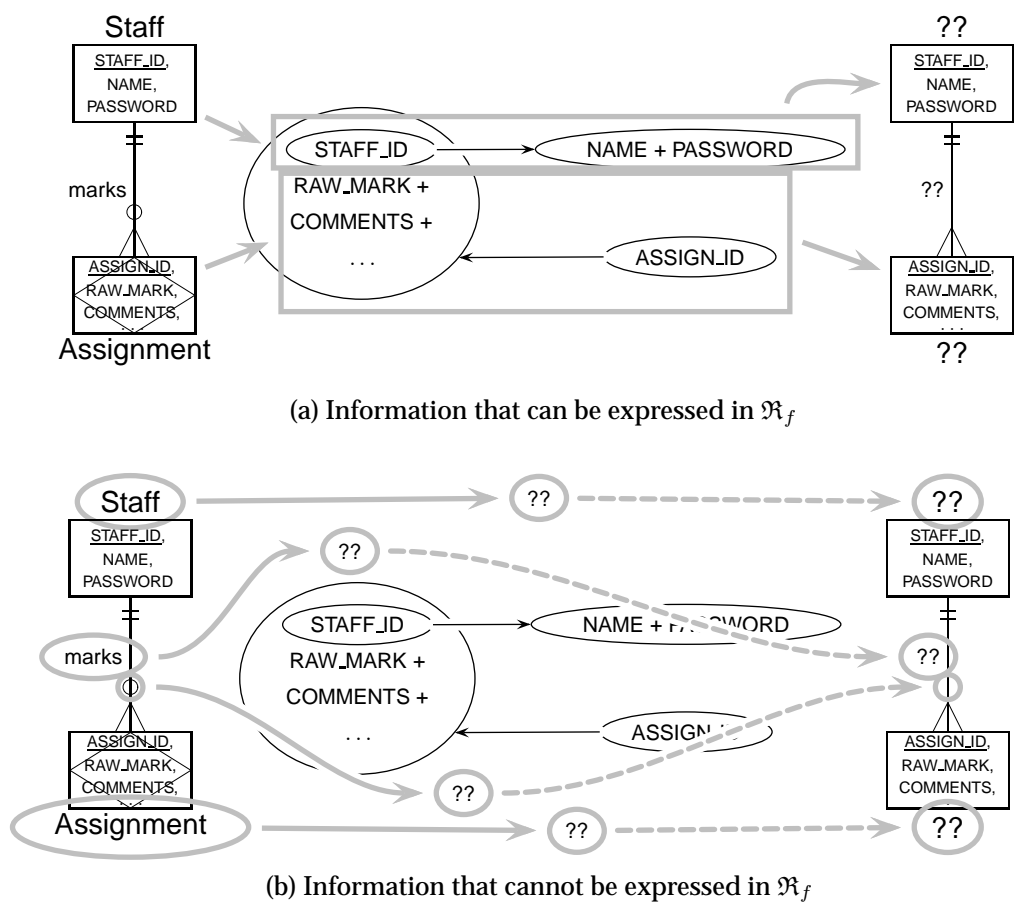
Although this translation is complete in the forward direction, this only implies that all information in  $\mathfrak{R}_f$  may be mapped to  $\mathfrak{R}_e$ . There is still some information useful to  $\mathfrak{R}_e$  that cannot be derived from  $\mathfrak{R}_f$ , for example:

- FDDs do not have any concept of entities, so there is little from which to derive meaningful entity names.
- It is impossible to accurately determine the optionality of a derived relationship.
- In general, the uniqueness of individual attributes within a bubble cannot be determined.
- The optionality of relationships is generally ambiguous. There will be no many-to-many relationships in the derived ERD because it is in at least fourth normal form as a result of the translation. One-to-many relationships may be inferred

from rule S16. Heuristic H2 may imply a one-to-one relationship, but this is not guaranteed (it is more likely if there is only one referencing domain flag).

- Type hierarchies are difficult to derive. If there are three or more domain flags referencing a single attribute (heuristic H2), then it is likely that this represents a type hierarchy, but there is no guaranteed way to determine this.
- There is no way of determining whether a derived type hierarchy is mutually exclusive or not, as functional dependencies cannot express this information.

Some of the information that cannot be expressed by  $\mathfrak{R}_f$  is shown in Figure 5.21 for the assessment marks viewpoint. In this example, a description has been translated from  $\mathfrak{R}_e$  to  $\mathfrak{R}_f$ , then back to  $\mathfrak{R}_e$ . Note that the Assignment associative entity becomes a regular entity when the FDD is translated back into an ERD.



**Figure 5.21:** Loss of information when translating from  $\mathfrak{R}_e$  to  $\mathfrak{R}_f$  (assessment marks viewpoint)



## 5.5 $\mathfrak{R}_e \rightleftharpoons \mathfrak{R}_d$

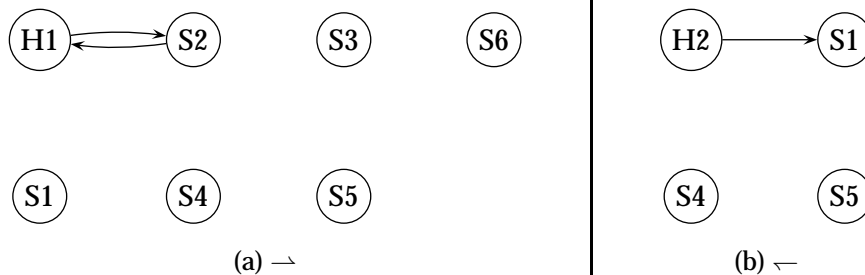
In this translation, descriptions are translated between an entity-relationship diagram expressed in Martin notation and a data flow diagram expressed in Gane and Sarson notation (Gane and Sarson, 1979). This translation is partial in both directions. That is:

$$\mathfrak{R}_e(E-R, ERD_{Martin}) \rightleftharpoons \mathfrak{R}_d(DataFlow, DFD_{G\&S}).$$

There are no restrictions on this translation in either direction. The rules and heuristics for this translation are not presented here to conserve space, and are summarised in Table 5.5 (full details of the rules may be found in Appendix E). The subsumption/exclusion graphs for both directions of the translation are shown in Figure 5.22.

**Table 5.5:** Summary of  $\mathfrak{R}_e \rightleftharpoons \mathfrak{R}_d$  rules

| Rule | Spec. | $\leftrightarrow$    | $\mathfrak{R}_e$ construct(s)                 | $\mathfrak{R}_d$ construct(s)                      |
|------|-------|----------------------|---|--|
| T1   | -     | $\rightleftharpoons$ | non-weak EENTITYTYPE                          | DFDATASTORE  |
| T2   | -     | $\rightleftharpoons$ | ERATTRIBUTEITEM                               | DFFIELDITEM  |
| S1   | T1    | $\Rightarrow$        | MARTINREGULARENTITY                           | GNSDATASTORE                                       |
| S2   | -     | $\rightarrow$        | MARTINASSOCIATIVEENTITY                       | GNSDATASTORE                                       |
| S3   | T1    | $\rightarrow$        | non-embedded<br>MARTINWEAKENTITY              | GNSDATASTORE                                       |
| S4   | T2    | $\Rightarrow$        | MARTINATTRIBUTE                               | GNSFIELD   |
| S5   | T2    | $\Rightarrow$        | MARTINATTRIBUTEGROUP                          | GNSFIELDGROUP                                      |
| S6   | -     | $\rightarrow$        | embedded<br>MARTINWEAKENTITY                  | GNSFIELDGROUP                                      |
| H1   | -     | $\nrightarrow$       | MARTINASSOCIATIVEENTITY                       | GNSDATAPROCESS + GNSDATASTORE +<br>GNSDATAFLOW     |
| H2   | -     | $\nrightarrow$       | MARTINRELATIONSHIP +<br>2 MARTINREGULARENTITY | GNSDATAPROCESS + 2 GNSDATASTORE +<br>2 GNSDATAFLOW |



**Figure 5.22:** Subsumption/exclusion graphs for the  $\mathfrak{R}_e \rightleftharpoons \mathfrak{R}_d$  translation

All rules in this translation are partial, due to the relatively low expressive overlap between the two representations. Rules S2 and S3 are unidirectional because it is

impossible to determine what type of entity a data store corresponds to. There are no excluded rules in either direction.

### 5.5.1 Observations

Rule S1 translates a regular entity to a data store (construct GNSDATASTORE). It could be argued that a regular entity may also map to an external entity (construct GNS-EXTERNALENTITY), but this is unlikely because external entities refer to things that are by definition ‘outside’ the description (Gane and Sarson, 1979; Evergreen Software Tools, 1995b). Similarly, resource stores are not considered, as they represent the storage of physical objects rather than data (Evergreen Software Tools, 1995b).

Data flows cannot be derived from an ERD, except in a somewhat limited fashion by heuristic H1, but once data flows have been defined manually (which could be achieved through enrichment), it may be possible to automatically determine the contents of data flows by comparing the contents of both the source and target elements for common fields. This would effectively be a form of post-translation enrichment. In the reverse direction, it is difficult to derive relationships from a DFD, but heuristic H2 can often produce useful relationships from collections of data stores and processes linked by data flows.

Resource flows cannot be derived from an ERD because resource flows model the movement of actual physical items as opposed to data, and are therefore outside the scope of an ERD. Similarly, multiple data processes cannot be derived from an ERD, as ERDs have little or no expression of process, let alone process parallelism. Some ordinary data processes may be derived using heuristic H1.

In summary, only a relatively small amount of information may be translated from a DFD to an ERD, in particular data stores. Data flows cannot normally be translated, as ERDs cannot express the flow of data, and similarly for processes. Some of this information can be made use of by heuristics, however.

## 5.6 Summary

In this chapter, the following translations were discussed:

- $\mathfrak{R}_e(E-R, ERD_{Martin}) \rightleftharpoons \mathfrak{R}_r(Relational, SQL/92)$ ,

- $\mathfrak{R}_f(\text{FuncDep}, \text{FDD}_{\text{Smith}}) \rightarrow \mathfrak{R}_e(\text{E-R}, \text{ERD}_{\text{Martin}}) / \mathfrak{R}_f \leftarrow \mathfrak{R}_e$  and
- $\mathfrak{R}_e(\text{E-R}, \text{ERD}_{\text{Martin}}) \rightleftharpoons \mathfrak{R}_d(\text{DataFlow}, \text{DFD}_{\text{G\&S}})$ .

Rules and heuristics were defined for the first translation; those for the remaining two translations are defined in Appendix E. Issues arising from the rules and heuristics, and restrictions on the translations were identified and discussed. From the rules defined, it appears that the first two translations will be of generally higher quality than the third, which will be verified in Chapter 8.

The rules in this chapter were expressed using the abstract translation notation developed in Chapters 3 and 4, but this notation is only useful (and intended) for discussing rules at a high level — it cannot be used to specify the implementation details of the rules. To implement a translation, these rules must be converted either manually or automatically into some form of program that can be executed.

In the next chapter is described the implementation of a prototype environment called *Swift*. *Swift* implements some of the translations defined in this thesis and has been used to test the translation-based approach to facilitating the use of multiple representations.



# Chapter 6

## Prototype implementation

### 6.1 Introduction

As part of the experimental work of this research, a prototype modelling environment that facilitates the use of multiple representations, known as *Swift*, was implemented by the author in order to test various aspects of the approach taken. In this chapter, the architecture of Swift and the issues arising from its implementation are discussed. Some of the work presented in this chapter has also been published by Stanger and Pascoe (1997b; 1997c).

The implementation architecture of Swift, which comprises three logical units, is discussed in general in Section 6.2. The description modelling unit, described in Section 6.3, provides the functionality for displaying and manipulating viewpoints and descriptions. The translation unit, described in Section 6.4, provides the ability to translate descriptions from one representation to another. The repository unit, described in Section 6.5, provides a persistent store for data dictionary information and an API for accessing this information.

Swift is implemented in Java. Java is a recent development in object-oriented programming, and the use of this language provides some useful benefits:

- Swift has potentially good cross-platform compatibility, as it may be run in any Java-compliant run-time environment.
- Swift is implemented as a client/server system, that is, the description modelling and translation units act as clients to the repository. In theory, these units could communicate with the repository across a network.

The latter point ties in with one of the design goals of Java, which is to support dynamic

loading of classes at run-time, including across a network (Gosling and McGilton, 1996). This provides the opportunity to extend the client with a new representation by adding classes at the server end that implement the new representation. These classes may then be loaded across the network at run-time by the client. This notion of loading representation-specific code at run-time is conceptually similar to the approach taken by Informix's DataBlades (Keeler, 1996; Informix Software, 1996), and to the approach taken by Configurable Data Modelling Systems (CDMSs), which allow the developer to customise the data model they are using to a particular domain (Cooper, 1991; Serano, 1994).

The repository is stored in a PostgreSQL<sup>1</sup> database. PostgreSQL was chosen because of its object capabilities and ready availability. The combination of loading Java classes across a network and PostgreSQL's large object features also provides the opportunity to store parts of Swift's code as attribute values in the repository itself.

The translations implemented in Swift are based on the rules defined in the previous chapter, but they have been implemented in an ad hoc manner. A more flexible approach would be to define translations using some form of specification language. The interface specification language VML (introduced in Chapter 2) was a possible candidate, and an attempt was made to integrate VML into Swift. This attempt and the issues arising from it are documented in Section 6.4.1.

Other miscellaneous implementation issues are discussed in Section 6.6, and an example of Swift in use is presented in Section 6.7, which includes a demonstration of the effect of heuristics on translations.

## 6.2 The implementation architecture of Swift

The primary goal of Swift is to facilitate the use of multiple data modelling representations to describe a single viewpoint, which is done by performing translations of descriptions between representations. The Swift environment should therefore include at least the following:

- a means of managing viewpoints, representations and descriptions;

---

<sup>1</sup>Formerly known as Postgres95.

- a means of managing translations;
- a means of managing the persistent storage of viewpoint data; and
- a user interface module.

In Figure 6.1 on the following page is shown an architecture for Swift that includes all of these parts. Swift itself can be divided into three logical units. The *description modelling unit* deals with the manipulation of viewpoints, representations and descriptions; the *translation unit* controls the translation process; and the *repository unit* manages the storage of and access to viewpoint data. In practice, the description modelling and translation units are implemented as components of a single front-end Java application, as illustrated by the top half of Figure 6.1. This application also provides the user interface for manipulating viewpoints and descriptions, and initiating and controlling translations.

The lower half of Figure 6.1 depicts the repository unit of Swift, which comprises two parts. The first of these is the repository itself, which can be further subdivided into three sub-parts:

1. the *data dictionary* stores information about viewpoints and descriptions, and is shown in black in the lower centre of Figure 6.1;
2. the *translation store* contains the Java classes that implement translations, and is shown in orange in the lower left of Figure 6.1; and
3. the *representation store* contains the Java classes that implement representations, and is shown in green in the lower right of Figure 6.1.

The last two sub-parts have not been implemented in Swift because of limitations in the Java to PostgreSQL interface at the time of implementation. These sub-parts are, however, peripheral to the translation process, so omitting them has little effect on the ability to perform translations with Swift.

The second part of the repository unit is the repository API (denoted by the arrows in Figure 6.1), which is used by the front end to access the data stored in the repository. Both the repository and the repository API are described in Section 6.5 on page 162.

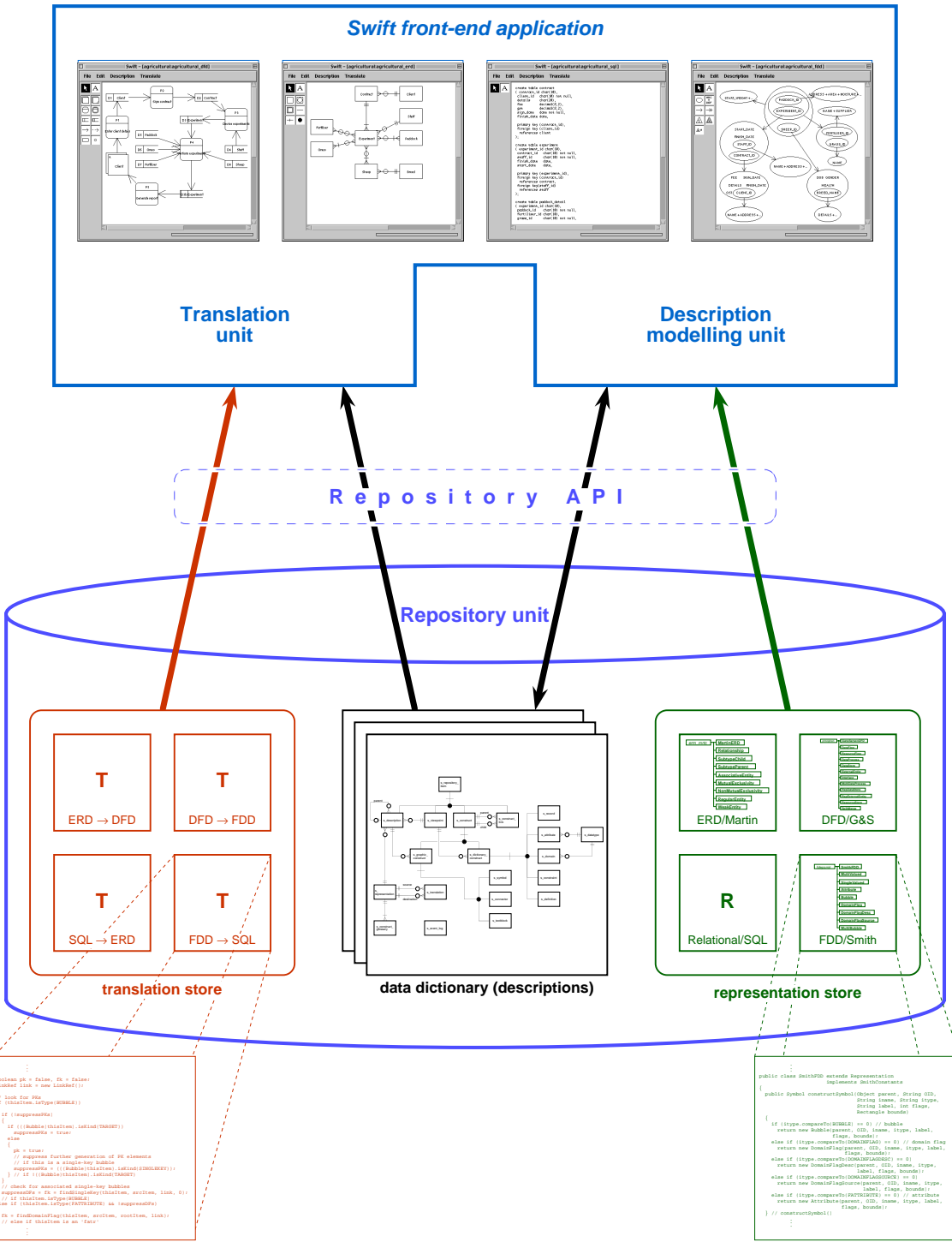


Figure 6.1: Swift's implementation architecture



## 6.2.1 Implementation language(s)

Java was chosen as the implementation language for Swift for several reasons, which will now be detailed. The implementation architecture to some degree determined the choice of language, but choosing Java also affected the implementation architecture. The initial criteria for the implementation language were:

- the language should be readily portable to different platforms;
- the language should be relatively easy to develop in;
- good user interface development tools should be available; and
- the language should preferably be object-oriented.

The first three criteria were chosen for fairly obvious reasons: portability of the resulting environment, and reduction of code complexity and development time. The fourth criterion provided the opportunity to make some additional useful design decisions within the implementation architecture. For example, the description modelling unit could define a set of generic methods for manipulating and presenting constructs that could then be overridden by methods contained within the classes for a particular representation.

Three languages were considered for implementing Swift: C++, Java and Tcl/Tk. The relevant features of each of these languages are summarised in Table 6.1 on the following page. C++ had the advantages of being familiar to the author, and of having a large code base available. Developing a system in C++ can be daunting, however, because of the sheer complexity of the language. A CASE tool is relatively simple in principle, so it would seem that the full power and complexity of C++ is not necessarily required. C++ also tends to be less portable than the other two languages, and the availability of user interface development tools varies depending on the development environment.

Java had the disadvantage of being new to the author, but was similar enough to C++ that this was not a major problem. Java is highly portable because of the Java Virtual Machine architecture, and is less complex than C++ while still retaining many of the useful features of C++. The standard Java development suite also includes a complete application framework for building user interfaces.

**Table 6.1:** Features of Java, C++ and Tcl/Tk with respect to implementing Swift

| Feature                | Java           | C++       | Tcl/Tk          |
|------------------------|----------------|-----------|-----------------|
| Portability            | excellent      | good      | excellent       |
| Development time       | moderate       | long      | short           |
| Complexity             | moderate       | high      | low to moderate |
| Object-oriented        | yes            | yes       | no              |
| Application robustness | excellent      | good      | excellent       |
| Familiarity            | similar to C++ | yes       | no              |
| Run-time performance   | good           | excellent | good            |
| Dynamic class loading  | yes            | no        | n/a             |
| Built-in networking    | yes            | no        | unknown         |
| Packages               | yes            | no        | unknown         |
| UI framework           | yes            | varies    | yes             |

Tcl/Tk also had the disadvantage of being unknown to the author, but it is highly portable, and it is generally faster and easier to develop systems with Tcl/Tk than with languages like C++ and Java (Ousterhout, 1998). Building user interfaces is also relatively easy with Tcl/Tk. The complete unfamiliarity of the language and its lack of object-oriented features argued against this choice, however. Tcl/Tk is also oriented more toward integrating existing tools rather than building new applications.

Having narrowed the choice to C++ and Java, attention was then focused on specific features of each language. Java has a distinct advantage over C++ in terms of robustness of applications because of its lack of pointers, which removes an entire class of errors related to memory management (Gosling and McGilton, 1996, Section 1.2.2). Conversely, applications developed in C++ are generally faster, although this was not a major concern for Swift.

Java provides some additional features that C++ does not, that were considered advantageous for the implementation of Swift, in particular:

- the ability to load classes dynamically at run-time;
- built-in network features; and
- the ability to group related source files into *packages*.

Java's ability to load classes dynamically at run-time means that only the required classes need to be loaded at any given time. This is particularly useful for the classes

that implement a representation, as they only need to be loaded when that representation is actually in use. Combining dynamic loading with Java's networking features means that it is possible to load classes across a network. This makes it possible to completely separate the front-end application from the repository, possibly even running them in a client/server fashion on separate machines. It also becomes possible to store the classes for representations and translations in the repository, reducing the size of the front-end application. While all of these things are possible in C++, they are not built-in features of the language and would require more code to be written in order to emulate them.

In addition, Java's *package* mechanism provides a useful means of grouping related classes together, for example, all the classes relating to a particular representation may be placed in the same package. Classes that are members of the same package may make reference to other classes within the same package without having to first import them (Flanagan, 1997). Packages may also be formed into hierarchies, for example, the `java.awt.event` package is a 'sub-package' of the `java.awt` package.

In conclusion, Java<sup>2</sup> was used because:

- It is relatively less complex to develop in than C++.
- Applications developed in Java tend to be more robust due to Java's elimination of pointer errors and memory leaks (Gosling and McGilton, 1996).
- Java is more portable than C++. There are some compatibility issues with different virtual machines, but this was not considered to be a major problem.
- Java is sufficiently similar to C++ that the learning curve was relatively small compared to that for Tcl/Tk.
- Java's networking features and dynamic loading of classes at runtime allow Swift to be distributed in a client/server fashion.
- Java provides other useful features such as the package mechanism that C++ does not provide.

---

<sup>2</sup>Specifically, version 1.1.3 of Sun's Java Development kit (JDK).

## 6.3 The description modelling unit

The ultimate aim of Swift is to provide a complete design environment that facilitates the use of multiple representations to describe a viewpoint. The functionality for manipulating viewpoints and descriptions is provided by the description modelling unit. The implementation architecture described earlier implies that the user interface for Swift is handled separately from both the translation and description modelling units. In practice, the design of the user interface is so intertwined with the mechanics of the description modelling unit that they cannot be easily separated. Design issues affecting the user interface are discussed in Section 6.3.1. Other implementation issues associated with the description modelling unit are discussed in Section 6.3.2.

### 6.3.1 Design issues

Swift's description modelling unit is used to manipulate and display constructs of a particular representation. There were two main choices of how to implement the user interface to this unit:

1. create a separate, specialised user interface module for each representation; or
2. create a single, generic user interface module and create separate 'plug-ins' for each representation.

The first approach has been implemented by several CASE tools, such as Sybase's Deft (O'Brien, 1992), and has the advantage of allowing the user interface to be tailored for each representation. This approach will generally only be practical if there are a small number of representations, however, because writing a custom user interface module for each new representation can be a relatively labour-intensive task. Deft has only a small number of representations (see Section A.5.4 on page 345), each of which is implemented as a separate Macintosh application. Even with code reuse, creating and maintaining these separate applications would not be a simple task. It is also possible that adding a new module may require modifications to existing modules to enable them to interoperate correctly. The end result is a user interface that is difficult to extend in a modular fashion.

The second approach, by contrast, provides better extensibility, and is an approach that has also been implemented by several CASE tools, such as Visible Systems' EasyCASE (Evergreen Software Tools, 1995c). A generic user interface module is developed that is then specialised according to the type of description being edited. In EasyCASE, this is done by populating the toolbar with symbols appropriate to the representation currently in use. Adding a new representation requires only the construction of an appropriate tool set and the definition of the structure and presentation of each construct in the representation.

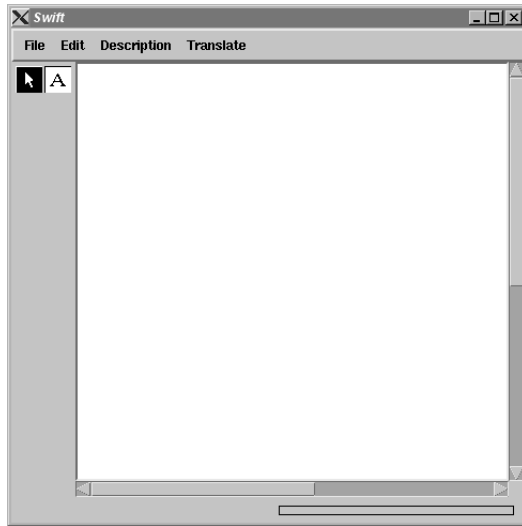
Facilitating the use of multiple representations is an important goal of this research, so it was important that this be reflected by Swift. Accordingly, the second approach was adopted, as it provides greater flexibility when implementing a user interface to handle multiple representations. Swift's user interface changes as different descriptions are loaded, as shown in Figure 6.2 on the following page.

### **6.3.2 Implementation issues**

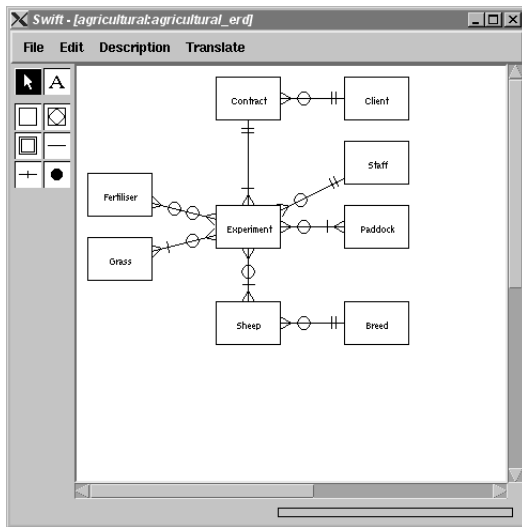
The description modelling unit of Swift comprises a collection of classes that implement the user interface, and classes that manipulate viewpoints, representations, descriptions and constructs. The user interface of the description modelling unit was implemented using the Java 1.1 abstract windowing toolkit (AWT). The JavaBeans component software framework was also considered (Flanagan, 1997, Chapter 10), but discarded because at the time of implementation it was still fairly new and unproven.

Viewpoints are implemented by the class `swift.model.Viewpoint`. Each viewpoint has associated with it a collection of constructs that are shared across all descriptions within that viewpoint, such as attributes, record structures and domains. This enforces a measure of consistency among descriptions, for example, the attribute *address* only needs to be defined once, and will have the same properties in all descriptions that reference it. These 'data dictionary' constructs are implemented as subclasses of the abstract class `swift.repn.DConstruct` and are loaded from the repository when the viewpoint is opened.

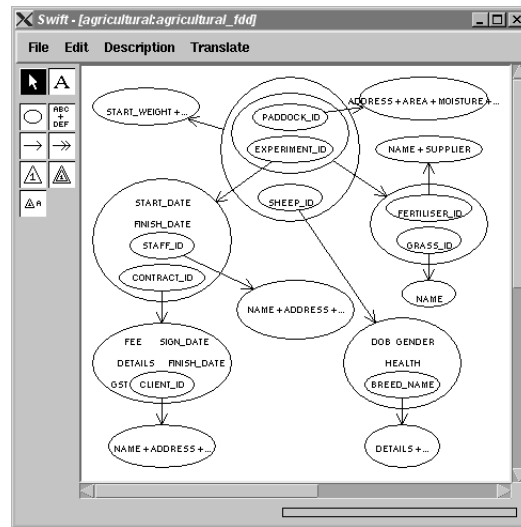
Descriptions are implemented by the class `swift.model.Description`. Each description has associated with it a particular representation and a collection of elements in-



(a) Generic interface



(b)  $\mathfrak{R}_e(E-R, ERD_{Martin})$  interface



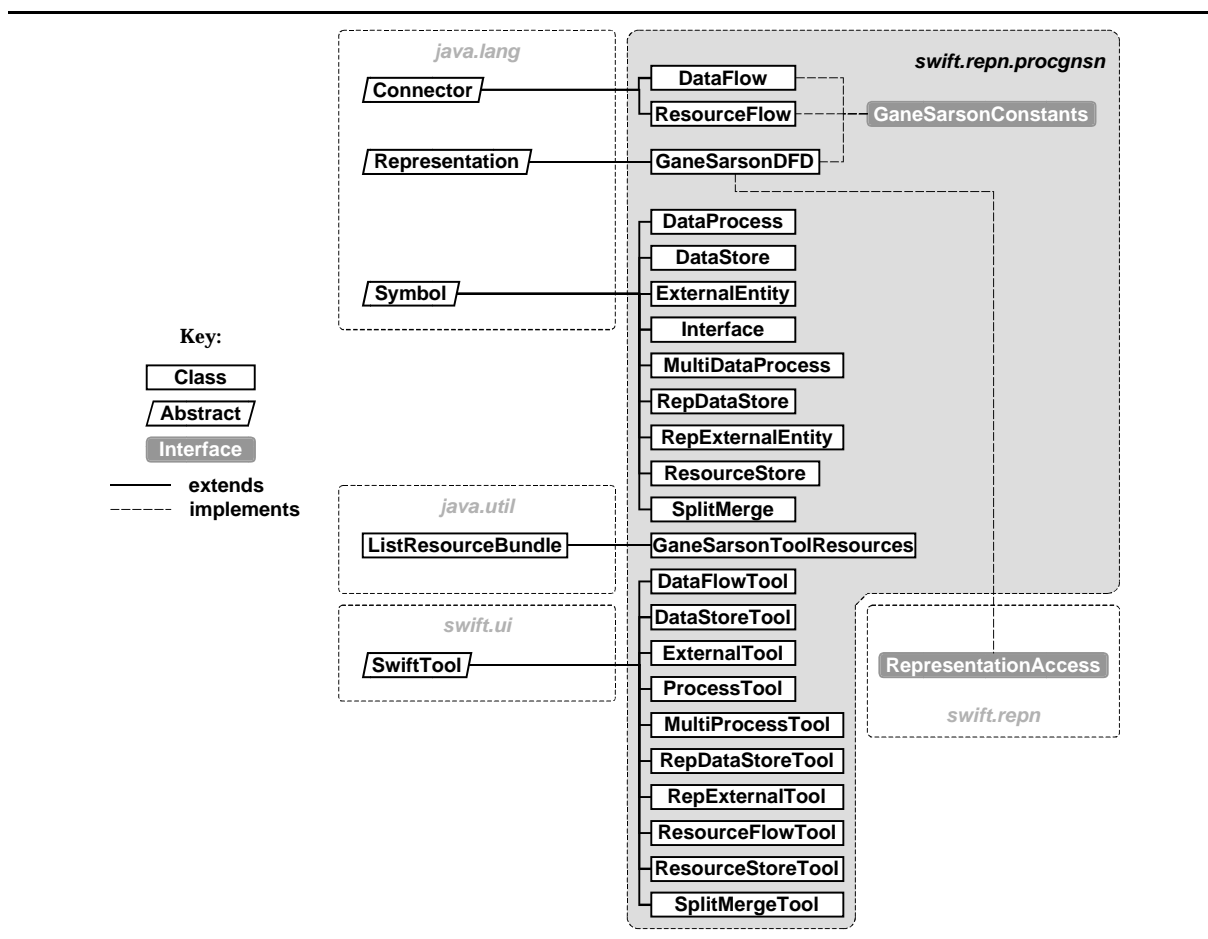
(c)  $\mathfrak{R}_f(FuncDep, FDD_{Smith})$  interface

**Figure 6.2:** The user interface to Swift

stantiated from the constructs of that representation (see below).

Each representation is implemented as a subclass of the abstract class `swift.repn.Representation`. All the classes relating to a representation are placed in the same package, for example, the representation  $\mathfrak{R}_d(DataFlow, DFD_{G\&S})$  is implemented by a collection of classes in the package `swift.repn.procgnsn`, as shown in Figure 6.3. (The complete class hierarchy for Swift may be found in Appendix H.)

The ‘graphic’ constructs of a particular representation are implemented as subclasses of `swift.repn.Symbol` and `swift.repn.Connector` (which are both subclasses of `swift.repn.GConstruct`). Thus, the `GNSEXTERNALENTITY` construct of  $\mathfrak{R}_d$  is implemented by the class `swift.repn.procgnsn.ExternalEntity`, and the `GNSRESOURCEFLOW` construct is implemented by the class `swift.repn.procgnsn.ResourceFlow`.



**Figure 6.3:** Java class hierarchy to implement the representation  $\mathfrak{R}_d(DataFlow, DFD_{G\&S})$  in Swift

The description modelling unit loads descriptions from the repository, where they are stored in a generic form (see Section 6.5 on page 162), and stores them in memory

using the classes outlined above. The procedure for loading and displaying a description is as follows:

1. Locate the data for the description in the repository.
2. Determine the description's representation and load the correct Java class files.
3. Load the data for each element of the description in turn, and pass them to a 'constructor' method in the representation class that returns an instance of the appropriate construct subclass.

Swift at present does not allow the sharing of techniques across multiple representations (see Figure 3.2 on page 43), as the core code of a representation is implemented in a single class combining both the technique and the scheme. This obviously implies code redundancy when implementing representations that share the same technique. This deficiency could be remedied by creating separate Technique and Scheme classes that either replace or enhance the existing Representation class.

Swift implements the representations  $\mathfrak{R}_r(\textit{Relational}, \textit{SQL}/92)$ ,  $\mathfrak{R}_e(\textit{E-R}, \textit{ERD}_{\textit{Martin}})$ ,  $\mathfrak{R}_f(\textit{FuncDep}, \textit{FDD}_{\textit{Smith}})$  and  $\mathfrak{R}_d(\textit{DataFlow}, \textit{DFD}_{\textit{G\&S}})$ . The representation  $\mathfrak{R}_s(\textit{DataFlow}, \textit{CDFD}_{\textit{SOFL}})$  has also been partially implemented. This small number of representations is enough to produce a reasonable initial number of translations, discussed next.

## 6.4 The translation unit

The translation unit of Swift provides the capability to translate descriptions from one representation to another. Translations in Swift follow the individual interfacing strategy, as noted in Section 2.4.3 on page 30, so a separate translation must be implemented for each pair of representations.

Translations in Swift are activated in an asynchronous manner, that is, the user must manually activate the translation as opposed to the environment activating the translation automatically. The effects this approach can have on maintaining consistency among related descriptions have already been discussed in Section 3.6 on page 58, but it should be reiterated here that Swift does not implement any form of consistency



maintenance, as the main reason for developing Swift was to test various ideas associated with the translation-based approach to facilitating the use of multiple representations. If Swift were to be developed further to become a full design environment, some form of consistency maintenance scheme would obviously need to be implemented.

One issue that has not been dealt with to this point is the issue of generating a schema from multiple descriptions. The translations discussed here translate a single source description into a single target description, that is, they are effectively one-to-one translations. Generating a schema could involve the translation of many source descriptions to a single target description, that is, a many-to-one translation. Swift does not as yet implement such many-to-one translations, due to the need to further examine the issues associated with performing many-to-one translations. Some of these issues are discussed in Section 10.5 on page 289.

Translations in Swift are currently defined in an ad hoc manner. A more flexible approach would be to use some form of specification language such as Amor's (1997) View Mapping Language (VML). An attempt by the author to integrate VML into Swift is discussed in Section 6.4.1. The rule evaluation strategy of Swift is discussed in Section 6.4.2, and other implementation details are discussed in Section 6.4.3.

### **6.4.1 Translation specification in Swift**

The translations in Chapter 5 are specified using the abstract rule notation, while Swift is implemented in Java. It was therefore necessary to convert the abstract rules into code that could be executed by Swift. For the purposes of the prototype, the rules were implemented directly in Java code in an ad hoc fashion. While relatively simple to implement, ad hoc translation specification is not an ideal solution and lacks flexibility.

A better alternative would be to adopt some form of language for specifying the rules of a translation. This translation specification could then be interpreted or compiled by Swift and used to perform translations. In Chapter 3 it was suggested that interface specification languages such as Amor's (1997) View Mapping Language (VML) could be used to specify translations of descriptions between representations. The author therefore decided that it would be useful to integrate some form of VML support into Swift.

Implementing a full VML parser in Swift was not feasible due to time constraints, so instead a copy of Amor's (1997) existing VML mapping system was obtained. This system was implemented in Snart, an object-oriented dialect of Prolog developed by Grundy (1993), which was in turn implemented on top of LPA MacProlog32 running under Mac OS. The author's intent was to use the VML mapping system to implement translations that had not been directly implemented in Swift, and then develop tools for translating between the Swift repository structures and the Snart persistent object format (Grundy, 1993; Amor, 1997).

Unfortunately, attempts to perform translations were thwarted by the VML mapping system running out of memory, even when allocated extremely large amounts (up to 200Mb). This problem persisted across several different hardware and software configurations, including different versions of the operating system (Mac OS 7.1, 7.5.2 and 8.1), different versions of the MacProlog32 environment (1.08d and 1.25)<sup>3</sup>, different hardware (a Power Macintosh G3/233, a Power Computing PowerWave 604/150, a Centris 660AV and an LCIII) and different RAM configurations. Attempts to run the system under an almost identical configuration to that used by Amor (1998, personal communication) also failed. At best, only a small part of any particular mapping could be executed before the mapping system overflowed the available heap space. The mappings tested by the author were less complex than several successfully tested by Amor using the same system, so the cause of these problems remains unknown. Even Amor (1998, personal communication) himself was unable to explain these results.

Despite these problems, several simple rules were successfully tested (for example, translating an E-R entity to a DFD data store), but more complex rules could not be tested. These results suggest that using VML to specify the details of translations is a valid approach, but this cannot be fully confirmed until a more robust mapping system is available.

One possible solution was to attempt to fix the existing mapping system, as the complete Prolog source code was available. This option was discarded, however, as the system was slow, memory-intensive and somewhat fragile. The mapping system was also tied to a single development platform (LPA MacProlog32 running under Mac OS),

---

<sup>3</sup>The author would like to thank Dr. John Grundy for providing access to version 1.08d of the MacProlog32 environment for testing purposes.

so it was not a general long-term solution. Other researchers are currently working toward implementing VML mapping systems using mainstream programming languages such as C++ (Price, 1995) and Java (Grundy, 1998), so it was felt that any further exploration in this direction by the author was an unnecessary duplication of effort.

It also became apparent during the attempts to use the VML mapping system that VML is not fully adequate to specify the translation of descriptions between representations. In particular, it is impossible in VML to specify both unidirectional rules and rule exclusion (see Chapter 4). In order to be useful for specifying the details of translations, VML would first need to be extended to deal with these issues.

Finally, the existing Swift implementation was sufficient to test the translation-based approach to facilitating the use of multiple representations (an example of Swift in use may be found in Section 6.7). Implementing a translation specification language in Swift, while useful, would not have generated any new insights, so the author decided to focus instead on defining extensions to VML in order to deal with issues such as rule exclusion. The outstanding issues with VML and the extensions required to address them will be described in Chapter 7. This extended version of VML is known as *VML-S* (the ‘S’ standing for ‘Swift’).

## 6.4.2 Rule evaluation in Swift

Amor’s (1997) VML mapping system evaluates rules by finding all rules that can potentially affect a particular element, then attempting to generate element combinations that match the rules’ headers and invariants (this process was described in Section 4.7 on page 94). It is expected that a *VML-S* mapping system will be similar, although this is not required. The rule evaluation process must be extended to handle exclusions among rules. If there are no exclusions among rules, then rules in a *VML-S* specification may be applied in any order, and the same rule may be applied several times during the translation. Exclusions will imply an ordering for the affected rules, but rules that are not affected by exclusions may still be applied in arbitrary order. Extensions to the original VML algorithms to deal with rule exclusions are described in Section 7.5 on page 201.

Translations in Swift are currently defined in an ad hoc fashion, and are free to

use whichever rule evaluation strategy seems appropriate. Most of the translations currently implemented in Swift follow the strategy of fully processing a single rule until there are no more construct collections that match that rule, which was noted as an alternate rule evaluation strategy in Section 4.7 on page 94. It is also sometimes more efficient to combine the processing of several similar rules into a single unit, and this is an approach that has also been used in Swift's translations.

For example, Swift's FDD to ERD translation activates rules in the following order:

1. All rules/heuristics that pertain to single-valued dependencies are processed.
2. All rules/heuristics that pertain to multivalued dependencies are processed.
3. All rules/heuristics that pertain to isolated bubbles are processed.
4. Any remaining rules/heuristics that do not fall into one of the above groups are processed.

This ordering reflects the translation's genesis from a modified form of Smith's Method for generating normalised relations from an FDD (Smith, 1985), and has proven to be an effective ordering.

Rule exclusion implies the removal of some rules from consideration when the rule(s) that exclude them are executed on a collection of elements. This has been achieved in Swift's translations by marking elements as they are processed. Rules that are activated later can, if necessary, check whether an element has already been processed, and terminate their execution if so. At present elements are marked as either 'processed' or 'not processed' using a boolean flag. A more general solution would be to tag elements with the rules that have processed them, allowing later rules to filter themselves if any rules that exclude them have already been applied to those elements.

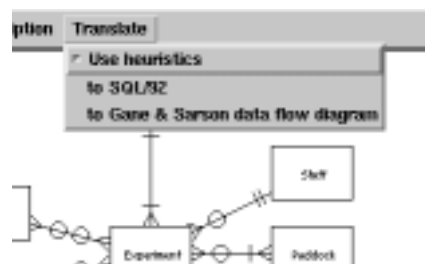
### **6.4.3 Implementation**

Translations in Swift are implemented in a similar manner to representations. A translation between two representations is implemented by a subclass of a generic Translation class, and is dynamically loaded when required. Each translation subclass comprises code that implements the rules and heuristics of that translation.

This approach is particularly effective when applied to the individual interfacing strategy that Swift follows, as each new representation requires the addition of a potentially large number of new translations. As noted in Section 2.4.3 on page 30, if there are  $n$  existing representations, and a new representation is added, up to  $n$  new translations must potentially be created. Defining each translation in its own class makes it easier to manage the collection of existing translations, and new translations may be added at any time without affecting other translations.

The description modelling unit includes a Translate menu, shown in Figure 6.4. This menu contains all the translations that are applicable to the representation of the current description (the Use heuristics menu item allows the heuristics of translations to be turned on and off.). In Figure 6.4, the active description is the Martin ERD  $D_1(V_{agri}, E-R, ERD_{Martin})$ , and the Translate menu contains two translations:

1.  $\mathfrak{R}_e(E-R, ERD_{Martin}) \rightarrow \mathfrak{R}_r(Relational, SQL/92)$ ; and
2.  $\mathfrak{R}_e(E-R, ERD_{Martin}) \rightarrow \mathfrak{R}_d(DataFlow, DFD_{G\&S})$ .

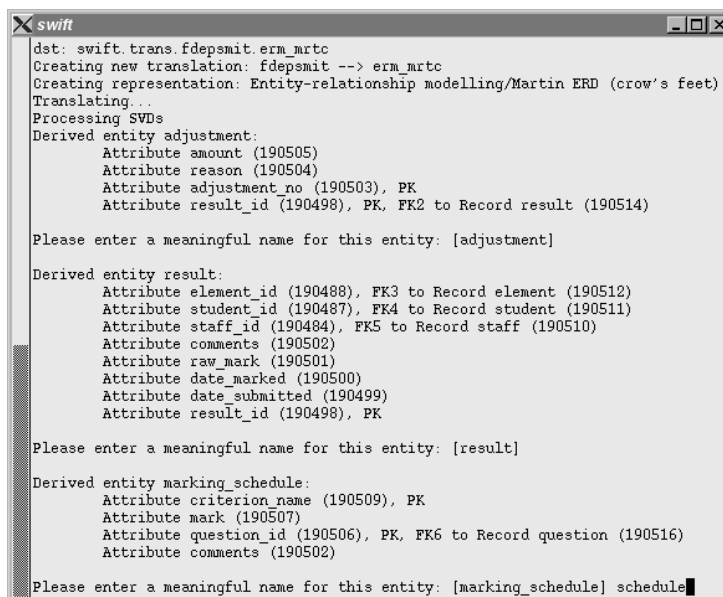


**Figure 6.4:** The Translate menu

Translations are initiated by choosing them from the Translate menu. This causes the appropriate translation class to be loaded into memory, and the translate() method of that class is called. This method takes the source description as an argument and returns the target description.

A major issue with translations identified in Section 4.6 on page 91 is the problem of information ‘gain’ during a translation. This can be solved by the enrichment process whereby ‘missing’ information is collected from the user in some way (see Section 4.6). Pre-translation enrichment can be performed by populating the repository with appropriate data prior to initiating a translation. The three example viewpoints used in this thesis are all ‘pre-enriched’ to some extent. Enrichment during the translation requires

user input. Since Swift is at present running primarily in a Unix environment, this input is dealt with using a basic text prompt, as can be seen in Figure 6.5, in which is shown the mid-point of a translation from an FDD to an ERD. Post-translation enrichment is performed by manipulating the description after the translation is complete.



```
swift
dst: swift.trans.fdepsmit.erm_mrtc
Creating new translation: fdepsmit --> erm_mrtc
Creating representation: Entity-relationship modelling/Martin ERD (crow's feet)
Translating...
Processing SVDs
Derived entity adjustment:
  Attribute amount (190505)
  Attribute reason (190504)
  Attribute adjustment_no (190503), PK
  Attribute result_id (190498), PK, FK2 to Record result (190514)

Please enter a meaningful name for this entity: [adjustment]

Derived entity result:
  Attribute element_id (190488), FK3 to Record element (190512)
  Attribute student_id (190487), FK4 to Record student (190511)
  Attribute staff_id (190484), FK5 to Record staff (190510)
  Attribute comments (190502)
  Attribute raw_mark (190501)
  Attribute date_marked (190500)
  Attribute date_submitted (190499)
  Attribute result_id (190498), PK

Please enter a meaningful name for this entity: [result]

Derived entity marking_schedule:
  Attribute criterion_name (190509), PK
  Attribute mark (190507)
  Attribute question_id (190506), PK, FK6 to Record question (190516)
  Attribute comments (190502)

Please enter a meaningful name for this entity: [marking_schedule] schedule
```

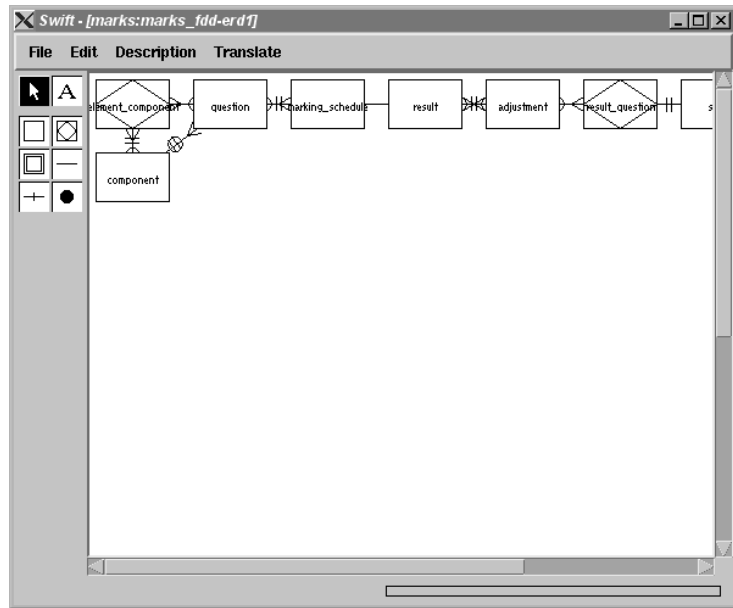
**Figure 6.5:** Enrichment during a translation

Only a small attempt has been made to rearrange the descriptions generated by translations into a more ‘readable’ form. Although many algorithms exist for laying out graph-like diagrams (such as Batini et al., 1985; Tamassia, 1985; Coleman and Parker, 1996; Cimikowski and Shope, 1996; DiBattista et al., 1997)<sup>4</sup>, Swift takes the simple (and quick) approach of positioning the symbols on a grid in the order in which they appear in memory, as shown in Figure 6.6. The user must then finish rearranging the description manually, which could prove unwieldy for large descriptions.

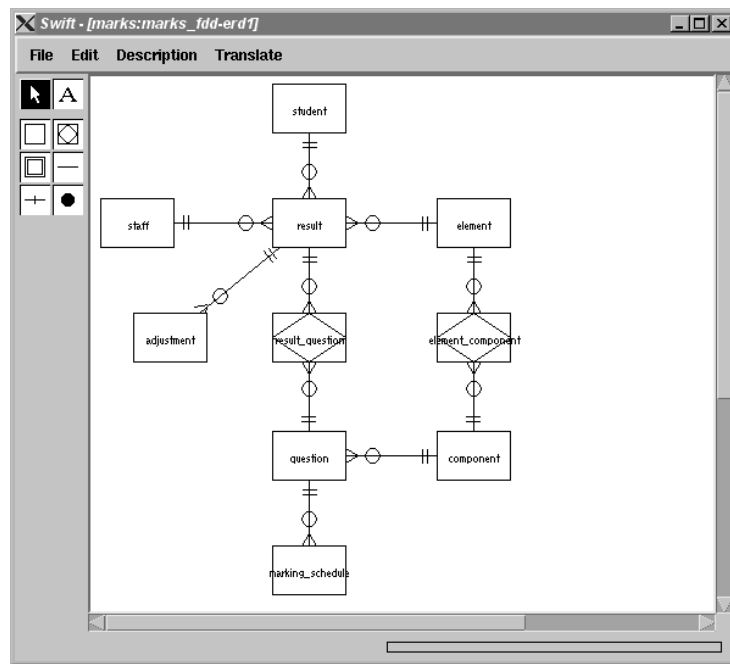
## 6.5 The repository unit

The repository unit comprises two parts, the repository itself, which is a persistent data store for all data relating to a particular viewpoint; and the repository API, which is used by external applications to access the data stored in the repository.

<sup>4</sup>The Java Development Kit also includes four applets that implement what appears to be a genetic algorithm for laying out graphs (Sun Microsystems, 1997).



(a) Immediately after translation



(b) After manual rearrangement

**Figure 6.6:** Rearranging the symbols of a description after a translation ( $\mathfrak{R}_f \rightarrow \mathfrak{R}_e$  on the university marks viewpoint)

The repository stores all information relating to descriptions, such as the elements of descriptions. Ultimately, it is intended that the repository will also store the Java classes for representations and translations, and possibly even parts of Swift itself. There are two ways of implementing such a repository:

- *internally*, that is, build a set of custom data structures and/or files for storing the repository data; or
- *externally*, that is, use already existing data management software, such as a database management system (DBMS), to store the repository data.

The latter option has the advantage of reducing the amount of work required to implement the repository, but the performance of a DBMS-based repository may not be as good as that of a custom-built repository that is tuned specifically for handling these kinds of data. This will generally be outweighed by the implementation advantages gained by using a DBMS, such as the ability to share models among several users with full concurrency and security control. Some CASE tools already follow a similar approach: EasyCASE stores its repository data in dBASE III Plus database files (Evergreen Software Tools, 1995c).

The repository stores data dictionary information about four main categories of object:

- *Viewpoints*, which correspond to the concept of a viewpoint discussed in Chapter 2.
- *Descriptions*, which correspond to the concept of a description discussed in Chapter 3. Typical descriptions might include entity-relationship diagrams or data flow diagrams.
- *Graphic constructs*, which are the 'visual' elements of a representation (for example, entities, data flows and attributes). Graphic constructs are a specialisation of a generic construct class, and are specialised further into three subclasses:
  - *Symbols*, for example, entities or data stores;
  - *Connectors* that link symbols, for example, data flows or relationships; and
  - *Text blocks* that contain blocks of text (not implemented in Swift).



- *Dictionary constructs*, which are the ‘non-visual’ elements of a representation (for example, attributes, domains, constraints). Dictionary constructs are also a specialisation of the generic construct class, and are specialised further into the following subclasses: *attributes*, *domains*, *records*, *constraints* and *definitions*.

These objects are stored within a database structure that is designed to be as generic as possible; an ERD for this database structure is shown in Figure 6.7 on the next page. In memory, the description modelling unit stores elements as instances of subclasses of either the Symbol or Connector classes, and ideally a similar situation would hold in the repository. This would reduce the impedance mismatch (Cattell, 1991) between the description modelling unit and the repository unit, as they would both be using nearly identical data structures. This of course depends on the type of DBMS used to implement the repository. A purely relational DBMS has no concept of subclasses, so some other approach would be required if a relational repository was used. The correspondences between repository entities and the Java classes used in the description modelling unit are listed in Table 6.2.

**Table 6.2:** Correspondence between repository entities and Swift classes

| Repository entity      | Swift class                      |
|------------------------|----------------------------------|
| s_repository_item      | n/a — internal to repository     |
| s_viewpoint            | swift.model.Viewpoint            |
| s_description          | swift.model.Description          |
| s_construct            | swift.repn.Construct             |
| s_graphic_construct    | swift.repn.GConstruct            |
| s_connector            | swift.repn.Connector             |
| s_symbol               | swift.repn.Symbol                |
| s_textblock            | not supported                    |
| s_dictionary_construct | swift.repn.DConstruct            |
| s_attribute            | swift.repn.Attribute             |
| s_domain               | not supported                    |
| s_record               | swift.repn.Record                |
| s_constraint           | swift.repn.Constraint            |
| s_definition           | not supported                    |
| s_construct_link       | n/a — built into Swift classes   |
| s_representation       | swift.repn.Representation        |
| s_translation          | swift.repn.Translation           |
| s_datatype             | partially supported but no class |
| s_construct_glossary   | n/a — reference only             |
| s_event_log            | not supported                    |

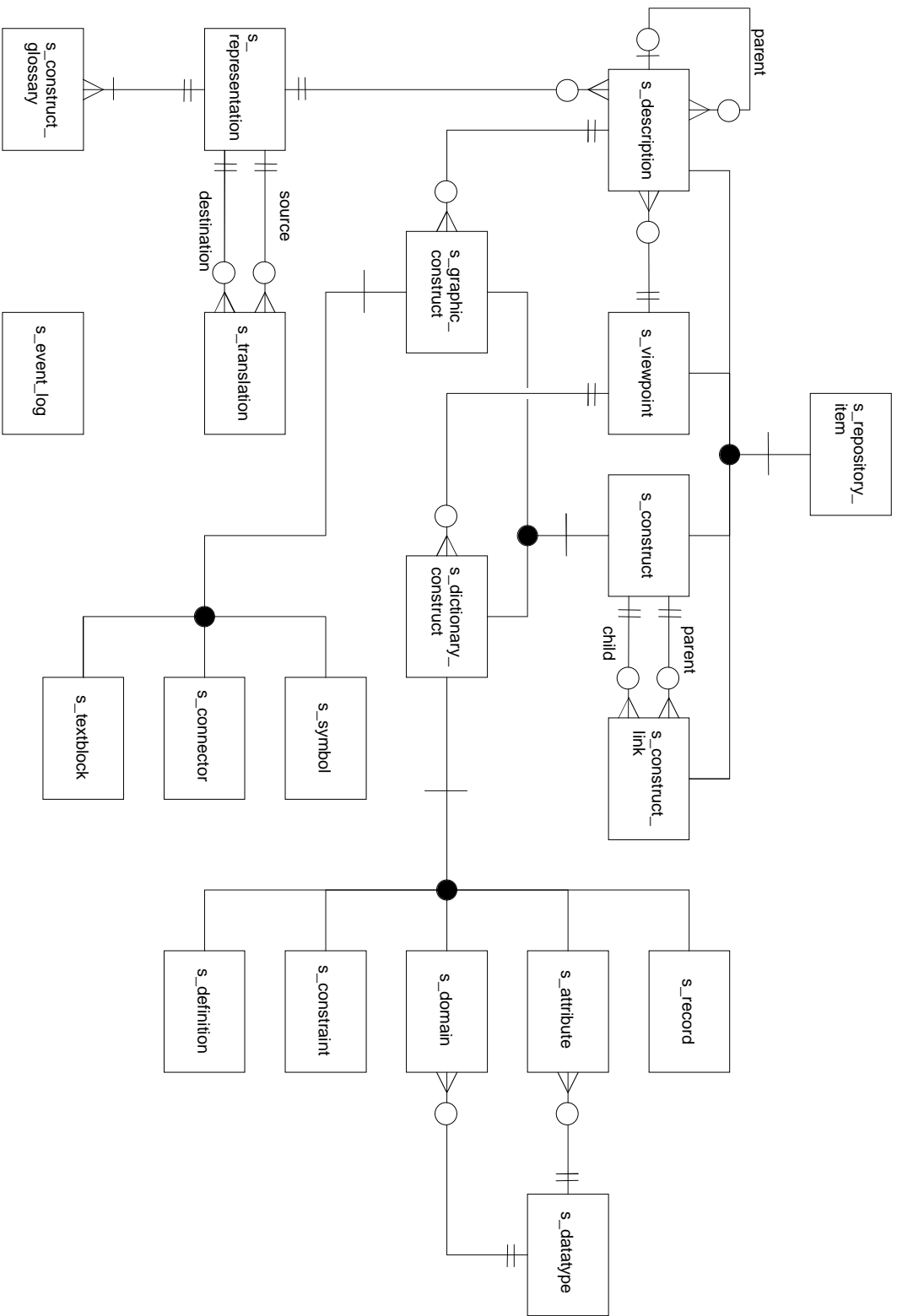


Figure 6.7: Structure of the repository

Four choices were available for implementing the repository unit: Oracle 7.2 running under Windows NT; PostgreSQL or Interbase running under Solaris; and Rdb 6.2 running under OpenVMS. Although the author was most familiar with Rdb, choosing Java as the implementation language immediately removed Rdb from the list, as there was at the time no means for communicating between a Java application and Rdb.

The next best options were Oracle and PostgreSQL. PostgreSQL was chosen over Oracle because of its object capabilities and the availability of source code, which allowed the potential for further tailoring of the DBMS in addition to PostgreSQL's abstract data type facilities. Interfaces were also readily available to allow Java to communicate with PostgreSQL.

It was expected that using PostgreSQL would allow the repository to be implemented in an object-oriented fashion as described above, but this was found not to be the case. Although PostgreSQL is an object-relational DBMS, it is at present more 'relational' than 'object'. For example, in a typical object-oriented environment, an instance of a class is also an instance of its superclass, for example, an instance of `DataStore` is also an instance of `Symbol`. Thus, a query to retrieve all instances of `Symbol` would be expected to also retrieve all instances of `DataStore`. This does not happen in PostgreSQL, however — the SQL query `select * from s_symbol` would not retrieve data from the `s_datastore` table, as these tables are not connected in any way apart from the super/subclass relationship. This means that there is no simple way to retrieve all symbols for a description, as the modelling unit would be required to query all the representation-specific subclasses in the repository.

As a result, the PostgreSQL repository does not implement representation-specific construct subclasses. Instead, elements are stored as instances of either the `s_symbol` or `s_connector` classes. Each instance is parameterised by a four-character *construct type* (the `ctype` attribute), which is used by the description modelling unit to parse the element data (Appendix G on page 451 lists details of the construct types that are defined for each representation). This is a useful approach that has been used in configurable data modelling systems (Serrano, 1994).

The structure of the repository as implemented in PostgreSQL is shown in Table 6.3 on the next page. The lack of representation-specific construct subclasses increases the impedance mismatch between the repository unit and the description modelling unit,

**Table 6.3: Repository database schema (PostgreSQL)**

|                                      |         |                                     |          |   |          |
|--------------------------------------|---------|-------------------------------------|----------|---|----------|
| <b>s_repository_item<sup>a</sup></b> |         | <b>s_textblock<sup>f</sup></b>      |          | <b>s_representation</b>                 |          |
| deleted                              | bool    | bounds                              | box      | technique                               | char4    |
|                                      |         | value                               | text     | techname                                | text     |
| <b>s_viewpoint</b>                   |         | <b>s_dictionary_construct</b>       |          | scheme                                  | char4    |
| vname                                | text    | viewpoint_id                        | oid      | schemename                              | text     |
| creator                              | text    |                                     |          | javaclass <sup>i</sup>                  | text     |
| <b>s_description</b>                 |         | <b>s_attribute</b>                  |          | <b>s_translation</b>                    |          |
| dname                                | text    | datatype <sup>g</sup>               | oid      | source                                  | oid      |
| dtype <sup>b</sup>                   | char4   | size                                | smallint | destination                             | oid      |
| representation                       | oid     | dp                                  | smallint | javaclass <sup>i</sup>                  | text     |
| parent_id <sup>c</sup>               | oid     | <b>s_domain<sup>f</sup></b>         |          | <b>s_datatype<sup>g</sup></b>           |          |
| viewpoint_id                         | oid     | datatype <sup>g</sup>               | oid      | tcode                                   | char4    |
| <b>s_construct</b>                   |         | size                                | smallint | tname                                   | char(20) |
| cname                                | text    | dp                                  | smallint | tdesc                                   | text     |
| ctype                                | char4   | <b>s_record</b>                     |          | <b>s_construct_glossary<sup>f</sup></b> |          |
| flags <sup>d</sup>                   | int4    | (no additional attributes)          |          | representation                          | oid      |
| <b>s_graphic_construct</b>           |         | <b>s_constraint</b>                 |          | ctype                                   | char4    |
| label <sup>e</sup>                   | text    | contype                             | char4    | fullname                                | text     |
| desc_id                              | oid     | <b>s_definition<sup>f</sup></b>     |          | <b>s_event_log<sup>f</sup></b>          |          |
| <b>s_connector</b>                   |         | value                               | text     | when                                    | abstime  |
| src_flags <sup>d</sup>               | int4    | <b>s_construct_link<sup>h</sup></b> |          | what                                    | char4    |
| dst_flags <sup>d</sup>               | int4    | parent                              | oid      | affects                                 | oid      |
| src_port <sup>f</sup>                | int2    | child                               | oid      | event                                   | oid      |
| dst_port <sup>f</sup>                | int2    | description                         | oid      |   |          |
| cpath <sup>f</sup>                   | point[] | linktype                            | char4    |   |          |
| <b>s_symbol</b>                      |         | flags                               | int4     |   |          |
| bounds                               | box     |                                     |          |   |          |

**Notes on Table 6.3:**

- <sup>a</sup> s\_repository\_item is the generic root class for all 'non-static' repository classes.
- <sup>b</sup> dtype is a four-character code that denotes the generic type of the description, for example 'erd\_' for an entity-relationship diagram.
- <sup>c</sup> parent\_id is the object identifier of the parent description, if any. This provides support for levelled descriptions such as data flow diagrams.
- <sup>d</sup> Flags are used to store representation-specific modifiers to an element's appearance or behaviour. For example, the cardinality and optionality of an E-R relationship is encoded in its src\_flags and dst\_flags.
- <sup>e</sup> label is a text value drawn on screen with the construct, for example, the name of a data store. If the label is empty, nothing is displayed. This is distinct from the construct's name, which is internal to Swift and not normally seen by the user.
- <sup>f</sup> Not supported in the current version of Swift.
- <sup>g</sup> s\_datatype is a lookup table encoding data types for use in s\_attribute and s\_domain. The schema generation process will use this information to generate appropriate data type definitions.
- <sup>h</sup> This table encodes the links between constructs: record contains attribute, symbol is source of connector, and so on.
- <sup>i</sup> javaclass will eventually contain the compiled Java class for the representation or translation. At present it contains only the class name, for example SmithFDD. The representation's package name is constructed by concatenating the technique and scheme attributes, for example, 'fdep' + 'smit'. This produces the fully qualified class name swift.repn.fdepsmit.SmithFDD, as all representations fall under the swift.repn package. A similar process occurs for translations.

which requires extra work on the part of the description modelling unit to parse the repository data. If Swift were to be ported to a true object-based repository such as ObjectStore (Object Design, 1999), this would probably change.

The repository API consists of a set of ‘wrapper’ calls layered on the Java Database Connectivity (JDBC) libraries of Java (Sun Microsystems, 1998b), and acts as an abstraction layer between the other units and the repository. The basic API is defined in the Repository class, which is then subclassed for specific DBMS types, such as relational or object DBMSs. At present, the only such subclass is SQLRepository for SQL-based relational DBMSs. These generic DBMS subclasses may then be further subclassed for specific DBMS products, such as PostgreSQL, which is implemented in Swift by the classes Postgres95101 and PostgreSQL621 for versions 1.0.1 and 6.2.1 respectively.

This abstraction allows the internal structure of the repository to be changed without overly affecting the other units, and also allows support for new repository DBMSs to be added without having to change the repository classes in the front end. The PostgreSQL repository classes use the JavaPostgres95 JDBC driver (McLean et al., 1997), which was originally developed as a Java implementation of PostgreSQL’s *libpq* programming interface (Lockhart, 1999), but has since evolved into a full JDBC driver.

The repository is currently managed by direct manipulation using SQL, but it is intended that a repository management unit will eventually be implemented as part of the Swift front-end to provide repository management facilities similar to that offered by commercial CASE tools.

It is also intended that the repository will eventually store many of the compiled Java classes that make up Swift, as noted in Table 6.3. This is particularly useful for the classes that define a representation’s behaviour, as these are not loaded until they are actually needed. This was not possible at the time of implementation, however, because the JavaPostgres95 driver did not fully support PostgreSQL’s large object features. This has therefore been left as an area for future work, and some ideas are outlined in Chapter 10. At present, the Java class name is stored in the repository as a string, which is retrieved and used to locate the appropriate Java class file at runtime.

## **6.6 Miscellaneous implementation issues**

### **6.6.1 Logging of operations**

At present Swift does not support the logging of operations on descriptions and view-points. This would be useful in order to provide an audit trail and a means of rolling back changes to descriptions. The logging mechanism would obviously need to be related to any consistency maintenance mechanism that was implemented.

Translations are carried out in the MViews framework by propagating update records throughout the affected viewpoint. These update records may be logged to show a series of updates made to a description (Grundy et al., 1997a). This is particularly important in MViews-based environments when information cannot be translated from one description to another (usually because of a mismatch in the expressive powers of the representations involved). The update records are expanded into a human-readable form that can be used as a basis for applying appropriate changes manually.

Although Swift does not implement a logging system, support for this has been included in the repository in the `s_event_log` class. This class is designed to track events as they occur in the repository (inserts, updates and deletes). It is intended that this will be maintained automatically via some form of trigger mechanism. The Swift front-end may then interpret this information as appropriate.

### **6.6.2 Extensibility**

A useful feature of an environment that supports multiple representations is the ability to easily add new representations. Adding new representations to Swift requires defining the Java classes that implement the representation and its constructs, and adding an entry to the `s_representation` class in the repository that defines the Swift internal identifiers for the representation and the name of the appropriate Java class. A similar process is undertaken to add new translations. Adding a new representation or translation in this way requires no changes to the Swift 'kernel', and is totally transparent to the user.

Shifting the repository to a different DBMS should also be relatively easy because of the generalised repository API. All that is required for a new repository DBMS is

to define a subclass of the appropriate Repository subclass. If an entirely new class of DBMSs is to be introduced, a new Repository subclass must also be defined. Repository subclasses that could be defined include object-oriented, object/relational, flat files and ODBC (Microsoft's *Open Database Connectivity* middleware).

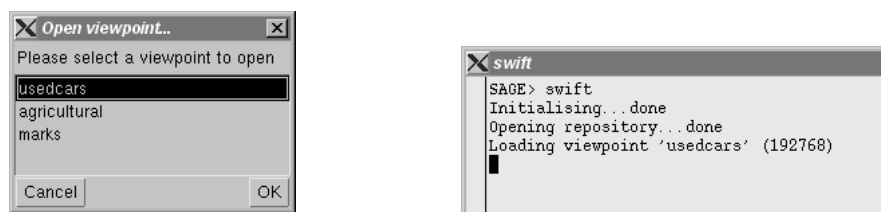
Extending Swift with new logical units in addition to the three already defined should also not be particularly difficult because of the modular nature of the application. The only bottleneck here is that the user interface for all the units is presently combined in the front-end application. This would require altering the front-end in order to add a new unit. This is not likely to happen very often, however.

It is anticipated that moving Swift to a component architecture such as JavaBeans (Flanagan, 1997, Chapter 10) will further enhance the extensibility of the environment.

## 6.7 Example of Swift in use

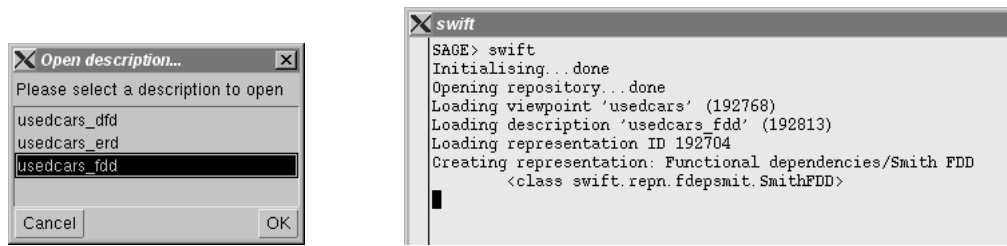
In this section is presented a brief example illustrating the use of Swift. Although some previous figures have shown screenshots from Swift, there has not been a complete example showing Swift being used to perform a translation.

Swift groups related descriptions together into a viewpoint. The used cars viewpoint is shown being loaded in Figure 6.8. Once a viewpoint has been loaded, individual descriptions within that viewpoint may be loaded. In Figure 6.9 on the following page the Smith FDD description  $D_1(V_{cars}, FuncDep, FDD_{Smith})$  of the used cars viewpoint is shown being loaded. As this description is loaded, the classes for the representation  $\mathfrak{R}_f(FuncDep, FDD_{Smith})$  are also loaded so that the description may be displayed on the screen and the user interface customised appropriately.



**Figure 6.8:** Loading a viewpoint in Swift

After a description is loaded, the Translate menu is populated with translations appropriate to the representation of the active description. In this example, as illustrated



**Figure 6.9:** Loading a description in Swift

in Figure 6.10, the Translate menu contains two translations:

1.  $\mathfrak{R}_f(\text{FuncDep}, \text{FDD}_{\text{Smith}}) \rightarrow \mathfrak{R}_r(\text{Relational}, \text{SQL}/92)$ ; and
2.  $\mathfrak{R}_f(\text{FuncDep}, \text{FDD}_{\text{Smith}}) \rightarrow \mathfrak{R}_e(\text{E-R}, \text{ERD}_{\text{Martin}})$ .



**Figure 6.10:** The Translate menu for an FDD

Suppose the second translation is chosen. The correct translation class is loaded and the active description is passed to the translation's `translate()` method. Although it is possible in the  $\mathfrak{R}_f \rightarrow \mathfrak{R}_e$  translation to derive default names for entities from the internal names of the FDD's dependencies, this may not always give the best result depending on the extent of pre-enrichment in the source description. User input may therefore be required to acquire meaningful entity names, as shown in Figure 6.11.

When the translation is complete, the symbols of the new description are partially rearranged and the description is displayed on the screen, as shown in Figure 6.12(a) on page 174. The user may then manually rearrange the description as required, as shown in Figure 6.12(b).

This example has briefly shown how Swift may be used to manipulate existing viewpoints. A more comprehensive case study showing how Swift may be used to build a new viewpoint is presented in Chapter 9.



```

X swift
Initialising... done
Opening repository... done
Loading viewpoint 'usedcars' (192768)
Loading description 'usedcars_fdd' (192813)
Loading representation ID 192704
Creating representation: Functional dependencies/Smith FDD
  <class swift.repn.fdepsmit.SmithFDD>
class swift.dd.Repository
src: swift.trans.fdepsmit.erm_mrtc
dst: swift.trans.fdepsmit.erm_mrtc
Creating new translation: fdepsmit --> erm_mrtc
Creating representation: Entity-relationship modelling/Martin ERD (crow's feet)
Translating...
Processing SVDs
Derived entity car_2:
  Attribute VIN (192902), PK
  Attribute registration (192894), FK2 to Record car (192905)

Please enter a meaningful name for this entity: [car_2] car_vin

Derived entity car:
  Attribute colour (192879)
  Attribute year (192904)
  Attribute VIN (192902), FK3 to Record car_2 (2)
  Attribute registration (192894), PK
  Attribute odometer (192892)
  Attribute model (192890)
  Attribute miles_km (192889)
  Attribute make (192888)
  Attribute list_price (192887)

Please enter a meaningful name for this entity: [car] █

X swift
  Attribute address (192876)
  Attribute phone (192893)
  Attribute name (192891)
  Attribute ird_number (192883), PK

Please enter a meaningful name for this entity: [staff]

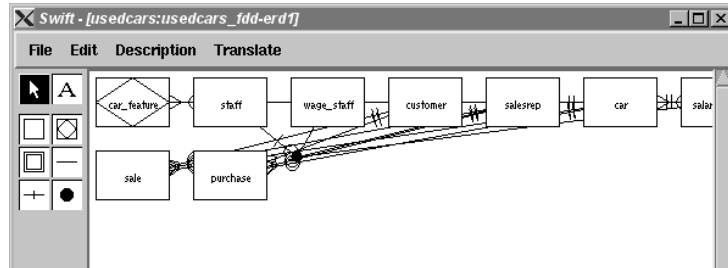
Processing MVDs
Derived entity car_feature_2car_feature_1:
  Attribute registration (192894), PK, FK14 to Record car (192905)
  Attribute feature_code (192884), PK, FK13 to Record feature (192907)

Please enter a meaningful name for this entity: [car_feature_2car_feature_1] car_feature

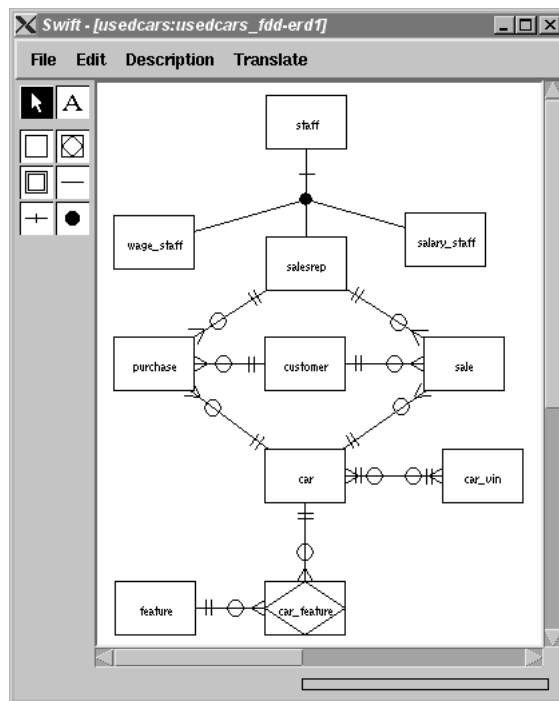
Processing isolated bubbles
Processing relationships
  purchase >0--+ customer
  salary_staff --0|- staff
  car_feature >0--+ car
  sale >0--+ salesrep
  purchase >0--+ salesrep
  wage_staff --0|- staff
  salesrep --0|- staff
  sale >0--+ customer
  sale >0--+ car
  purchase >0--+ car
  car_vin >0--+ car
  car >0--+ car_vin
  car_feature >0--+ feature
Rearranging diagram
█

```

Figure 6.11: The beginning and end of the translation process



(a)

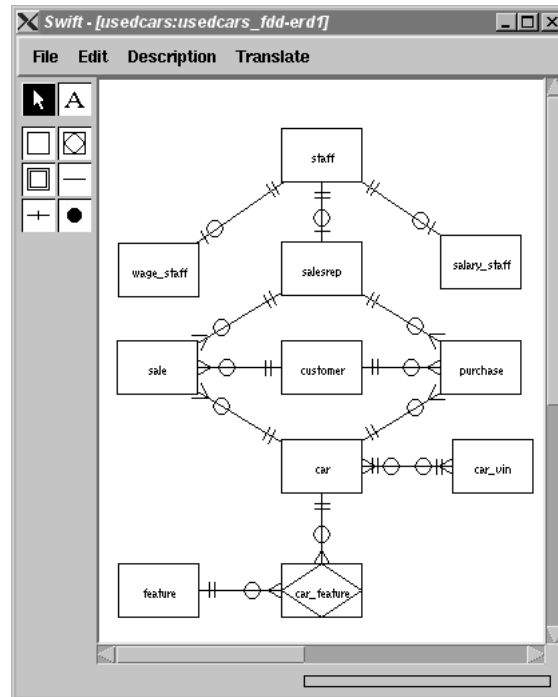


(b)

**Figure 6.12:** The target description: (a) as generated by Swift; (b) after manual rearrangement by the user

### 6.7.1 The effect of heuristics on translations

The example above was executed with heuristics active. If heuristics are deactivated, the  $\mathfrak{R}_f \rightarrow \mathfrak{R}_e$  translation produces the description shown in Figure 6.13. When compared with the description in Figure 6.12(b), it can be seen that the type hierarchy between the Staff, Wage\_staff, Salesrep and Salary\_staff entities has not been generated.



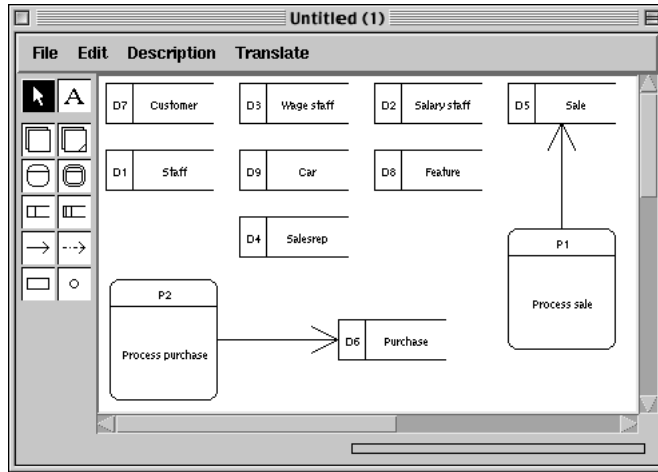
**Figure 6.13:** Effect of heuristics on the  $\mathfrak{R}_f \rightarrow \mathfrak{R}_e$  translation (used cars viewpoint)

A similar effect can be seen in the  $\mathfrak{R}_e(E-R, ERD_{Martin}) \rightarrow \mathfrak{R}_d(DataFlow, DFD_{G\&S})$  translation, as illustrated in Figure 6.14 on the following page. Note the generation of process elements in Figure 6.14(a) when heuristics are active.

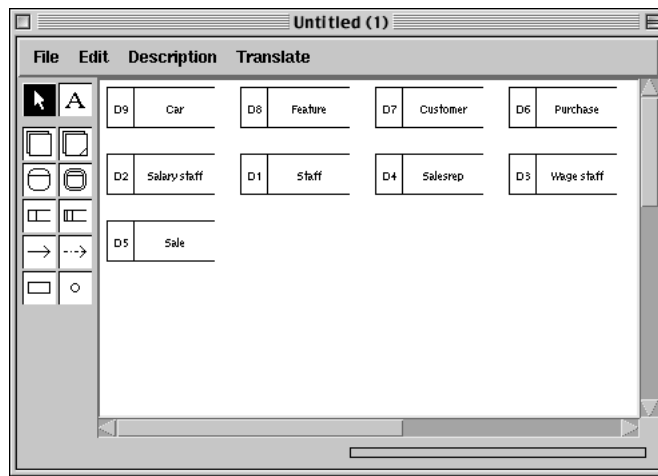
From these examples, it can be intuitively seen that heuristics do have a positive impact on the quality of the translations. A method for measuring relative translation quality will be developed in Chapter 8 to verify this intuition.

## 6.8 Summary

In this chapter was discussed the implementation of a prototype modelling environment (called *Swift*) that facilitates the use of multiple representations. Swift is implemented in Java and comprises three logical units: the description modelling unit, the



(a)



(b)

**Figure 6.14:** Effect of heuristics on the  $\mathfrak{R}_e \rightarrow \mathfrak{R}_d$  translation: (a) with heuristics; (b) without heuristics

translation unit and the repository unit. The former two units are implemented as a single application that communicates with the repository, which is stored in a PostgreSQL database. Swift takes advantage of the dynamic class loading capabilities of Java to allow representation and translation classes to be loaded at runtime.

Swift provides a useful demonstration of the approach followed in this research, namely using translations between representations to facilitate the use of multiple representations. Two major issues have been identified that need to be explored further. First, a need for some form of translation specification language was identified. VML was selected for this purpose, but was found to be lacking with respect to some of the issues identified in Chapter 4, in particular rule exclusion and the specification of unidirectional rules. Some modifications must also be made to the translation process described in Chapter 4, due to limitations in Amor's (1997) implementation of VML. An extended version of VML, known as *VML-S*, is defined in the next chapter.

The second issue arising from this chapter is measuring the effect that heuristics have on translation quality. Experimentation with Swift has shown intuitively that heuristics have a positive impact on translation quality. A formal means of verifying this intuition will be developed and applied in Chapter 8.



# Chapter 7

## Translations using VML-S

### 7.1 Introduction

In Chapter 5, rules for the translation  $\mathfrak{R}_e(E-R, ERD_{Martin}) \Rightarrow \mathfrak{R}_e(Relational, SQL/92)$  were defined using the abstract translation notation from Chapter 4. This notation does not, however, allow the detailed specification of the mappings between the constructs in a rule, and in particular does not allow the specification of mappings between the *properties* of those constructs, so some other form of language or notation is required. It was decided in Chapter 6 that Amor's (1997) View Mapping Language (VML) would provide a useful way to specify a translation between two representations. An overview of the language is presented in Section 7.2.

As discussed in Chapter 6, however, an attempt by the author to integrate VML support into Swift highlighted some outstanding issues with VML that complicate the use of VML in the context of specifying description and element translations. These issues are partly a result of VML's orientation towards schema translation rather than defining mappings between constructs of representations, and partly a result of incomplete specification in some areas that were not central to Amor's research. Extensions to VML are therefore defined in Section 7.3 to deal with these issues. This extended version of VML is named *VML-S* ('S' stands for 'Swift'), and can be used to specify translations between different representations in a more detailed manner.

While the abstract notation cannot be used to specify the details of rules, it has proven to be useful in the initial stages of developing a VML-S specification. A correspondence between the abstract notation and VML-S specifications is identified in Section 7.4.

None of the extensions defined in Section 7.3 have as yet been incorporated into

a VML implementation, but the implementation issues arising from these extensions have been examined, and algorithms have been developed to support the extensions. These issues and algorithms are presented in Section 7.5.

## 7.2 An overview of VML

The *View Mapping Language* (VML) is a high-level declarative language for specifying mappings between schemas (Amor, 1997, Chapter 5). It was designed to address many of the shortcomings with other languages identified by Amor (1997, Section 4.2), and meets all of Amor's requirements for translation specification languages listed in Section 2.4.4 on page 32. VML was originally implemented using an object-oriented derivative of Prolog known as Snart (Grundy, 1993), and thus inherits the declarative syntax of Prolog, while also providing some object-oriented features, such as complex type handling.

A mapping specification in VML comprises an *inter\_view* definition, followed by a collection of *inter\_class* definitions. The *inter\_view* definition specifies the schemas that are to be mapped between and some simple constraints on the mapping, while the *inter\_class* definitions specify how to map particular objects between the two schemas. In Figure 7.1 is shown an example of a mapping specification that maps staff data from a relational schema to an object-oriented schema (this example is derived from the used cars viewpoint described in Appendix C). In the relational schema, staff data are stored in a single Staff table, whereas in the object-oriented schema, staff data are stored in instances of one of three subclasses of the Employee class.

The first four items of an *inter\_view* definition specify the *model\_ids* of the two schemas to be mapped between and what is known as the *model\_type* for each schema. Each schema is identified by a schema name and an optional version number enclosed in braces ({}). In Figure 7.1(c), the *model\_ids* are `usedcars_SQL` and `usedcars_ODL` respectively. The *model\_type* entries for each schema effectively specify the directions in which the mapping may be applied, and must be one of the following:

- *read\_only*, meaning that data may be mapped out of the schema, but not into it;
- *read\_write*, meaning that data may be mapped both into and out of the schema; or



---

```

create table staff
( employee_no char(7),
  name char(80),
  address char(80),
  phone char(12),
  job_code char(2)
  check (job_code in
    ('SR', 'WS', 'SS')),
  -- salesrep columns
  commission_rate numeric(3, 2),
  -- salary staff columns
  salary integer,
  -- wage staff columns
  hourly_rate numeric(5, 2),
  hours_per_week numeric(4, 2)
  primary key (employee_no)
);

```

**(a) Relational (SQL/92)  
schema**

```

interface employee
{ extent employees;
  key IRD_number;
  attribute character[7] IRD_number;
  attribute character[70] name;
  attribute character[100] address;
  attribute character[12] phone;
}

interface salesrep:employee
{ extent salesreps;
  key employee::IRD_number;
  attribute integer[4,2] commission;
}

interface waged:employee
{ extent wagedstaff;
  key employee::IRD_number;
  attribute integer[5,2] rate;
  attribute integer[3,1] normal_hrs;
  attribute integer[3,1] timehalf_hrs;
  attribute integer[3,1] double_hrs;
}

interface salaried:employee
{ extent salariedstaff;
  key employee::IRD_number;
  attribute integer salary;
}

```

**(b) Object-oriented  
(ODL) schema**

```

inter_view(usedcars_SQL, read_write, usedcars_ODL, read_write, complete).

inter_class
( [staff], [employee],
  equivalences
  ( employee_no = ird_number,
    name = name,
    address = address,
    phone = phone
  )
).

inter_class
( [staff], [salesrep],
  inherits(inter_class([staff], [employee])),
  invariants(job_code = 'SR'),
  equivalences(commission_rate = commission)
).

inter_class
( [staff], [salaried],
  inherits(inter_class([staff], [employee])),
  invariants(job_code = 'SS'),
  equivalences(salary = salary)
).

inter_class
( [staff], [waged],
  inherits(inter_class([staff], [employee])),
  invariants(job_code = 'WS'),
  equivalences
  ( hourly_rate = rate,
    hours_per_week = normal_hrs + timehalf_hrs + double_hrs
  )
).

```

**(c) VML mapping**

---

**Figure 7.1: Example of a VML mapping specification**

- *integrated*, which applies when an integrated model (interchange format) is used.

Thus, in Figure 7.1(c), the *model\_type* for both schemas is *read\_write*, which effectively means that the mapping is bidirectional.

The last item in an *inter\_view* definition is the *map\_type*, which specifies how complete the movement of data needs to be between the two schemas. It may be either *complete*, that is, all objects in one schema must be mapped to objects in another, or *partial*, that is, only some of the objects need to be mapped.

The first two items of an *inter\_class* definition specify the objects (referred to by Amor as ‘entities’) that are being mapped on both the left and right hand sides of the mapping. Together, these items form the ‘header’ of the *inter\_class* definition. Thus, the header of the first *inter\_class* definition in Figure 7.1(c) specifies a mapping between the Staff ‘object’ in the SQL schema and the Employee object in the ODL schema. The rest of the *inter\_class* definition comprises up to four parts: an optional *inherits* clause, an optional list of *invariants*, an optional list of *equivalences* and an optional list of *initialisers*.

VML *inter\_class* definitions may import the definitions of other *inter\_class* definitions using the optional *inherits* clause. This clause is similar to the `#include` preprocessor directive in C or the `import` command in Java, in that it incorporates the definition of the inherited rule into the current rule; it does not provide inheritance in the object-oriented sense. It is typically used to share common specifications across several *inter\_class* definitions.

Invariants specify constraints on a mapping, as described in Section 4.2 on page 68, and may be applied in two ways:

1. When applied to the source schema, they specify the pre-conditions that must be satisfied before a particular *inter\_class* definition may be applied to objects of the source schema. For instance, the invariants of the [staff], [salesrep] *inter\_class* definition in Figure 7.1(c) specify that Staff rows may only be mapped to Salesrep instances when the value of Staff.job\_code is equal to ‘SR’.
2. When applied to the target schema, they specify post-conditions on the newly mapped objects. Using the same example as above, it can be seen that when mapping Salesrep instances to Staff rows, Staff.job\_code must be set to ‘SR’.

Initialisers specify initial values for attributes of new objects created in the target schema by a mapping, including the definition of constructors for object-oriented models. A major use of initialisers is to specify defaults for information that does not exist or is implicit in the source schema. These initial values may be altered later by equivalences.

Equivalences specify the actual mappings between objects in two schemas. Most of the equivalences in Figure 7.1(c) specify only simple equalities between object attributes, but VML also allows for more complex specifications, including pointers, function calls, list and array references, iteration over lists and arrays, procedures, method invocation and temporary variables (Amor, 1997, Section 5.2.6). The [staff], [waged] *inter\_class* definition in Figure 7.1(c) has an interesting equivalence that maps the hours worked. In the Waged ODL class, the total hours worked is stored in three attributes representing normal hours (*normal\_hrs*), time and a half hours (*timehalf\_hrs*), and double time hours (*double\_hrs*), whereas in the Staff SQL table, the total hours worked is stored in a single attribute (*hours\_per\_week*). The mapping of hours worked from Waged to Staff is straightforward: simply sum *normal\_hrs*, *timehalf\_hrs* and *double\_hrs* and store the result in *hours\_per\_week*. The reverse mapping is not so obvious, however, as there is no indication as to how the value of the *hours\_per\_week* attribute is actually calculated. A solution could be to add a constraint to the newly-created Waged instance that restricts the sum of *normal\_hrs*, *timehalf\_hrs* and *double\_hrs* to being equal to the value of *hours\_per\_week*.

VML has many other powerful features (Amor, 1997):

- VML allows the specification of arbitrary mappings. Most mappings may be specified declaratively, but it is possible to call Prolog procedures for more complex mappings.
- A graphical notation (VML-G) is defined to allow expression of mappings at a high level of abstraction, although this notation can become difficult to read for complex mappings, as noted in Section 4.4 on page 78.
- VML is a declarative language, so mappings may be applied in arbitrary order.
- Mappings may either create a new schema, or incrementally update an existing schema.

- Conditional mappings that map part of a set of objects according to specified criteria may be defined using *bijections* (see Section 7.3.1).

### 7.2.1 Using VML to specify translations between representations

VML was originally designed for specifying mappings between schemas, but as discussed in Section 4.4 on page 78, it is possible to apply VML to representation definitions in order to specify translations between representations. An *inter\_view* definition specifies a description translation between two representations. The *model\_type* entries of the *inter\_view* specify the directions in which the translation may be applied, and the *map\_type* entry specifies the completeness of the translation (partial or complete). An *inter\_class* definition specifies an element translation between two representations. The header of an *inter\_class* comprises two lists of constructs, one from each representation. Invariants, initialisers and equivalences refer to the properties of constructs, but are otherwise treated no differently than they would be in ‘normal’ VML usage.

## 7.3 Outstanding issues with VML

VML was successfully designed to specify the mapping of objects between schemas. While the above discussion implies that it is relatively simple to use VML to specify translations between representations, there are some issues with VML that interfere with this, which will now be discussed. In particular, the following issues must be addressed:

- the specification of unidirectional rules (Section 7.3.1);
- the handling of rule exclusion (Section 7.3.2);
- no algorithm for translating lists of elements (Section 7.3.3); and
- the inability to refer to the same construct more than once in an *inter\_class* header (Section 7.3.4).

In this section, each of these issues will be discussed, and extensions to VML proposed to solve them. This extended version of VML is known as *VML-S*, where the ‘S’

stands for ‘Swift’ (see Chapter 6). Most of the extensions involve adding new clauses to VML’s *inter\_class* syntax; the modified syntax for the VML-S *inter\_class* definition is shown in Backus-Naur form (BNF) in Figure 7.2. Complete syntax for the extensions is given in the following sections. A full BNF syntax definition for VML-S may be found in Appendix F.

---

```

⟨inter_class_def⟩ ::= inter_class( ⟨class_list⟩ , ⟨class_list⟩
                                [, ⟨label⟩] [, ⟨inherits⟩] [, ⟨direction⟩]
                                [, ⟨excludes⟩] [, ⟨invariants_def⟩]
                                [, ⟨equivalences_def⟩] [, ⟨initialisers_def⟩] ) .
⟨class_list⟩ ::= [ [⟨class_list_name⟩ { , ⟨class_list_name⟩ } ] ]
⟨class_list_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⟩ } )

```

---

**Figure 7.2:** BNF syntax of the VML-S *inter\_class* definition

### 7.3.1 Unidirectional rules

Many translations are bidirectional in nature, that is, the ‘source’ and ‘target’ are interchangeable. If the specification language used only allows unidirectional translations, then a bidirectional translation must be defined as two separate translations, one for the ‘forward’ (left-to-right) direction and one for the ‘reverse’ (right-to-left). This could potentially result in inconsistencies between the two translations (Amor, 1997, p. 72).

The ability to specify translations in a bidirectional manner is thus very important, and was a strong argument in favour of using VML. A useful side effect of using a bidirectional specification language is that the process of specifying a translation in, for example, the forward direction results in much, if not all, of the reverse translation being specified simultaneously.

Some translations, however, may only be appropriate in one direction. VML allows the specification of unidirectional description translations by defining a representation as *read\_only* in the *inter\_view* definition. This means that information may only be

read from descriptions expressed using that representation, not written to them. VML does not, however, support the specification of unidirectional *element* translations. It is impossible to specify that a particular *inter\_class* or even that a particular equivalence within an *inter\_class* is unidirectional. Indeed, an example given by Amor (1997, Section E.3.1) suffers from the latter problem — equivalences in VML are always bidirectional, but one of the equivalences in the example is noted in a comment as only working in one direction.

An example of the first problem is shown in Figure 7.3(a), in which is shown part of the *inter\_class* definition for the following heuristic from the  $\mathfrak{R}_e(E-R, ERD_{Martin}) \rightleftharpoons \mathfrak{R}_r(Relational, SQL/92)$  translation:

$$\mathfrak{R}_e [\text{MARTINIDENTIFIER}] \leftarrow \mathfrak{R}_r [\text{SQL92UNIQUE}] \quad (\text{H1})$$

Heuristics are always unidirectional, but this cannot be specified in VML.

An example of the second problem is shown in Figure 7.3(b), in which is shown part of the *inter\_class* definition for the following rule from the  $\mathfrak{R}_f(FuncDep, FDD_{Smith}) \rightarrow \mathfrak{R}_e(E-R, ERD_{Martin}) / \mathfrak{R}_f \leftarrow \mathfrak{R}_e$  translation:

$$\begin{aligned} \mathfrak{R}_f [\text{FDFUNCTIONALSOURCE}, \text{FDFUNCTIONAL}, \\ \text{FDFUNCTIONALTARGET}] \rightarrow \mathfrak{R}_e [\text{ERENTITYTYPE}] \\ \mathfrak{R}_f [\text{FDFUNCTIONALSOURCE}, \text{FDFUNCTIONAL}, \\ \text{FDFUNCTIONALTARGET}] \leftarrow \mathfrak{R}_e [\text{ERENTITYTYPE}] \end{aligned} \quad (\text{T1})$$

The equivalences of this rule include a call to the user-defined function `get_nonkey`, which extracts the non-key attributes from a `MARTINREGULARENTITY` element. This function only makes sense in the reverse direction, as it only applies to constructs in  $\mathfrak{R}_e$ . The unidirectional nature of this equivalence cannot be specified in VML.

What is therefore needed is some way to specify the direction of both *inter\_class* definitions and individual equivalences within an *inter\_class*. The translation process will use this information to apply only those rules and equivalences that are appropriate to the direction of the translation being executed; rules and equivalences that are unidirectional in the opposite direction are ignored. The direction of an element translation is specified in VML-S using the new `direction` clause of the *inter\_class* definition, and the directions of individual equivalences are specified using one of three new equivalence operators.

---

```

inter_class
( [martinidentifier], [sql92unique],
  invariants
  ( martinidentifier.partial = false )
  equivalences
  ( martinidentifier.name = sql92unique.name,
    martinidentifier.attributes[] = sql92unique.columns[],
    martinidentifier.identifiedItem = sql92unique.columns[]=>relation
  )
).

```

(a) Unidirectional *inter\_class* definition

```

inter_class
( [fdfunctionalsource, fdfunctional, fdfunctionaltarget], [erentitytype],
  invariants
  ( fdfunctionalsource = fdfunctionalsource,
    fdfunctional.destination = fdfunctionaltarget,
    ...
  ),
  equivalences
  ( fdfunctional.name = erentitytype.name,
    ...
    get_nonkey(fdfunctionaltarget.attributes, erentitytype.attributes)
  ),
  initialisers
  ( ... )
).

```

(b) Unidirectional *equivalence* clause

---

**Figure 7.3:** Unidirectional translations that cannot be fully specified using VML

### The direction clause

The optional `direction` clause allows the specification of the direction(s) in which a rule may be applied. The allowable values are `=>>` (unidirectional forward), `<<=` (unidirectional reverse) and `<=>` (bidirectional). If this clause is omitted, `<=>` is assumed. The syntax and use of this clause is shown in Figure 7.4 on the next page.

### Enhanced equivalence syntax

The `direction` clause addresses the need to specify the direction of an *inter\_class* definition, but does not address the need to specify the direction of individual equivalences. The following binary equivalence operators are therefore proposed:

**a = b** means that the mapping between a and b may be applied in the same direction(s) as specified by the `direction` clause of the *inter\_class*.

**a <=> b** means that the mapping between a and b may be applied in both forward and reverse directions.

---

**Syntax:**         $\langle \text{direction} \rangle ::= \text{direction}(\langle \text{dir\_map\_op} \rangle)$   
                   $\langle \text{dir\_map\_op} \rangle ::= \Rightarrow | \Leftarrow | \Leftrightarrow$

**Example:**

```
inter_class
( [martinidentifier], [sql92unique],
  direction (<<=),
  invariants
  ( martinidentifier.partial = false )
  equivalences
  ( martinidentifier.name = sql92unique.name,
    martinidentifier.attributes[] = sql92unique.columns[],
    martinidentifier.identifiedItem = sql92unique.columns[]=>relation
  )
).
```

---

**Figure 7.4:** The `direction` clause

**a  $\Rightarrow$  b** means that the mapping between a and b may only be applied in the forward direction.

**a  $\Leftarrow$  b** means that the mapping between a and b may only be applied in the reverse direction.

These operators apply to equivalences that contain simple equations, but they do not deal with equivalences that contain bijections, function calls (predicates) or procedural mapping specifications (using the built-in `map_to_from` predicate).

In mathematical terms, a *bijection* is a mapping  $f : S \rightarrow T$  in which a distinct element of  $S$  maps to a distinct element of  $T$ , and every element of  $T$  can be mapped from at least one element of  $S$  (Clapham, 1990)<sup>1</sup>. VML bijections are derived from this definition and are effectively bidirectional conditional mappings (Amor, 1997, Section 5.2.6.10), so some way of specifying unidirectional bijections must be defined. (The concept of a ‘unidirectional bijection’ may at first seem contradictory, but the ‘bi-’ in ‘bijection’ refers to the fact that a bijection is both an injection and a surjection, so the usage ‘unidirectional bijection’, while somewhat counter-intuitive, is correct.)

The simplest way in which to specify unidirectional bijections is to allow the three new equivalence operators ( $\Leftrightarrow$ ,  $\Rightarrow$  and  $\Leftarrow$ ) to be applied as unary prefix operators, for example,  $\Rightarrow \text{bijection}(\dots, \dots)$ . This approach also has the advantage that the same syntax may be used for unidirectional function calls, as shown in Figure 7.5.

---

<sup>1</sup>Mappings that have only the former property are known as *one-to-one* or *injective*, while mappings that have only the latter are referred to as *onto* or *surjective*.



The `map_to_from` function for specifying procedural mappings does not need to be altered as it is already possible to specify a unidirectional procedural mapping by passing `true` as one of the arguments to `map_to_from` (Amor, 1997, p. 96).

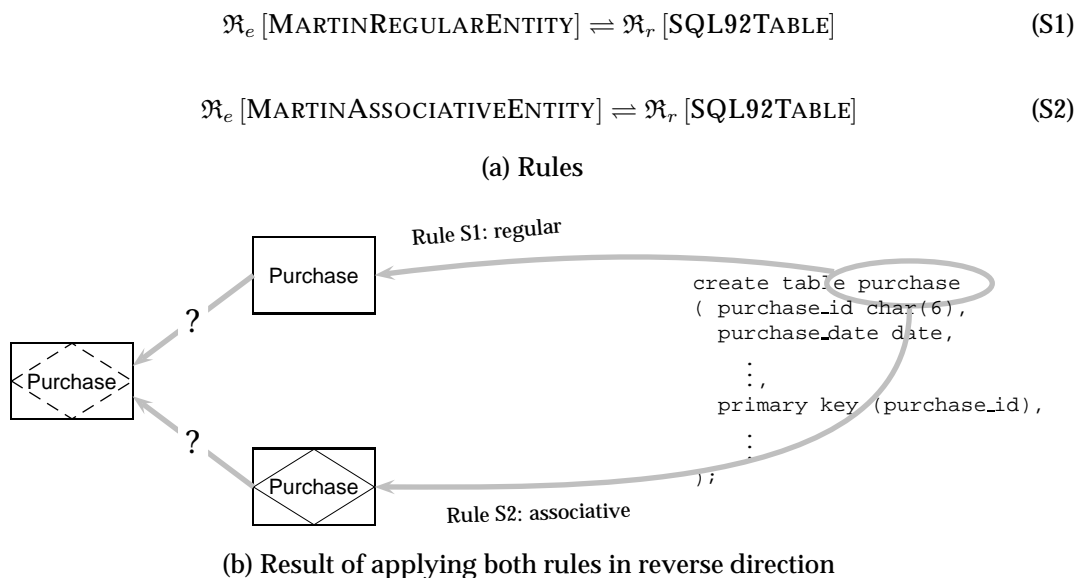
Equivalences that contradict the direction of the containing *inter\_class* are ignored rather than treated as errors, as it is sometimes possible for specialisations of a bidirectional technique-level rule to be unidirectional (for example, rule S3 of the  $\mathfrak{R}_e \rightleftharpoons \mathfrak{R}_d$  translation). The syntax and use of the new equivalence operators is shown in Figure 7.5.



**Figure 7.5:** New directional equivalence operators

### 7.3.2 Rule exclusion

Earlier in Section 4.7.1 on page 98, the problem and importance of rule exclusion was discussed. VML provides no mechanism for dealing with excluded rules. Consider the two rules from the  $\mathfrak{R}_e \rightleftharpoons \mathfrak{R}_r$  translation shown in Figure 7.7 (rule S2 has been made bidirectional in this example for the purposes of illustration). Each rule subsumes the other in the reverse direction, so there is potential for the rules to conflict. If both these rules were applied in the reverse direction, the current VML translation process would create an element that was either a regular entity or an associative entity, depending on the order in which the rules were applied. There is no guarantee that the rules will be applied in the same order every time, so this situation is undesirable. The first solution suggested in Section 4.7.1 was to choose one rule to be bidirectional and make the remaining rules unidirectional. This solution was adopted for the definition of rule S2 in Chapter 5, but it is, of course, impossible to specify this in VML as VML does not allow the explicit specification of unidirectional rules.



**Figure 7.7:** An example of rule exclusion

The new `direction` clause introduced in VML-S could be used to specify rule S2 as unidirectional, but a more general solution suggested in Section 4.7.1 is to explicitly specify the exclusions among rules, which is a feature that VML also does not support. This second solution is preferable in general, as the first solution is not appropriate for

all excluded rules, and also tends to obscure the exclusions among rules. That is, rules are made unidirectional for no other reason than that they are excluded by some other rule.

The introduction of explicit exclusions among rules requires the ability to uniquely identify a particular rule. In VML, an *inter\_class* definition is identified by its header, which is the combination of the two construct lists. Several *inter\_class* definitions may share the same header, so they are further distinguished by their invariants (Amor, 1997, Section 5.2.1). Rule headers may therefore not be unique, and including the invariants as part of the identification is obviously not practical, as the number of invariants may vary considerably for each rule. Some other means of uniquely and concisely identifying rules is required, which is provided in VML-S by the `label` clause of the *inter\_class* definition. Rule exclusions are specified using the `excludes` clause of the *inter\_class* definition.

### The `label` clause

The optional `label` clause is similar in principle to the `label` statement in many programming languages, and allows *inter\_class* definitions to be explicitly named. The syntax and use of this clause is shown in Figure 7.8.

---

**Syntax:**                    `<label> ::= label( <inter_class_id> )`  
                              `<inter_class_id> ::= <simple_id>`

**Example:** `inter_class`  
`( [eridentifier], [ralternatekey],`  
`label (t6_id_ak),`  
`direction (<<=),`  
`:`  
`).`

---

**Figure 7.8:** The `label` clause

Labels may be used anywhere that an *inter\_class* header is used to identify a rule; at present, the only such places are in the `inherits` and `excludes` clauses. Label names must be unique within an *inter\_view* definition.

## The `excludes` clause

The optional `excludes` clause allows the explicit specification of rule exclusions for each direction of a translation. It specifies a list of rules, identified by their labels, that are excluded by the current rule in a particular direction. If the current rule is applied in the specified direction, the excluded rules should not be applied. If a direction is not specified for a rule, then the exclusion is assumed to apply in both directions. The syntax and use of this clause is shown in Figure 7.9.

---

**Syntax:**             $\langle \text{excludes} \rangle ::= \text{excludes}(\langle \text{inter\_class\_id} \rangle [\langle \text{dir\_map\_op} \rangle] \{, \langle \text{inter\_class\_id} \rangle [\langle \text{dir\_map\_op} \rangle] \})$

**Example:**

```
inter_class
( [martinrelationship, ertypeitem[2]],
  [sql92foreignkey[2], sql92unique[2]],
  label (s12_rel_0101),
  inherits (relationship_ll_base),
  excludes (h1_id_unique <=>, s10_rel_01xN <=>),
  invariants
    :
  ).
```

---

**Figure 7.9:** The `excludes` clause

The VML-S translation process will obviously need to be extended to take advantage of this new rule exclusion information. Rules that exclude others should be considered before the rules that they exclude, that is, rule exclusions imply an ordering to rule evaluation. Compare this with the VML translation process, where rules may (in theory) be applied in arbitrary order. The subsumption/exclusion graph introduced in Section 4.7.1 provides a means of describing this ordering information, and is incorporated into the extended translation algorithms in Section 7.5.

### 7.3.3 Translating construct lists

The header of an *inter\_class* definition comprises two lists of constructs, one for each representation. Within these lists, constructs may be:

- individual, representing a single element (for example, `[some_construct]`);
- grouped, representing an unordered set of elements of the same type (for example, `[group(some_construct)]`); or

- ‘listed’, representing an ordered list of elements of the same type (for example, `[some_construct[]]`).

The existing VML syntax allows the specification of all three, but Amor’s VML mapping system implementation handles only the first two cases, and no algorithms are defined for dealing with the third case.

Grouped constructs cannot be used in place of lists, as a grouped construct is an unordered set, whereas a list is ordered. A typical reason for using a list as opposed to a grouped construct would be when a specific number of identifiable elements are to be translated (for instance, two `ERENTITYTYPE` elements), where each has specific properties depending on their position in the list. The VML-S mapping system algorithms are therefore extended in Section 7.5.2 to handle ‘listed’ constructs in an *inter\_class* header.

### 7.3.4 Using the same construct multiple times

While defining the  $\mathfrak{R}_f(\text{FuncDep}, \text{FDD}_{\text{Smith}}) \rightarrow \mathfrak{R}_e(\text{E-R}, \text{ERD}_{\text{Martin}})$  translation (see Appendix E), the author encountered one heuristic that could not be specified declaratively using VML. Heuristic H2 translates a collection of attributes tagged with the same domain flag into an E-R type hierarchy, and is denoted by:

$$\begin{aligned} &\mathfrak{R}_f[\text{FDATTRIBUTESET}_1, \dots, \text{FDATTRIBUTESET}_n, \\ &\quad \text{SSDOMAINFLAG}, \text{SSSINGLEKEYBUBBLE}, \text{SSATTRIBUTE}_a, \\ &\quad \text{SSATTRIBUTE}_1, \dots, \text{SSATTRIBUTE}_n] \rightarrow \mathfrak{R}_e[\text{MARTINTYPEHIERARCHY}] \end{aligned} \quad (\text{H2})$$

Ideally, it would be possible to define an *inter\_class* header with a single `SSATTRIBUTE` construct (representing `SSATTRIBUTEa` above) and a separate list of `SSATTRIBUTE` constructs (representing `SSATTRIBUTE1, …, SSATTRIBUTEn` above), that is:

```
inter_class
( [fdattributeset[], ssdomainflag, sssinglekeybubble, ssattribute, ssattribute[]],
  [martintypehierarchy],
  ...
).
```

Unfortunately, VML allows each distinct construct to appear only once in the *inter\_class* header, so `ssattribute` cannot be repeated. This is not an unreasonable restriction — the *inter\_class* definition above would be very confusing to work with, as it

would be difficult (if not impossible) to determine which `ssattribute` construct was being referred to in the body of the *inter\_class*. Thus, in standard VML the definition must be written as:

```
inter_class
( [fdattributeset[], ssdomainflag, sssinglekeybubble, ssattribute[]],
  [martintypehierarchy],
  ...
).
```

Here, construct `SSATTRIBUTEa` is included within the list of `SSATTRIBUTE` constructs, which can make access to `SSATTRIBUTEa` difficult. It could be assumed that the first item of the list is `SSATTRIBUTEa`, while the remainder of the list contains the remaining `SSATTRIBUTE` constructs. Unfortunately, the only list iteration mechanism that VML provides is the declarative `[]`-form that iterates over an entire list. That is, there is no simple declarative mechanism in VML that would allow the first item of a list to be treated separately from the rest.

This restriction is particularly relevant to the specification of this heuristic because the `SSATTRIBUTE` constructs would only be referred to in the invariants. VML does not allow procedures in the invariants section (Amor, 1997, p. 96), but user-defined functions are allowed if they reference attributes from only one schema. Thus, it might be possible to specify some of this heuristic procedurally using VML.

The solution adopted in VML-S is to allow the specification of *aliases* for construct names in an *inter\_class* header, in much the same way as SQL allows the definition of aliases for table names in a `select` statement (Groff and Weinberg, 1994, p. 153). Constructs in a VML-S *inter\_class* header may have an alias name associated with them, which allows the same construct to appear multiple times in a single header, but with different names. The syntax and use of construct aliases is shown in Figure 7.10. Alias names must be unique within an *inter\_class*.

## 7.4 Converting the abstract notation into VML-S

The abstract translation notation defined in Chapter 4 is convenient for expressing translations at a high level, and also provides a useful template for building VML-S translation specifications. A description translation corresponds directly to a VML-S

---

**Syntax:**

$$\langle \text{class\_list\_name} \rangle ::= \text{group}(\langle \text{class\_name} \rangle) [\langle \text{alias\_name} \rangle]$$

$$| \langle \text{class\_name} \rangle [\langle \text{alias\_name} \rangle]$$

$$\langle \text{alias\_id} \rangle ::= \langle \text{simple\_id} \rangle$$

$$\langle \text{alias\_name} \rangle ::= \langle \text{alias\_id} \rangle [ [ \langle \text{list\_id} \rangle \{ , \langle \text{list\_id} \rangle \} ] ]$$

**Example:**

```

inter_class
( [fdattributeset[], ssdomainflag, sssinglekeybubble,
  ssattribute parent, ssattribute[] children],
  [martintypehierarchy],
  :
  member(children[], fdattributeset[].attributes),
  member(parent, sssinglekeybubble.attributes),
  :
  ).

```

---

**Figure 7.10:** Aliases for constructs in the *inter\_class* header

*inter\_view* definition, for example, the description translation expression:

$$\mathfrak{R}_e(E-R, ERD_{Martin}) \mapsto \mathfrak{R}_d(DataFlow, DFD_{G\&S})$$

corresponds to the following *inter\_view* definition:

```
inter_view(er_martinerd, read_only, df_ganesarsondfd, read_write, partial).
```

All items of the *inter\_view* definition may be determined directly from the high-level notation:

- the *model\_id* may be built from the names of the representation's technique and scheme;
- the direction of the translation determines the *model\_type* for both representations — in a unidirectional translation the source representation is `read_only` and the target representation is `read_write`, whereas in a bidirectional translation both representations are `read_write`; and
- the completeness of the translation determines the *map\_type*, either `partial` or `complete`.

An element translation (rule or heuristic) corresponds directly to the header of a VML-S *inter\_class* definition, for example, the rule:

$$\mathfrak{R}_e[\text{MARTINASSOCIATIVEENTITY}] \mapsto \mathfrak{R}_d[\text{GNSDATASTORE}]$$

corresponds to the header of the following *inter\_class* definition:

```
inter_class
( [martinassociativeentity], [gnsdatastore],
  label (some_label),
  direction (=>>),
  ...
).
```

The direction of the *inter\_class* is determined from the direction of the translation operator in the notation, as shown above.

The way in which constructs are arranged in the *inter\_class* header is determined from the high-level notation as follows:

- An individual construct in the notation corresponds to an individual construct in the VML-S specification.
- A fixed-size collection of constructs of the same type in the notation corresponds to a fixed-length list of that construct in VML-S.
- An arbitrarily-sized but ordered collection of constructs of the same type corresponds to an unspecified-length list of that construct in VML-S.
- An arbitrarily-sized and unordered collection of constructs of the same type in the notation corresponds to a `group` of that construct in VML-S.

Thus, the element translation expression<sup>2</sup>:

$$\mathfrak{R}_e(E-R, ERD_{Martin})[\text{MARTINREGULARENTITY}, \text{MARTINIDENTIFIER}, \\ \text{MARTINATTRIBUTE}_m, \dots, \text{MARTINATTRIBUTE}_n] \rightarrow \\ \mathfrak{R}_f(FuncDep, FDD_{Smith})[\text{FDATTRIBUTESET}_1, \text{FDATTRIBUTESET}_2, \\ \text{SSSINGLEVALUED}]$$

corresponds to the following *inter\_class* definition (assuming that the `MARTINATTRIBUTE` constructs are not ordered):

```
inter_class
( [martinregularentity, martinidentifier, group(martinattribute)],
  [fdattributeset[2], sssinglevalued],
  direction (=>>),
  ...
).
```

---

<sup>2</sup>This translation has been invented for the purposes of this example — it does not appear in the  $\mathfrak{R}_e \rightarrow \mathfrak{R}_f$  description translation.



The invariants, and to some extent the initialisers, are used to specify the pre- and post-conditions of a rule, while the equivalences specify the precise mappings between constructs in terms of their properties (construct properties were discussed in Section 3.2 on page 42). These parts of the *inter\_class* definition cannot be derived from the high-level notation, which is to be expected — the abstract notation deliberately does not describe any of this information for the reasons discussed in Section 4.4.2 on page 80.

The VML-S implementation of a rule defined in the previous chapter will now be examined in more detail.

### 7.4.1 Specification of $\mathfrak{R}_e \rightleftharpoons \mathfrak{R}_r$ rule S8

This rule, when applied in the forward direction ( $\rightarrow$ ), expands a many-to-many relationship between two E-R entities by introducing an intermediate table that links the two corresponding SQL tables, as illustrated in Figure 7.11 on the next page. In the reverse direction ( $\leftarrow$ ), any such intermediate table that is found is translated into a many-to-many relationship. The definition of this rule is:

$$\begin{aligned}
 &\mathfrak{R}_e[\text{MARTINRELATIONSHIP}, \text{ERTYPEITEM}_1, \text{ERTYPEITEM}_2] \rightarrow \\
 &\quad \mathfrak{R}_r[\text{SQL92TABLE}, \text{SQL92PRIMARYKEY}, \\
 &\quad \quad \text{SQL92FOREIGNKEY}_1, \text{SQL92FOREIGNKEY}_2] \\
 &\mathfrak{R}_e[\text{MARTINRELATIONSHIP}, \text{ERTYPEITEM}_1, \text{ERTYPEITEM}_2] \leftarrow \\
 &\quad \mathfrak{R}_r[\text{SQL92TABLE}, \text{SQL92PRIMARYKEY}, \\
 &\quad \quad \text{SQL92FOREIGNKEY}_1, \text{SQL92FOREIGNKEY}_2]
 \end{aligned} \tag{S8}$$

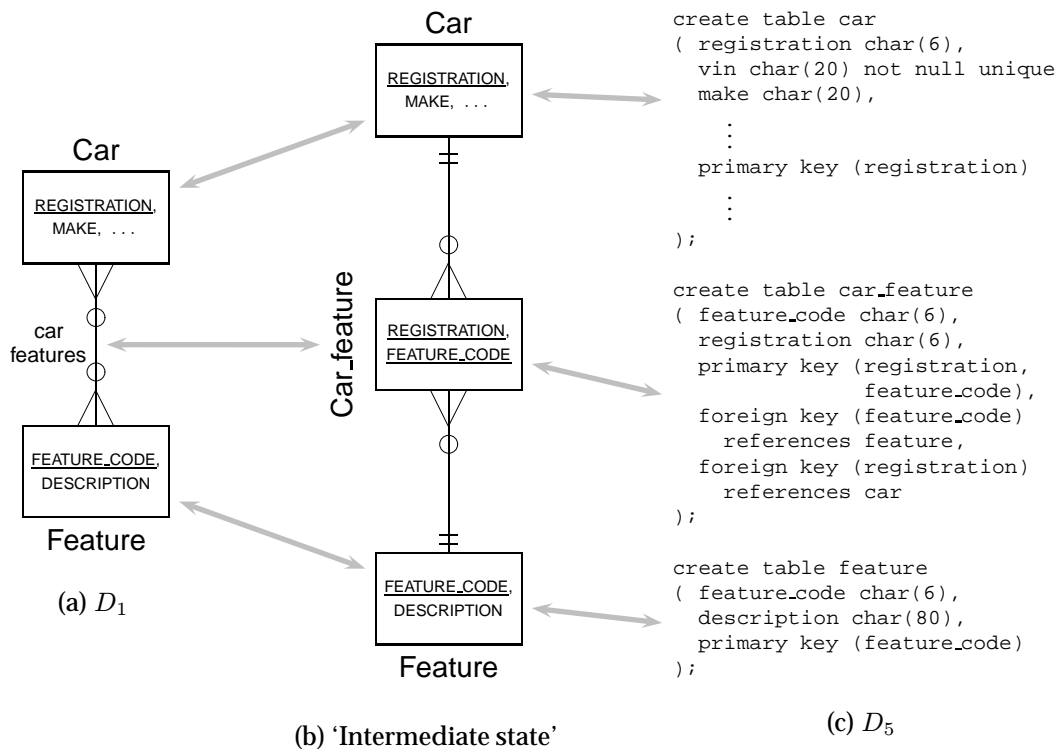
Applying the correspondences identified above, this rule can be converted to the following initial VML-S *inter\_class* definition:

```

inter_class
( [martinrelationship, ertypeitem[2]],
  [sql92table, sql92primarykey, sql92foreignkey[2]],
  label (s8_rel_xMxN),
  ...
).

```

(Alternatively, construct aliases could be used instead of a list of ERTYPEITEM constructs, but this would not have any real effect on the resultant rule specification.)



**Figure 7.11:** Translating many-to-many relationships using  $\mathfrak{R}_e \rightleftharpoons \mathfrak{R}_r$  rule S8 (used cars viewpoint)

Examining Figure 7.11, it can be seen that the elements of  $D_1$  corresponding to the constructs in the *inter\_class* header are:

- martinrelationship = relationship car features;
- ertypeitem[1] = Car and ertypeitem[2] = Feature (or vice versa);
- sql92table = table car\_feature;
- sql92primarykey = the primary key of table car\_feature;
- sql92foreignkey[1] = the foreign key from car\_feature to car (or vice versa); and
- sql92foreignkey[2] = the foreign key from car\_feature to feature (or vice versa).

The first step is to determine any invariants that apply to the mapping. First, the relationship (car features) must be specified as a many-to-many relationship. Cardinality

and optionality are specified in the representation  $\mathfrak{R}_e(E-R, ERD_{Martin})$  by the `srcOpt`, `srcCard`, `dstOpt` and `dstCard` properties of the `MARTINRELATIONSHIP` construct. Since the optionality of both ends of the relationship is lost in the forward direction, there is no need to specify this (in the reverse direction, initialisers are used to set the optionality of the relationship). The `srcCard` and `dstCard` properties should, however, be greater than one, that is:

```
invariants
( martinrelationship.srcCard > 1,
  martinrelationship.dstCard > 1,
  ...
```

Next, the entities must be correctly associated with the relationship using the source and destination properties of `MARTINRELATIONSHIP`. In this example, it will be assumed that `Car` is the ‘source’ entity of the relationship and `Feature` is the ‘target’ entity. That is:

```
...
martinrelationship.source = ertypeitem[1],
martinrelationship.target = ertypeitem[2],
...
```

On the SQL side of the translation, the primary key element must be associated with the newly generated table `car_feature`. In addition, the columns of the primary key and the columns of the table must be identical. Thus:

```
...
sql92table.primaryKey = sql92primarykey,
sql92table.attributes[] = sql92primarykey.attributes[],
...
```

Similarly, the foreign keys must be associated with the `car_feature` table. In addition, the concatenation of the columns of both foreign keys should be the same as the columns of the primary key. That is:

```
...
member(sql92foreignkey[], sql92table.foreignKeys),
sql92foreignkey[].relation = sql92table,
append(sql92foreignkey[2].columns, sql92foreignkey[1].columns,
       sql92primarykey.attributes)
)
```

Now the equivalences between the various constructs and their properties must be identified. First, the name of the relationship (`car features`) corresponds to the name of the intermediate table (`car_feature`). Similarly, the names for the foreign keys may

be built from the names of the entities (the representation definition includes names for many constructs even though these are not displayed). For instance, a name for the foreign key from `car_feature` to `car` could be built by appending the string `'_fk'` to the name of the `Car` entity, giving `'car_fk'`. Note, however, that this is only applicable in the forward direction. In the reverse direction the names of the entities are not derived from the names of the foreign keys; rather they are mapped from the names of the tables by rule S1. Thus:

```

equivalences
( martinrelationship.name = sql92table.name,
  =>> append('_fk', ertypeitem[1].name, sql92foreignkey[1].name),
  =>> append('_fk', ertypeitem[2].name, sql92foreignkey[2].name),
  ...

```

Next, the columns of the `car_feature` table are determined by concatenating the attributes of the entity identifiers of `Car` and `Feature`. The entity identifier of `Car` comprises the single attribute `registration` and the entity identifier of `Feature` comprises the attribute `feature_code`, so the columns of `car_feature` are `registration` and `feature_code`. As with the foreign key names, this mapping is only applicable in the forward direction; in the reverse direction, the attributes of `Car` and `Feature` are again determined by rule S1. This mapping cannot be dealt with by a simple equation, as two lists of attributes must be concatenated then mapped. It may be possible to do this using a temporary variable, but the approach taken here is to define the mapping as a user-defined function `combine_attr`, which takes as arguments the identifiers of the two entities and produces an SQL table with appropriate columns.

```

...
=>> combine_attr(ertypeitem[1].identifier, ertypeitem[2].identifier, sql92table),
...

```

Next, the entities `Car` and `Feature` must be mapped to the tables `car` and `feature`. Entities are already translated by other rules, so there is no particular need to duplicate this here. Instead, all that is required is to specify that the `Car` entity maps to the table referenced by the `car_fk` foreign key, and `Feature` maps to the table referenced by the `feature_fk` foreign key. A similar situation applies with the identifiers of the two entities, which map to the referenced primary keys of the foreign keys. That is:

```

...
ertypeitem[1] = sql92foreignkey[1].refTable,
ertypeitem[2] = sql92foreignkey[2].refTable,
ertypeitem[1].identifier = sql92foreignkey[1].refPK,
ertypeitem[2].identifier = sql92foreignkey[2].refPK,
...

```

The last equivalence that needs to be dealt with is to ensure that the attributes of the Car identifier map to the columns of `car_fk`, and similarly with Feature and `feature_fk`:

```
...
  ertypeitem[1].identifier=>attributes[] = sql92foreignkey[1].columns[],
  ertypeitem[2].identifier=>attributes[] = sql92foreignkey[2].columns[]
),
```

Finally, information that is required by the target description but cannot be extracted from the source description is dealt with using initialisers. First, the optionality of the relationship must be determined when translating in the reverse direction. Since the required optionality information cannot be extracted from SQL, the only solution is to choose some suitable default and apply that in an initialiser. A reasonable default for optionality is to make it optional. The cardinality of the relationship is already known to be many, but the invariants only specify it as being greater than one. Initialisers can be used to set the cardinality to two. Thus:

```
initialisers
( martinrelationship.srcCard = 2,
  martinrelationship.dstCard = 2,
  martinrelationship.srcOpt = 0,
  martinrelationship.dstOpt = 0
)
```

The complete rule definition is shown in Table 7.1 on the following page. Once the other rules in the translation have been defined, it is possible to check for conflicts among the rules and thus determine any exclusions among the rules. This particular rule conflicts with rules S1, S4 and S10 in the reverse direction. An appropriate `excludes` clause has been added to the rule definition shown in Table 7.1.

## 7.5 Extensions to the translation process

In this section are presented the extensions to the translation process required to support the VML-S extensions described above. An overview of the original VML mapping process was presented in Section 4.7 on page 94. Amor (1997) implemented this process in four passes, as shown in Figure 7.12 on page 203.

VML has incomplete support for unidirectional translations and no support for rule exclusion. The addition of rule and equivalence directions and rule exclusion provides

**Table 7.1:** VML-S specification of  $\mathfrak{R}_e \rightleftharpoons \mathfrak{R}_r$  rule S8

---

```

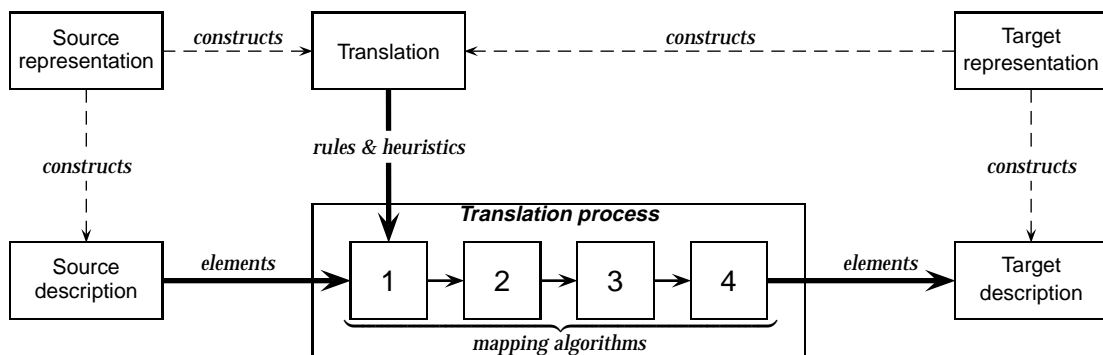
inter_class
( [martinrelationship, ertypeitem[2]], [sql92table, sql92primarykey, sql92foreignkey[2]],
  label (s8_rel_xMxN),
  excludes (s1_regular_table <=<, s4_id_pk <=<, s10_rel_01xN <=<),
  invariants
  ( martinrelationship.srcCard > 1,
    martinrelationship.dstCard > 1,
    martinrelationship.source = ertypeitem[1],
    martinrelationship.target = ertypeitem[2],
    sql92table.primaryKey = sql92primarykey,
    sql92table.attributes[] = sql92primarykey.attributes[],
    member(sql92foreignkey[], sql92table.foreignKeys),
    sql92foreignkey[].relation = sql92table,
    append(sql92foreignkey[2].columns, sql92foreignkey[1].columns,
      sql92primarykey.attributes)
  ),
  equivalences
  ( martinrelationship.rolename = sql92table.name,
    =>> append('_fk', ertypeitem[1].name, sql92foreignkey[1].name),
    =>> append('_fk', ertypeitem[2].name, sql92foreignkey[2].name),
    =>> combine_attr(ertypeitem[1].identifier, ertypeitem[2].identifier, sql92table),
    ertypeitem[1] = sql92foreignkey[1].refTable,
    ertypeitem[2] = sql92foreignkey[2].refTable,
    ertypeitem[1].identifier = sql92foreignkey[1].refPK,
    ertypeitem[2].identifier = sql92foreignkey[2].refPK,
    ertypeitem[1].identifier=>attributes[] = sql92foreignkey[1].columns[],
    ertypeitem[2].identifier=>attributes[] = sql92foreignkey[2].columns[]
  ),
  initialisers
  ( martinrelationship.srcCard = 2,
    martinrelationship.dstCard = 2,
    martinrelationship.srcOpt = 0,
    martinrelationship.dstOpt = 0,
    <=< append(ertypeitem[2].name, ertypeitem[1].name, martinrelationship.name)
  )
).

```

---

an extra method of filtering out rules and equivalences that cannot be applied to a collection of source elements, and rule exclusion also implies an ordering to rule evaluation. The information required to perform this filtering and ordering is described by the subsumption/exclusion graphs for the translation. Thus, before any translation can be performed, the subsumption/exclusion graphs for that translation must be generated. Subsumption/exclusion graphs may be generated at any time before a translation takes place, so this step is separate from the translation process. Graphs may be saved for future re-use and will only need to be regenerated if the translation is changed. Algorithms for constructing subsumption/exclusion graphs are presented in Section 7.5.1.

The original VML translation algorithm (Amor, 1997, Table 10.3) has been extended to make use of the information stored in the subsumption/exclusion graphs, pro-



**Pass Steps**

- One:** determine all combinations to be mapped from the source description  
create elements for first construct in *inter\_class* header specification  
apply appropriate initialisers, equivalences and invariants
- Two:** attach or create elements for all other constructs 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 resolution  
re-calculate determined equivalences

**Figure 7.12:** Amor's (1997) four-pass translation process

ducing Algorithm 7.1 on the next page; the extensions to the original algorithm are underlined. The main points to note in this algorithm are:

1. An ordering step has been added on lines 8–9 that uses the subsumption/exclusion graph to determine in what order rules should be evaluated. The rules are ordered such that non-subsumed, non-excluded rules are evaluated first, followed by subsumed but non-excluded rules, followed by subsumed and excluded rules.
2. Exclusion handling has been added on lines 23–24 and 29–30, which removes excluded rules from consideration when the rule that excludes them is applied.
3. Direction checking has been added in several places in order to remove inappropriate rules and equivalences from consideration (for example, lines 5–7).
4. The modified algorithm only details the process for *create* mappings, that is, mappings that create new descriptions/elements rather than modifying or deleting

existing ones. This is partly because the processes for *delete* and *modify* mappings are similar to those for *create* mappings, and partly because the full implications of *delete* and *modify* mappings have not been examined in this thesis.

---

Algorithm 7.1: Modified translation algorithm

---

PERFORMMAPPINGS(*d*)

**Input:** The direction of the translation *d*.

**Output:**

```

(1)  foreach aggregated mapping to be mapped
(2)    switch aggregated mapping
(3)    case create:
(4)       $K \leftarrow$  newly created (key) element
(5)       $I \leftarrow$  find inter_class definitions referencing construct of K
(6)        (or one of its generalisations) and that appear in the
(7)        subsumption/exclusion graph for d
(8)      order I according to the information in the
(9)      subsumption/exclusion graph for d
(10)     foreach inter_class r in I
(11)        $G \leftarrow$  INITIALELEMENTGROUPS(K, r)
(12)        $C \leftarrow$  GENERATECOMBINATIONS(G, r)
(13)       foreach combination in C
(14)         if K is grouped in r's header
(15)           match with existing mappings of this type
(16)           re-evaluate grouped elements
(17)           re-evaluate invariants if K's construct involved
(18)           if invariants are violated
(19)             dissolve existing mapping
(20)           else
(21)             re-calculate affected equivalences that have
(22)             direction d or  $\leq \Rightarrow$ 
(23)             if r excludes other inter_classes
(24)               remove excluded inter_classes from I
(25)           else
(26)             create required elements in other data store
(27)             apply initialisers that have direction d or  $\leq \Rightarrow$ 
(28)             solve equivalences that have direction d or  $\leq \Rightarrow$ 
(29)             if r excludes other inter_classes
(30)               remove excluded inter_classes from I
(31)       case modify:
(32)         ...
(33)       case delete:
(34)         ...

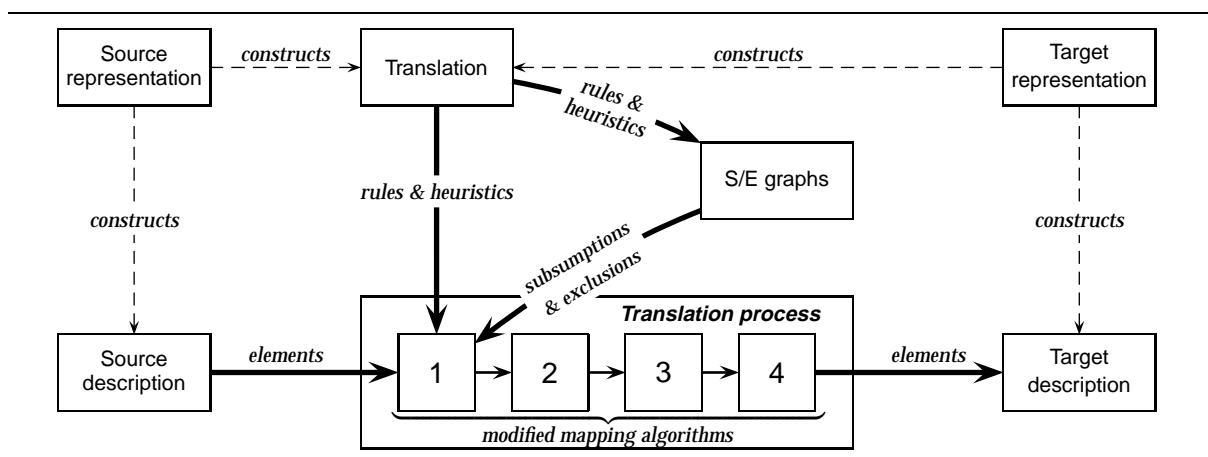
```

---

The translation process must also be extended to handle 'listed' constructs in an *inter\_class* header, as discussed in Section 7.3.3. Appropriately extended versions of Amor's (1997) algorithms are presented in Section 7.5.2.



In summary, the only major new step in performing a translation is the generation of the subsumption/exclusion graphs, as shown in Figure 7.13. The four-pass translation process remains unchanged, but the algorithms within this process have been modified to take advantage of the subsumption/exclusion graphs, and also to handle ‘listed’ constructs in *inter\_class* headers. A demonstration of these extended algorithms may be found in Appendix F.



**Figure 7.13:** The modified translation process

### 7.5.1 Building subsumption/exclusion graphs

The addition of the `excludes` clause to the *inter\_class* definition makes it possible to define the exclusions among rules, from which the subsumption/exclusion graphs may be built using Algorithm 7.2 on the following page.

An important part of Algorithm 7.2 is detecting whether one rule subsumes another. A rule  $r_i$  subsumes another rule  $r_j$  in a particular direction if the source constructs of  $r_j$  are a subset ( $\subseteq$ ) of the source constructs of  $r_i$ , and the invariants of  $r_i$  do not contradict those of  $r_j$  (see Section 4.7.1 on page 98). Algorithm 7.3 on the following page specifies how to detect rule subsumption. Note that invariant checking is not detailed in this algorithm, as this has already been dealt with by Amor (1997). This algorithm also assumes that both rules may be applied in the same direction, so any calling algorithm must ensure that this condition holds. This algorithm has been implemented with simulated invariant checking to verify that it produces correct results.

---

**Algorithm 7.2: Build a subsumption/exclusion graph for a translation**

---

**Input:** A collection of VML-S rule definitions  $R = \{r_1, r_2, \dots, r_n\}$ , and a direction  $d$ .

**Output:** A subsumption/exclusion graph  $G_d = (\mathcal{V}, \mathcal{E})$  for  $R$  in direction  $d$ .

BUILDSEGRAPH( $R, d$ )

$\mathcal{V} \leftarrow \emptyset$

$\mathcal{E} \leftarrow \emptyset$

**foreach** rule  $r_i$  of  $R$  whose direction is  $d$  or  $\langle = \rangle$

  create vertex  $v_i$  in  $\mathcal{V}$

**foreach** rule  $r_i$  of  $R$  whose direction is  $d$  or  $\langle = \rangle$

**foreach** rule  $r_j$  of  $R$  whose direction is  $d$  or  $\langle = \rangle$ ,  $i \neq j$

**if** SUBSUMES( $r_i, r_j$ )

**if**  $r_j$  is in  $r_i$ 's exclusion list

        create edge  $e_{ij} : v_i \xrightarrow{p} v_j$  in  $\mathcal{E}$

**else**

        create edge  $e_{ij} : v_i \rightarrow v_j$  in  $\mathcal{E}$

**return** ( $\mathcal{V}, \mathcal{E}$ )

---

---

**Algorithm 7.3: Determine whether one rule subsumes another**

---

**Input:** Two rules  $r_i$  and  $r_j$ ,  $i \neq j$ , and a direction  $d$  in which both rules may be applied.

**Output:** **True** if  $r_i$  subsumes  $r_j$  in direction  $d$ , otherwise **false**.

SUBSUMES( $r_i, r_j, d$ )

$\sigma_i \leftarrow$  source constructs of  $r_i$  with respect to  $d$

$\sigma_j \leftarrow$  source constructs of  $r_j$  with respect to  $d$

**if** number of constructs in  $\sigma_i \geq$  number of constructs in  $\sigma_j$

**foreach** item  $c_j$  of  $\sigma_j$

**if**  $\sigma_i$  does not contain  $c_j$

**return false**

**foreach** invariant  $p_i$  of  $r_i$

**foreach** invariant  $p_j$  of  $r_j$

**if**  $p_i$  contradicts  $p_j$

**return false**

**return true**

**return false**

---

## 7.5.2 Modified algorithms for mapping list structures

Algorithm 7.1 on page 204 calls the algorithms `INITIALELEMENTGROUPS` and `GENERATECOMBINATIONS`, which are modifications of two of Amor's algorithms (1997, Tables 10.5 and 10.6). The first step of the translation algorithm is to determine all rules that might affect the element currently under consideration, referred to by Amor as the *key* element. For each matching rule, an initial collection of elements that may be potentially affected is generated by the algorithm `INITIALELEMENTGROUPS`. All combinations of these elements that satisfy the rule's invariants are then generated by the algorithm `GENERATECOMBINATIONS`.

There are three cases to consider when determining the initial element groups for a mapping:

1. individual constructs: `inter_class([construct1], ... );`
2. grouped constructs: `inter_class([construct1, group(construct2)], ... );` and
3. 'listed' constructs: `inter_class([construct1, construct3[2]], ... ).`

Amor's original algorithms already handle the first two cases; the third case will now be discussed.

The original VML syntax definition implies that the syntax `some_construct[n]` (where  $n$  is an integer) is legal in an *inter\_class* header, but this was apparently not what was intended (Amor, 1998, personal communication). Instead, it was expected that lists would be specified without an explicit length in the *inter\_class* header (that is, `some_construct[ ]`), and that the length would be specified by an invariant (that is, `count(some_construct[ ]) = 2`). This seems somewhat cumbersome however, especially when a mechanism for specifying list lengths in the header is already allowed by the syntax. Reducing the number of invariants will also reduce the amount of time required to evaluate a translation. The VML-S syntax therefore explicitly allows the specification of list lengths in *inter\_class* headers.

If the length  $l$  of the list is known, generating the initial element groups is simply a matter of finding all  $n$  elements corresponding to that construct, and from these elements generating all possible lists of length  $l$ . There are two possibilities to consider

here: either the listed construct is the same as the construct of the key element (or one of its generalisations), or it is not. Consider the following *inter\_class* header:

```
inter_class
([martinrelationship, ertypeitem[2]], [sql92foreignkey],
...

```

and suppose that the following elements exist in the source description: the regular entities Student and Staff, the associative entity Assignment and a relationship marks connecting Staff and Assignment. Now suppose that the key element is Staff. This is instantiated from the construct MARTINREGULARENTITY, which is a specialisation of ERTYPEITEM and is the same as the listed construct. Conversely, if the relationship marks is the key element, this is different from the listed construct.

If the listed construct is the same as the construct of the key element, the key element must occur at least once in every generated list, so the number of lists generated is  $n^l - (n - 1)^l$ . If the listed construct is *not* the same as the construct of the key element, the key element does not occur in the generated lists, and the number of lists generated is  $n^l$ . Thus, if the number of elements found is six and the list length is two, the total number of lists generated is eleven when the listed construct is the same as the key construct, and thirty-six otherwise.

For example, given the three elements Staff, Student and Assignment (which can all be generalised to the same construct ERTYPEITEM), the key element Student, and a list length of two, the following five ( $3^2 - 2^2$ ) lists would be generated: {Staff, Student}, {Student, Staff}, {Student, Student}, {Student, Assignment}, {Assignment, Student}. Conversely, if the key element were marks (drawn from the construct MARTINRELATIONSHIP), the following nine ( $3^2$ ) lists would be generated: {Staff, Staff}, {Staff, Student}, {Staff, Assignment}, {Student, Staff}, {Student, Student}, {Student, Assignment}, {Assignment, Staff}, {Assignment, Student}, {Assignment, Assignment}. A detailed example of this list generation process may be found in Section F.2 on page 427.

Sometimes the length of the list may not be specified. If so, the only option is to generate all possible lists of lengths one to the number of elements found, which could result in a very large number of lists. If the number of elements found is  $n$ , the total number of lists generated will be  $\sum_{i=1}^n n^i - (n - 1)^i$  when the listed construct is the same as the key construct, and  $\sum_{i=1}^n n^i$  when it is not. Thus, if six elements are found,

the total number of lists generated is 36,456 when the listed construct is the same as the key construct, and 55,986 otherwise. If the position of elements in the list is not significant, a `group` should probably be used instead of an indeterminate-length list.

Algorithm 7.4 is a modified version of Amor's (1997, Table 10.5) algorithm for determining initial element groups. The additions are on lines 5–20. Algorithm 7.5 on the next page is a slightly modified version of Amor's (1997, Table 10.6) algorithm for generating possible element combinations. Amor originally used the term 'cross-product' to describe how the element set  $E$  and the result set  $R$  were combined. The operation that is performed is not strictly a cross-product, although the result appears similar. The actual operation carried out is shown in Algorithm 7.6 on the following page.

---

Algorithm 7.4: Determine initial element groups

---

```

INITIALELEMENTGROUPS( $K, H$ )
Input: The key element  $K$  and an inter_class header  $H$ .
Output: A list of initial element groups for  $H$ , based on  $K$ .
(1)   foreach construct  $C$  in  $H$  (in reverse order)
(2)     if  $C$  is grouped
(3)       find all elements of  $C$ 
(4)       return list of elements in a list
(5)     else if  $C$  is 'listed'
(6)       find all elements of  $C$  ( $n$  elements)
(7)       if list length  $l$  is specified
(8)         if  $C$  is construct of  $K$ 
(9)           generate all possible lists of length  $l$  in which  $K$  occurs
(10)            at least once, using found elements
(11)         else
(12)           generate all possible lists of length  $l$ , using found elements
(13)       else
(14)         if  $C$  is construct of  $K$ 
(15)           generate all possible lists of lengths 1– $n$  in which
(16)              $K$  occurs at least once, using found elements
(17)         else
(18)           generate all possible lists of lengths 1– $n$ ,
(19)             using found elements
(20)         return list of generated lists in a list
(21)     else if  $C$  is construct of  $K$ 
(22)       return  $K$  in a list
(23)     else
(24)       find all elements of the named type
(25)       return them in a list

```

---

The key change in Algorithm 7.5 is how the invariants are applied to elements. When an individual element (that is, not grouped or 'listed') of a combination fails an

---

### Algorithm 7.5: Generate element combinations for the *inter\_class*

---

GENERATECOMBINATIONS( $G, H$ )

**Input:** A list of initial element groups  $G$  and an *inter\_class* header  $H$ .

**Output:** Those element groups that remain after filtering by invariants.

- (1) current result set  $R \leftarrow \emptyset$
  - (2) **foreach** set of elements  $E$  in  $G$  relating to a construct  $C$  in  $H$
  - (3)     determine set of invariants  $I_E$  relating purely to construct  $C$
  - (4)     apply the invariants  $I_E$  to the set  $E$
  - (5)     COMBINE( $E, R$ )
  - (6)     determine the set of invariants  $I_R$  applying to constructs
  - (7)         incorporated in  $R$
  - (8)     apply the invariants  $I_R$  to  $R$
  - (9) **return**  $R$
- 

### Algorithm 7.6: Combine two lists of elements

---

COMBINE( $E, R$ )

**Input:**  $E$  is a list of elements or lists, and  $R$  is a list of lists.

**Output:** The combined list.

- (1) result list  $L \leftarrow \emptyset$
  - (2) **if**  $R = \emptyset$
  - (3)     add  $E$  to  $L$
  - (4) **else**
  - (5)     **foreach** item  $e$  of  $E$
  - (6)         **foreach** list  $r$  of  $R$
  - (7)             add  $e$  to the front of list  $r$
  - (8)             add list  $r$  to  $L$
  - (9) **return**  $L$
- 

invariant, the entire combination is removed from consideration (Amor, 1997, p. 177). For grouped elements, the invariants are applied to each element in the group and are used to reduce the number of elements in the group (Amor, 1997, p. 177). Similarly, for listed elements the invariants are applied to each generated list of elements and are used to reduce the number of lists. If the number of lists is reduced to zero, the entire combination is removed from consideration.

## 7.6 Summary

At the end of the previous chapter, two issues requiring further exploration were identified. The first, extending VML, was discussed in this chapter. Several issues were identified that complicate the specification of description and element translations in

VML. Some of these were a result of VML's focus on specifying schema translations rather than description translations, while others were the result of an incomplete implementation of the original VML specification. An extended version of VML, known as *VML-S*, was defined to address these issues. A correspondence between the abstract notation defined in Chapter 4 and VML-S specifications was identified, and it was demonstrated how a VML-S rule specification can be built.

Most of the VML-S extensions were to VML's *inter-class* syntax. The `direction` clause was defined to address VML's inability to specify unidirectional rules, and three new directional equivalence operators were defined to allow the specification of unidirectional equivalences. The `label` and `excludes` clauses were defined to allow the explicit specification of rule exclusion. Construct aliases were introduced to allow the same construct to appear multiple times in a rule header.

The translation process described in Chapter 4 was also extended and changes and additions were made to some of Amor's (1997) original translation algorithms to rectify known problems and support the new VML-S syntax. An algorithm was also specified for generating subsumption/exclusion graphs for a set of rules.

None of the VML-S extensions described in this chapter have been implemented, with the exception of the algorithm for building subsumption/exclusion graphs. This is primarily because other researchers are already working on reimplementing VML in Java (Grundy, 1998, personal communication). Grundy has expressed an interest in the extensions described here, so any implementation as part of this research would be an unnecessary duplication of effort. Despite the lack of a VML-S implementation, VML-S has been successfully used to define the rules of the translations discussed in Chapter 5 and Appendix E — see Appendix F for details.

The second issue identified at the end of the previous chapter was measuring the effect of heuristics on translation quality. In the next chapter, a method for measuring translation quality will be defined in order to determine what impact heuristics may have on translation quality.





# Chapter 8

## Measuring the quality of translations

### 8.1 Introduction

The previous chapters have outlined an approach to translating descriptions between multiple representations. An important aspect of this approach is the use of heuristics to improve translation quality. In order to determine whether the use of heuristics has an impact on translation quality, a measure of translation quality must be developed. The concepts of *relative information capacity* and *schema intension graphs* described in this chapter can be used to determine the category of expressive overlap between two representations, and to determine the relative quality of two or more translations.

Hull's (1986) concept of relative information capacity, described in Section 8.2, provides a basis for comparing the information content of schemas in the context of schema integration. Miller et al.'s (1994b) schema intension graphs are a tool that may be used to determine the relationship between the relative information capacities of two schemas, and are also described in Section 8.2. The representation definitions used in this thesis are a form of schema, so relative information capacity may also be applied to representations. In particular, the author has found that the expressive power of a representation can be characterised by the information capacity of its definition (discussed in Section 8.3). Consequently, schema intension graphs may also be used to determine the category of expressive overlap between two representations.

From this basis, the author has developed two methods for measuring the relative quality of translations, which are defined in Section 8.4. These methods make use of schema intension graphs and produce a collection of measurements for a translation that may be compared with those of other translations. The first method was found to produce results that did not match the experimental evidence gained from using

Swift, so the second method was developed as a refinement of the first to remedy this. Using this refined method, it will be shown in Section 8.5 that the use of heuristics has a positive impact on the quality of translations.

## 8.2 Relative information capacity (Hull, 1986)

A schema conveys information about the phenomenon that it models. Hull (1986) defined the concept of *relative information capacity* to describe the information content of different schemas. The relative information capacity of a schema determines the set of all valid instances of that schema (Hull, 1986; Miller et al., 1993; Miller, 1994). Relative information capacity, as the name implies, is not a quantitative measure; rather it is an indicator of the relative information content of different schemas.

Relative information capacity can be used as a basis for identifying the differences between schema instances, and can therefore be used as an aid to schema integration and translation (Miller, 1994). Miller et al. (1993) defined relative information capacity in terms of information capacity preserving mappings (or just information preserving mappings) and equivalence preserving mappings. An *information preserving mapping* from schema  $S_1$  to schema  $S_2$  maps every element of  $S_1$  to an element of  $S_2$ , but only some elements of  $S_2$  to elements of  $S_1$ .  $S_2$  has an information capacity greater than  $S_1$ , and is said to *dominate*  $S_1$ , denoted  $S_1 \preceq S_2$ . An *equivalence preserving mapping* from  $S_1$  to  $S_3$  maps every element of  $S_1$  to an element  $S_3$  and vice versa (that is, both the mapping and its inverse are information preserving). The information capacities of  $S_1$  and  $S_3$  are identical, and the schemas are said to be *equivalent*, denoted  $S_1 \equiv S_3$ .

Since relative information capacity is a relative measure, it can only be effectively used in a comparative manner. To this end, (Miller et al., 1994b) defined the *schema intension graph* formalism to provide a tool for comparing the information capacity of two schemas. A schema intension graph (SIG) describes the associations between domains in a schema, and provides a way to “reason about constraints on *collections of entities* in an instance of the entity set, rather than about the internal structure of a single entity” (Miller et al., 1994b, p. 3). To determine the equivalence (or lack thereof) of two schemas  $S_1$  and  $S_2$ , schema intension graphs  $G_{S_1}$  and  $G_{S_2}$  are built for each schema. One of these graphs (for example,  $G_{S_1}$ ) is then manipulated by a series of

information and/or equivalence preserving transformations in an attempt to produce a transformed graph ( $G'_{S_1}$ ) that is isomorphic to  $G_{S_2}$ . If an isomorphism cannot be produced, there is no direct relationship between the relative information capacities of  $S_1$  and  $S_2$ . If an isomorphism *can* be produced, then depending on the series of transformations used,  $S_1$  either dominates  $S_2$  (or vice versa), or the two schemas are equivalent.

Note that Miller et al. (1994a) use stricter definitions of equivalence and dominance than that used here; strictly speaking, the discussion of equivalence and dominance in this section should properly refer to *absolute* dominance and equivalence, whereas the following discussion on SIG transformations should properly refer to *internal* dominance and equivalence (Miller et al., 1994a). The distinction, however, is not crucial to the discussion in this chapter.

### 8.2.1 Building schema intension graphs (Miller et al., 1994b)

The relative information capacities of two schemas may be compared using schema intension graphs (Miller et al., 1994b). Schema intension graphs provide an alternate representation for describing schemas, and comprise:

- a collection of nodes representing typed domains (that is, sets of data values);
- a collection of labelled edges representing binary relations between domains; and
- annotations on the edges that represent simple constraints on the binary relations.

Nodes may be *simple* (atomic) or *constructed* (composite). Simple nodes are denoted by the name of the domain that they represent. Constructed nodes represent a collection of domains and are built using the  $\times$  (cross product) and  $+$  (union) operators. Edges are denoted by plain lines, and represent binary relations that may be composed to form a *path* between two nodes (the notation  $e_2 \circ e_1$  means ‘edge  $e_1$  followed by edge  $e_2$ ’). Some edges may be designated as *selection* edges, labelled  $\sigma$ , or *projection* edges, labelled  $\pi$ . These two edge types are analogous to the relational operators of the same name (Codd, 1972b).

In addition, edges may be annotated with up to four properties denoting simple constraints on the binary relation represented by the edge. There are four properties

used to annotate edges: totality, surjectivity, functionality and injectivity. These properties are defined as follows (Miller et al., 1994b, p. 6):

Let  $S_A$  and  $S_B$  be two sets. A binary relation  $r : A - B$  is *total* (denoted  $e : A \dashv B$ ) if it is defined for all elements of  $S_A$ ; *surjective* ( $e : A \dashv B$ ) if it is defined for all elements of  $S_B$ ; *functional* ( $e : A \longrightarrow B$ ) if an element of  $S_A$  determines at most one element of  $S_B$ ; and *injective* ( $e : A \longleftarrow B$ ) if an element of  $S_B$  determines at most one element of  $S_A$ .

Note that the correct usage of these terms is determined by the ‘direction’ in which the binary relation is ‘read’. That is, the binary relation  $e : A \longrightarrow B$  is functional if read from left to right, but injective if read from right to left. A *bijection* is defined as an edge that is total, surjective, functional and injective (Miller et al., 1994b, p. 6).

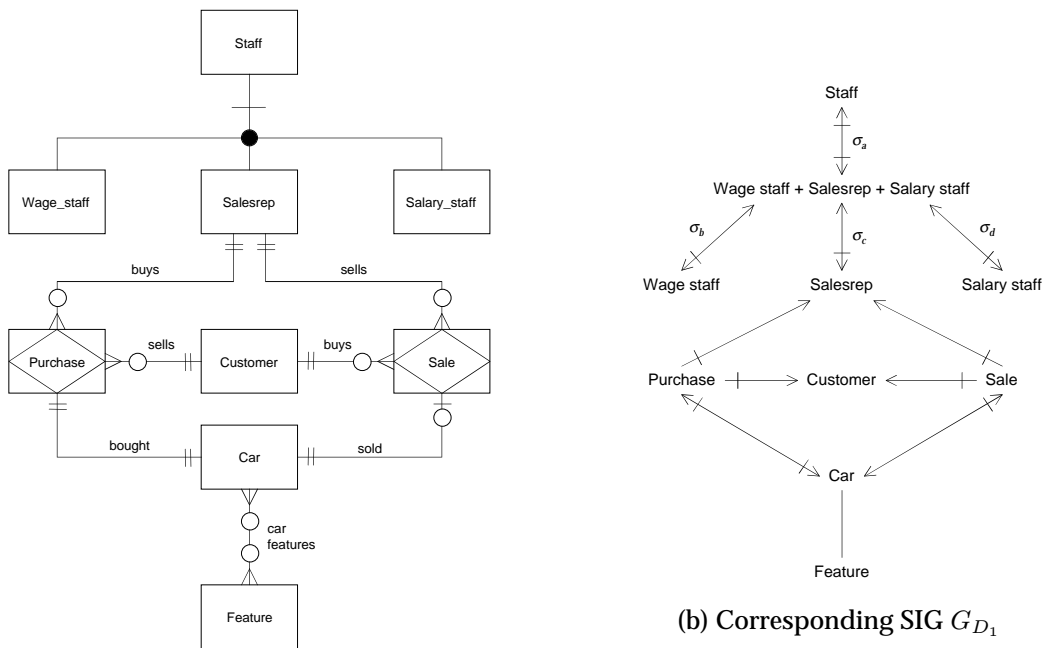
In Figure 8.1 is shown an ERD description ( $D_1$ ) and corresponding schema intension graph ( $G_{D_1}$ ) for the used cars viewpoint (described in Section C.1). The author has developed the following process for deriving a SIG from an ERD:

- Each entity corresponds to a SIG node. This node represents the set of all instances of the entity, and can thus be thought of as the ‘domain’ of that entity. Thus, the Feature entity corresponds to the SIG node Feature.
- The Staff entity has three subtypes: Wage\_staff, Salesrep and Salary\_staff. The constructed node Wage staff + Salesrep + Salary staff (referred to hereafter as ‘the union’) corresponds to the set of all instances of the three subtypes.
- Every instance of Staff must also be an instance of the union, and vice versa. Thus, the binary relation Staff — Wage staff + Salesrep + Salary staff is one-to-one (functional and injective) and defined for all elements in both directions (total and surjective). The instances of the union also form a trivial selection of the instances of Staff, so the edge is a selection edge.
- Every instance of the union is an instance of exactly one of Wage staff, Salesrep or Salary staff. Conversely, every instance of Wage staff, Salesrep and Salary staff is an instance of the union. The binary relation Wage staff + Salesrep + Salary staff — Wage staff is therefore one-to-one (functional and injective), is defined

for all elements of Wage staff (surjective), but not defined for all elements of the union (not total). The instances of Wage staff form a selection of the instances of the union, so the edge is a selection edge. The edges between the union and the other two subtypes are treated similarly.

- A relationship from entity  $E_1$  to entity  $E_2$  translates to a SIG edge as follows:
  - If the  $E_1$  (respectively  $E_2$ ) end of the relationship has a cardinality of one, the corresponding SIG edge is injective (respectively functional).
  - If the  $E_1$  (respectively  $E_2$ ) end of the relationship is mandatory, the corresponding SIG edge is total (respectively surjective).

Thus, the SIG edge corresponding to the relationship from Customer to Sale is injective and surjective, while the SIG edge corresponding to the relationship from Car to Purchase is a bijection.



**Figure 8.1:** E-R description of the used cars viewpoint and corresponding schema intention graph

## 8.2.2 Comparing schema intension graphs (Miller et al., 1994b)

It is possible to determine the relationship between the relative information capacities of two schemas by comparing SIGs for the two schemas. Intuitively, if two SIGs are isomorphic, then the corresponding schemas are equivalent (Miller et al., 1994b, p. 11). An isomorphism occurs between two SIGs when the graphs match both structurally and semantically. That is, two SIGs must be more than just structurally identical — the semantics of the SIG nodes must also be compatible. For example, the semantics of the construct  $\mathfrak{R}_e(E-R, ERD_{Martin})$  [MARTINREGULARENTITY] are compatible with the semantics of the construct  $\mathfrak{R}_d(DataFlow, DFD_{G\&S})$  [GNSDATASTORE], so SIG nodes corresponding to these constructs are isomorphic. Conversely, the semantics of the construct  $\mathfrak{R}_d$  [GNSEXTERNALENTITY] are not compatible with those of the construct  $\mathfrak{R}_e$  [MARTINREGULARENTITY], so SIG nodes corresponding to these constructs are not isomorphic.

Miller et al. (1994b, Section 5) identify three possible results from comparing two SIGs:

1. The SIGs are isomorphic; or they are not immediately isomorphic, but one can be manipulated by a series of equivalence preserving transformations (described below) until it is isomorphic to the other. The two schemas are equivalent.
2. The SIGs are not immediately isomorphic, but one can be manipulated by a series of information capacity augmenting transformations (described below) until it is isomorphic to the other. The transformed schema is dominated by the other.
3. The SIGs are not isomorphic and cannot be transformed to be isomorphic. The information capacities of the two schemas cannot be compared.

Batini et al. (1992, p. 145) follow a similar classification scheme with respect to schema transformations: they refer to the first case as *information-preserving* transformations, the second as *augmenting* transformations and the third as *noncomparable* transformations.

Miller et al. (1994a, Section 5) describe several equivalence preserving and information capacity augmenting transformations on SIGs, which are summarised in Table 8.1. If two SIGs are not immediately isomorphic, these transformations may be applied in

**Table 8.1:** Summary of SIG transformations

| Transformation                               | Type        | Effect on information capacity |
|--|-------------|--------------------------------|
| Delete annotation or $\pi/\sigma$ constraint | annotation  | augmented                      |
| Move/copy edge                               | composition | augmented                      |
| Create/delete duplicate node                 | selection   | no change                      |
| Create bijective selection edge              | selection   | no change                      |
| Delete bijective selection edge              | selection   | augmented                      |

an attempt to make one of the SIGs isomorphic to the other. Transforming the SIG for a schema is effectively the same as transforming the schema itself, but SIGs provide a useful abstract formalism within which to perform these transformations. The transformations that may be applied to SIGs are:

**Annotation transformations** allow the removal of annotations or projection and selection constraints from edges. If schema  $S_1$  can be transformed into schema  $S_2$  by such a transformation, then the information capacity of  $S_2$  is greater than that of  $S_1$  (that is,  $S_1 \preceq S_2$ ).

**Composition transformations** allow edges to be moved and copied around the graph. Consider an edge  $e_1$  that connects nodes  $n_1$  and  $n_2$ . If there is a functional edge (or path of functional edges) from some other node  $n_3$  to  $n_2$ , the edge  $e_1$  may be moved across this edge/path so that it connects nodes  $n_1$  and  $n_3$  instead. A variation of this allows  $e_1$  to be duplicated across the functional path, giving a new edge  $e'_1$  that connects nodes  $n_1$  and  $n_3$  while leaving the original edge intact. The annotation of the resulting edge is determined by the composition of the edges in the path across which it is moved or duplicated. If an edge with a particular annotation (such as surjectivity) is composed with an edge that does not have that annotation, the resulting edge does not have the annotation. For example, the composition of the edges  $\longrightarrow$  (functional) and  $\dashv$  (surjective) would be an edge with no annotations. If schema  $S_1$  can be transformed into schema  $S_2$  by such a transformation, then the information capacity of  $S_2$  is greater than that of  $S_1$  (that is,  $S_1 \preceq S_2$ ).

**Selection transformations** allow the creation and deletion of new nodes and edges as follows:

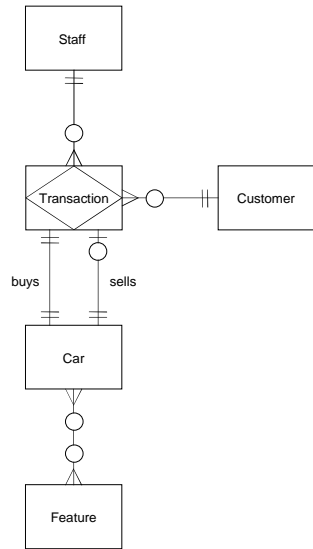
- New nodes are created by duplicating existing nodes. To enforce the restriction that the new node is a duplicate of an existing node, a bijective selection edge must connect the new node with the node it duplicates.
- Nodes may only be deleted if they are attached to exactly one other node by a single bijective selection edge, that is, they are effectively a duplicate of the other node.
- A bijective selection edge may be created that connects a node to itself.
- A bijective selection edge connecting two nodes may be deleted.

If schema  $S_1$  can be transformed into schema  $S_2$  by a node creation/deletion or edge creation transformation, then the information capacities of  $S_2$  and  $S_1$  are identical (that is,  $S_2 \equiv S_1$ ). If  $S_1$  can be transformed into  $S_2$  by an edge deletion transformation, then the information capacity of  $S_2$  is greater than that of  $S_1$  (that is,  $S_1 \preceq S_2$ ). This result seems somewhat counter-intuitive, as noted by Qian (1995), but is nonetheless correct (Miller et al., 1994a, p. 17). This is because a bijective selection edge between two nodes denotes that the nodes correspond to the same domain, so removing the edge also removes this constraint.

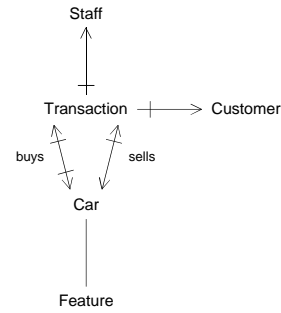
For example, consider the alternative description  $D_2$  of the used cars viewpoint shown in Figure 8.2(a). Comparing this with the description in Figure 8.1(a) on page 217, it can be seen that this description does not include subtypes for staff, and purchases and sales are modelled as a single Transaction entity. The SIG corresponding to this description ( $G_{D_2}$ ) is shown in Figure 8.2(b). Intuitively, it could be guessed that the information capacities of the two E-R descriptions might be equivalent, or perhaps that the information capacity of the first description is greater than that of the second. Since the SIG transformations described above never decrease information capacity, an attempt will now be made to transform  $G_{D_2}$  until it is isomorphic to  $G_{D_1}$ .

The first step is to create new nodes in  $G_{D_2}$  to correspond to the additional nodes in  $G_{D_1}$ . As shown in Figure 8.3(a) on page 222, the Transaction node is duplicated to produce the new node Sale, connected by the bijective selection edge  $\sigma_1$ . Similarly, the





(a)  $D_2(V_{cars}, E-R, ERD_{Martin})$



(b) Corresponding SIG  $G_{D_2}$

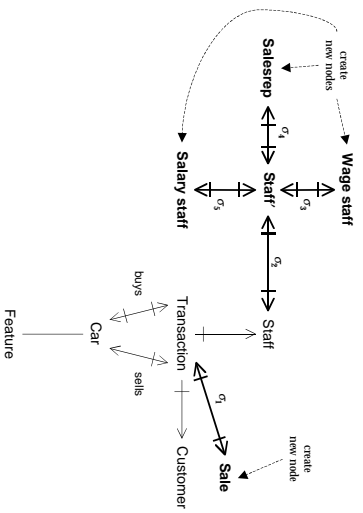
**Figure 8.2:** Alternate E-R description of the used cars viewpoint and corresponding SIG

Staff node can be duplicated to produce the node Staff' and the bijective selection edge  $\sigma_2$ . Staff' is further duplicated to produce the nodes Wage staff, Salesrep and Salary staff (with associated edges  $\sigma_3$ ,  $\sigma_4$  and  $\sigma_5$  respectively).

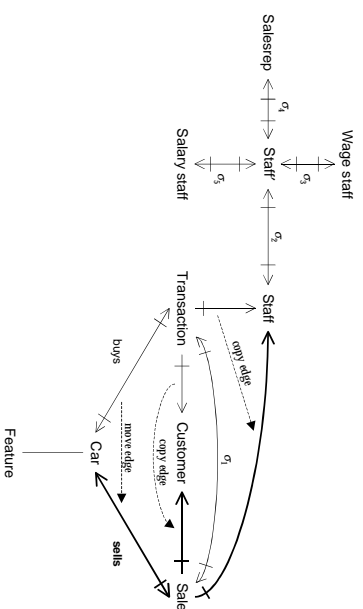
The next step is to move or copy edges in  $G_{D_2}$  so that they correspond more closely to edges in  $G_{D_1}$ . For example, the sells edge in  $G_{D_2}$  may be moved across selection edge  $\sigma_1$  so that it connects nodes Sale and Car, as shown in Figure 8.3(b). The edge connecting Staff and Transaction may be copied across selection edge  $\sigma_1$  to produce a similar edge between Sale and Staff. The edge connecting Customer and Transaction may also be copied in this manner, producing the transformed SIG shown in Figure 8.3(b).

The transformed SIG is now closer to  $G_{D_1}$ , but there are still some obvious discrepancies. For instance, the edges from the Transaction and Sale nodes to Staff should be connected to Salesrep instead. This can be remedied by moving both edges across the path  $\sigma_4 \circ \sigma_2$ , producing the SIG shown in Figure 8.3(c).

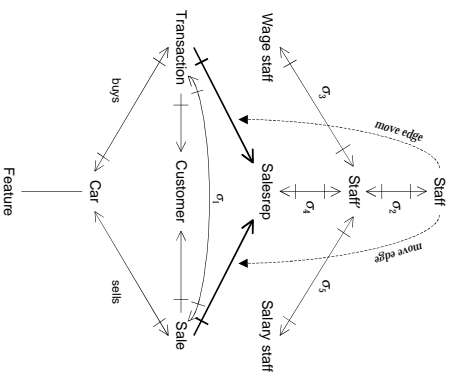
Another discrepancy is the bijective selection edge ( $\sigma_1$ ) connecting nodes Transaction and Sale. An edge deletion transformation can remove this edge. The only remaining discrepancies are the annotations on the edges  $\sigma_3$ ,  $\sigma_4$  and  $\sigma_5$ . Three annotation deletion transformations will remove the offending annotations, producing the



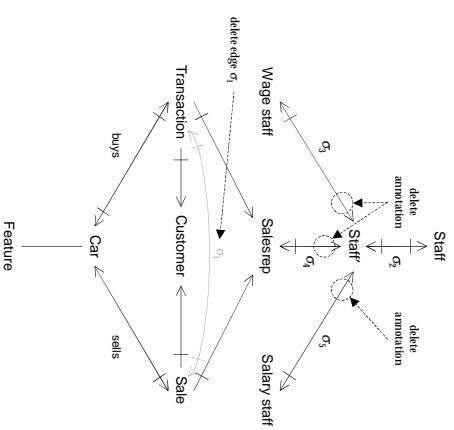
(a) Step 1: create nodes and associated edges



(b) Step 2: move and duplicate edges



(c) Step 3: move more edges



(d) Step 4: delete  $\sigma_1$  and delete annotations

**Figure 8.3:** Transforming a subgraph of a SIG

SIG shown in Figure 8.3(d). This SIG is isomorphic to  $G_{D_1}$ , because the structures of the two SIGs are identical and the semantics of the nodes are compatible.

The first set of transformations described above produces no change in information capacity, while the remaining transformations all result in an increase in information capacity. The information capacity of  $D_1$  is therefore greater than that of  $D_2$ , that is,  $D_2 \preceq D_1$ . Note that the order of transformations given above is not the only order in which the transformations could be applied, although there are some restrictions on the ordering. For example, the creation of the various Staff nodes could have been delayed until step 3, and the moving and copying of edges could have been done in a different order. Conversely, selection edge  $\sigma_1$  cannot be deleted before the transformations in step 2 are completed. The final result will be the same regardless.

### **8.3 Categorising expressive overlap using schema intention graphs**

Relative information capacity was originally developed with application to instances of schemas in mind, rather than descriptions. Representations are defined in this thesis using an extended entity-relationship approach, as described in Section 3.4 on page 49, so a representation definition is effectively a schema describing the constructs of the representation. Relative information capacity can therefore also be applied to descriptions.

The information capacity of a schema determines the set of all valid instances of that schema within the context of the phenomenon that it models (Miller, 1994). A schema defines a set of valid database structures for a given phenomenon. Combined with the set of all data relating to the phenomenon, the schema may be used to generate all possible database states for the phenomenon. This is analogous to grammars for languages: a grammar defines a set of valid sentence structures for a language, and combined with the set of all possible words for the language, the grammar may be used to generate all valid sentences for the language (Chomsky, 1978; Holtzman, 1994).

By extension, the information capacity of a representation definition determines the set of all valid descriptions that may be expressed using the representation. That is,

it determines the extent of what may be expressed using that representation. This is effectively identical to the notion of expressive power introduced by the author in Chapter 3. That is, the expressive power of a representation can be characterised by the information capacity of the representation's definition. This means that SIGs may be used to determine any relationship between the expressive powers of two representations, and thus the nature of the expressive overlap between those representations.

Let  $\mathfrak{R}_p(T_1, S_1)$  and  $\mathfrak{R}_q(T_2, S_2)$  be two representations, and let  $D_1$  and  $D_2$  be descriptions expressed using  $\mathfrak{R}_p$  and  $\mathfrak{R}_q$  respectively. The expressive overlap between  $\mathfrak{R}_p$  and  $\mathfrak{R}_q$  will fall into one of the four categories identified in Section 3.5 on page 55: disjoint, intersecting, inclusive or equivalent. Suppose that SIGs are built corresponding to the definitions of both representations, and that one of the SIGs is then transformed as described in the previous section in an attempt to create an isomorphism. If it is found that the information capacities of both definitions are identical (result 1 in the previous section), then the expressive powers of  $\mathfrak{R}_p$  and  $\mathfrak{R}_q$  are equivalent. If it is found that the information capacity of one definition (for example, that for  $\mathfrak{R}_p$ ) is greater than that of the other (result 2 in the previous section), then the expressive power of  $\mathfrak{R}_p$  is inclusive of  $\mathfrak{R}_q$ , or vice versa. If no comparison can be drawn between the information capacities of the two definitions (result 3 in the previous section), then the expressive overlap between  $\mathfrak{R}_p$  and  $\mathfrak{R}_q$  falls into either the disjoint or intersecting categories, but it is otherwise impossible to distinguish between these two categories. An intersecting expressive overlap is obviously of more interest than a disjoint expressive overlap, so it would be useful to be able to detect this.

The author has identified the following solution to this inability to distinguish between intersecting and disjoint expressive overlap. An equivalent or inclusive expressive overlap implies that the entirety of one representation definition is either identical to or contained within the other representation definition. These two cases may be detected using the SIG transformation method by transforming an *entire* SIG. An intersecting expressive overlap implies that some subset of one representation definition is either identical to or contained within some subset of the other representation definition. In SIG terms, this equates to being able to transform one or more *subgraphs* of the SIG until they are isomorphic to one or more subgraphs in the other SIG.

Three results of comparing two SIGs were described in the previous section. The first two results are unchanged when using SIGs to determine the expressive overlap of representations. The author has modified the third result as follows:

3. The two SIGs are not isomorphic and cannot be transformed to be isomorphic. The expressive overlap of the two representations may be determined as follows:
  - (a) If one or more subgraph(s) of the SIGs are either isomorphic, or can be transformed to be isomorphic, the two representations have an intersecting expressive overlap.
  - (b) If no subgraphs of the SIGs are isomorphic, and cannot be transformed to be isomorphic, the two representations have a disjoint expressive overlap.

The expressive overlap between two representations determines the completeness of translations between those representations, as discussed in Section 3.5 on page 55. If  $\mathfrak{R}_p$  and  $\mathfrak{R}_q$  are equivalent, it is possible to translate completely in both directions, that is,  $\mathfrak{R}_p \leftrightarrow \mathfrak{R}_q$ . If  $\mathfrak{R}_p$  is inclusive of  $\mathfrak{R}_q$  (respectively,  $\mathfrak{R}_q$  inclusive of  $\mathfrak{R}_p$ ), then the possible translations are  $\mathfrak{R}_q \rightarrow \mathfrak{R}_p$  and  $\mathfrak{R}_p \rightarrow \mathfrak{R}_q$  (respectively,  $\mathfrak{R}_p \rightarrow \mathfrak{R}_q$  and  $\mathfrak{R}_q \rightarrow \mathfrak{R}_p$ ). If  $\mathfrak{R}_p$  intersects with  $\mathfrak{R}_q$ , then it is possible to translate partially in both directions, that is  $\mathfrak{R}_p \rightleftharpoons \mathfrak{R}_q$ . Finally, if  $\mathfrak{R}_p$  and  $\mathfrak{R}_q$  are disjoint, it is typically impossible to translate a description from  $\mathfrak{R}_p$  to  $\mathfrak{R}_q$  or vice versa.

Ideally, the process of detecting SIG isomorphism and performing SIG transformations would be automated. Miller et al. (1994a, Theorems 4.3 and 4.4) show that the problem of testing for SIG equivalence is in general undecidable, and note that the problem of detecting graph isomorphism may be NP-complete. This may not be a major problem in practice, however; Miller et al. (1994a, p. 20) discuss how reasonably efficient algorithms may be developed. Unfortunately, this is only worsened by the introduction of subgraph isomorphism, which introduces the additional problem of detecting potentially isomorphic subgraphs.

Use of SIG isomorphism to determine the expressive overlap of two representations may not, however, be particularly amenable to automation in other ways, as it may not always give a complete analysis of the expressive overlap. SIG isomorphism gives good results when the two representations being compared have similar constructs at a similar level of ‘granularity’. Consider the representations  $\mathfrak{R}_e(E-R, ERD_{Martin})$

and  $\mathfrak{R}_e(DataFlow, DFD_{G\&S})$ . Both of these have constructs analogous to the concepts of ‘entity’ (ERENTITYTYPE and DFDATASTORE respectively) and ‘attribute’ (ERATTRIBUTE and DFFIELD respectively). Now consider  $\mathfrak{R}_e$  and  $\mathfrak{R}_f(FuncDep, FDD_{Smith})$ . While both have constructs analogous to the concept of ‘attribute’ (ERATTRIBUTE and FDATTRIBUTE),  $\mathfrak{R}_f$  does not have a *single* construct directly analogous to the concept of ‘entity’. Nevertheless, it is possible to map ERENTITYTYPE constructs to and from a *collection* of  $\mathfrak{R}_f$  constructs. In other words, a single construct of  $\mathfrak{R}_e$  may map to a collection of constructs of  $\mathfrak{R}_f$  and vice versa.

It may be possible to resolve this issue by taking advantage of the SIG concept of constructed nodes. In the SIG for a representation definition, simple nodes represent individual constructs, constructed nodes represent collections of constructs and edges represent associations between constructs. In the SIGs in this chapter, constructed nodes are only used to represent specialisation/generalisation associations between constructs, but they could also be used to represent the ‘collective constructs’ described above. It is unclear at present whether this constitutes a change to the representation definition, or merely an ‘optimisation’ of the representation definition so that it may more easily be compared with other representations, and has been left as an area for further investigation (see Chapter 10).

Another issue that arises is that it may be possible to define a representation in more than one way. A representation definition in this thesis is effectively a schema describing the constructs of the representation. The collection of constructs that comprise the representation will always remain constant, as changing this collection will result in the definition of a different representation. It may, however, be possible to define the properties of these constructs in different ways. For example, in the representation definitions in this thesis, the associations between constructs are typically expressed using properties on both sides of the association. Thus, in  $\mathfrak{R}_r(Relational, SQL/92)$ , the RMRELATION construct has a foreignKeys property that comprises a list of associated RMFOREIGNKEY elements, while the RMFOREIGNKEY construct has a reciprocal relation property that links to the associated RMRELATION element. An alternative definition of  $\mathfrak{R}_r$  might omit the foreignKeys property while retaining the relation property.

Such variations may have a small effect on the expressive power of a representation, but will not affect the methods defined in this chapter, as these are defined in

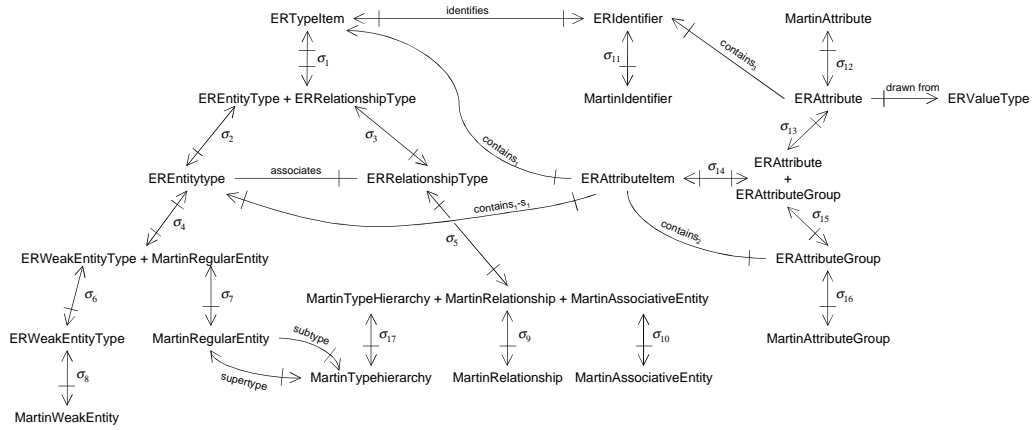
terms of constructs rather than the properties of constructs. That is, the methods defined here assume that a construct is atomic. Defining the properties of a representation in a different manner should therefore have no effect on the results produced by these methods, although it will obviously have an effect on the VML-S specifications of translation rules to or from the representation. If the definition of a representation is changed in such a way as to produce a different collection of constructs, it is no longer the same representation, so the results of applying the methods defined here may differ. This is, however, to be expected.

### 8.3.1 Example of determining expressive overlap

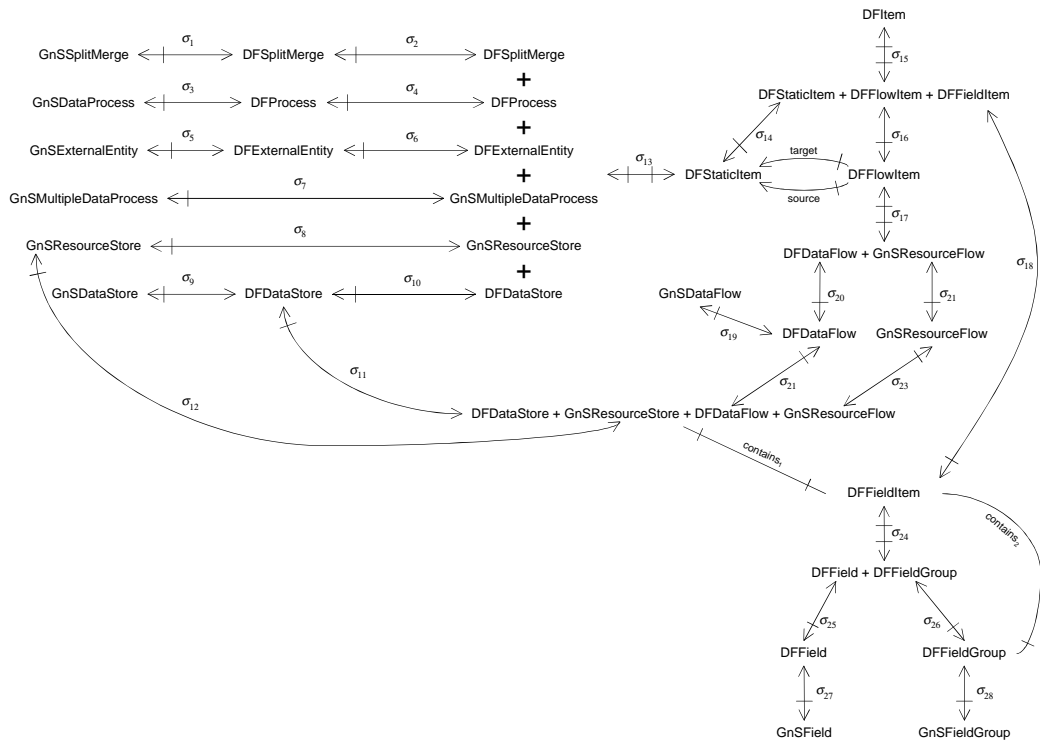
Consider the representations  $\mathfrak{R}_e(E-R, ERD_{Martin})$  and  $\mathfrak{R}_d(DataFlow, DFD_{G\&S})$ . The SIGs corresponding to the definitions of these representations are shown in Figure 8.4 on the following page. Intuitively, these two SIGs do not appear immediately isomorphic, and closer examination shows that indeed they are not. The next step is to find any subgraphs that might potentially be isomorphic. The subgraph of the  $\mathfrak{R}_e$  SIG comprising nodes  $\{ERAttributeItem, ERAttribute + ERAttributeGroup, ERAttribute, MartinAttribute, ERAttributeGroup, MartinAttributeGroup\}$  is directly isomorphic to the subgraph of the  $\mathfrak{R}_d$  SIG comprising nodes  $\{DFFieldItem, DFField + DFFieldGroup, DFField, GnSField, DFFieldGroup, GnSFieldGroup\}$ . The structures of the two subgraphs are identical, and the semantics of the nodes are compatible.

Examining the semantics of the remaining nodes in each SIG, the following additional node correspondences may be readily identified:  $MartinRegularEntity \Leftrightarrow GnSDataStore$  and  $EREntityType \Leftrightarrow DFDataStore$ . Although at first glance it might appear that  $ERTypeItem$  corresponds to  $DFItem$ , this is not so, because  $DFItem$  includes attributes while  $ERTypeItem$  does not. That is, the semantics of the two nodes are different. The rules for the  $\mathfrak{R}_e \rightleftharpoons \mathfrak{R}_d$  translation (excluding heuristics) also reveal that the  $\mathfrak{R}_e$  nodes  $MartinWeakEntity$  and  $MartinAssociativeEntity$  both correspond to the  $\mathfrak{R}_d$  node  $GnSDataStore$ . Heuristics are not included, as they do not form a part of the ‘natural’ expressive overlap.

These correspondences produce the two subgraphs of the  $\mathfrak{R}_d$  SIG shown in Figure 8.5(a) on page 229. The right-hand subgraph is already isomorphic to a subgraph



(a)  $\mathfrak{R}_e(E-R, ERD_{Martin})$



(b)  $\mathfrak{R}_d(DataFlow, DFD_{G\&S})$

**Figure 8.4:** SIGs for  $\mathfrak{R}_e(E-R, ERD_{Martin})$  and  $\mathfrak{R}_d(DataFlow, DFD_{G\&S})$





of the  $\mathfrak{R}_e$  SIG. The left-hand subgraph may be transformed until it is isomorphic to two subgraphs of the  $\mathfrak{R}_e$  SIG, by applying the series of transformations shown in Figure 8.5. First, as shown in Figure 8.5(b), a collection of new nodes are created. The node  $DFDataStore'$  is created to correspond to the  $\mathfrak{R}_e$  node  $ERWeakEntityType + MartinRegularEntity$ . Similarly,  $WeakDataStore$  is created to correspond to  $ERWeakEntityType$ ,  $GnSAssocStore$  is created to correspond to  $MartinAssociativeEntity$  and  $GnSWeakStore$  is created to correspond to  $MartinWeakEntity$ .

Next, as shown in Figure 8.5(c), the selection edge  $\sigma_c$  is deleted to separate the  $GnSAssocStore$  node from the  $GnSDataStore$  node. The selection edge  $\sigma_9$  is moved across the selection edge  $\sigma_a$ , so that  $GnSDataStore$  is connected to  $DFDataStore'$  instead of  $DFDataStore$ . Edge  $\sigma_9$  is then copied and moved across the selection edges  $\sigma_d$  and  $\sigma_b$  respectively, so that  $GnSWeakStore$  is attached to  $WeakDataStore$ .

Finally, as shown in Figure 8.5(d), the selection edge  $\sigma_d$  is deleted and the totality annotation on selection edge  $\sigma_b$  is removed. This produces the subgraphs shown in Figure 8.6(b), which are isomorphic to the subgraphs of the  $\mathfrak{R}_e$  SIG shown in Figure 8.6(a). Since at least one isomorphic subgraph may be found in each SIG, but the two SIGs are not totally isomorphic,  $\mathfrak{R}_e$  and  $\mathfrak{R}_d$  have intersecting expressive powers<sup>1</sup>.

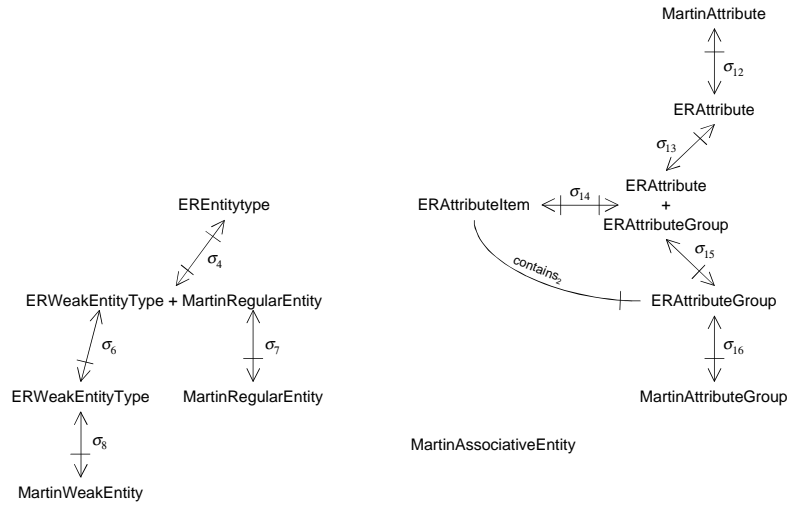
### 8.3.2 Analysis

The categorisation method described above was applied to the three pairs of representations  $\{\mathfrak{R}_e(E-R, ERD_{Martin}), \mathfrak{R}_r(Relational, SQL/92)\}$ ,  $\{\mathfrak{R}_f(FuncDep, FDD_{Smith}), \mathfrak{R}_e(E-R, ERD_{Martin})\}$  and  $\{\mathfrak{R}_e(E-R, ERD_{Martin}), \mathfrak{R}_r(DataFlow, DFD_{G\&S})\}$ , which correspond to the three translations discussed in Chapter 5. It was found that the expressive overlap of each pair of representations falls into the intersecting category. This was expected for  $\{\mathfrak{R}_e, \mathfrak{R}_r\}$  and  $\{\mathfrak{R}_e, \mathfrak{R}_r\}$ , but was not expected for  $\{\mathfrak{R}_f, \mathfrak{R}_e\}$ , because the translation between these two representations is complete in the forward direction. Rather, it was expected that the expressive power of  $\mathfrak{R}_e$  would be inclusive of the expressive power of  $\mathfrak{R}_f$ .

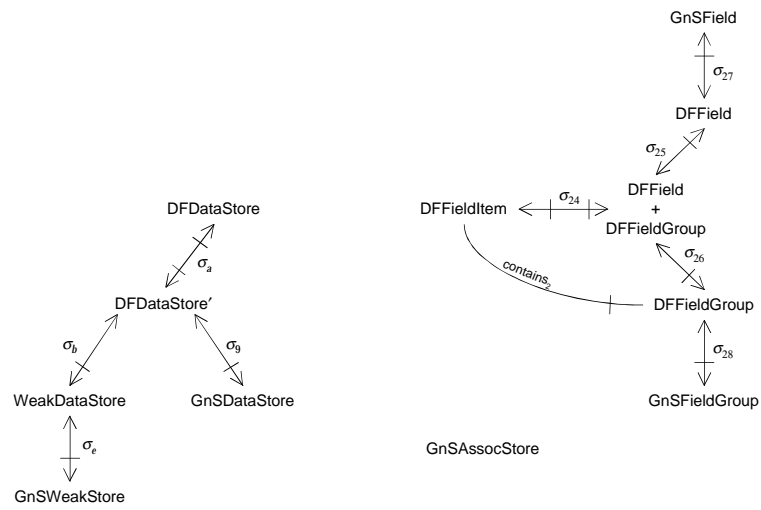
The translation  $\mathfrak{R}_f \rightarrow \mathfrak{R}_e/\mathfrak{R}_f \leftarrow \mathfrak{R}_e$  was identified in Chapter 5 as being complete in the forward direction because all of its rules are complete in the forward direction.

---

<sup>1</sup>Strictly speaking, the expressive power of the  $\mathfrak{R}_e$  subset is inclusive of that of the  $\mathfrak{R}_d$  subset, but this does not affect the result.



(a) Corresponding subgraphs from  $\mathfrak{R}_e$  SIG



(b) Transformed subgraphs from  $\mathfrak{R}_d$  SIG

**Figure 8.6:** Isomorphism between SIG subgraphs

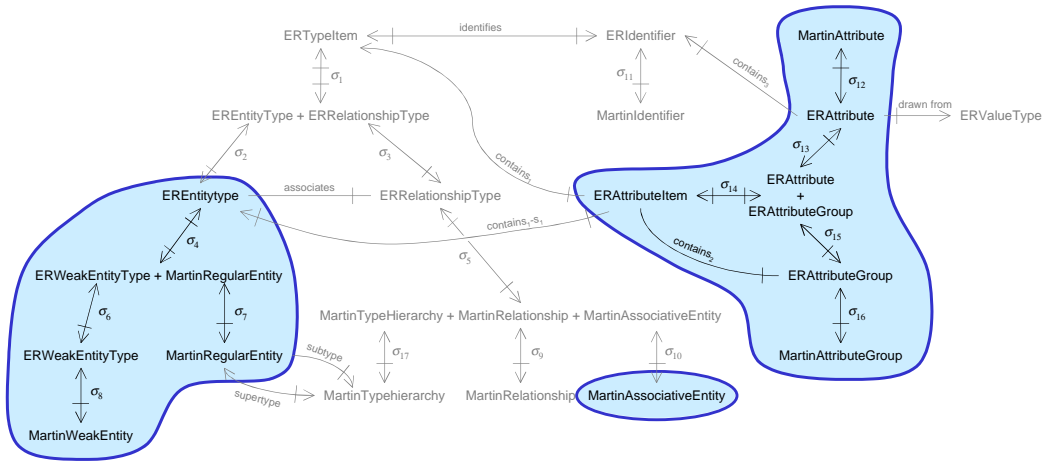
It may be that there are as yet undiscovered rules for this translation that are not complete. That is, the forward translation may not be complete. Alternatively, this unexpected result may be caused by the granularity issue identified earlier. This seems more likely, given that it appears possible to translate functional dependency descriptions completely to E-R descriptions, and many of the rules of this translation involve ‘collective constructs’. This is not a simple issue to resolve and has been left as an area for future research (see Chapter 10).

A final possibility is that this method is sensitive to the way in which representations are defined. The  $\mathfrak{R}_f$  representation definition has many constructs (such as the `FDSOURCE` construct) that exist only to provide convenient generalisations of other constructs. None of these constructs are explicitly used in rules, so it could be argued that they are not absolutely necessary. That is, the variant of  $\mathfrak{R}_f$  used here may be more complex than necessary. Of course, removing constructs from the definition of  $\mathfrak{R}_f$  will by definition produce a new representation  $\mathfrak{R}'_f$  that has different associations among its constructs.

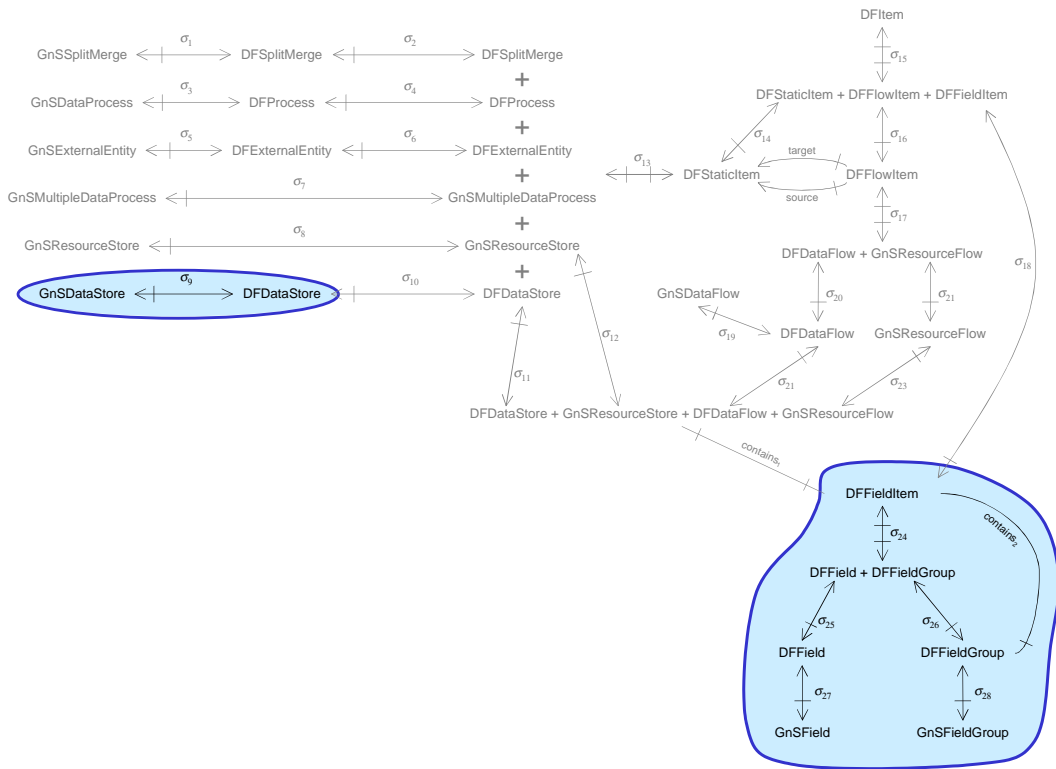
## 8.4 Measuring the relative quality of translations

The process described above for categorising the expressive overlap of two representations also provides a possible method for measuring the relative quality of translations. Consider the isomorphisms between  $\mathfrak{R}_d(\text{DataFlow}, \text{DFD}_{G\&S})$  and  $\mathfrak{R}_e(\text{E-R}, \text{ERD}_{\text{Martin}})$  identified in Figure 8.6 on the preceding page, and suppose that a description is to be translated from  $\mathfrak{R}_d$  to  $\mathfrak{R}_e$ , or vice versa. The untransformed subgraphs of the  $\mathfrak{R}_d$  SIG shown in Figure 8.5(a) on page 229 and the corresponding subgraphs of the  $\mathfrak{R}_e$  SIG shown in Figure 8.6(a) are indicated by the blue shaded areas in Figure 8.7. These shaded areas are derived from the isomorphism identified above, so they represent the expressive overlap between the two representations. This expressive overlap may be increased by the use of heuristics, which will be shown in Section 8.5.

Simple nodes of the SIG for a representation definition correspond to constructs of that representation, constructed nodes correspond to collections of constructs and the edges between nodes correspond to the associations between constructs. Counting the total number of nodes within an isomorphic subgraph gives an indication of the de-



(a)  $\mathfrak{R}_e(E-R, ERD_{Martin})$



(b)  $\mathfrak{R}_d(DataFlow, DFD_{G\&S})$

**Figure 8.7:** Expressive overlap between  $\mathfrak{R}_e$  and  $\mathfrak{R}_d$

gree of expressive overlap of the two representations, whereas counting the number of simple nodes gives an indication of the proportion of individual constructs that may potentially be mapped from one representation to the other. Similarly, counting the number of edges gives an indication of the number of associations between constructs that may potentially be mapped, while counting the number of non-selection (non- $\sigma$ ) edges gives an indication of the number of ‘useful’ associations between constructs that may potentially be mapped (selection edges are at present used here only to represent specialisation and generalisation of constructs). These numbers will then give an indication of the quality of any translations between the two representations. This is not an absolute measure of a translation’s quality because it is based on relative information capacity, which is a relative measure. It should therefore be considered as a measure of the *relative* quality of a translation that is only meaningful when compared with the relative qualities of other translations.

Thus, for the  $\mathfrak{R}_e \rightleftharpoons \mathfrak{R}_d$  translation, a visual comparison of the two subgraphs shown in Figure 8.7 gives the results shown in Table 8.2. The ‘#’ columns show the number of nodes, constructs, edges and non- $\sigma$  edges obtained by applying the process outlined above. The ‘%’ columns show these numbers as a proportion of the total number of nodes, constructs, edges and non- $\sigma$  edges (which are shown at the head of the table). These numbers indicate that a description translation from  $\mathfrak{R}_e$  to  $\mathfrak{R}_d$  will in general be of relatively higher quality than a description translation from  $\mathfrak{R}_d$  to  $\mathfrak{R}_e$ .

**Table 8.2:** Relative quality measurements for the  $\mathfrak{R}_e \rightleftharpoons \mathfrak{R}_d$  translation

---

SIG for  $\mathfrak{R}_e(E-R, ERD_{Martin})$  has 21 nodes, 17 distinct constructs, 26 edges and 9 non- $\sigma$  edges.  
 SIG for  $\mathfrak{R}_d(DataFlow, DFD_{G\&S})$  has 26 nodes, 21 distinct constructs, 32 edges and 4 non- $\sigma$  edges.

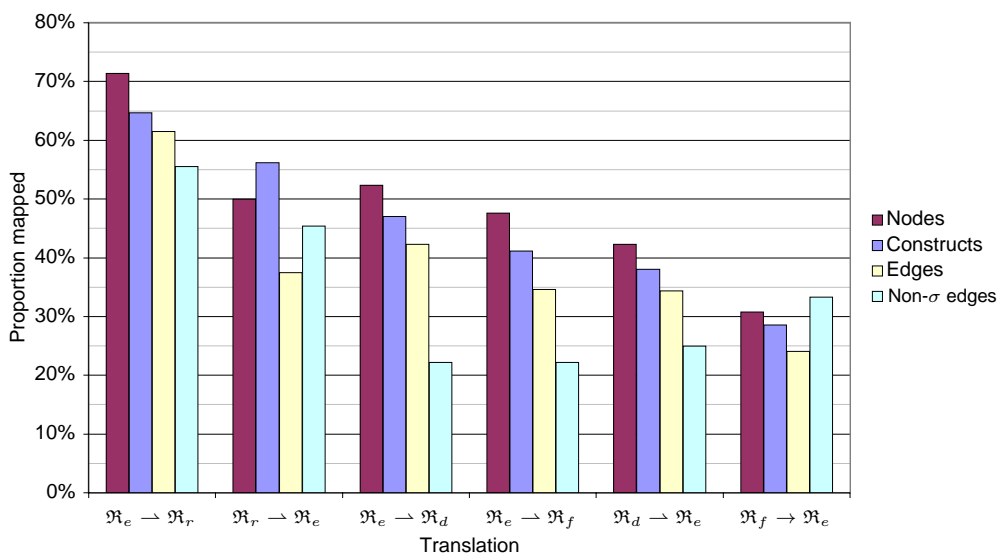
| Direction                                   | Nodes |       | Constructs |       | Edges |       | Non- $\sigma$ edges |       |
|---|-------|-------|------------|-------|-------|-------|---------------------|-------|
|   | #     | %     | #          | %     | #     | %     | #                   | %     |
| $\mathfrak{R}_e \rightarrow \mathfrak{R}_d$ | 11    | 52.38 | 8          | 47.06 | 11    | 42.31 | 2                   | 22.22 |
| $\mathfrak{R}_d \rightarrow \mathfrak{R}_e$ | 11    | 42.31 | 8          | 38.10 | 11    | 34.38 | 1                   | 25.00 |

---

### 8.4.1 Initial analysis

The relative quality measurement described above has been applied to the three translations discussed in Chapter 5. The analysis for  $\mathfrak{R}_e \rightleftharpoons \mathfrak{R}_d$  was presented above, and analyses for the remaining two translations may be found in Appendix E. The results

of these analyses are summarised in Figure 8.8, in which is shown the relative quality for each translation in both the forward and reverse directions. The results in the graph have been sorted in approximate descending order of relative quality. For each translation, the proportion of nodes, distinct constructs, edges and non-selection (non- $\sigma$ ) edges that may be mapped from the source representation to the target representation is shown.



**Figure 8.8:** Summary of initial relative translation quality measurements (without heuristics)

Based on an examination of the translation definitions and experimentation with the Swift environment, the initial intuitive expectation was that the translations would be ranked in terms of quality in approximately the following order:  $\mathfrak{R}_e \rightleftharpoons \mathfrak{R}_r$ ,  $\mathfrak{R}_f \rightarrow \mathfrak{R}_e$ ,  $\mathfrak{R}_e \rightarrow \mathfrak{R}_f$  and  $\mathfrak{R}_e \rightleftharpoons \mathfrak{R}_d$ . As can be seen from Figure 8.8, however, this is not borne out by the analysis. Instead, it can be seen that the  $\mathfrak{R}_f \rightarrow \mathfrak{R}_e$  translation has the worst relative quality, which was unexpected, as experimentation with the prototype environment suggested a much higher quality than that shown in Figure 8.8. Conversely, the  $\mathfrak{R}_e \rightleftharpoons \mathfrak{R}_r$  and  $\mathfrak{R}_e \rightleftharpoons \mathfrak{R}_d$  translations are ranked in the expected order. It is also interesting to observe the differences in relative quality between the forward and reverse translations. For example, the relative quality of the  $\mathfrak{R}_r \rightarrow \mathfrak{R}_e$  translation is lower than that of the  $\mathfrak{R}_e \rightarrow \mathfrak{R}_r$  translation, which was expected because of the problems inherent in reverse engineering SQL schemas.

The most likely reason for this discrepancy between expectations and results is the

granularity problem identified in Section 8.3. A considerable proportion of the rules for the  $\mathfrak{R}_f \rightarrow \mathfrak{R}_e/\mathfrak{R}_e \rightarrow \mathfrak{R}_f$  translation (see Appendix E) involve ‘collective constructs’ rather than individual constructs. If this is the problem, then these ‘collective constructs’ must be taken into account during the measurement.

### 8.4.2 Improving the relative quality measurement

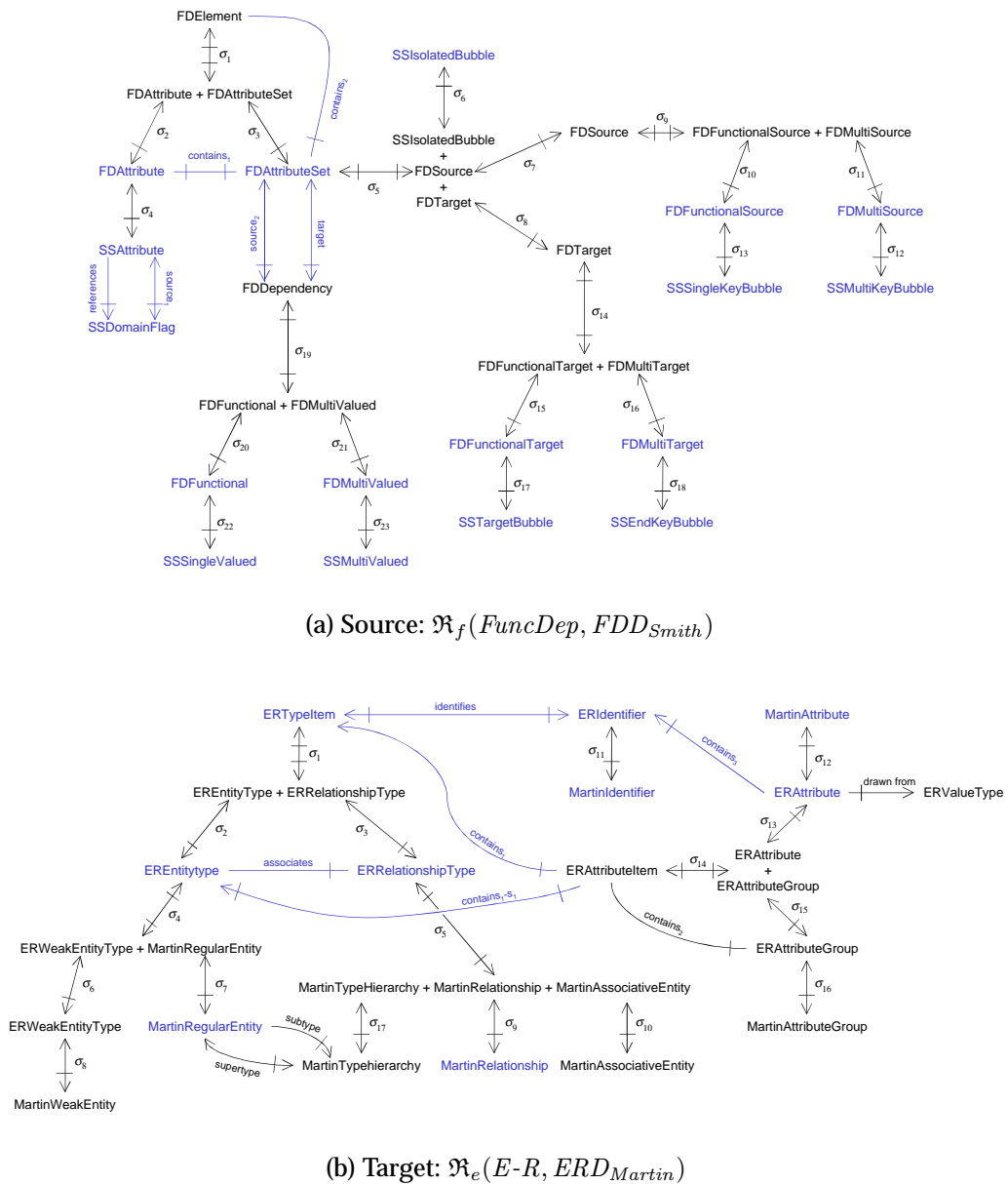
Improving the relative quality measurement can be achieved by addressing the issue of different levels of granularity across representations. The information for determining the ‘collective constructs’ needed to resolve this issue already exists in the rules and heuristics of the translation definitions in Chapter 5 and Appendix E, and may be used without altering the SIGs for the representations. It is therefore possible to modify the previously defined relative quality measurement to take into account the ‘collective constructs’. SIGs for each representation definition are built as usual. The following process is then applied for the forward and reverse directions of the translation:

1. In the source SIG, tag all construct nodes that correspond to source constructs of a rule that can be applied in this direction.
2. In the target SIG, tag all construct nodes that correspond to target constructs of a rule that can be applied in this direction.
3. For each non- $\sigma$  edge in both SIGs:
  - (a) if both end nodes are tagged, also tag the edge; or
  - (b) if only one end node  $n_1$  is tagged but there is a path of selection edges from the other end node  $n_2$  to a similarly-tagged node corresponding to a specialisation of the construct represented by  $n_2$ , also tag the edge (this means that there is at least one tagged specialisation of the untagged construct); or
  - (c) if neither end node is tagged, but there are paths of selection edges from both end nodes to similarly-tagged nodes corresponding to specialisations of the constructs represented by the end nodes, also tag the edge.

This process is illustrated in Figure 8.9 for the translation  $\mathfrak{R}_f(\text{FuncDep}, \text{FDD}_{\text{Smith}}) \rightarrow \mathfrak{R}_e(\text{E-R}, \text{ERD}_{\text{Martin}})$ ; tagged nodes and edges are indicated by colouring them blue.



Note that this process only considers construct nodes and non-selection edges, which is to be expected. Composite nodes and selection edges are at present only used in the SIGs to represent generalisation/specialisation, which is taken into account by steps 3(b) and 3(c) above. Heuristics are not included in Figure 8.9 and will be examined shortly.



**Figure 8.9:** Applying the modified relative quality measurement (ignoring heuristics)

The final step is to count all tagged source nodes, source edges, target nodes and target edges. The results of this process for the  $\mathfrak{R}_f \rightarrow \mathfrak{R}_e/\mathfrak{R}_e \rightarrow \mathfrak{R}_f$  translation are

shown in Table 8.3. The numbers in the ‘source’ columns show the proportion of source constructs and associations between constructs that can be translated from the source representation by the translation (the ‘#’ and ‘%’ columns show the actual number of tagged constructs and edges and proportion of the total number of constructs and edges respectively). The numbers in the ‘target’ columns show the proportion of target constructs and associations that can be generated in the target representation by the translation.

**Table 8.3:** Relative quality measurements for the  $\mathfrak{R}_f \rightarrow \mathfrak{R}_e/\mathfrak{R}_e \rightarrow \mathfrak{R}_f$  translation using the modified measurement method

---

SIG for  $\mathfrak{R}_f$  (*FuncDep*, *FDD<sub>Smith</sub>*) has 21 construct nodes and 6 non- $\sigma$  edges.  
 SIG for  $\mathfrak{R}_e$  (*E-R*, *ERD<sub>Martin</sub>*) has 17 construct nodes and 9 non- $\sigma$  edges.

---

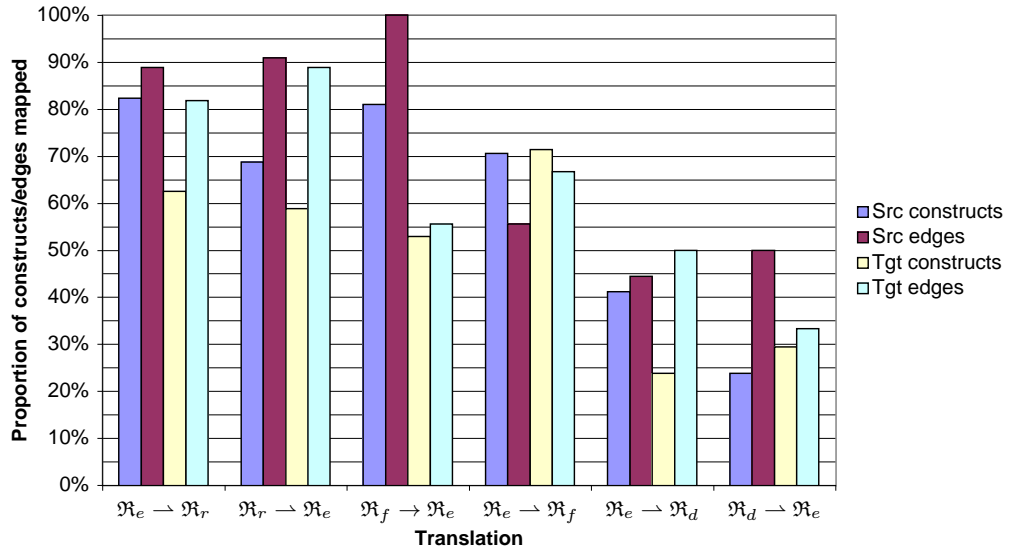
| Direction                                   | Tagged            |       | Tagged       |        | Tagged            |       | Tagged       |       |
|---|-------------------|-------|--------------|--------|-------------------|-------|--------------|-------|
|   | source constructs |       | source edges |        | target constructs |       | target edges |       |
|   | #                 | %     | #            | %      | #                 | %     | #            | %     |
| $\mathfrak{R}_f \rightarrow \mathfrak{R}_e$ | 17                | 80.95 | 6            | 100.00 | 9                 | 52.94 | 5            | 55.56 |
| $\mathfrak{R}_e \rightarrow \mathfrak{R}_f$ | 12                | 70.59 | 5            | 55.56  | 15                | 71.43 | 4            | 66.67 |

---

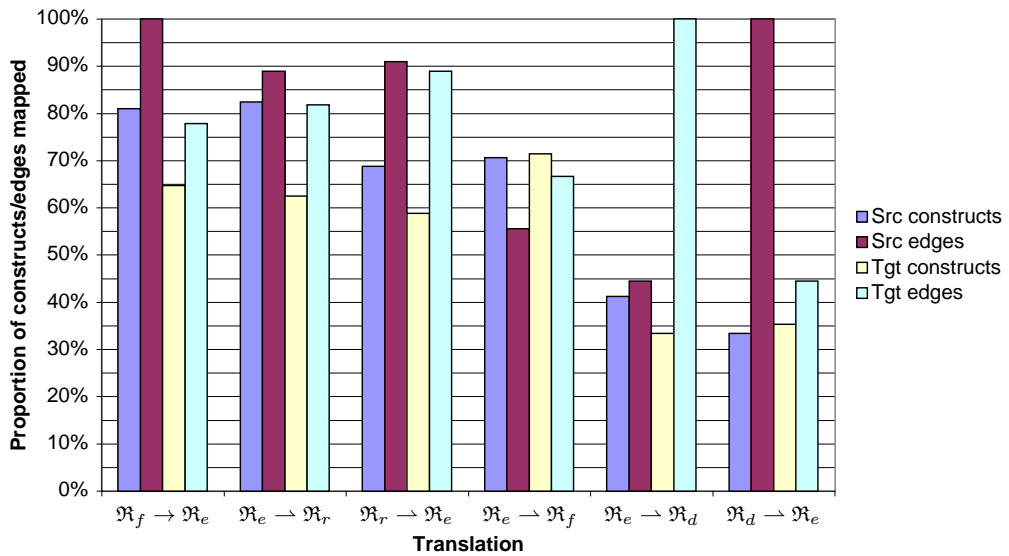
Applying the modified relative quality measurement to the three translations gives the results shown in Figure 8.10(a). Once again, the translations have been ranked in approximate descending order of relative quality, and this time the order matches the original expectations. Further refinement of the relative quality measurement is discussed in Chapter 10, where the influence of the number of complete and partial rules in a translation is discussed.

## 8.5 The effect of heuristics on translation quality

Heuristics may allow an expansion of the expressive overlap between two representations, as they can map additional constructs between the representations that otherwise could not be mapped. That is, heuristics allow more information to be mapped from one representation to another, which should have a beneficial impact on the quality of translations between the representations. In theory, the introduction of heuristics could potentially even convert a partial translation into a complete translation, although this will depend on the nature of the translation and the heuristics involved.



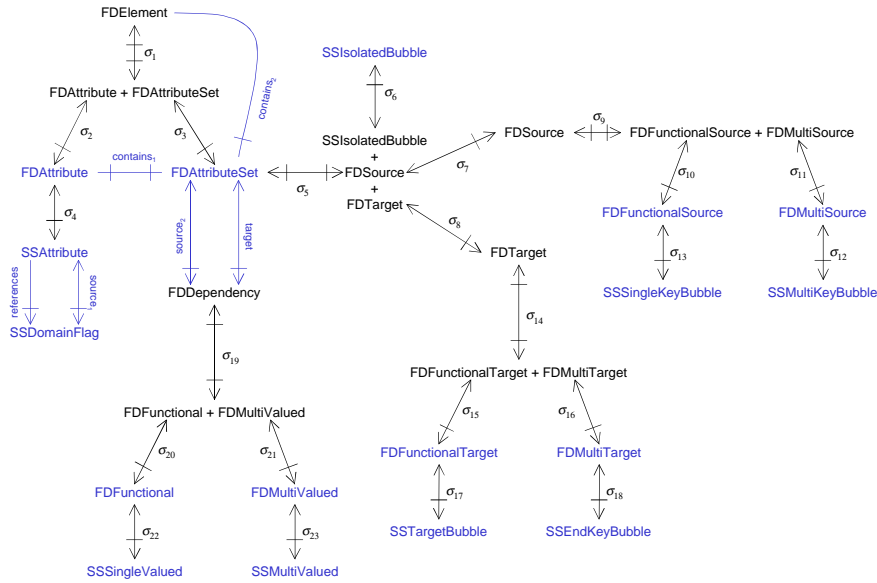
(a) Without heuristics



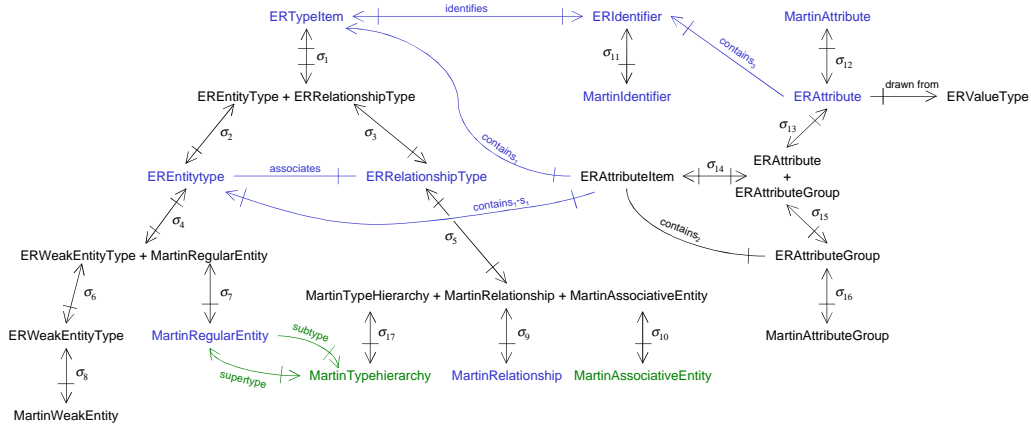
(b) With heuristics

**Figure 8.10:** Summary of modified relative translation quality measurements

In Figure 8.11 is shown the impact of heuristics on the relative quality of the translation  $\mathfrak{R}_f(FuncDep, FDD_{Smith}) \rightarrow \mathfrak{R}_e(E-R, ERD_{Martin})$ . The new nodes and edges that are translated by heuristics are coloured green. The results are summarised in Table 8.4.



(a) Source:  $\mathfrak{R}_f(FuncDep, FDD_{Smith})$



(b) Target:  $\mathfrak{R}_e(E-R, ERD_{Martin})$

**Figure 8.11:** Applying the relative quality measurement (with heuristics)

Since heuristics are unidirectional, they can obviously only affect the relative quality of a translation when it is evaluated in the appropriate direction. The impact of heuristics on the relative qualities of the three translations discussed above is shown in Figure 8.10(b) on the preceding page, in approximate descending order of relative

**Table 8.4:** Relative quality measurements for the  $\mathfrak{R}_f \rightarrow \mathfrak{R}_e/\mathfrak{R}_e \rightarrow \mathfrak{R}_f$  translation (with heuristics)

---

SIG for  $\mathfrak{R}_f$  (*FuncDep*, *FDD<sub>Smith</sub>*) has 21 construct nodes and 6 non- $\sigma$  edges.  
 SIG for  $\mathfrak{R}_e$  (*E-R*, *ERD<sub>Martin</sub>*) has 17 construct nodes and 9 non- $\sigma$  edges.

| Direction                                   | Tagged            |       | Tagged       |        | Tagged            |       | Tagged       |       |
|---|-------------------|-------|--------------|--------|-------------------|-------|--------------|-------|
|   | source constructs |       | source edges |        | target constructs |       | target edges |       |
|   | #                 | %     | #            | %      | #                 | %     | #            | %     |
| $\mathfrak{R}_f \rightarrow \mathfrak{R}_e$ | 17                | 80.95 | 6            | 100.00 | 11                | 64.71 | 7            | 77.78 |
| $\mathfrak{R}_e \rightarrow \mathfrak{R}_f$ | 12                | 70.59 | 5            | 55.56  | 15                | 71.43 | 4            | 66.67 |

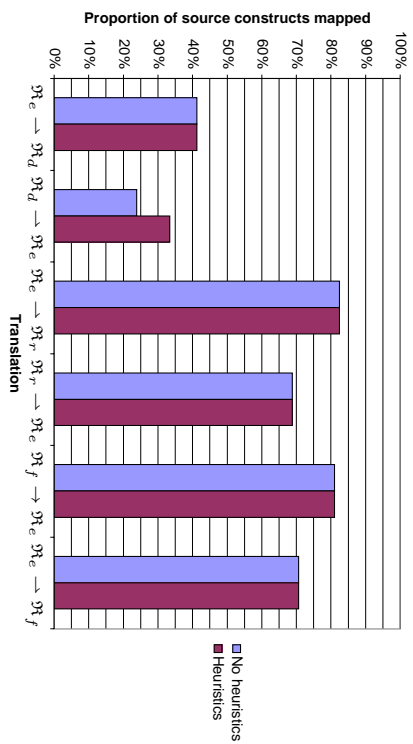
---

quality. The only change in the ordering is that the  $\mathfrak{R}_f \rightarrow \mathfrak{R}_e$  translation moves into first position; the relative ordering of the remaining translations is unchanged. Despite this, the impact of heuristics upon the relative quality of some translations is dramatically evident. The effect of heuristics was also evident when applying translations in the Swift prototype environment (see Section 6.7.1 on page 175).

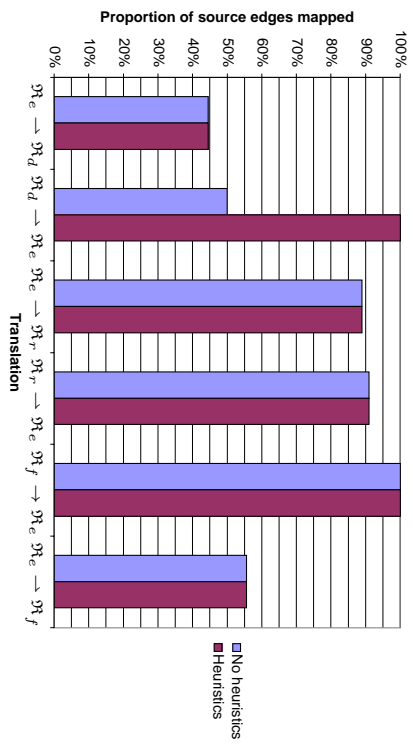
The impact of heuristics on individual translation quality is shown more clearly in Figure 8.12 on the following page, which breaks down the impact in terms of the source constructs and edges mapped, shown in Figures 8.12(a) and 8.12(b), and the target constructs and edges generated, shown in Figures 8.12(c) and 8.12(d). There is no impact on the  $\mathfrak{R}_r \rightarrow \mathfrak{R}_e$  and  $\mathfrak{R}_e \rightarrow \mathfrak{R}_f$  translations when heuristics are included, because there are no heuristics to be applied in those directions. The remaining translations are affected to varying degrees, depending on the number of heuristics and whether these heuristics allow the mapping of any additional constructs. For instance, the heuristics of the  $\mathfrak{R}_e \rightarrow \mathfrak{R}_d$  translation do not result in the mapping of any additional source constructs or edges, but do result in additional information being generated in the target.

The measurements undertaken above have shown that heuristics can result in more constructs and edges being mapped from the source representation, and more constructs and edges being generated in the target representation. Translation quality is defined in this thesis as how well a translation maps constructs from the source representation to the target representation (see Section 2.4.1 on page 28). The above results show that including heuristics in a translation increases the amount of information translated, and thus improves the quality of the translation.

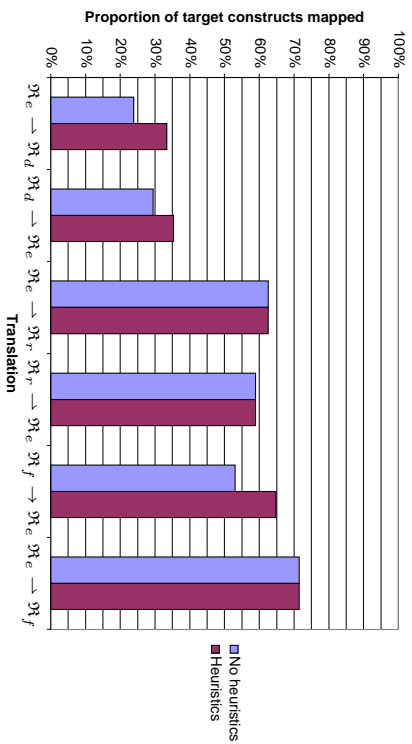
It should be remembered, however, that this definition of quality applies only to



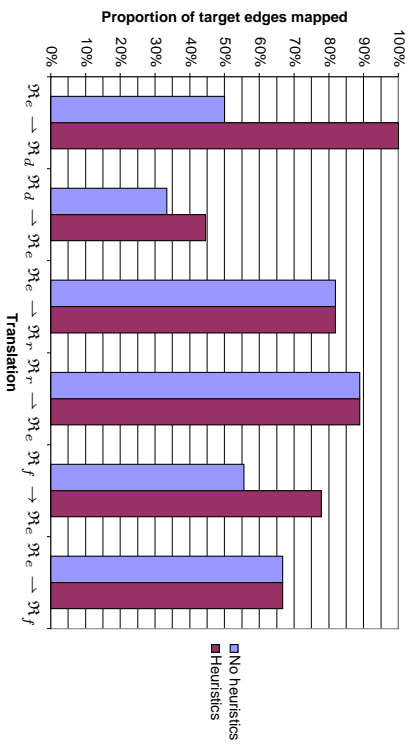
(a) Source constructs



(b) Source edges



(c) Target constructs



(d) Target edges

Figure 8.12: Impact of heuristics on relative translation quality

translations of descriptions, and does not take into consideration the effect that a heuristic may have on the semantics of a *viewpoint*. By definition, a heuristic may not always produce a semantically consistent result. Including heuristics in the enrichment process reduces the likelihood of such inconsistent results being produced, but it may still be possible for heuristics to be inappropriately applied, thus producing an inconsistent viewpoint.

The SIG formalism does not capture the changes in viewpoint semantics caused by heuristics. This is because SIGs only model representations, not viewpoints, and it is the total content of a viewpoint that determines whether a semantic change is consistent or not. This line of inquiry, while interesting, has been left as an area for future research.

## 8.6 Summary

In this chapter, methods for categorising the expressive overlap of two representations and measuring the relative quality of translations were defined and applied. These methods are based on Hull's (1986) concept of *relative information capacity*, which provides a means of describing the relative information content of different schemas, and Miller et al.'s (1994b) notion of a *schema intension graph*, which provides a means of measuring differences in relative information capacity.

It was shown that the expressive power of a representation may be characterised by the information capacity of the representation's definition, which means that the methods developed for comparing relative information capacity of schemas are also applicable to representations. In particular, it was shown in Section 8.3 that the category of expressive overlap between two representations may be determined using schema intension graphs. From this basis, the author defined a method using schema intension graphs for measuring the relative quality of translations between representations. This measure was further refined and used to show that heuristics have a positive impact on translation quality.

There are outstanding issues with both the method for categorising expressive overlap and the relative quality measure, which will be discussed further in Chapter 10. In the next chapter, the approach taken in this thesis will be evaluated with respect to the goals of the thesis.





# Chapter 9

## Evaluation of the proposed modelling approach

### 9.1 Introduction

In the previous chapters has been outlined an approach to modelling a viewpoint based on the use of multiple representations and translating descriptions between these representations. A prototype implementation was described, and a method for measuring the relative quality of translations was defined and applied. In this chapter, this approach is evaluated against several goals of the thesis.

The goals of the thesis are:

1. Improve a viewpoint in terms of depth and detail by using multiple representations to describe the viewpoint.
2. Facilitate the use of multiple representations within a viewpoint by translating descriptions between representations.
3. Develop a consistent and unified terminology for discussing representations and translations between them. This has already been discussed in detail in Chapters 3 and 4, and will not be discussed further here.
4. Develop effective ways of specifying translations. This has already been discussed in Chapters 4, 5 and 7.
5. Use translations to enable the highlighting of potential inconsistencies between descriptions of a viewpoint.

6. Identify ways of improving translation quality. This was examined in detail in the previous chapter.
7. Show that the approach presented here is novel and practicable.

The translation-based approach is evaluated against goals 1, 2 and 5 in Section 9.2, in which is presented a complete case study of using Swift to build a new viewpoint. This study will show examples of improving the depth and detail of a viewpoint, facilitating the use of multiple representations and highlighting potential inconsistencies between descriptions.

The novelty of the approach (the first part of goal 7) is examined in Section 9.3. A survey of five commercial CASE tools was undertaken to determine the extent to which commercially available tools facilitate the use of multiple representations, and the results are summarised here. This summary is followed by a more detailed examination of the three research-based approaches that were briefly discussed in Chapter 2: MViews, MDM and ORECOM. The approach taken in this thesis is then compared with the surveyed CASE tools and the three research-based approaches.

The practicability of the approach (the second part of goal 7) is discussed in Section 9.4. A general discussion of the tractability of the translation-based approach is presented, followed by a series of empirical experiments that test the time complexity of translations within the Swift environment.

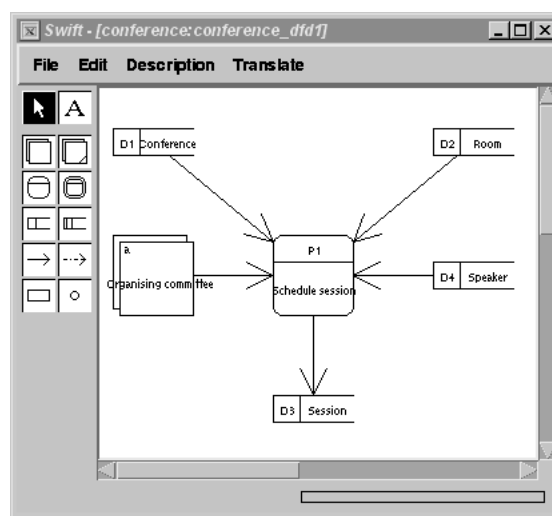
## 9.2 Case study

In this section, the translation-based approach to facilitating the use of multiple representations is evaluated using a case study. The case study comprises the building of a single viewpoint for a small conference centre that hosts conferences for other organisations. An analyst has been hired to design a conference management system for the conference centre. The analyst interviews relevant stakeholders and collects a large amount of information about user requirements. She then proceeds to build a viewpoint  $V_{conf}$  describing these requirements.

The conference centre merely provides the facilities and organisational infrastructure for running a conference; they do not organise the conferences themselves. Clients

who wish to make use of the centre’s facilities are expected to submit details of the conference, such as a proposed timetable of conference sessions, a list of speakers and the number of attendees. The conference centre assigns conference sessions to appropriate rooms and organises other aspects of the day-to-day running of the conference, such as ensuring that appropriate equipment is installed in each room for each session.

The analyst starts with the process of assigning rooms for conference sessions, and builds a data flow diagram (DFD) description  $D_1(V_{conf}, DataFlow, DFD_{G\&S})$  to represent this process; this DFD is shown in Figure 9.1. The analyst populates the data stores with the attributes shown in Table 9.1.



**Figure 9.1:** DFD description  $D_1(V_{conf}, DataFlow, DFD_{G\&S})$  for the process of assigning conference sessions to rooms

**Table 9.1:** Attributes for  $D_1(V_{conf}, DataFlow, DFD_{G\&S})$

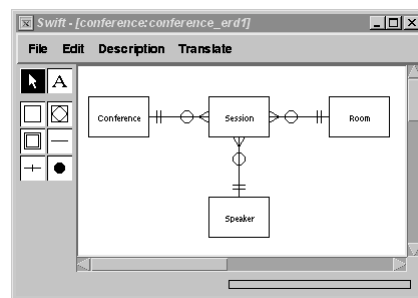
|                     |         |                |        |
|---------------------|---------|----------------|--------|
| <b>Conference</b>   |         | <b>Speaker</b> |        |
| conference ID       | string  | speaker ID     | string |
| conference name     | string  | speaker name   | string |
| number of attendees | integer |                |        |
| <b>Room</b>         |         | <b>Session</b> |        |
| room number         | string  | session ID     | string |
| room name           | string  | session date   | date   |
| capacity            | integer | session time   | time   |

The analyst then sets out to identify any relationships between ‘objects’ in the viewpoint. This information cannot be expressed by the representation  $\mathfrak{R}_d(DataFlow,$

$DFD_{G\&S}$ ) used for description  $D_1$ , so the analyst translates the description  $D_1$  into an entity-relationship diagram (ERD) description  $D_2(V_{conf}, E-R, ERD_{Martin})$ .

The data stores of  $D_1$  are mapped to entities in  $D_2$  using rule S1, and in a typical DFD to ERD translation this is all that would be mapped. Heuristics may be used, however, to translate additional elements of the DFD. For example, the data flows from the Conference data store to the Session data store via the Schedule session process can be mapped by heuristic H2 to a relationship between the Conference and Session entities. The same applies to the data flows from Room to Session, and from Speaker to Session.

Each time a heuristic is applied, the proposed mapping is presented to the analyst, who must then decide whether this is a valid mapping or not. In this case, after consulting her analysis notes, she decides that all three mappings identified in the previous paragraph are valid and accepts them. The ERD description  $D_2$  produced by the translation is shown in Figure 9.2.

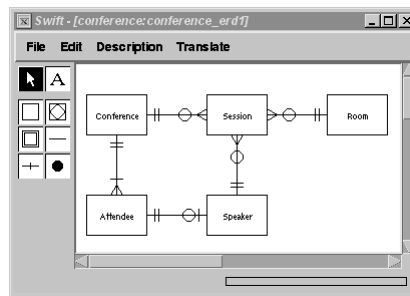


**Figure 9.2:** ERD description  $D_2(V_{conf}, E-R, ERD_{Martin})$  produced by translating description  $D_1$

Comparing description  $D_2$  against her notes, the analyst realises that the ERD does not cater for conference attendees. She also notes that all speakers are attendees, but not all attendees are speakers, and identifies three options for modelling this:

1. Make speakers and attendees separate subtypes of a Person entity;
2. Make speakers and attendees separate entities linked by a one-to-one relationship; or
3. Combine speakers and attendees into a single entity, with a flag attribute to indicate whether an attendee is a speaker.

After further discussion with the clients, she decides that there is insufficient difference between the data requirements for speakers and attendees to justify either of the first two options, and adopts the third option. The Speaker entity is therefore renamed to Attendee and the attributes `is_speaker` and `registered` are added to this entity. A conference may have many attendees, so a one-to-many relationship is added from Conference to Attendee. The modified ERD ( $D'_2$ ) is shown in Figure 9.3.

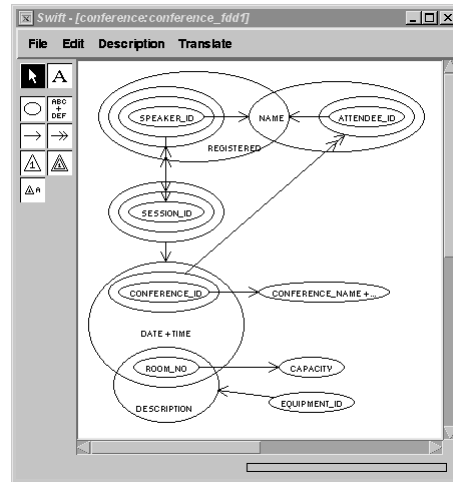


**Figure 9.3:** ERD description  $D'_2$  produced by modifying  $D_2$

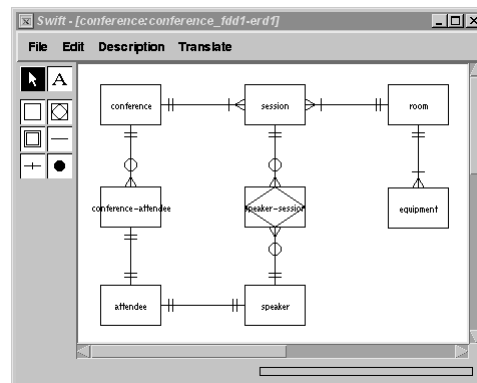
The analyst now decides to evaluate the consistency of the viewpoint  $V_{conf}$  by building a new functional dependency diagram (FDD) description  $D_3(V_{conf}, FuncDep, FDD_{Smith})$ , translating  $D_3$  to produce a new ERD description  $D_4(V_{conf}, E-R, ERD_{Martin})$ , and comparing  $D_4$  with with the existing ERD description  $D'_2$ . From her analysis notes, she derives the following list of dependencies:

- *attendee ID*  $\rightarrow$  *session ID*
- *session ID*  $\rightarrow$  *attendee ID*
- *session ID*  $\rightarrow$  *room number, date, time, conference ID*
- *conference ID*  $\rightarrow$  *conference name, number of attendees*
- *conference ID*  $\rightarrow$  *attendee ID*
- *room number*  $\rightarrow$  *capacity*
- *attendee ID*  $\rightarrow$  *name, is speaker, registered*
- *equipment ID*  $\rightarrow$  *room number, description*

The FDD description ( $D_3$ ) corresponding to these dependencies is shown in Figure 9.4. This FDD is then translated into the ERD description ( $D_4$ ) shown in Figure 9.5. Note that only the Attendee-session associative entity is produced by the application of a heuristic; the remainder of the ERD is produced by the application of ‘plain’ rules.



**Figure 9.4:** FDD description  $D_3(V_{conf}, FuncDep, FDD_{Smith})$



**Figure 9.5:** ERD description  $D_4(V_{conf}, E-R, ERD_{Martin})$  produced by translating description  $D_3$

This new ERD ( $D_4$ ) is clearly different from  $D'_2$  shown in Figure 9.3. The analyst uses her knowledge of the user requirements to determine whether these differences are caused by inconsistencies between the two descriptions. The following potential inconsistencies have been highlighted as a result of applying the translation:

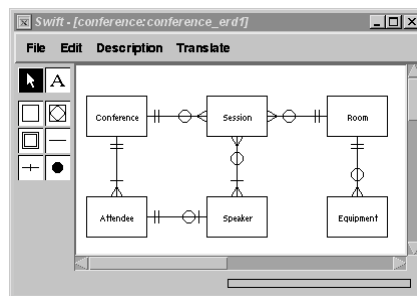
1. The optionality of several of the relationships is different in both descriptions.
2. The Conference-attendee, Attendee-speaker and Equipment entities all appear in  $D_4$  but not in  $D'_2$ .

The analyst decides that the differences in optionality are caused by limitations of the translation rather than being actual inconsistencies. In addition, the relationships attached to the Conference-attendee entity in  $D_4$  prove upon examination to be consistent with the relationship between Conference and Attendee in  $D'_2$ . Further examination reveals that Conference-attendee does not add any new data requirements and is therefore unnecessary.

The Attendee-session entity in  $D_4$  is more interesting as it corresponds to the relationship between Session and Attendee in  $D'_2$ . Comparing the relationships in both ERDs, the analyst realises that there is indeed an inconsistency. The relationship in  $D'_2$  is a one-to-many, but the relationships attached to Attendee-session in  $D_4$  imply a many-to-many relationship between Session and Attendee. Upon reflection, the analyst realises that there should indeed be a many-to-many relationship between Session and Attendee, and amends  $D'_2$  appropriately.

The Equipment entity in  $D_4$  is totally new; it does not correspond to anything in  $D'_2$ . The analyst returns her notes and discovers that she inadvertently omitted a requirement for rooms to have various pieces of equipment assigned to them. The Equipment entity should therefore be added to  $D'_2$ .

The amendments described above produce the description  $D''_2$  shown in Figure 9.6.



**Figure 9.6:** ERD description  $D''_2$  produced by correcting inconsistencies

### 9.2.1 Discussion

The example presented in this section, while relatively simple, has illustrated the following important points. First, using multiple representations to describe the viewpoint  $V_{conf}$  has produced a viewpoint with greater detail than would otherwise be possible using a single representation. For example, it would be impossible to describe

the data flow information embodied in the DFD description  $D_1$  if only an ERD representation were used to describe the viewpoint. In addition, the use of heuristics has allowed the extraction of information implicit in the source description that might not otherwise have been translated, for example, the relationships produced when translating description  $D_1$  into the ERD description  $D_2$ . That is, the depth and detail of the viewpoint has been increased as a direct result of applying a translation.

Second, the use of translations has facilitated the use of multiple representations by allowing the analyst to easily create new descriptions based on already existing descriptions. Even in cases where the quality of the translation is not particularly good (such as for the DFD to ERD translation), enough information is translated from the source description to provide a useful template on which to build, as illustrated by the translation from description  $D_1$  to  $D_2$  described above.

Third, the use of translations between representations has enabled the analyst to more easily identify inconsistencies between descriptions within the viewpoint, as illustrated by the comparison between the ERD descriptions  $D'_2$  and  $D_4$  above. It would be more difficult to identify the inconsistencies between the ERD description  $D'_2$  and the FDD description  $D_3$  if the translation were not available.

### **9.3 Novelty of the approach**

The idea of performing translations between different modelling representations is not new, and has been implemented in many tools. Indeed, many commercial CASE tools include translations, but these are often only used as the final step of the database construction process, for example, generation of SQL schemas from a database design. A goal of this thesis was to use translations to facilitate the use of multiple representations, which is something that many CASE tool do not do. In Section 9.3.1, the results of a survey of five 'conventional' CASE tools are presented (the detail of the survey may be found in Appendix A).

Other researchers have also worked on the idea of performing translations between representations. In Chapter 2 were briefly discussed three other approaches similar to that presented here. These three approaches are examined in more detail in Sections 9.3.2, 9.3.3 and 9.3.4.



The approach taken in this thesis is then compared to the three research approaches and conventional tools in Section 9.3.5.

### 9.3.1 Survey of ‘conventional’ CASE tools

In this section, the results of a survey of five ‘conventional’ CASE tools are presented (full details of the survey may be found in Appendix A). The tools were surveyed to determine the extent to which they facilitate the use of multiple representations and their support for translations between representations. The tools surveyed were: Visible Systems’ EasyCASE, Visible Analyst and EasyER/EasyOBJECT; Sybase’s Deft; and MetaCase’s MetaEdit. EasyCASE and Deft were chosen because full versions were readily accessible; the remaining tools were chosen because comprehensive demonstration versions were available for download from the Internet.

Reiner (1992, p. 445) also surveyed a wide range of CASE tools (including Deft and EasyCASE), and found that many “depend on a set of fairly independent diagrams, without a coherent underlying methodology”, and that “there is little visualization [*sic*] of progress through the design process, or system tracking of alternative designs”. This implies a lack of support for facilitating the use of multiple representations.

The results of the CASE tool survey are summarised in Table 9.2 on the following page. Three of the CASE tools surveyed partially facilitate the use of multiple representations through the use of a common data dictionary that allows sharing of information between descriptions. All tools support multiple schemes for various techniques, and both Deft and EasyCASE allow the user to perform trivial scheme translations.

Most of the tools surveyed do not support non-trivial scheme translations, such as scheme translations between representations with different expressive powers. The only exception is EasyER/EasyOBJECT, which supports non-trivial scheme translations, but the translations it does support are very simple, allowing only one-to-one mappings between constructs. That is, it is not possible to map collections of source constructs to a single target construct or vice versa, as occurs, for example, in the  $\mathfrak{R}_f \rightarrow \mathfrak{R}_e/\mathfrak{R}_f \leftarrow \mathfrak{R}_e$  translation. It is also only possible to modify existing descriptions using these translations, not create new descriptions.

The only technique translations supported by most of the tools are from E-R repre-

**Table 9.2:** Summary of ‘conventional’ CASE tool features

|   | EasyCASE               | Visible Analyst      | EasyER/<br>EasyOBJECT | Deft                   | MetaEdit |
|---|------------------------|----------------------|-----------------------|------------------------|----------|
| No. of representations supported            | 23 <sup>a</sup>        | 10 <sup>a</sup>      | 13 <sup>a</sup>       | 9 <sup>a</sup>         | 15       |
| Supports trivial scheme translations        | yes                    | no                   | yes                   | yes                    | no       |
| Supports non-trivial scheme translations    | no                     | no                   | yes                   | no                     | no       |
| Supports technique translations             | yes <sup>b</sup>       | yes <sup>b</sup>     | yes <sup>c</sup>      | yes <sup>d</sup>       | no       |
| Facilitates use of multiple representations | partial <sup>e,f</sup> | partial <sup>f</sup> | yes <sup>g</sup>      | partial <sup>e,f</sup> | no       |

**Notes on Table 9.2:**

- <sup>a</sup> Each of these tools supports a large number of dialects of SQL, which have been treated as a single representation for the purposes of this summary. Although each dialect is, strictly speaking, a different representation, the large number of SQL dialects tends to skew the results in favour of tools that support many SQL dialects. For example, treating SQL dialects as separate representations would give EasyER/EasyOBJECT a total of fifty-seven representations, forty-five of which are very similar. Compare this with MetaEdit, which does not support SQL at all, but nevertheless supports a wide variety of representations.
- <sup>b</sup> E-R ↔ relational only.
- <sup>c</sup> Simple translations of corresponding constructs only.
- <sup>d</sup> E-R → relational only (the reverse is provided by an optional add-on product).
- <sup>e</sup> Integrated data dictionary shares information across descriptions.
- <sup>f</sup> Supports at least one technique translation.
- <sup>g</sup> Simple one-to-one mappings only, and translations do not allow the generation of new descriptions.

sentations to a database language (usually some dialect of SQL). Visible Analyst also provides the generation of COBOL or C source code from a collection of descriptions, and EasyER/EasyOBJECT provides very simple technique translations between E-R and object representations. EasyCASE, Visible Analyst and EasyER/EasyOBJECT directly support bidirectional translations from E-R to SQL; Deft provides this feature in an optional add-on module.

It could be argued that the data dictionary in many of these tools provides similar functionality to the author’s proposed translation approach, in that the data dictionary acts as a central repository for common information and ties together several different descriptions. In a contemporary CASE tool it is quite possible to create a new description that should share attributes with others, but does not. This can happen

when an unwary developer ‘recreates’ existing attributes with new names. The translation approach advocated here cannot prevent this, but it does provide a mechanism for highlighting such inconsistencies, as noted in Section 4.5 on page 86. It can also ameliorate the problem by allowing the developer to translate an existing description into the desired target representation, resulting in a template for further development. Such an activity is not supported by any of the CASE tools reviewed.

In addition, the data dictionaries of conventional tools provide a means of sharing common elements across *descriptions*, but not of associating constructs across *representations*. In effect, the data dictionaries of conventional tools work at a lower level of abstraction than the approach proposed in this thesis.

### 9.3.2 The MViews approach

The MViews framework (Grundy and Hosking, 1993b; Grundy and Hosking, 1993a; Grundy and Hosking, 1997) is particularly relevant to the work in this thesis, for several reasons:

- it is a recent and ongoing project;
- it supports multiple textual and graphical representations;
- multiple descriptions of a viewpoint may be constructed, and translations performed between them via an automatic update propagation mechanism; and
- the update propagation mechanism helps maintain design integrity by ensuring that multiple descriptions are kept as consistent as possible.

MViews appears to deal with many of the issues described in this thesis. The discussion in this section therefore focuses on how MViews differs from the author’s work.

Many of the concepts in MViews are derived from earlier work on integrated software development environments, such as PECAN (Reiss, 1985) and FIELD (Reiss, 1990b; Reiss, 1990a), and therefore the terms used in MViews are derived from software engineering rather than viewpoint-oriented methods. There is however a correspondence between the two terminologies, as noted in Table 3.4 on page 65. Current work with MViews is focused in the areas of (Grundy et al., 1996; Grundy and Hosking, 1996; Grundy, 1998; Hosking and Grundy, 1995; Hosking et al., 1995):

- architectural support for multiple viewpoint representations ('views' in MViews parlance);
- maintaining consistency between multiple graphical and textual representations;
- human interface issues such as how to present inconsistencies to users, how to support their interaction with inconsistencies, and so on;
- collaborative multiple viewpoint system issues; and
- various applications — mostly software engineering, but also building/architectural design, user interfaces and process modelling.

That is, the main focus is on the *implementation issues* that arise from facilitating the use of multiple representations. Translating between representations is part of this, and work in this area has concentrated on the issue of maintaining consistency across multiple descriptions (Grundy and Hosking, 1996). Related research has been undertaken in the area of specifying translations, resulting in the mapping specification language VML (Amor, 1997) that was used as the basis of the VML-S language defined in Chapter 7. Other than this, there has been little exploration of the translation process itself. Grundy (1998, personal communication) has stated that the translations already implemented were considered to be “straightforward”, and for this reason the issues arising from the translation process itself were not explored.

In contrast, a major focus of this research is on the *process* of translating descriptions between representations within a single viewpoint, and how the use of translations impacts upon the information system design and implementation process. Although the Swift prototype environment was implemented as part of the experimental work, it is not the main focus of the research and is intended merely as a vehicle for testing assertions about the translation process. A major aspect of the translation process is the quality of translations, which is something that the MViews researchers have not explored in depth. If something cannot be translated automatically by an MViews-based environment, the user is notified and must make appropriate changes manually. One contribution of this thesis is the use of heuristics and enrichment to improve the quality of translations (see Chapter 4).

MViews uses an integrated data model (IDM), which corresponds to the interchange format interfacing strategy described in Section 2.4.3 on page 30, although the MViews IDM uses an interesting hierarchical approach to integrating different representations (Grundy and Venable, 1995a, Figure 1). Each environment implemented using MViews has its own integrated data model, which is defined using a variant E-R representation named CoCoA (Venable, 1993; Venable and Grundy, 1995). Overlaps between the representations to be integrated are identified and used to create several partial integrated data models, which are themselves integrated to create the full IDM.

This hierarchical approach may alleviate some of the potential pitfalls of the interchange format interfacing strategy that were identified in Chapter 2, although adding a new representation still requires changes to the IDM (Grundy, 1998, personal communication). Conversely, an approach based on the individual interfacing strategy allows a new representation to be added with no effect on existing representations or the repository. It is also not clear how the integrated approach would scale to a large number of widely dissimilar representations. The published work describes tools that include only two or three fairly similar representations, although it is suggested that it is possible to integrate dissimilar representations using this approach (Grundy and Venable, 1995b).

MViews-based environments generally follow a synchronous approach to translations, although this is not required by the underlying consistency mechanism (Grundy et al., 1996; Grundy and Hosking, 1997). Thus, as changes are made to descriptions, these changes are automatically propagated to other affected descriptions in the viewpoint. The potential pitfalls of this approach have already been discussed in Section 3.6 on page 58, and will not be discussed further here. Swift's translations are asynchronous, so it is possible for descriptions to become temporarily inconsistent with each other. Translations are manually activated by the user at appropriate points, thus allowing more flexibility in the use of translations.

Translations in MViews-based environments have in the past been defined in an ad hoc fashion, but there is now a push to integrate VML support, which should result in more precisely specified translations. Grundy has already expressed interest in incorporating some of the ideas developed in this thesis (Grundy, 1998, personal communication).

Recent work with MViews has been directed toward implementing a Java component-based environment for building multi-view editing systems (Grundy et al., 1997b). This environment comprises three parts: a visual notation editor called Build-ByWire, a 'componentware' toolkit called JViews, and a visual editor for building JViews components called JComposer. JViews includes support for a repository based on an object-oriented DBMS, although it is not stated which one (Grundy et al., 1997b). MViews' approach to repository implementation is similar to that for its integrated data model: individual repositories are created for each representation, and these are then integrated to form an integrated repository (Grundy and Venable, 1995a).

In summary, the approach taken in this thesis, while similar to that taken with the MViews framework, has focused more on the issues arising from the translation process and less on the implementation issues for dealing with multiple representations and translating between them. MViews has a greater emphasis on maintaining consistency among descriptions, and tools implemented using MViews generally follow a synchronous approach to consistency maintenance. MViews is built around an interchange format interfacing strategy using an integrated data model that is built specifically for each environment, whereas this research follows the individual interfacing strategy that provides potentially higher quality translations. The individual interfacing strategy can also provide more flexibility in the use of translations and makes it easier to extend an environment with new representations.

### 9.3.3 The MDM approach

Atzeni and Torlone's (1997) MDM environment facilitates the use of multiple representations using a translation-based approach. Representations are defined using a highly-abstract metamodel derived from Hull and King's (1987) Generic Semantic Model, which was in turn derived from the IFO model (Abiteboul and Hull, 1987). This metamodel comprises at least the following four metaconstructs (Atzeni and Torlone, 1993):

**Lexical:** allows the definition of constructs whose elements have printable values, such the RMDOMAIN construct.

**Abstract:** allows the definition of constructs whose elements have non-printable values, such as the `ERENTITYTYPE` or `DFEXTERNALENTITY` constructs.

**Aggregation:** allows the definition of constructs whose elements comprise sets of tuples of simpler elements. The components of an aggregation may comprise lexical, abstract or aggregation constructs. The `RMRELATION` construct is an example of an aggregation.

**Function:** allows the definition of constructs whose elements are functions from one or more elements to another element. The `RMATATTRIBUTE` construct is an example of a function that maps `RMRELATION` constructs to `RMDOMAIN` constructs.

Atzeni and Torlone (1996b) later added the *grouping* and *hierarchy* metaconstructs, but no definition of these was provided. It is claimed that these six metaconstructs are adequate to define most representations (Atzeni and Torlone, 1996b); if a representation is discovered with constructs that do not fit into the existing metamodel, it must be extended with new metaconstructs. It is expected, however, that this would not happen very often.

Atzeni and Torlone (1996b, p. 122) claim that constructs corresponding to the same metaconstruct will have the same semantics. This seems reasonable when comparing, for example, an `ERENTITYTYPE` construct with an object-oriented class — these both correspond to the *abstract* metaconstruct, and have similar semantics. It is not so clear, however, when comparing an `ERENTITYTYPE` construct with a `DFEXTERNALENTITY` construct. While both of these would appear to correspond to the *abstract* metaconstruct, the semantics of the two constructs are quite different. An external entity represents something that is by definition ‘outside’ the description, yet it may provide data required by other elements in the description. Entities, by contrast, are ‘internal’ to a description. In fact, it can be shown that the `ERENTITYTYPE` construct corresponds more closely to the `DFDATASTORE` construct (see Appendix E).

The likely reason for this problem is that Atzeni & Torlone have focused purely on ‘structural’ semantic data models, in particular the E-R approach, the relational model and the functional model, and have ignored process-oriented models such as data flow and state-transition modelling. The Generic Semantic Model, from which Atzeni & Torlone’s metamodel is derived, was developed for use in a survey of se-

mantic data modelling approaches (Hull and King, 1987) and also does not consider process-oriented models. One solution could be to define new metaconstructs corresponding to various process-oriented constructs.

Every ‘instance’ of the MDM environment has a collection of representations (referred to as ‘models’) defined within it. These representations are combined to form a ‘supermodel’, which is then used to store descriptions within the environment. That is, MDM follows the interchange format interfacing strategy. The number of translations required by such a strategy is normally  $2n$ , where  $n$  is the number of representations. MDM only requires  $n$  translations, however, because the supermodel is built by taking the union of all representations currently in the system. That is, the supermodel is inclusive of all the other representations:  $\forall i, \mathfrak{R}_i \preceq \mathfrak{R}_{supermodel}$ . Any description that is an instance of representation  $\mathfrak{R}_i$  is also an instance of  $\mathfrak{R}_{supermodel}$ , so the only translations required are those from the supermodel to individual representations. A side-effect of this arrangement is that the supermodel must be regenerated every time a new representation is added.

Translations in MDM are automatically derived from a collection of elementary translations that “implement the standard translations between simple combinations of constructs” (Atzeni and Torlone, 1997, p. 528), although it is not clear what they mean by ‘standard’. These elementary translations are assumed to be correct. Translations are defined as a collection of rules, specified using a graph-based formalism that defines mappings between groups of constructs, known as *patterns*. The basic translations are also highly abstract, in keeping with the abstract metamodel, for example, replacing  $n$ -ary aggregations with binary aggregations and replacing complex attributes with simple attributes.

The major feature of the approach taken with MDM is that translations are defined in the context of a formal lattice framework (Atzeni and Torlone, 1995), which means that it is possible to formally verify various properties of translations. A translation is *correct* if it always produces a valid description in the target representation. The lattice framework also provides a measure of the quality of translations, as the lattice imposes a partial order on sets of patterns. Atzeni and Torlone define ‘quality’ as the number of rules used to complete a translation; the smaller this number, the higher the quality of the translation. A *minimal* translation is a correct translation that uses



the least possible number of rules. A minimal translation is also *optimal* if no other minimal translations exist. Atzeni & Torlone do not appear to have explored the issue of rule exclusion, although they do mention the concept of subsumption with respect to patterns (Atzeni and Torlone, 1996a). It may be that the highly abstract nature of their metamodel reduces the overlaps among rules and thus eliminates rule exclusion.

In summary, the approach taken with MDM has focused mainly on the formal specification of representations and translations, and the derivation of various formal properties of those translations. The approach is restricted to semantic data models and does not consider other modelling approaches such as process-oriented models. In contrast, the approach followed in this research has from the outset explicitly included a variety of different modelling approaches. The abstract approach to defining constructs in MDM allows representations and translations to be defined in a more ‘bottom-up’ fashion, as opposed to the ‘top-down’ approach followed in this thesis, but the assumption that constructs drawn from the same metaconstruct have the same semantics may not always hold.

### **9.3.4 The ORECOM approach**

Su et al.’s (1992) ORECOM is an object-oriented model that has been used to build a multiple-representation schema translation environment; the name stands for ‘Object-oriented Rule-based Extensible COre Model’. The main goals in implementing this approach were to support the integration of multiple heterogeneous data sources and/or schemas for use in heterogeneous database systems, and the exploration of schema and data model translation, particularly between object-oriented and some of the richer semantic data models. ORECOM supports at least the following representations: IDEF-1X (National Institute of Standards and Technology, 1993), NIAM (Verheijen and van Bekkum, 1982), EXPRESS (ISO-IEC, 1992b), SDM (Hammer and McLeod, 1981), OMT (Rumbaugh et al., 1991) and OSAM\* (Su et al., 1989). All of these are ‘structural’ representations — there is no mention of process-oriented representations such as data flow or state-event modelling.

ORECOM follows the interchange format interfacing strategy (Su et al., 1992, pp. 5–8) — all descriptions are stored using the ORECOM model. The individual interfacing strategy was not used for several reasons, two of which are particularly relevant here.

First, it is claimed that translations under the individual interfacing strategy are not specified in a uniform manner, due to the translation algorithms being representation-specific. This has been successfully addressed in this thesis by the use of VML-S as a uniform translation specification language. Second, it is claimed that translation algorithms cannot be shared across individual translations, for example, “two mappings between E-R and the relational model . . . need two different algorithms; one for each direction”. Again, this has been addressed in this thesis by the use of VML-S to specify bidirectional translations, and also to some extent by the use of technique-level rules, which allow some rules to be shared across several translations.

A representation is converted into ORECOM by decomposing its constructs and semantic constraints into collections of primitive ORECOM constructs and constraints. ORECOM has three primitive structural constructs: *objects* and *classes*, which are identical to the concepts of object and class in object-oriented programming; and *associations*, which are binary relationships between classes. A class is either a *domain* class, which represents atomic data types like integers and characters, or an *entity* class, which can represent complex data types such as ‘person’ or ‘address’. *N*-ary associations are represented in ORECOM by a collection of binary associations and an associated collection of ORECOM constraints to capture the additional semantic properties of the *n*-ary association.

Semantic constraints of representations (such as cardinality and inheritance) are expressed in ORECOM by a collection of primitive constraints known as *micro-rules*. These are triggered by certain operations on a class, and capture those semantic constraints that are not captured by the structural constructs. Specification of a particular representation’s semantic constraints can require a large number of these micro-rules, however, making comparison of representations difficult (Su et al., 1992, p. 16). The solution used in ORECOM is to group collections of micro-rules into higher-level semantic constraints that occur across many different representations. These collections are known as *macros*, and include the following (Su and Fang, 1993, Section 3.2): membership, participation, cardinality, inheritance, privacy<sup>1</sup> and dependency. Macros can effectively be thought of as representation-independent semantic constraints.

---

<sup>1</sup>This is the concept of visibility used in many object-oriented programming languages (for example, public vs. private methods in C++), and is typically restricted to object-oriented representations.

ORECOM follows a similar approach to extensibility as that taken by Atzeni and Torlone (1997) with MDM — when a new representation introduces constructs and constraints that cannot be decomposed into ORECOM constructs and constraints, the model may be extended by defining new primitive constructs and constraints. As with MDM, it is expected that this would not happen very often.

An environment for performing schema translations has been built around ORECOM. Since all representations are defined using ORECOM primitives, the data model translation system can readily compare the constructs and constraints of the source and target representations. From this comparison is built an equivalence matrix that identifies the closest matches between constructs in each representation. The schema translation system then uses this table to translate schemas from one representation to the other.

In summary, ORECOM follows an approach similar to that taken by Atzeni and Torlone (1993) in that both use an abstract metamodel to define representations, and translations between representations are automatically derived from representation definitions. The major difference is that ORECOM is based on an object-oriented model rather than a mathematical one. ORECOM's micro-rules are, however, defined using a first-order calculus, which provides scope for proving the correctness of translations. The main focus of work with ORECOM has been on performing translations between 'structural' semantic data models and object-oriented models, especially in the context of heterogeneous database systems. In particular, Su and Fang (1993) have identified the following possible applications:

- schema sharing;
- schema translation in the context of a heterogeneous DBMS;
- schema integration;
- schema verification and optimisation;
- semantics modification and extension before conversion, that is, extending the semantics of a description before translating it to another representation (analogous to the concept of pre-enrichment discussed in Section 4.6 on page 91); and

- helping users to learn new modelling representations (all representations are decomposed into a standard form, making it easier to compare representations).

### 9.3.5 Discussion

Of the five ‘conventional’ tools surveyed, only one (EasyER/EasyOBJECT) facilitates the use of multiple representations to any useful extent, and even this support is limited, and possibly unintentional. The three research-based approaches discussed address the use of multiple representations in varying ways. The MViews approach is focused on maintaining consistency among multiple descriptions. The MDM approach is focused on the formal specification of representations and translations, and the derivation of formal properties of translations. The ORECOM approach is focused on the translation of schemas between representations for the purpose of building heterogeneous database systems.

The individual interfacing strategy is used in this research as a basis for defining translations. Those conventional tools that provide translations also follow this approach, although this is more likely a result of their limited translation support than any conscious design decision<sup>2</sup>. The three research approaches all follow the interchange format interfacing strategy. The MDM and ORECOM approaches are particularly interesting in this respect — their use of extremely abstract or primitive constructs to define representations may ameliorate the problem of interchange format complexity (Su et al., 1992, p. 7). This is because neither approach attempts to define the interchange format in terms of existing modelling constructs; rather, modelling constructs are decomposed into primitive forms. All three approaches still suffer from the issue of extending the interchange format to handle new representations, although Atzeni and Torlone (1996a) partially address this by automating the process of generating the new interchange format.

One goal of this research was to enable the highlighting of potential inconsistencies within a viewpoint using translations. Translations may be used to highlight inconsistencies among descriptions by translating descriptions expressed using different representations into the same representation, then comparing them (see Section 4.5 on

---

<sup>2</sup>Many CASE tools are now embracing the Unified Modelling Language (Rational Software Corporation, 1997) as an interchange format for object-oriented design.

page 86). Of the approaches and tools surveyed, only MViews makes any mention of design consistency, but this is oriented toward *maintaining* consistency among descriptions as they evolve in parallel, rather than *highlighting* potential inconsistencies among a collection of descriptions. The other approaches have not explored this issue.

Another goal of this research was to identify ways of improving translation quality. All three research-based approaches have some notion of translation quality. The MDM approach in particular has an inherent notion of translation quality (Atzeni and Torlone, 1996a), but this is defined in terms of the number of rules required to perform a translation, rather than how well a translation maps constructs from one representation to another. Atzeni and Torlone (1996a) do define the notion of a *preferable* translation, but other than this, none of the three research approaches discuss methods of how translation quality might be improved. The author introduced the concept of a heuristic in Chapter 4 as one possible mechanism for improving the quality of translations, which was shown to be effective in Chapter 8. None of the three research approaches use heuristics, and it would probably be difficult to introduce heuristics into either the MDM or ORECOM approaches, as both of these rely on translations being automatically derived from representation definitions. Heuristics by their very nature cannot be derived in this manner. MDM may be more amenable to the inclusion of heuristics, as it builds translations from a collection of predefined ‘basic’ translation rules. Translation definition in MViews-based environments follows an approach similar to that followed in this thesis, so incorporating heuristics into the MViews framework or its derivatives should be possible.

The second approach to improving translation quality discussed in Chapter 4 is the process of enrichment. None of the three research approaches make any explicit mention of enrichment *during* a translation, which can result in a higher-quality translation than would otherwise be possible. One of the proposed applications of ORECOM was to support the pre-enrichment of schemas before translation (Su and Fang, 1993), and it seems likely that pre-enrichment is supported by all three approaches. Post-enrichment involves the manual modification of descriptions after they have been translated, so all three approaches support it by definition. When information cannot be directly translated, MViews environments can provide hints to aid post-enrichment by generating a textual description of what changes are required.

Translation rules in this thesis are expressed at a high level using an abstract notation, while detailed specifications of rules are expressed using an extended variant of Amor's (1997) View Mapping Language (VML), known as VML-S. MViews does not have a high-level abstract notation, and translation rules were originally specified in a somewhat ad hoc manner. There has, however, been recent work on incorporating a new version of VML into the MViews frameworks (Grundy, 1998, personal communication). Both MDM and ORECOM use an algebraic notation for expressing translation rules. MDM's algebraic notation may also be expressed in graphical form. All of these notations are, however, low-level and very detailed. None of the three approaches defines a high-level abstract notation like that described in this thesis.

Of the three research approaches, only MDM has a notion of rule subsumption, which differs from that presented in Chapter 4. None of the three approaches identify the issue of rule exclusion, which can have a significant impact on how rules are evaluated during a translation. Rule exclusion may not be an issue in either MDM or ORECOM, however, as translation rules are automatically generated in both approaches rather than being defined in advance.

In summary, the approach followed in this thesis compares well with existing approaches, and introduces the following novel aspects:

- an abstract notation for expressing translation rules at a high level;
- the issue of rule exclusion and ways of dealing with it, including extensions to VML;
- use of translations to highlight potential inconsistencies between descriptions within a viewpoint; and
- use of heuristics and enrichment to improve the quality of translations.

Table 9.3 summarises the differences in approach between this research and MViews, MDM and ORECOM.

**Table 9.3:** Comparison of research approaches

|  | <b>This research</b>         | <b>MDM</b>                           | <b>MViews</b>      | <b>ORECOM</b>             |
|--|------------------------------|--------------------------------------|--------------------|---------------------------|
| <b>Interfacing strategy<sup>a</sup></b>      | individual                   | interchange format                   | interchange format | interchange format        |
| <b>Underlying framework<sup>b</sup></b>      | viewpoint framework          | graph theoretic framework            | unknown            | object-oriented framework |
| <b>Translation specification<sup>c</sup></b> | abstract notation/VML-S      | abstract notation                    | ad hoc/VML         | n/a <sup>d</sup>          |
| <b>Consistency mechanism<sup>e</sup></b>     | asynchronous                 | unknown                              | synchronous        | asynchronous              |
| <b>Translation propagation<sup>f</sup></b>   | rules & heuristics           | rules <sup>d</sup>                   | update records     | equivalence matrix        |
| <b>Formal properties<sup>g</sup></b>         | relative translation quality | correctness, completeness, 'quality' | n/a <sup>h</sup>   | correctness               |

**Notes on Table 9.3**

<sup>a</sup> Discussed in Section 2.4.3 on page 30.

<sup>b</sup> Discussed in Section 2.3 on page 15 and Chapter 3 on page 39.

<sup>c</sup> Discussed in Section 4.4 on page 78 and Chapter 7.

<sup>d</sup> Translation rules are automatically derived.

<sup>e</sup> Discussed in Section 3.6 on page 58.

<sup>f</sup> Discussed in Section 4.2.

<sup>g</sup> Discussed in Section 8.4 on page 232.

<sup>h</sup> Work on MViews has focused on consistency maintenance methods rather than formal proofs.

## 9.4 Practicability of the approach

One of the goals of this research is to show that the translation-based approach to facilitating the use of multiple representations is practicable. A general discussion of the tractability of the approach is presented in Section 9.4.1, with an emphasis on the time complexity of translations with respect to the number of elements in the source description. The factors affecting the time complexity of translations are identified, and four predictions are made with respect to the time complexity of translations. These predictions are then empirically tested by performing various timing tests with Swift; the design, implementation and results of these tests are discussed in Section 9.4.2.

### 9.4.1 Tractability of translations

There are two major concerns with the tractability of the approach presented in this thesis:

1. the  $O(n^2)$  number of interfaces implied by the individual interfacing strategy; and
2. how well translations scale to more complex source descriptions.

Much of this thesis has focused on the translation process and in particular the *quality* of translations. The individual interfacing strategy was adopted by the author because it allows for the best quality translations compared to other interfacing strategies (Pascoe and Penny, 1990). The number of interfaces required by the interfacing strategy was a secondary concern. Unfortunately, the individual interfacing strategy requires  $n(n-1)/2$  interfaces in order to provide a complete set of translations between  $n$  representations, which can rapidly become problematic even for relatively small  $n$ . For example, the four representations discussed in this thesis would require a total of only six interfaces, but adding only five more representations would increase the number of interfaces required to thirty-six. Even if many of the interfaces were trivial (that is, changing only the appearance of the notation), the number of interfaces required could still rapidly become unwieldy.

One solution to this issue would be to accept a decrease in potential translation quality and adopt the interchange format interfacing strategy, which would reduce the total number of interfaces required to  $2n$ . Alternatively, it may be possible to replace certain interfaces with composite translations without affecting overall translation quality; this has been left as an area for future research and is discussed further in Section 10.8 on page 295.

Of more concern here is how well translations scale to more complex descriptions, particularly with respect with the time  $t$  taken to perform translations. This time is affected to varying degrees by the following factors:

1. the number of rules available for the translation;
2. the complexity of the translation itself, characterised by the internal complexity of its rules and the amount of rule subsumption;



3. the number of elements in the source description; and
4. the complexity of the source description, characterised by the level of ‘interconnectedness’ of its elements.

When performing a translation in a particular direction, some number of rules  $r_a$  will be available for use in that direction. Note that  $r_a$  may be less than the total number of rules  $r$ , as some rules may be unidirectional in the opposite direction and hence inapplicable. The value of  $r_a$  will typically not change very often, but the number of rules actually *used* during a translation may be less than  $r_a$ , as not all available rules may be applicable to the elements of the source description. That is, the number of rules actually used during a translation is highly variable and determined by the source description, while the number of available rules is effectively constant. Consequently, the size and complexity of the source description are likely to have a greater impact than the number of rules in determining the translation time  $t$ .

The complexity of a translation is affected by the internal complexity of its rules and the amount of rule subsumption. The internal complexity of an individual rule is determined by the number of source and target constructs in the rule definition. The simplest possible rule maps a single source construct to a single target construct; adding more source and target constructs increases the internal complexity of the rule. Rule subsumption (see Section 4.7.1 on page 98) also affects the complexity of a translation. If one rule subsumes another, then potentially both rules must be checked to see if they are applicable, which will take more time.

The simplest possible translation therefore comprises rules that map only single source constructs to single target constructs, and has no subsumed rules. Consider the translation  $\mathfrak{R}_e(E-R, ERD_{Martin}) \rightarrow \mathfrak{R}_d(DataFlow, DFD_{G\&S})$ , which was summarised in Table 5.5 on page 141: six out of seven available rules are as simple as possible, and there are only two rule subsumptions. Contrast this with  $\mathfrak{R}_f(FuncDep, FDD_{Smith}) \rightarrow \mathfrak{R}_e(E-R, ERD_{Martin})$ , which was summarised in Table 5.4 on page 136: three out of fifteen available rules are as simple as possible, and there are twenty-two rule subsumptions. The latter translation is more complex, and should therefore take longer to perform than the former translation. The complexity of a translation is effectively a multiplier on the number of rules  $r_a$  and will typically not change very often, so the

complexity of a translation can be considered constant with respect to its effect on  $t$ .

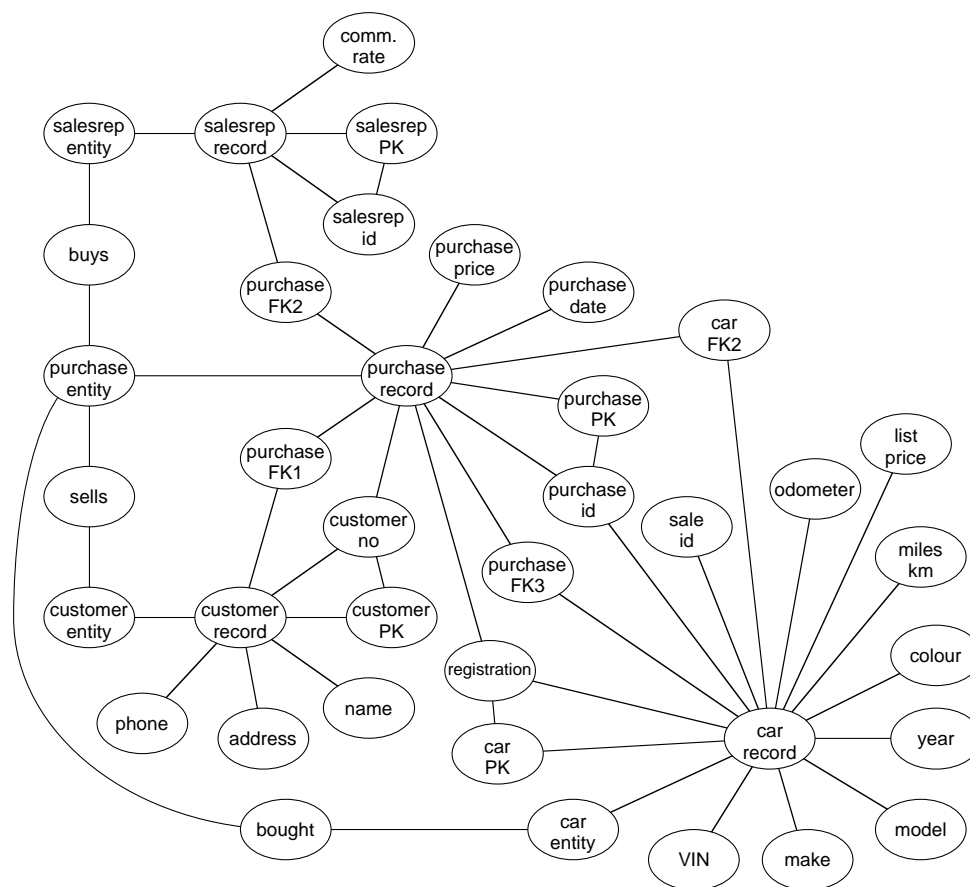
The number of source elements  $n$  will vary widely for each source description. Some descriptions may have only a few elements, while others may have hundreds or thousands of elements. Not all of these elements may be translated, however, as their associated constructs may have no analogue in the target representation. The wide variability of  $n$  and its impact on the number of rules used suggests that  $n$  will have a more significant impact on translation times than either the number of rules used or the complexity of the translation.

The relationship between  $n$  and  $t$  is affected by the level of ‘interconnectedness’ of elements in the source description. Elements become interconnected by being associated with each other in a description (for example, an entity is associated with its attributes, and vice versa). For identical values of  $n$ , a description that has a higher level of element interconnectedness should take longer to translate than a description with a lower level of element interconnectedness.

It is possible to represent the interconnectedness of elements in a description using a non-directed graph  $I = (\mathcal{V}, \mathcal{E})$  where  $\mathcal{V}$  is a set of vertices representing the elements of the description, and  $\mathcal{E}$  is a set of edges representing the associations between elements. An edge  $e_{ij} = (v_i, v_j)$  represents an association between the elements represented by  $v_i$  and  $v_j$ . An example of such an ‘interconnectedness graph’ is shown in Figure 9.7 for the subset of the used cars ERD description shown in Figure 9.8(a) on page 272.

The number of associations in which an element participates is represented in the interconnectedness graph  $I$  by the degree of the vertex corresponding to the element. By averaging the degrees of all vertices in  $I$ , it is possible to gain an indication of the overall level of element interconnectedness  $i$  in a description. The value of  $i$  represents the average number of times an individual element is associated with other elements in the description. It could be argued that the median degree provides a more ‘stable’ indication of interconnectedness, as it is not skewed by a small number of highly-interconnected elements (for example, thirty elements of degree one and one element of degree thirty produces an average of 1.94 but a median of one). The median degree, however, may not differentiate between descriptions with different levels of interconnectedness. For example, the FDD shown in Figure 9.8(d) on page 272 must have a higher level of interconnectedness than the FDD shown in Figure 9.8(c), because it

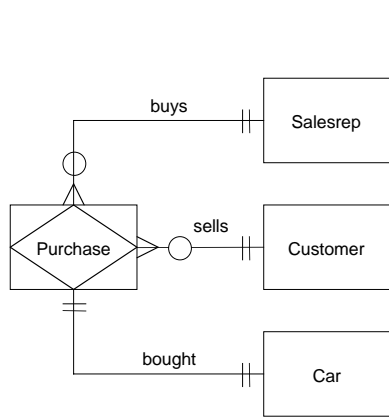
contains more nested elements. Despite this, both FDDs have a median degree of two, whereas the average degrees are 2.17 and 1.8 respectively. In effect, the median degree ‘filters out’ highly-interconnected elements.



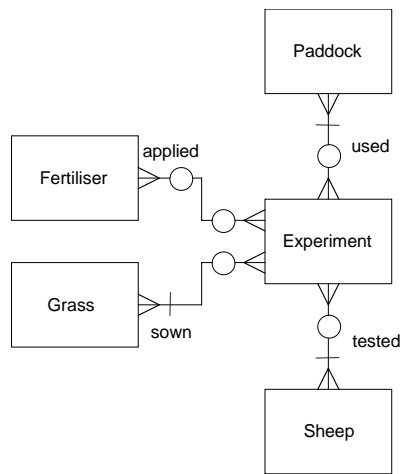
**Figure 9.7:** The interconnectedness of elements in a description

One particular manifestation of element interconnectedness is that it may be possible for elements to be associated in such a way that a single element is processed by several rules, including multiple instances of the *same* rule. For example, the Experiment entity in the unnormalised agricultural ERD (see Figure 9.8(b) on the next page) participates in four many-to-many relationships with four different entities. If this description were translated by  $\mathfrak{R}_e(E-R, ERD_{Martin}) \rightarrow \mathfrak{R}_r(Relational, SQL/92)$ , the Experiment entity would be processed four times by rule S8, each time in combination with a different entity and relationship.

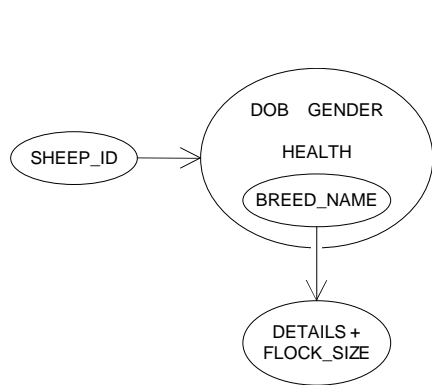
For the simplest possible translation,  $t$  should be proportional to the number of source elements, that is,  $O(n)$ .  $\mathfrak{R}_e \rightarrow \mathfrak{R}_d$  is an example of such a translation. More complex translations raise the possibility of elements being processed several times in



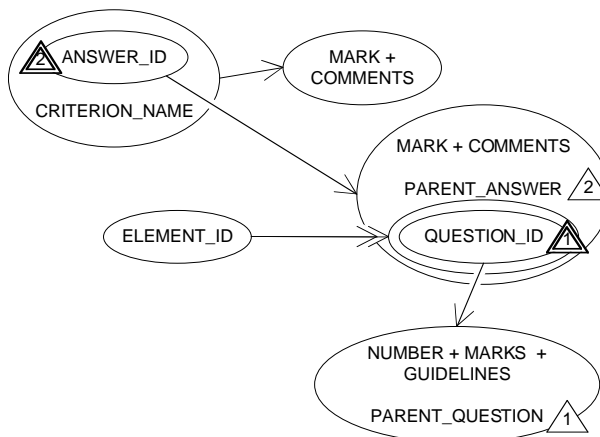
(a) 'Used cars' base ERD



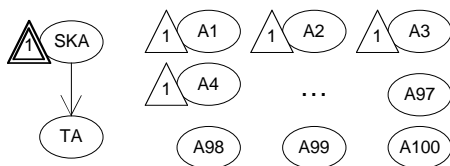
(b) 'Agricultural' base ERD



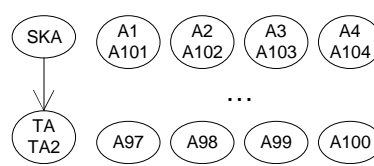
(c) 'Agricultural' base FDD



(d) 'Marks' base FDD



(e) 'More interconnected' FDD



(f) 'Less interconnected' FDD

**Figure 9.8:** Descriptions used for translation time testing

different combinations with other elements. In the worst case, every element would be combined with every other element, yielding a time complexity of  $O(n^2)$ . It is unlikely, however, that every element will be combined with every other, so the translation time  $t$  will typically be proportional to some fraction of  $n^2$ .  $\mathfrak{R}_f \rightarrow \mathfrak{R}_e$  is an example of such a translation. It is unclear at present at which point the complexity of a translation would be sufficient to change its time complexity from  $O(n)$  to  $O(n^2)$ .

### 9.4.2 Complexity testing

Four predictions can be identified from the discussion in the previous section:

1. The translation  $\mathfrak{R}_f(\text{FuncDep}, \text{FDD}_{\text{Smith}}) \rightarrow \mathfrak{R}_e(\text{E-R}, \text{ERD}_{\text{Martin}})$  is more complex than  $\mathfrak{R}_e(\text{E-R}, \text{ERD}_{\text{Martin}}) \rightarrow \mathfrak{R}_d(\text{DataFlow}, \text{DFD}_{\text{G\&S}})$ , and should therefore take longer to perform.
2. As the level of interconnectedness ( $i$ ) of elements in the source description increases, so too should the translation time  $t$ .
3. The time  $t$  taken to perform a simple translation such as  $\mathfrak{R}_e \rightarrow \mathfrak{R}_d$  should be  $O(n)$ , where  $n$  is the number of source elements.
4. The time  $t$  taken to perform a more complex translation such as  $\mathfrak{R}_f \rightarrow \mathfrak{R}_e$  should be  $O(n^2)$ .

To test these predictions, a series of experiments were carried out using Swift. Six distinct viewpoints were defined, which are summarised in Table 9.4 on the following page. The first four viewpoints were designed to test the effect of increasing  $n$  on the translation time  $t$ , and were built by taking small subsets of the sample viewpoints described in Appendix C. For each viewpoint, a base description was defined; these descriptions are shown in Figures 9.8(a)–9.8(d). The elements of each base description were then replicated to simulate progressively larger descriptions. This allowed  $n$  to increase without affecting the level of interconnectedness  $i$ .

The two remaining viewpoints were designed to directly test the effect of element interconnectedness on translation time, and were artificially generated. Both viewpoints comprised a collection of functional dependency descriptions whose level of

**Table 9.4:** Summary of test viewpoints

| Viewpoint           | Translation                                 | Range of $n$ | Interconnectedness ( $i$ ) |
|---------------------|---|--------------|----------------------------|
| 'Used cars' ERDs    | $\mathfrak{R}_e \rightarrow \mathfrak{R}_d$ | 38–3040      | 2.53                       |
| 'Agricultural' ERDs | $\mathfrak{R}_e \rightarrow \mathfrak{R}_d$ | 47–3760      | 2.89 <sup>a</sup>          |
| 'Agricultural' FDDs | $\mathfrak{R}_f \rightarrow \mathfrak{R}_e$ | 20–1600      | 1.80                       |
| 'Marks' FDDs        | $\mathfrak{R}_f \rightarrow \mathfrak{R}_e$ | 36–1440      | 2.17                       |
| More complex FDD    | $\mathfrak{R}_f \rightarrow \mathfrak{R}_e$ | 307–408      | 1.34–2.00                  |
| Less complex FDD    | $\mathfrak{R}_f \rightarrow \mathfrak{R}_e$ | 307–408      | 1.34–1.51                  |

**Notes on Table 9.4:**

- <sup>a</sup> Due to incomplete implementation of the  $\mathfrak{R}_e \rightarrow \mathfrak{R}_d$  translation, not all of this description is translated, resulting in an effective interconnectedness of 2.29. This is discussed further on page 275.

interconnectedness  $i$  gradually increased, but at differing rates. Descriptions for the fifth viewpoint were generated as follows:

1. Create a single-key bubble and a target bubble, each containing a single attribute (SKA and TA respectively).
2. Create a functional dependency linking the single-key and target bubbles.
3. Create 100 isolated bubbles, each containing a single attribute (A1–A100).
4. Attach a domain flag to the single-key attribute SKA, and also to some number of the isolated bubbles' attributes.

Part of one of these descriptions is shown in Figure 9.8(e). The intent was to produce a single highly-interconnected element (the domain flag). While  $n$  could not be kept constant, the impact of  $n$  was reduced by minimising the number of elements added to the description as  $i$  increased.

Descriptions for the sixth viewpoint were generated as follows:

1. Create a single-key bubble containing a single attribute (SKA), and a target bubble containing two attributes (TA and TA2).
2. Create a functional dependency linking the single-key and target bubbles.
3. Create 100 isolated bubbles, each containing a single attribute (A1–A100).
4. Add a second attribute ( $A_r$ ) to some number  $r$  of the isolated bubbles.

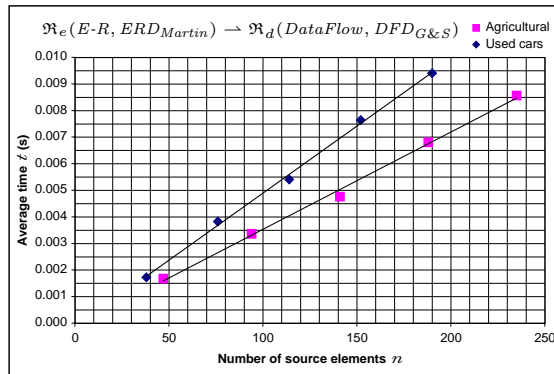
This had the effect of producing descriptions with the same numbers of elements as those in the fifth viewpoint, but with smaller  $i$  for identical  $n$ . Part of one of these descriptions is shown in Figure 9.8(f).

All tests were run under the default Java virtual machine settings, which provide up to 16 megabytes of heap space. The  $\mathfrak{R}_e(E-R, ERD_{Martin})$  descriptions were translated into  $\mathfrak{R}_d(DataFlow, DFD_{G\&S})$  and the  $\mathfrak{R}_f(FuncDep, FDD_{Smith})$  descriptions were translated into  $\mathfrak{R}_e$ . For each description, the translation was performed fifty times and the execution times averaged. Using Microsoft Excel, these values were then plotted against  $n$ , and Excel's 'trendline' feature was used to fit a curve to the observed data. The results of the tests are shown in Figure 9.9 on the next page.

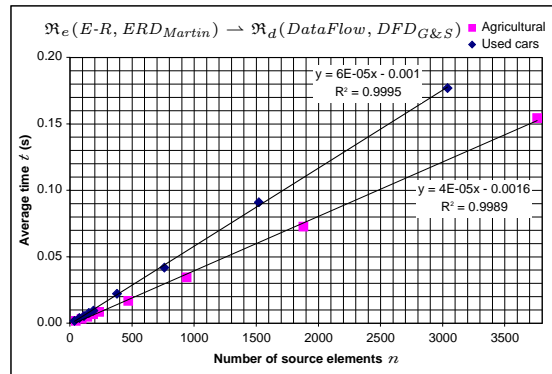
It can be seen from Figures 9.9(a)–9.9(d) that the  $\mathfrak{R}_f \rightarrow \mathfrak{R}_e$  translation takes considerably longer than the  $\mathfrak{R}_e \rightarrow \mathfrak{R}_d$  translation, which confirms prediction 1. Even for small values of  $n$ , the time taken to perform  $\mathfrak{R}_f \rightarrow \mathfrak{R}_e$  is at least an order of magnitude greater than that for  $\mathfrak{R}_e \rightarrow \mathfrak{R}_d$ .

Next, it can be seen from Figures 9.9(a) and 9.9(b) that the time taken to perform the translation  $\mathfrak{R}_e \rightarrow \mathfrak{R}_d$  appears linear with respect to  $n$ , which is strong evidence in favour of prediction 3. This could have been tested further using even larger descriptions than those already used, but this proved impractical due to the very long time required to load large descriptions into Swift (on the order of about 30 minutes for the largest description tested). The loading delay appears to be caused by the Java-Postgres95 interface, and could perhaps be improved by reimplementing the repository using a pure object DBMS.

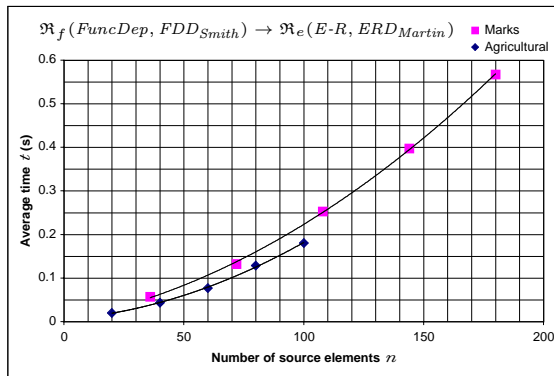
Unexpectedly, the translations for the used cars ERDs took longer than those for the agricultural ERDs, despite the agricultural ERDs having higher  $i$  (2.89) than the used cars ERDs (2.53). The agricultural ERDs have higher  $i$  because they are unnormalised, whereas the used cars ERDs are normalised. In particular, the Experiment entity has an unnormalised internal structure. On further investigation, the author discovered that the  $\mathfrak{R}_e \rightarrow \mathfrak{R}_d$  translation had not been fully implemented and did not deal with unnormalised structures. As far as the translation was concerned, the unnormalised structures within the Experiment entity did not exist, producing an effective  $i$  of 2.29 instead of 2.89. This reduced value of  $i$  is less than that for the used cars ERDs, and hence consistent with prediction 2.



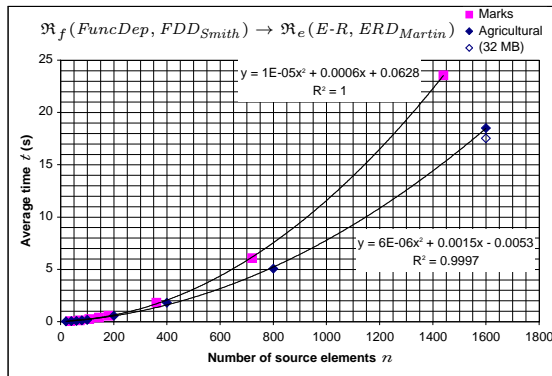
(a)  $n$  vs.  $t$  (ERD descriptions,  $n \leq 5$ )



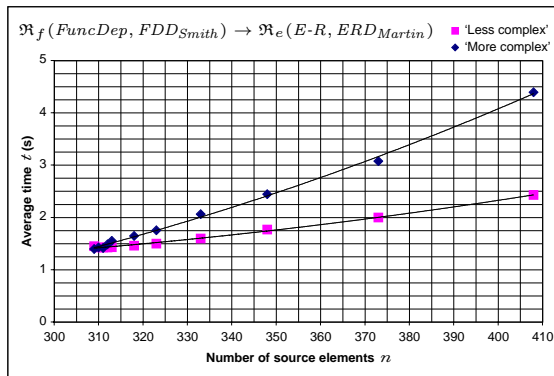
(b)  $n$  vs.  $t$  (ERD descriptions)



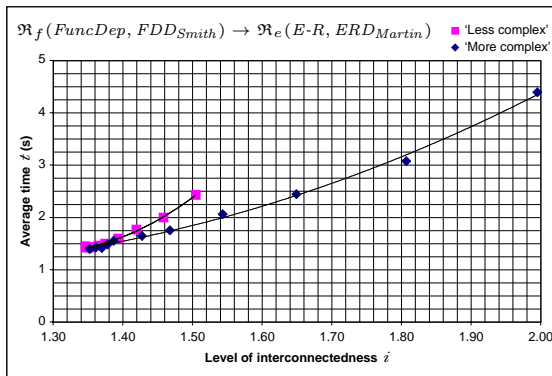
(c)  $n$  vs.  $t$  (FDD descriptions,  $n \leq 5$ )



(d)  $n$  vs.  $t$  (FDD descriptions)



(e)  $n$  vs.  $t$  (interconnectedness)



(f)  $i$  vs.  $t$

**Figure 9.9:** Results of Swift translation time testing



The results for  $\mathfrak{R}_f \rightarrow \mathfrak{R}_e$  are shown in Figure 9.9(c) and 9.9(d), where the time taken to perform the translations is clearly not linear in  $n$ . The best-fit curve for both viewpoints is an order two polynomial, that is, the translation time is proportional to  $n^2$ , which supports prediction 4. It should be noted however, that the  $x^2$  coefficients of the polynomials are both very small (on the order of  $10^{-5}$  for the marks FDDs and  $10^{-6}$  for the agricultural FDDs). That is, while  $t$  is proportional to  $n^2$ , this does not begin to have a significant effect until  $n$  is very large, on the order of about 1,000 elements (the largest ‘normal’ description used in this thesis has 109 elements). The predicted time for a Marks FDD containing 1,000 elements is approximately 11.7 seconds, which is still reasonable for practical use. Performance could probably be further improved by compiling the translations to native machine code rather than executing them through the Java virtual machine.

There was a possibility that the  $n^2$  curve was caused by Java’s garbage collection mechanism, although this seemed unlikely given the results of the ERD tests. To eliminate garbage collection as a factor, the tests were re-run twice on the largest agricultural FDD, first with thirty-two megabytes of memory allocated to Java, and then with 128 megabytes. Allocating thirty-two megabytes reduced the translation time by approximately one second (indicated by the hollow diamond in Figure 9.9(d)). Allocating 128 megabytes reduced the time by a further quarter of a second. Neither reduction was sufficient to produce a linear relationship between  $n$  and  $t$ .

The curve for the marks FDDs is noticeably steeper than that for the agricultural FDDs. As with the first pair of translations, this is because the marks FDDs have a higher degree of interconnectedness (2.17) compared with the agricultural FDDs (1.8). This is further evidence in support of prediction 2.

Prediction 2 is further borne out by the results shown in Figure 9.9(e), where the more interconnected description takes longer to translate than the less interconnected description for identical values of  $n$ . Interestingly, when the interconnectedness values of these descriptions are plotted against  $t$ , as shown in Figure 9.9(f), the two curves are not identical, as might be expected. Rather, the curve for the ‘less interconnected’ description is steeper. The author is at present unable to explain this phenomenon; it may imply some additional factor that has not yet been identified. Further investigation and experimentation with a wider range of descriptions is therefore required.

## 9.5 Summary

In this chapter, the approach taken in this thesis was evaluated against the goals of the thesis. In Section 9.2, a case study was presented of building a new viewpoint using Swift, showing how the proposed approach achieves various thesis goals. In Section 9.3, the approach taken here was compared with that taken by other tools and environments to evaluate the novelty of the author's approach, and several novel aspects of the author's approach were identified. In Section 9.4 the tractability of the translation-based approach to facilitating the use of multiple representations was discussed, and the time complexity of translations in Swift was examined.

The discussion and evaluation presented in this and earlier chapters will be used to draw conclusions in Chapter 11. Before drawing conclusions, however, there are many issues raised in this thesis which have been left as areas for further work. These unresolved issues are discussed in the next chapter, and possible directions for future research are identified.

# Chapter 10

## Further research

### 10.1 Introduction

In this thesis have been examined several major issues associated with using multiple representations to describe a viewpoint, including the issues of improving translation quality and of using translations as a means of highlighting potential design inconsistencies. There remain, however, many issues that were either peripheral to the main thrust of the research, or which have not be explored for other reasons such as time constraints. These unresolved issues are outlined in this chapter, and possible directions for future research are identified.

The issues discussed in this chapter have been organised into several categories. Representations are a key part of the framework underlying this research, and a deliberate effort has been made in this research to include a reasonably diverse collection of representations. Despite this, the representations discussed in this thesis form but a tiny fraction of the total number of representations in current use, so it would be useful to extend the approach used here to include more representations, as discussed in Section 10.2. The issue of how best to define representations is also discussed.

Improving the quality of translations is an important aspect of this research. The relative quality measurement defined in Chapter 8 provides a useful means of comparing the relative quality of translations, but it could be improved in several ways, which are discussed in Section 10.3.

Swift, the prototype modelling environment developed as part of this research, was developed as a means of testing various aspects of the translation-based approach. Consequently, Swift is well-developed in some areas, but less well-developed in others. In addition, some parts of Swift, such as the rule evaluation strategy and the

underlying repository can potentially be replaced with other approaches. Some alternatives are discussed in Section 10.4.

A typical function of CASE tools is to generate a database schema, and it is expected that this will be no different for an environment that facilitates the use of multiple representations. The major difference is that schema generation in current CASE tools typically involves only a single source description, whereas in an environment that facilitates the use of multiple representations, a schema could potentially be generated from multiple source descriptions. This process is discussed in Section 10.5.

The synchronous and asynchronous approaches to maintaining consistency among descriptions were identified in Chapter 3. This area has already been explored by other researchers (Grundy and Hosking, 1994; Grundy et al., 1996; Grundy and Hosking, 1996; Hosking and Grundy, 1995), so it was not examined in any great depth here. Nevertheless, there do remain some outstanding issues in this area, which are discussed in Section 10.6.

VML-S was introduced in Chapter 7 as an extension of Amor's (1997) View Mapping Language to deal with specifying mappings between constructs of representations. The definition of VML-S in this thesis works well for this purpose, as demonstrated in Chapter 7 and Appendix F, but it could be improved further, as discussed in Section 10.7. In particular, the inheritance mechanism derived from VML could be improved upon, but this could require a major redesign of the language.

Other miscellaneous issues are discussed in Section 10.8.

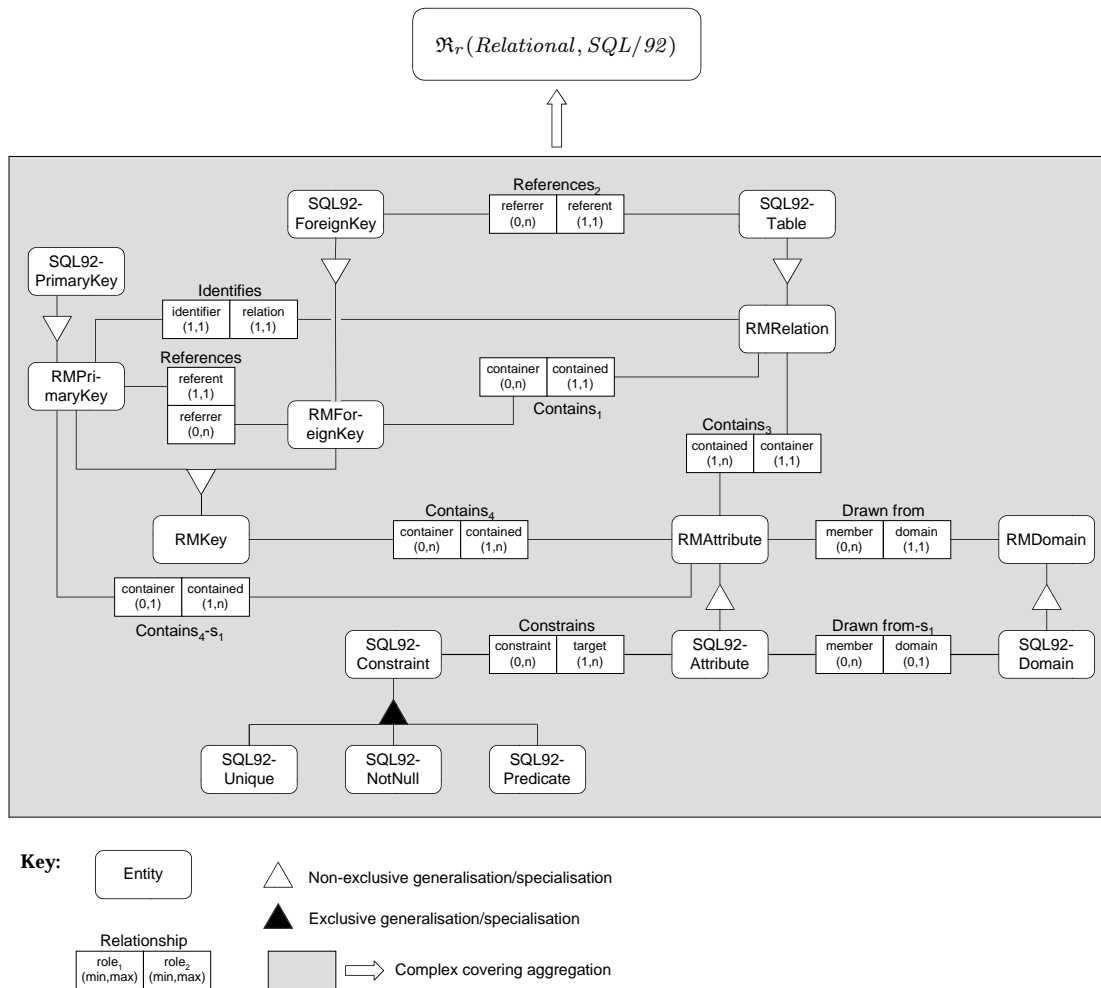
## 10.2 Representation issues

There are some issues with respect to representations that have not been fully addressed in this thesis. The most obvious issue is the number of representations used, as only four representations have been detailed in this thesis:  $\mathfrak{R}_e(E-R, ERD_{Martin})$ ,  $\mathfrak{R}_f(FuncDep, FDD_{Smith})$ ,  $\mathfrak{R}_r(Relational, SQL/92)$  and  $\mathfrak{R}_d(DataFlow, DFD_{G\&S})$ . An obvious first step would be to expand the number of representations considered. This would provide the opportunity to define a greater range of translations and further test the efficacy of the relative quality measure developed in Chapter 8. Other possible representations include: more sophisticated E-R representations such as ERC+

(Spaccapietra and Parent, 1992); variations on the relational technique; other semantic models such as SDM (Hammer and McLeod, 1981) and IFO (Abiteboul and Hull, 1987); functional data models (Shipman, 1981); state-event representations such as state-transition diagrams; more sophisticated data flow representations such as SOFL (Liu et al., 1998); and object-oriented representations such as the ODMG model (Cattell et al., 1997) and the Unified Modelling Language (Rational Software Corporation, 1997; Muller, 1997). It would also be useful to define a representation for schema intension graphs.

All of the representations mentioned above are either formal or semi-formal (see Section 2.3.2 on page 19). Informal representations were deliberately excluded from consideration in this thesis because of their unstructured and often ill-defined nature, which makes them inherently more difficult to translate in an automated manner than the structured and well-defined formal and semi-formal representations. Darke and Shanks (1997a; 1997b) have recently begun to explore the issues associated with using informal representations in user viewpoints, and it would be interesting to apply their results to the approach used in this thesis. At the opposite end of the spectrum, it would also be useful to consider the addition of formal specification languages such as Z (Brien and Nicholls, 1992). Grundy and Hosking (1995) have already explored the issues associated with maintaining consistency between Object-Z specifications (Duke et al., 1991) and Snart programs.

Representations are defined in this thesis using a variant of Martin's (1990) ERD notation as implemented by the EasyCASE tool (Evergreen Software Tools, 1995b). Venable's (1993) meta-modelling language CoCoA is a potentially more useful tool for this purpose, as it was designed with this type of modelling in mind. CoCoA is a variant form of E-R representation that introduces the concept of a *complex covering aggregation*, which allows designers to model the aggregation of entities into composite entities (Venable, 1993; Venable and Grundy, 1995). Unfortunately, CoCoA was discovered too late to be of use in this research, but it will be examined as a potential replacement for the approach currently used. In Figure 10.1 on the next page is shown how the representation  $\mathfrak{R}_r(\textit{Relational}, \textit{SQL}/92)$  could be defined using CoCoA (attributes are not included for clarity); compare this with the definition from this thesis shown in Figure 10.2 on page 283.



**Figure 10.1:**  $\mathfrak{R}_r(\text{Relational}, \text{SQL}/92)$  definition using CoCoA

Finally, a more coherent approach needs to be developed to naming techniques and schemes for use in the abstract notation. The names of techniques and schemes mentioned in this thesis were chosen in a somewhat arbitrary manner, with the only requirement being that each technique and scheme have a unique name. A broader survey of the techniques and schemes available should be undertaken to provide a framework for this naming system. A similar approach should also be applied to the names of constructs within representations.

### 10.3 Quality improvement

The method developed in Chapter 8 for measuring the relative quality of translations provides a useful indication of the relative quality of two or more translations, but it

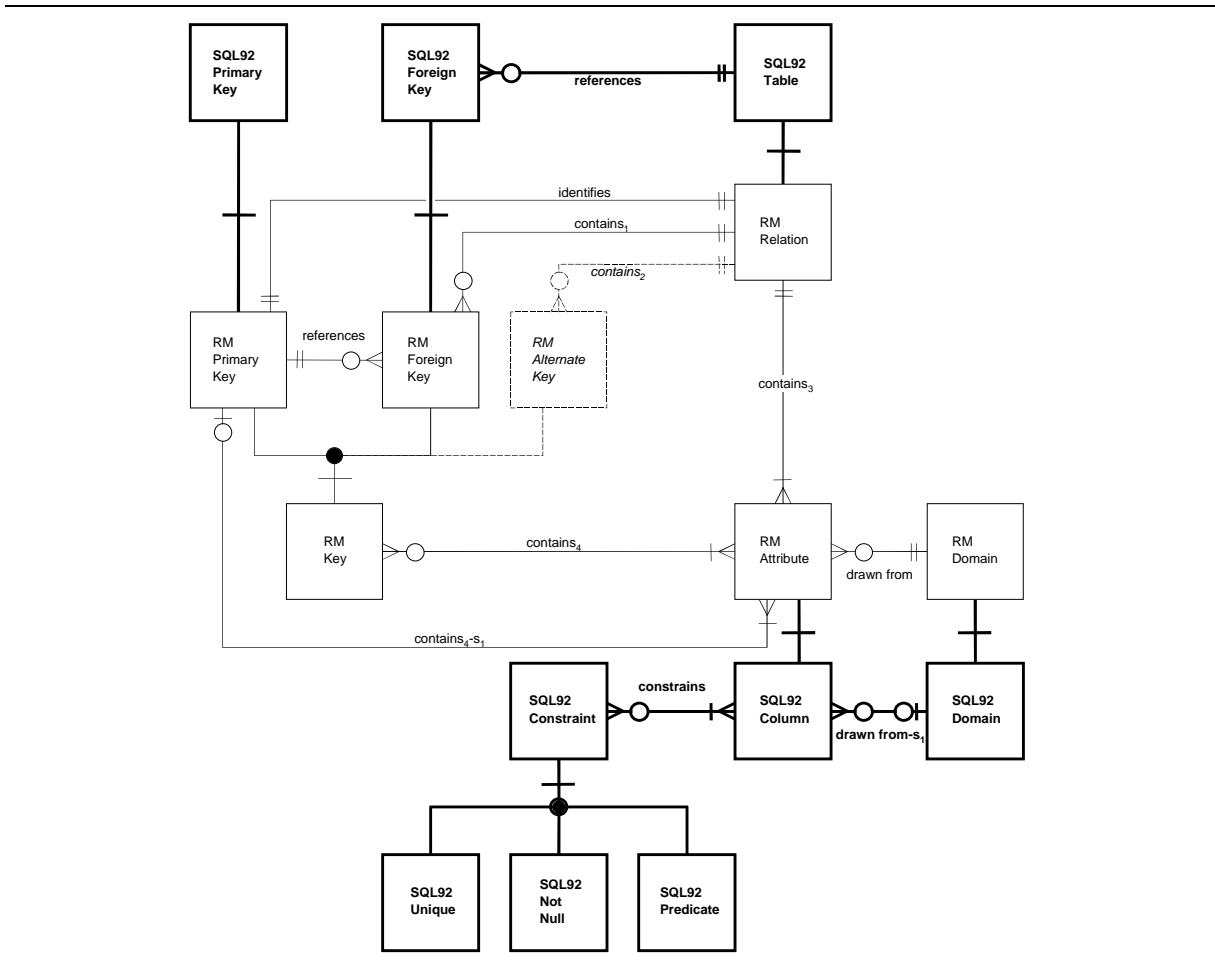


Figure 10.2:  $\mathfrak{R}_r(\text{Relational}, \text{SQL}/92)$  as defined in this thesis

does suffer from some problems. The original form of the measurement, based purely on isomorphism of schema intension graphs (SIGs), was found to generate unexpected results. The translation  $\mathfrak{R}_f(\text{FuncDep}, \text{FDD}_{\text{Smith}}) \rightarrow \mathfrak{R}_e(\text{E-R}, \text{ERD}_{\text{Martin}}) / \mathfrak{R}_f \leftarrow \mathfrak{R}_e$  was found to be of similar relative quality to the translation  $\mathfrak{R}_e(\text{E-R}, \text{ERD}_{\text{Martin}}) \equiv \mathfrak{R}_d(\text{DataFlow}, \text{DFD}_{\text{G\&S}})$ , contradicting experimental evidence that implied the former translation was of much higher quality.

The most likely reason for this is that the SIG isomorphism approach only measures the *explicit* mappings between *single* constructs — often there may also be *implicit* mappings between *groups* of constructs (the ‘collective constructs’ referred to in Section 8.3 on page 223). This is particularly apparent in the  $\mathfrak{R}_f \rightarrow \mathfrak{R}_e / \mathfrak{R}_f \leftarrow \mathfrak{R}_e$  translation, as there are few direct mappings between individual constructs of either representation. Indeed, the only obvious direct mapping is between the FDATTRIBUTE construct and the ERATTRIBUTE construct. If constructs of  $\mathfrak{R}_f$  are grouped in various ways, how-

ever, it becomes possible to define quite a comprehensive translation between the two representations (see Appendix E).

The solution followed in this thesis was to include the rules and heuristics in the measurement process. The SIG isomorphism method can be used to determine the category of expressive overlap between two representations, and also to determine the direct mappings between individual constructs. Mappings between groups of constructs are then determined by tagging construct nodes in a SIG according to how they are used in rules. An alternative approach might be to use the constructed node concept of schema intension graphs to model collective constructs. The major issue with doing this, however, is that modifying a SIG to include collective construct information may invalidate any measurements based on analysing the SIG.

This issue arises because the collective constructs of a representation  $\mathfrak{R}_p$  are defined in terms of the other representation ( $\mathfrak{R}_q$ ) that is being translated to or from. This means that the collective constructs of  $\mathfrak{R}_p$  with respect to  $\mathfrak{R}_q$  will differ from those with respect to some other representation  $\mathfrak{R}_s$ ; indeed, there may be no collective constructs with respect to  $\mathfrak{R}_s$ . It is therefore not sensible to store collective construct information with a representation definition, as it is only meaningful with respect to another representation. Since this information cannot be stored in the representation definition, modifying the representation's SIG to include collective construct information may mean that the SIG no longer models the same representation.

One possible solution might be to define a kind of 'multi-layer' SIG, in which the collective construct information is defined as a layer on top of the basic representation SIG. The unfortunate side-effect of this, however, is that some representations will be treated differently from others for measurement purposes, depending on whether or not they include collective constructs.

One limitation of the relative quality measure defined here is that it does not consider the completeness of the individual rules of a translation. Suppose that two translations are measured using the approach described in Chapter 8, and are found to have identical relative qualities. Now suppose that one translation comprises only partial rules, whereas the other comprises only complete rules. It seems obvious that the latter translation should be of higher relative quality than the former, yet this is not reflected by the relative quality measurement. This situation almost occurs with the  $\mathfrak{R}_f \rightarrow \mathfrak{R}_e$



and  $\mathfrak{R}_e \rightarrow \mathfrak{R}_r$  translations: the relative quality of the former is slightly higher than that of the latter, and the rules for the former are all complete, while some of the rules for the latter are partial.

One approach to resolving this issue could be to define some sort of weighting factor based on the proportions of complete and partial rules in a translation, which could then be applied to the relative quality measurements. One very simple formula for such a weighting factor might be:

$$\frac{\# \text{ complete rules}}{\# \text{ rules}} + \frac{\# \text{ partial rules}}{2 \times \# \text{ rules}}$$

Unfortunately, taking the completeness of rules into account again raises the issue of being able to define a representation in different ways. This is not an issue with the existing measurement methods because they consider a construct to be ‘atomic’ (see Section 8.3), and this issue arises only at the construct property level. The completeness of a rule is partly determined by how well it maps the properties of the constructs involved, so introducing rule completeness as a measurement criterion effectively means that constructs are no longer considered ‘atomic’. Consequently, different definitions for the same representation could give differing results.

Another issue with the relative quality measurement method is that although the use of heuristics will improve the quality of *translations*, it may also degrade the internal consistency of the *viewpoint*, as heuristics can sometimes generate semantically inconsistent results (although this can be ameliorated by including heuristics in the enrichment process, as discussed in Section 4.6 on page 91). This potential degradation falls outside the definition of translation quality (see Section 2.4.1 on page 28), so it cannot be measured by the SIG method. In addition, the SIG method deals only with the mappings between constructs of representations and is thus totally independent of the source description. Conversely, the potential viewpoint consistency degradation caused by heuristics is dependent on context, and is thus highly dependent on the source description. It therefore seems unlikely that the SIG method could be extended to measure viewpoint consistency degradation.

A final issue is that the relative quality measurement cannot measure any translation quality improvements from by the enrichment process described in Section 4.6 on page 91. This is because the effects of enrichment occur at the description level, not the

representation level. Given that the effects of enrichment will likely vary on a case-by-case basis, it may be difficult to define a general measure to determine the impact of enrichment on the quality of a translation. If such a measure were defined, it would have to deal with translations where enrichment is *required* in order to successfully complete the translation.

## 10.4 Implementation issues

The Swift prototype environment was intended mainly to demonstrate the utility of the translation process adopted in this thesis and is therefore incomplete in many areas. It is expected, however, that the Swift architecture will be used as the basis of a more comprehensive tool. The first likely objective will be to reimplement Swift using the JViews framework (Grundy et al., 1997b), which will provide a considerable amount of functionality related to performing translations and maintaining consistency between descriptions without having to write new code. Extensions and enhancements to the Swift environment can then be built on this base.

Swift's repository is currently stored in a PostgreSQL database, but several limitations in the way PostgreSQL implements its object features has meant that the repository is not as effective as it could be. Swift has been totally abstracted from the underlying repository database, which makes it possible to quickly port Swift to other repository DBMSs. It is expected that a pure object DBMS will provide a more effective repository; this assertion will be tested using ObjectStore. Oracle 8, which provides object-relational features, is also a possible alternative.

Storing the repository in a pure object DBMS may make it possible to use representation definitions to automatically extend the repository. If Swift were ported to a pure object DBMS in its present form, appropriate construct classes would need to be manually defined in the repository every time a new representation was added. Representations are defined using what is effectively a special form of description, so it should in theory be possible to parse this description and automatically generate appropriate repository classes for storing elements. This could even be achieved by a translation from the representation used for defining other representations to the Object Definition Language (ODL). It would be necessary, however, to extend representation definitions

with information on how constructs are drawn; such information is not included in representation definitions at present.

Another important feature of Swift that was not implemented was the ability to store in the repository the Java classes that implement the behaviour of representations, constructs and translations. This could not be implemented in the Swift because full support for PostgreSQL's large object features was not available in the JavaPostgres95 JDBC driver. Once this support becomes available, however, it should just be a matter of creating a subclass of `java.lang.ClassLoader` (Sun Microsystems, 1998a) to allow the loading of Java classes directly from the repository. This will make Swift much easier to extend with new representations and translations; simply storing the Java classes in the repository will immediately make the new representation or translation available to Swift without having to make any modifications to the environment.

The rule selection and evaluation strategy followed in this thesis is a slightly modified form of that used by Amor's (1997) VML mapping system. That is, for each element in a description, all rules that may apply to that element are determined, then an attempt is made to build element collections that match each of these rules. All rules that cannot be matched are deleted, and the rest are applied to the element collections built. An alternative approach identified in Section 4.7 on page 94 was to take each rule, find all collections of elements that match the source constructs of the rule, then apply the rule to each such collection. There may also be other rule selection and evaluation strategies not identified here. It would be interesting to test the different rule selection and evaluation strategies in combination with both synchronous and asynchronous consistency maintenance mechanisms. It may even be possible to combine different strategies to provide a more flexible approach.

Subsumption/exclusion graphs form an important part of the rule selection and evaluation process described in Section 4.7 on page 94. These graphs can indicate the exclusions among rules and are used to determine the order in which rules should be evaluated. The author's experience has shown that it is sometimes possible to define the rules of a translation in more than one way. Subsumption/exclusion graphs could therefore provide a means of optimising rules by indicating to translation definers the extent of interdependency among rules. Translations with less rule interdependency should be more efficient.

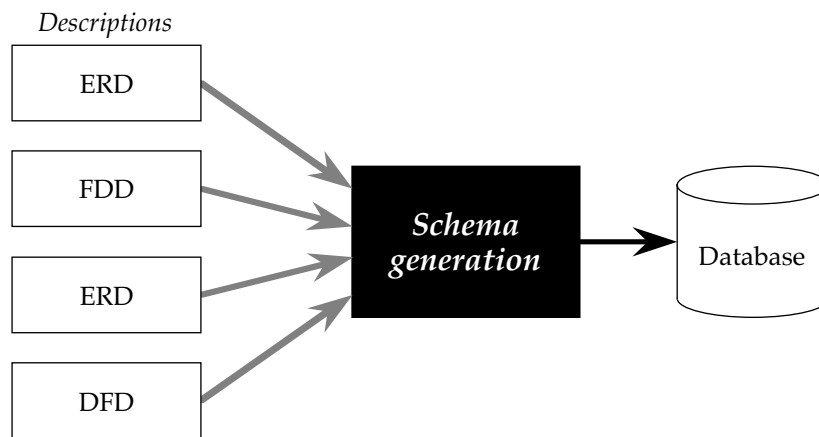
It is interesting that there is no discernible pattern to the subsumption/exclusion graphs for the translations defined in this thesis, and even the subsumption/exclusion graphs for the two directions of a translation can be completely different (see Chapter 5). Some translations have a large number of exclusions, while others have none. It is as yet unknown whether there is some property of a translation that might indicate whether it has a large number of exclusions, or whether this is just a side effect of the rule definition approach used here.

VML-S was introduced in Chapter 7 to provide a way of defining the details of translations, but this extended version of VML has not yet been implemented and is thus not included in Swift. Translations in Swift are at present manually coded in Java based on the original rules. It would obviously be desirable to incorporate VML-S into Swift in order to remove this extra step. The main reason for the lack of a VML-S implementation is that Grundy (1998, personal communication) has already stated that he is working on porting the VML mapping system to Java, and has expressed an interest in the extensions outlined here. Any implementation as part of this research would therefore have been an unnecessary duplication of effort.

Finally, it would be useful to automate the SIG-based methods for categorising the expressive overlap of representations and measuring the relative quality of translations. The refined relative quality measure described in Section 8.4.2 on page 236 should be amenable to automation, but it is likely that the process for categorising expressive overlap can be only partially automated. This is because the process depends to a large extent on human intervention to determine likely candidate subgraphs to be tested for isomorphism. Miller et al. (1994a, theorems 4.3 and 4.4) show that the problem of testing for SIG equivalence is in general undecidable, and note that the problem of detecting graph isomorphism may be NP-complete. This is the worst case, however, and (Miller et al., 1994a, p. 20) have discussed how reasonably efficient algorithms could be developed. One step of the process that can be relatively easily automated is that of generating the SIGs to be compared. The schema intension graph formalism is effectively just another representation, and it was shown in Section 8.2.1 on page 215 that translating an ERD to a SIG is relatively simple. It should therefore not be difficult to incorporate SIGs and the translations to generate them into Swift.

## 10.5 Schema generation from multiple descriptions

A typical function of a CASE tool is to take a description and generate a corresponding database schema, typically an SQL schema of some sort. This process generally only involves a single source description, although some information from other descriptions may be included depending on the degree of integration of the data dictionary. A key element of this research is the use of multiple representations to describe a viewpoint, which leads to multiple descriptions of the viewpoint. The schema generation process should ideally incorporate as much of the viewpoint as possible in order to provide a richer schema, as shown in Figure 10.3, so some way of incorporating multiple descriptions into the schema generation process must be identified.

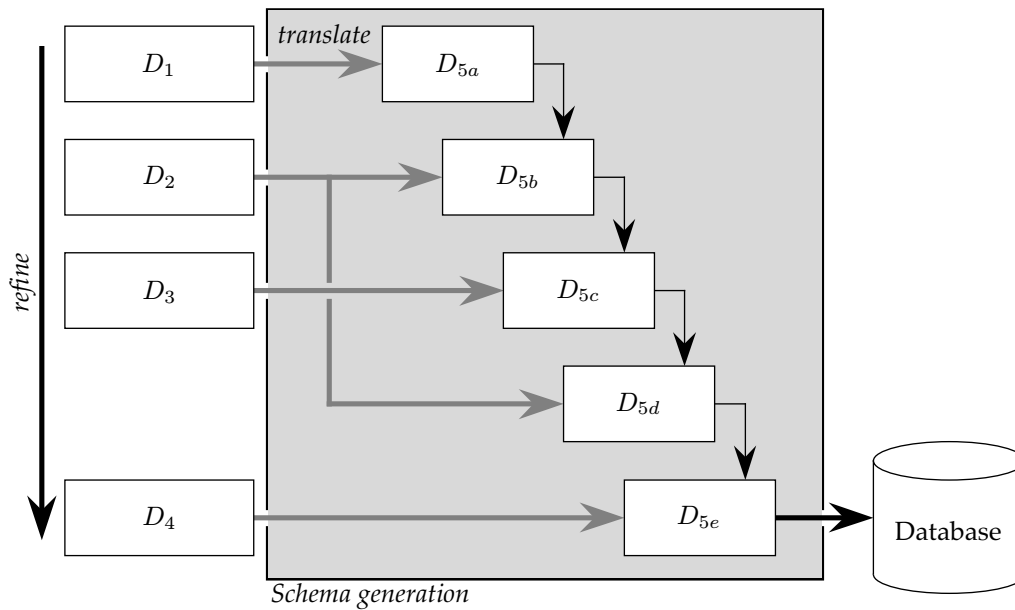


**Figure 10.3:** Schema generation from multiple representations

Generating a schema effectively becomes a multi-step process, in which we must determine how best to use the different descriptions that comprise the viewpoint. One approach could be to refine the generated schema in a stepwise fashion according to the information contained within each description, as illustrated in Figure 10.4. The refinement operator is denoted by the symbol  $\mapsto$  and the refinement of two descriptions to form a third is denoted by:

$$\left. \begin{array}{l} D_1(V, T_1, S_1) \\ D_2(V, T_2, S_2) \end{array} \right\} \mapsto D_3(V, T_3, S_3).$$

In the example shown in Figure 10.4 on the next page, the sources are the two entity-relationship descriptions  $D_1(V, E-R, ERD_{Martin})$  and  $D_3(V, E-R, ERD_{Chen})$ , the



**Figure 10.4:** Progressive refinement of a generated schema

functional dependency description  $D_2(V, FuncDep, FDD_{Smith})$  and the data flow description  $D_4(V, DataFlow, DFD_{G\&S})$ . The target representation for schema generation is  $\mathfrak{R}_r(Relational, SQL/92)$ . The first step could be to generate the SQL description  $D_{5a}(V, Relational, SQL/92)$  from the E-R description  $D_1$ . The next step could then be to normalise this description using the information contained within the FDD  $D_2$ . This refinement is expressed (in slightly abbreviated form) as:

$$\begin{array}{l}
 D_1(V, E-R, ERD_{Martin}) \rightarrow D_{5a}(V, Relational, SQL/92) \\
 D_2(V, FuncDep, FDD_{Smith}) \left\{ \begin{array}{l} \rightarrow D_{5b}(V, Relational, SQL/92). \end{array} \right.
 \end{array}$$

The refinement process continues for each remaining description of the viewpoint ( $D_3$  and  $D_4$  in Figure 10.4), at each step producing a further refined SQL description ( $D_{5c}, D_{5d}, D_{5e}$ ). The final SQL description  $D_{5e}$  is then used to generate the database. Because of the extra information that multiple descriptions can provide, this process could produce a ‘better’ schema than if it had been generated from just a single description, although it must be remembered that not all of this extra information may be able to be expressed in the target representation.

There is no particular significance to the order of refinements used in the example above; indeed, a different order might prove more appropriate or efficient. In addition, there is no need for all the steps to generate a description using the ultimate target representation. An alternative set of steps for the example in Figure 10.4 could be:

- translate  $D_4$  into  $D_6(V, E-R, ERD_{Martin})$ ;
- refine  $D_1, D_3$  and  $D_6$  to give  $D_7(V, E-R, ERD_{Martin})$ ;
- translate  $D_7$  into  $D_{8a}(V, Relational, SQL/92)$ ; and
- refine  $D_{8a}$  using  $D_2$  to give  $D_{8b}(V, Relational, SQL/92)$  as the final output.

This process would be expressed as follows:

$$\begin{array}{l}
 D_4(V, DataFlow, DFD_{G\&S}) \rightarrow D_6(V, E-R, ERD_{Martin}) \\
 D_1(V, E-R, ERD_{Martin}) \\
 D_3(V, E-R, ERD_{Chen})
 \end{array}
 \left. \vphantom{\begin{array}{l} D_4 \\ D_1 \\ D_3 \end{array}} \right\} \rightarrow D_7(V, E-R, ERD_{Martin}),$$
  

$$\begin{array}{l}
 D_7(V, E-R, ERD_{Martin}) \rightarrow D_{8a}(V, Relational, SQL/92) \\
 D_2(V, FuncDep, FDD_{Smith})
 \end{array}
 \left. \vphantom{\begin{array}{l} D_7 \\ D_2 \end{array}} \right\} \rightarrow D_{8b}(V, Relational, SQL/92).$$

Automatic optimisation of this process could be a particularly interesting area of research. It is also possible that changing the refinement order may result in a different final schema, but it seems likely that this would indicate a problem with the consistency of the viewpoint.

## 10.6 Consistency maintenance

As noted in Chapter 3, there are two major approaches to maintaining consistency among descriptions: either synchronously or asynchronously. It should be noted that although the asynchronous and synchronous approaches have been discussed separately in this thesis, they are not necessarily mutually exclusive; rather both are appropriate in different circumstances. It may not always be appropriate to maintain consistency between two descriptions; for example, suppose a developer translates an ERD to an FDD in order to experiment with the FDD. In a synchronous environment, this could lead to unwanted changes in the original ERD (see Figure 3.11 on page 62). Developers may also wish to explore alternate design paths, in which case maintaining consistency with other descriptions is probably not desired.

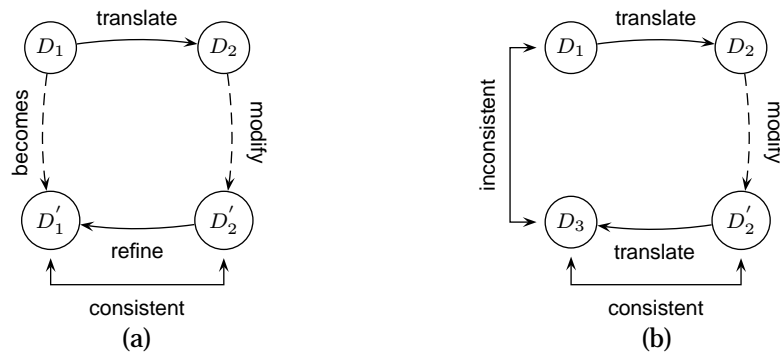
Rather than design an environment that exclusively follows one approach or the other, it would seem sensible to allow users to determine the level of consistency required among descriptions. Managing this could become rather complex, although many of the issues will be similar to those encountered in version management tools. One issue peculiar to this area, however, is the sensitivity of descriptions to changes in a synchronous environment (see Section 3.6 on page 58). Small, incremental changes to the source description may, up to a certain critical point, result in small incremental changes to the target description. Once this critical point is reached, however, a small change in the source description can result in a major change in the target description (see Figure 3.10 on page 61). What determines this critical point is an open question. It may be dependent on the representations in question, or it may be more closely related to the structure of the descriptions being manipulated. One possible solution could be to introduce some form of transaction management to the design process, as suggested in Section 3.6. The update record approach used in the MViews/JViews frameworks (Grundy et al., 1996) may provide a useful solution.

Another issue that arises from an environment with ‘tunable’ consistency maintenance is that it may be possible to optimise translations for one approach or the other. That is, some translations may be better optimised for the asynchronous approach, whereas others will be better optimised for the synchronous approach. An obvious solution to this is to store translations as VML-S specifications only, and generate translations that are optimised for the current consistency maintenance regime as required. The disadvantage of this is the overhead required to generate translations, although some form of caching mechanism could be used, similar to the library of translations in Atzeni and Torlone’s (1997) MDM environment. It is also unclear at present how much optimisation would be required for each approach, which could render the question moot.

The asynchronous approach can add an extra layer of complexity if new descriptions are generated by translations rather than updating existing ones. Consider the description translation  $D_1(V, E-R, ERD_{Martin}) \rightarrow D_2(V, FuncDep, FDD_{Smith})$ . If  $D_2$  is modified (giving  $D_2'$ ), then this modified FDD could be used to either refine the existing  $D_1$  (giving  $D_1'$ ), or to generate a new ERD description  $D_3$ . In the former case, the resulting descriptions  $D_1'$  and  $D_2'$  are consistent with each other, as shown in Fig-



ure 10.5(a). In the latter case, however,  $D'_2$  and  $D_3$  are consistent with each other, but neither is consistent with  $D_1$ , as shown in Figure 10.5(b). In some situations (such as exploring alternate design paths) this may be desirable, but in general this would lead to an inconsistent viewpoint. The latter case becomes even more complex if a fourth E-R description  $D_4$  is generated from  $D_2$  before it is modified. If  $D_1$ ,  $D_3$  and  $D_4$  should be consistent with each other, how may this be achieved?



**Figure 10.5:** Effects of asynchronous translations: (a) when translations refine existing descriptions; (b) when translations create new descriptions

It was suggested in Section 4.6 on page 91 that the potential consistency degradation caused by heuristics could be ameliorated by bundling the application of heuristics into the enrichment process, thus allowing user confirmation of all heuristics before they are applied. It may still be possible, however, for the user to incorrectly allow the application of a heuristic, thus producing an inconsistent viewpoint. The use of translations to highlight potential inconsistencies within a viewpoint will be of particular use in this situation, and therefore requires further examination.

A final issue not resolved in this thesis is that some translations may change the level of abstraction of a description. For example, if an ERD is translated to an FDD and back to an ERD, the second ERD is effectively an implementation-level ERD, whereas the first is a conceptual-level ERD (Tsichritzis and Klug, 1978). While both of these ERDs will be consistent with respect to the information contained within them, they will *not* be consistent with respect to the way that information is portrayed. This is an issue that will need to be addressed in any future research.

## 10.7 Outstanding issues with VML-S

Although the syntax and behaviour of VML-S were defined in Chapter 7, the language has not yet been implemented. This has resulted in several issues with VML-S not being fully examined.

An issue that will need to be addressed at some point is the generation of possible element groups for arbitrary-length lists of constructs (see Section 7.5.2 on page 207). If the header of a VML-S *inter\_class* definition contains the entry `construct [ ]`, then all possible lists of elements of type `construct` must be generated. Since the maximum length of the list is unknown, the number of possible lists to be generated can rapidly grow extremely large (up to  $\sum_{i=1}^n n^i$  for  $n$  elements). This will need to be addressed by any implementation of VML-S. It may be possible to replace such expressions with groups instead.

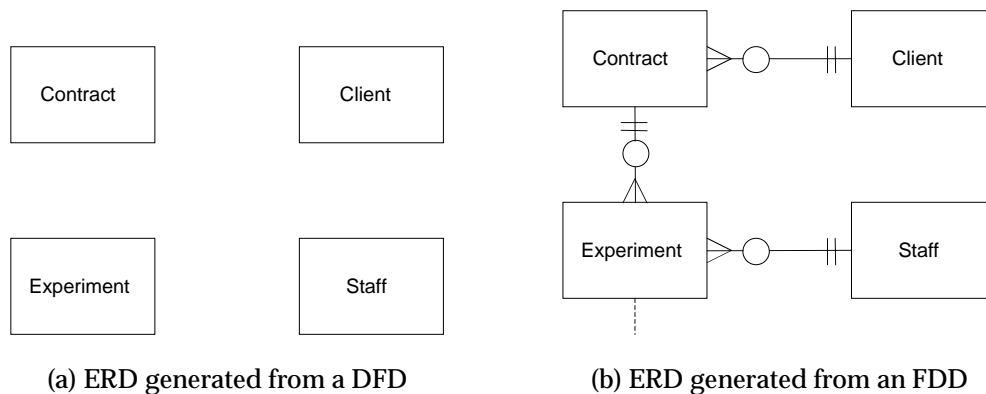
The discussion above of optimising translations for either synchronous or asynchronous use raises the interesting question of how this optimisation should be carried out. The VML-S specification of a translation is independent of the consistency maintenance mechanism used, but the actual implementation cannot be. One approach would be to extend the VML-S mapping system with some form of built-in optimisation, but this would probably be rather slow. An alternative is to build an ‘optimising compiler’ that converts the VML-S specifications either directly into machine-executable code, or into some other language such as Java (it might even be possible to compile directly from VML-S to Java byte-code).

A final issue with VML-S is that of rule specialisation. In this thesis, rule specialisation has been defined in VML-S using the `inherits` clause of the *inter\_class* definition. As noted in Section 7.2 on page 180, this clause merely incorporates the definition of the inherited rule into the current rule, rather than providing inheritance in the object-oriented sense. Nevertheless, this clause does provide a workable mechanism for defining specialised scheme-level rules, and was used in the definitions in Chapter 5 and Appendix E. It may, however, be appropriate to extend the `inherits` clause to provide a form of inheritance closer to that used in object-oriented programming languages. This would provide a more natural and flexible way of defining specialised rules than at present.

## 10.8 Other issues

The research described in this thesis has been deliberately limited to a *single* viewpoint. An obvious extension is therefore to explore the use of multiple representations in the context of *multiple* viewpoints. As the use of multiple representations provides a more comprehensive description of a viewpoint, it is to be expected that this will provide more information that will be beneficial to the viewpoint integration process.

Related to this is the issue of highlighting potential inconsistencies in a viewpoint, discussed in Section 4.5 on page 86. Potential inconsistencies within a viewpoint may be highlighted by translating descriptions into the same representation, and then determining whether the resulting descriptions are consistent. At present, this determination is carried out by the developer. Automatically determining the consistency of two descriptions has been largely ignored in this thesis, although methods from schema integration would probably prove useful here. Schema intension graphs could also prove useful, as they were originally developed as a means of determining whether two schemas have differing relative information capacities. SIGs may not be a complete solution, however, as the descriptions to be tested may be consistent, but not equivalent, as shown in Figure 10.6.



**Figure 10.6:** Two description fragments that are consistent but not equivalent

In Section 2.4.3 on page 30, it was noted that the major disadvantage of the individual interfacing strategy is the number of interfaces required to provide a complete set of translations between all representations. A complete set of translations will by definition provide the best quality, as each pair of representations will have a translation

tuned specifically to that pairing. It may, however, be possible to define a smaller set of interfaces that does not affect the overall quality of translations, depending on the expressive overlap between the representations involved. Consider the three representations  $\mathfrak{R}_m$ ,  $\mathfrak{R}_n$  and  $\mathfrak{R}_o$ , where the expressive power of  $\mathfrak{R}_m$  is inclusive of  $\mathfrak{R}_n$  (that is,  $\mathfrak{R}_n \preceq \mathfrak{R}_m$ ). Any translation from  $\mathfrak{R}_n$  to  $\mathfrak{R}_m$  will be complete, as noted in Section 3.5 on page 55. If the expressive power of  $\mathfrak{R}_n$  is equivalent (or inclusive) to that of  $\mathfrak{R}_o$ , then there is no need to define a direct translation from  $\mathfrak{R}_o$  to  $\mathfrak{R}_m$ , as the composite translation  $\mathfrak{R}_o \rightarrow \mathfrak{R}_n \rightarrow \mathfrak{R}_m$  will have the same effect. While this will in theory reduce the total number of translations required, the extent of this reduction in practice remains unclear.

It was suggested in Chapter 5 that translations can leverage the capabilities of other translations. For example, normalisation was removed from the  $\mathfrak{R}_e(E-R, ERD_{Martin}) \rightarrow \mathfrak{R}_r(Relational, SQL/92)$  translation because the  $\mathfrak{R}_f(FuncDep, FDD_{Smith}) \rightarrow \mathfrak{R}_e(E-R, ERD_{Martin})$  translation already provided normalisation. Including normalisation in both translations is an unnecessary duplication of effort, so it is sensible to remove normalisation from the  $\mathfrak{R}_e \rightarrow \mathfrak{R}_r$  and instead leverage the normalisation in the  $\mathfrak{R}_f \rightarrow \mathfrak{R}_e$  translation. This could be done in two ways:

1. by translating an ERD ‘through’  $\mathfrak{R}_f$  in order to normalise it, and then generating SQL from the normalised ERD, that is,  $\mathfrak{R}_e \rightarrow \mathfrak{R}_f \rightarrow \mathfrak{R}_e \rightarrow \mathfrak{R}_r$ ; or
2. by translating an ERD to an FDD, then translating that directly to SQL, that is,  $\mathfrak{R}_e \rightarrow \mathfrak{R}_f \rightarrow \mathfrak{R}_r$ .

The second approach would obviously only be feasible if the  $\mathfrak{R}_f \rightarrow \mathfrak{R}_r$  translation existed.

Leveraging existing translations in this manner can reduce the complexity of translations by removing functionality that is not strictly necessary, and also removes unnecessary duplication of effort. It may even be possible to take this a step further and abstract common features (such as normalisation) out of translations completely, and place them in a ‘library’ of shared translation operations that could then be called upon by any other translation. This is similar in principle to the way in which the rules of a translation are divided into scheme-level rules specific to that translation

and technique-level rules that are shared across related translations, but is probably closer to the macro concept in ORECOM (Su and Fang, 1993).

## 10.9 Summary

In this chapter, several outstanding issues have been identified and possible directions for future research identified. The current research has by necessity included only a small number of representations, so this is an obvious area for improvement. The CoCoA modelling representation was presented as a possible alternative for defining the constructs of representations.

The relative quality measurement developed in Chapter 8 provides a useful indication of the relative quality of different translations, but does not consider all aspects of a translation. It is expected that including properties such as the completeness of rules into the measurement will result in a more accurate measure of relative quality.

Various outstanding issues with the Swift prototype were discussed, and possible areas for extension identified. One such area was the integration of VML-S into Swift, which has not been done because other researchers are already working in this area (Grundy, 1998, personal communication). In addition, VML-S could potentially be redesigned to allow better definition of technique- and scheme-level rules.

The issue of generating a useful schema from multiple source descriptions using different representations was discussed, and the concept of *refinement* was introduced as a possible mechanism for achieving this. Some outstanding issues with maintaining consistency among multiple descriptions were also identified and discussed.

In the next chapter, the results of this research are summarised and conclusions are drawn.



# Chapter 11

## Conclusion

The goals of this thesis were:

1. Improve a viewpoint in terms of depth and detail by using multiple representations to describe the viewpoint.
2. Facilitate the use of multiple representations within a viewpoint by translating descriptions between representations.
3. Develop a consistent and unified terminology for discussing representations and translations between them.
4. Develop effective ways of specifying translations.
5. Use translations to enable the highlighting of potential inconsistencies between descriptions of a viewpoint.
6. Identify ways of improving translation quality.
7. Show that the approach presented here is novel and practicable.

Each of these goals will now be examined in turn.

### **11.1 Improving the depth and detail of a viewpoint**

The first goal of this thesis was to improve a viewpoint in terms of depth and detail by using multiple representations to describe the viewpoint. Use of multiple representations to describe a viewpoint was discussed in Chapters 2 and 3, and the effect of using multiple representations to describe a viewpoint was explored in the case study in Chapter 9.

It was shown in the case study that adding new descriptions expressed using different representations to a viewpoint allows expression of more information than would be possible if a single representation were used to describe the viewpoint. For example, information on how data flow through a system cannot be expressed using purely ‘structural’ representations such as entity-relationship diagrams. Such information can only be included in a viewpoint by using appropriate representations.

The author’s approach to using multiple representations is based on performing translations between descriptions expressed in different representations. Heuristics were introduced in Chapter 4 as a means of improving the quality of translations, and have the beneficial side-effect of also improving the depth and detail of a viewpoint. Heuristics can extract and make explicit information that is implicit in the source description, thus improving the depth and detail of the viewpoint. This was illustrated in the case study in Chapter 9 by the translation of a data flow diagram (DFD) into an entity-relationship diagram (ERD). The heuristics of this translation allowed the generation of relationships between derived entities that would otherwise not have been generated. The heuristics have extracted information that is implicit in the source description, thus enhancing the depth and detail of the viewpoint.

In summary, the use of multiple representations to describe a viewpoint improves the depth and detail of a viewpoint by allowing additional information to be expressed, while use of heuristics in translations improves depth and detail by extracting implicit information from source descriptions that might otherwise be missed.

## **11.2 Using translations to facilitate the use of multiple representations**

The second goal of the thesis was to facilitate the use of multiple representations within a viewpoint by translating descriptions between representations. The translation process and issues arising from it were discussed in Chapter 4.

It was shown in the case study in Chapter 9 that translations facilitate the use of multiple representations by allowing an analyst to easily create new descriptions based on already existing descriptions. That is, the translation produces a new description



that can be used as a template for further development by the analyst. For example, in the case study a DFD description was translated into an ERD description, which was then extended by adding new entities and relationships. Even when translations are of relatively low quality, such as translating a DFD to an ERD, enough information is translated from the source description to provide a useful basis on which to build.

### 11.3 Terminology framework

The third goal of the thesis was to develop a consistent and unified terminology for discussing representations and translations between them. The area of facilitating the use of multiple representations is a relatively new one, and the author is aware of only three other groups who have worked on this problem (Atzeni and Torlone, 1993; Grundy, 1993; Su et al., 1992). Interestingly, all three groups developed their work independently over the same approximate time period (1991–94). Atzeni and Torlone (1995) commented at the time that there was “not much literature related to the problem”. This independent development and the variety of fields from which the projects were initiated has resulted in very diverse terminologies. Different groups have used the same term to mean different things, or have used several terms to denote the same concept, as was shown in Table 3.4 on page 65.

A useful research contribution of this thesis has therefore been the definition of a well-defined and integrated terminology for describing the use of multiple representations, based upon a viewpoint framework (Finkelstein et al., 1989; Easterbrook, 1991a; Darke and Shanks, 1995b). The concepts of *representation*, *technique*, *scheme* had previously been used in a somewhat imprecise fashion; formal definitions of these were provided in Chapter 3. The concepts of *description*, *construct* and *element* were also introduced in Chapter 3 to provide a complete terminology that has been used throughout the thesis.

The concept of a viewpoint also provides a useful context within which to discuss and perform description translations, and it is hoped that this thesis may act as a focus point for bringing together the various projects that have been undertaken by other researchers in this area.

## 11.4 Translation specification

The fourth goal of the thesis was to develop effective ways of specifying translations. It can be useful to specify translations both at an abstract level and at a more detailed ‘implementation’ level. An abstract notation was defined in Chapters 3 and 4 to allow the specification of representations, descriptions and translations at a high level. This notation provides a concise means of expressing representations, descriptions and translations, and can also be used as a template for building more detailed low-level specifications.

An extended version of Amor’s (1997) View Mapping Language (VML), known as *VML-S*, was defined in Chapter 7 for the purpose of building detailed low-level specifications of translations. VML was chosen because of its many useful features, such as its declarative nature and the ability to define bidirectional translations with a single specification. VML was originally designed for specifying translations of data values between schemas, but a representation definition is effectively a schema describing the constructs of a representation, so VML can also be used to specify translations of descriptions between representations.

In its original form, VML did not adequately deal with important translation issues such as rule exclusion. It was therefore necessary to define several extensions to both the VML syntax and the algorithms used by the VML mapping system (see Chapter 7 and Appendix F). These extensions were not implemented by the author as it would have duplicated the effort already underway by Grundy (1998) to incorporate VML into the JViews framework. Despite the lack of an implementation, *VML-S* has been successfully used to specify three translations (see Appendix F).

## 11.5 Highlighting potential inconsistencies within a viewpoint

The fifth goal of the thesis was to use translations to enable the highlighting of potential inconsistencies between descriptions of a viewpoint. The issue of maintaining consistency between multiple descriptions of a phenomenon was discussed in Chapter 3. This was expanded on in Chapter 4 to produce a process for using translations to

highlight potential inconsistencies between descriptions of a viewpoint. The efficacy of this process was examined in Chapter 9.

In order to highlight potential inconsistencies between descriptions, they must first be translated so that they are expressed using the same representation. The resultant descriptions are then compared by the analyst for discrepancies, which may indicate an inconsistency between the original descriptions. An example of this process was presented in the case study in Chapter 9.

While it was shown in the case study that this process can be used to highlight *potential* inconsistencies between descriptions of a viewpoint, the process cannot be used to *detect* inconsistencies with any degree of certainty. This is because of the high degree of semantic knowledge required to determine whether a particular discrepancy actually represents an inconsistency. As a result, this process will be difficult to automate. In addition, the target representation for comparison should ideally have greater expressive power than the representations of the source descriptions, so that no information is 'lost' during the translations. This may not always be possible, however.

Nevertheless, the use of translations to highlight potential inconsistencies is still useful and may even highlight possibilities that have not occurred to or have been misinterpreted by the analyst. For example, in the case study the analyst realised after applying the comparison process that there was an entity missing from the original ERD description, and that a relationship in the ERD had been defined incorrectly. A viewpoint must by definition be internally consistent, so being able to highlight potential inconsistencies between descriptions of a viewpoint is important.

## 11.6 Improving translation quality

The sixth goal of the thesis was to identify ways of improving translation quality, because the efficacy of the translation-based approach to facilitating the use of multiple representations is highly dependent on the quality of translations. The quality of a translation is defined as how completely it maps constructs of its source representations onto constructs of its target representation, and was introduced in Chapter 2.

Two novel mechanisms for improving translation quality, *heuristics* and *enrichment* were introduced in Chapter 4. Heuristics are translation rules that *usually* produce a

semantically consistent result, but may sometimes produce a semantically inconsistent result. Use of heuristics should therefore always be under the control of the analyst, to ensure that heuristics are applied appropriately. This can be done by incorporating application of heuristics into the enrichment process discussed below.

Heuristics allow a translation to translate more information than would normally be possible. For example, when a domain flag in a functional dependency diagram (FDD) is referenced more than twice, this can be mapped to a type hierarchy when translating to an ERD. Without the heuristic, the same source structure would instead be mapped to a collection of one-to-one relationships. Another example is generating relationships when translating from a DFD to and ERD, as was illustrated in the case study in Chapter 9. Without this heuristic, these relationships would not be produced.

In Chapter 8, a method for measuring the relative quality of translations was developed from Hull's (1986) concept of relative information capacity. While this does not provide an objective measure of the quality of an individual translation, it does provide a way of comparing the quality of two or more translations relative to each other. This measure was used in Chapter 8 to show that use of heuristics can improve the quality of translations.

An alternate approach to improving the quality of translations is that of enrichment. This method deals with the potential 'gain' of information that occurs when it is required to generate constructs in the target representation that have no analogues in the source representation. Enrichment can occur, before, during and after a translation, and involves the provision of additional information that may be used by the translation. Existing approaches to translating between multiple representations have considered both pre- and post-enrichment, but do not appear to have considered enrichment during a translation.

Enrichment falls outside the scope of the relative translation quality measurement, as it applies to specific descriptions rather than representations in general. Although this means that the effect of enrichment on translation quality cannot be easily measured, enrichment should always have a positive effect on the quality of a translation, as it provides information that the translation could otherwise never generate.

In summary, two novel mechanisms for improving translation quality, heuristics and enrichment, have been introduced in this thesis. It has been shown that heuristics

improve the quality of translations, and it seems likely that enrichment will also improve the quality of translations, although no measure has as yet been devised to show this.

## **11.7 Novelty and practicability of the approach**

The final goal of the thesis was to demonstrate the novelty and practicability of the approach presented. In Chapter 9, the following novel aspects of the approach outlined in this thesis were identified:

- An abstract notation for expressing representations, descriptions and translations at a high level (Chapters 3 and 4), and extensions to Amor's (1997) View Mapping Language to support the detailed low-level specification of translations between representations (Chapter 7).
- An analysis of the issues (such as rule exclusion) that arise when translating descriptions from one representation to another, and possible ways of addressing these issues, such as subsumption/exclusion graphs and extensions to VML (Chapter 4).
- The use of heuristics and enrichment during translations to improve the quality of translations (Chapter 4).
- The basis of a method for using translations to detect inconsistencies within a viewpoint (Chapter 4).

The following three novel contributions can also be identified:

- A novel method for measuring the relative quality of translations, derived from Miller et al.'s (1994b) schema intension graphs was described in Chapter 8.
- A novel framework for discussing the use of multiple representations, based on viewpoint concepts was described in Chapter 3.
- A diverse range of representations have been included in this research from the outset.

This last point requires further explanation. Other researchers in this area have focused on semantic data models such as the E-R approach and the relational model. That is, they have focused mainly on modelling the *structure* of data but not other aspects, such as how data flow through a system. Both Atzeni and Torlone (1993) and Su et al. (1992) explicitly state that they only consider semantic data models, which may limit the applicability of their environment to other modelling approaches such as data flow modelling or state-transition modelling. Grundy and Venable (1995b) note that their approach can support more diverse representations such as DFDs, but it is not clear from their published work whether this has been implemented in practice.

This research has included a diverse range of representations from the outset. The four representations used in this thesis are drawn from different modelling paradigms:  $\mathfrak{R}_e(DataFlow, DFD_{G\&S})$  is drawn from data flow modelling;  $\mathfrak{R}_e(E-R, ERD_{Martin})$  is a semantic data model;  $\mathfrak{R}_f(FuncDep, FDD_{Smith})$  provides a more functional style of modelling; and  $\mathfrak{R}_r(Relational, SQL/92)$  is representative of an ‘implementation’-oriented data model. This research therefore provides a good foundation for further work in the use of diverse multiple representations.

### 11.7.1 Practicability

There were two major concerns with the practicability of the approach presented in this thesis:

1. the  $O(n^2)$  number of interfaces implied by the individual interfacing strategy; and
2. how well translations scale to more complex source descriptions.

The first issue arose because of a deliberate choice on the author’s part to focus on the quality of translations rather than the number of interfaces required. A significant reduction in the number of interfaces required is probably only possible by using a different interfacing strategy, which will result in lower-quality translations. This is a definite shortcoming of the approach presented here, and will need to be addressed by any future work.

The scalability of translations was demonstrated in Chapter 9 using the Swift prototype. The time taken to complete simpler translations was shown to be  $O(n)$ , while

the time taken to complete more complex translations was shown to be  $O(n^2)$ . Despite this, experiments with the Swift prototype showed that this is not a critical issue in practice, as translations of very large descriptions were still completed in a reasonable timeframe.

## 11.8 Closing remarks

In closing, this thesis has produced the following research contributions:

- A well-defined and integrated terminology for discussing multiple representation issues, derived from viewpoint research.
- The use of multiple diverse representations to describe a viewpoint, including process-oriented representations such as data flow diagrams, in addition to the more typical semantic data models.
- An abstract notation for expressing representations, descriptions and translations at a high level, and extensions to Amor's (1997) View Mapping Language to support the detailed low-level specification of translations between representations.
- An analysis of the issues (such as rule exclusion) that arise when translating descriptions from one representation to another, and possible solutions to these issues.
- The novel use of heuristics and enrichment during translations to improve the quality of translations, and a method for measuring the relative quality of translations.
- The basis of a method for using translations to highlight potential inconsistencies between descriptions within a viewpoint.

As outlined above, the goals of the thesis have been achieved, although some issues will require further investigation. A prototype environment has been implemented that supports the use of multiple representations and allows translations to be performed between these representations. Important issues surrounding the translation process, such as rule exclusion, have been identified and possible solutions presented.

It has been demonstrated that the use of heuristics in translations results in an increase in the quality of translations, that translations can facilitate the use of multiple representations and that using multiple representations to describe a viewpoint can improve the viewpoint in terms of depth and detail.



# References

**Note:** URLs are correct as of the date of printing, but may change in future.

Abiteboul, S. and Hull, R. (1987). IFO: A formal semantic database model, *ACM Transactions on Database Systems* **12**(4): 525–565.

Altmann, R., Hawke, A. and Marlin, C. (1988). An integrated programming environment based on multiple concurrent views, *Australian Journal of Computing* **20**(2): 65–72.

Amor, R. W. (1997). *A Generalised Framework for the Design and Construction of Integrated Design Systems*, PhD thesis, Department of Computer Science, University of Auckland, Auckland, New Zealand.

**URL:** [ftp://helios.bre.co.uk/pub/ra\\_phd/](ftp://helios.bre.co.uk/pub/ra_phd/)

Amor, R. W. (1998). Personal communication.

Amor, R. W., Augenbroe, G., Hosking, J., Rombouts, W. and Grundy, J. (1995). Directions in modelling environments, *Automation in Construction* **4**: 173–187.

**URL:** [http://www.cs.auckland.ac.nz/~john-g/papers/a\\_in\\_c95.ps.gz](http://www.cs.auckland.ac.nz/~john-g/papers/a_in_c95.ps.gz)

Armstrong, W. (1974). Dependency structures of data base relationships, in J. L. Rosenfeld (ed.), *IFIP Congress '74 (Information Processing '74)*, North-Holland, Stockholm, Sweden, pp. 580–583.

Atzeni, P. and Torlone, R. (1993). A metamodel approach for the management of multiple models and the translation of schemes, *Information Systems* **18**(6): 349–362.

Atzeni, P. and Torlone, R. (1995). Schema translation between heterogeneous data models in a lattice framework, in R. Meersman and L. Mark (eds), *Database Applications Semantics, Sixth IFIP TC-2 Working Conference on Data Semantics (DS-6)*, IFIP, Chapman & Hall, London, Stone Mountain, Atlanta, Georgia, USA, pp. 345–361.

**URL:** <http://www.dia.uniroma3.it/~atzeni/psfiles/ifip.ps.gz>

- Atzeni, P. and Torlone, R. (1996a). Management of multiple models in an extensible database design tool, in P. Apers, M. Bouzeghoub and G. Gardarin (eds), *Fifth International Conference on Extending Database Technology (EDBT'96)*, Vol. 1057 of *Lecture Notes in Computer Science*, Springer-Verlag, Avignon, France, pp. 79–95.  
**URL:** <http://www.dia.uniroma3.it/~atzeni/psfiles/edbt96.ps.gz>
- Atzeni, P. and Torlone, R. (1996c). MDM: A multiple-data-model tool for the management of heterogeneous database schemes, part I, Handout material for the TMR Seminar on Metamodeling, Paris, France.  
**URL:** <http://www.dia.uniroma3.it/~atzeni/psfiles/parigi1.ps.gz>
- Atzeni, P. and Torlone, R. (1996b). MDM: A multiple-data-model tool for the management of heterogeneous database schemes, part II, Handout material for the TMR Seminar on Metamodeling, Paris, France.  
**URL:** <http://www.dia.uniroma3.it/~atzeni/psfiles/parigi2.ps.gz>
- Atzeni, P. and Torlone, R. (1997). MDM: A multiple-data-model tool for the management of heterogeneous database schemes, in J. M. Peckman (ed.), *SIGMOD 1997 International Conference on the Management of Data*, ACM, ACM Press, Tucson, Arizona, pp. 528–531.  
**URL:** <http://www.dia.uniroma3.it/~atzeni/psfiles/sigmodPreprint.ps.gz>
- Batini, C., Ceri, S. and Navathe, S. B. (1992). *Conceptual Database Design: An Entity-Relationship Approach*, Benjamin/Cummings, Redwood City, California.
- Batini, C., Furlani, L. and Nardelli, E. (1985). What is a good diagram? A pragmatic approach, in P. P. Chen (ed.), *Fourth International Conference on the Entity-Relationship Approach*, IEEE Computer Society Press/North Holland, Chicago, Illinois, pp. 312–319.
- Batini, C. and Lenzerini, M. (1984). A methodology for data schema integration in the entity relationship model, *IEEE Transactions on Software Engineering* **SE-10**(6): 650–663.
- Batini, C., Lenzerini, M. and Navathe, S. (1986). A comparative analysis of methodologies for database schema integration, *ACM Computing Surveys* **18**(4): 323–364.

- Batra, D. and Antony, S. R. (1994). Effects of data model and task characteristics on designer performance: A laboratory study, *International Journal of Human-Computer Studies* **41**: 481–508.
- Batra, D. and Srinivasan, A. (1992). A review and analysis of the usability of data management environments, *International Journal of Man-Machine Studies* **36**: 395–417.
- Beeri, C., Fagin, R. and Howard, J. H. (1977). A complete axiomatization for functional and multivalued dependencies in database relations, in D. C. Smith (ed.), *1977 ACM SIGMOD International Conference on Management of Data*, ACM, New York, Toronto, Canada, pp. 47–61.
- Borowski, E. and Borwein, J. (1989). *Dictionary of Mathematics*, Collins, Glasgow.
- Brien, S. M. and Nicholls, J. E. (1992). Z base standard, *Technical Monograph PRG-107*, Oxford University Computing Laboratory, Oxford, UK.  
**URL:** <ftp://ftp.comlab.ox.ac.uk/pub/Zforum/zstandard1.0.ps.Z>;  
<ftp://ftp.comlab.ox.ac.uk/pub/Zforum/zstandard-annex1.0.ps.Z>
- Brooks, F. P. (1975). *The Mythical Man-Month: Essays on Software Engineering*, Addison-Wesley, Reading, Massachusetts.
- Brown, M. H. (1992). Zeus: A system for algorithm animation and multi-view editing, *Research Report 75*, Digital Equipment Corporation, Systems Research Center, Palo Alto, California.  
**URL:** <ftp://ftp.digital.com/pub/DEC/SRC/research-reports/SRC-075.pdf>
- Campbell, D. (1992). Entity-relationship modeling: One style suits all?, *DATABASE* **23**(3): 12–18.
- Cattell, R. (1991). *Object Data Management*, Addison-Wesley, Reading, Massachusetts.
- Cattell, R., Barry, D. K. and Bartels, D. (1997). *The Object Database Standard: ODMG 2.0*, Morgan Kaufmann Series in Data Management Systems, Morgan Kaufmann, Los Altos, California.

- CDIF Technical Committee (1994a). CASE Data Interchange Format — Overview, *EIA Interim Standard EIA/IS-106*, Electronic Industries Association, Arlington, Virginia.
- CDIF Technical Committee (1994b). CDIF framework for modeling and extensibility, *EIA Interim Standard EIA/IS-107*, Electronic Industries Association, Arlington, Virginia.
- CDIF Technical Committee (1994c). CDIF integrated meta-model: Foundation subject area, *EIA Interim Standard EIA/IS-111*, Electronic Industries Association, Arlington, Virginia.
- CDIF Technical Committee (1995a). CDIF integrated meta-model: Common subject area, *EIA Interim Standard EIA/IS-112*, Electronic Industries Association, Arlington, Virginia.
- CDIF Technical Committee (1995b). CDIF integrated meta-model: Data flow model subject area, *EIA Interim Standard EIA/IS-115*, Electronic Industries Association, Arlington, Virginia.
- CDIF Technical Committee (1996a). CDIF integrated meta-model: Data modeling subject area, *EIA Interim Standard EIA/IS-114*, Electronic Industries Association, Arlington, Virginia.
- CDIF Technical Committee (1996b). CDIF integrated meta-model: State/event model subject area, *CDIF Draft STEV-V8*, Electronic Industries Association, Arlington, Virginia.
- Checkland, P. (1981). *Systems Thinking, Systems Practice*, John Wiley & Sons, Chichester, England.
- Chen, P. P.-S. (1976). The entity-relationship model — Toward a unified view of data, *ACM Transactions on Database Systems* 1(1).
- Chen, P. P.-S. (1977). *The Entity-Relationship Approach to Logical Database Design*, number 6 in *The Q.E.D. Monograph Series*, Q.E.D. Information Sciences, Inc., Wellesley, Massachusetts.
- Chomsky, N. (1978). *Syntactic Structures*, Peter Lang Publishing.

- Cimikowski, R. and Shope, P. (1996). A neural-network algorithm for a graph layout problem, *IEEE Transactions on Neural Networks* 7(2): 341–345.
- Clapham, C. (1990). *The Concise Oxford Dictionary of Mathematics*, Oxford University Press, Oxford, UK.
- Clark, S. (1992). Transformr: A prototype STEP exchange file migration tool, *National PDES Testbed Report Series NISTIR 4944*, U.S. Department of Commerce, National Institute of Standards and Technology, Washington, D.C.
- Codd, E. (1970). A relational model of data for large shared data banks, *Communications of the ACM* 13(6).
- Codd, E. (1972a). Further normalization of the data base relational model, in R. Rustin (ed.), *Data Base Systems*, Courant Computer Science Symposia Series 6, Prentice-Hall, Englewood Cliffs, New Jersey, pp. 33–64.
- Codd, E. (1972b). Relational completeness of data base sublanguages, in R. Rustin (ed.), *Data Base Systems*, Courant Computer Science Symposia Series 6, Prentice-Hall, Englewood Cliffs, New Jersey, pp. 65–98.
- Codd, E. (1979). Extending the database relational model to capture more meaning, *ACM Transactions on Database Systems* 4(4).
- Codd, E. (1988a). Fatal flaws in SQL, part one, *Datamation* (August 15): 45–48.
- Codd, E. (1988b). Fatal flaws in SQL, part two, *Datamation* (September 1): 71–74.
- Codd, E. (1990). *The Relational Model for Database Management Version 2*, Addison-Wesley, Reading, Massachusetts.
- Coleman, M. and Parker, D. (1996). Aesthetics-based graph layout for human consumption, *Software — Practice and Experience* 26(12): 1415–1438.
- Cooper, R. (1991). Configurable data modelling systems, in H. Kangassalo (ed.), *Ninth International Conference on the Entity-Relationship Approach*, Elsevier Science Publishing Company, Lausanne, Switzerland, pp. 57–74.

- Darke, P. and Shanks, G. (1994). Viewpoint developments for requirements definition: An analysis of concepts, issues and approaches, *Working Paper 21/94*, Department of Information Systems, Monash University, Melbourne, Australia.
- Darke, P. and Shanks, G. (1995a). Understanding stakeholder viewpoints in requirements definition: A framework for viewpoint development, *Working Paper 37/95*, Department of Information Systems, Monash University, Melbourne, Australia.
- Darke, P. and Shanks, G. (1995b). Viewpoint development for requirements definition: Towards a conceptual framework, *Sixth Australasian Conference on Information Systems (ACIS'95)*, Perth, Australia, pp. 277–288.
- Darke, P. and Shanks, G. (1996). Stakeholder viewpoints in requirements definition: A framework for understanding viewpoint development approaches, *Requirements Engineering* 1: 88–105.
- Darke, P. and Shanks, G. (1997a). Managing user viewpoints in requirements definition, *Eighth Australasian Conference on Information Systems (ACIS '97)*, Adelaide, Australia, pp. 495–508.
- Darke, P. and Shanks, G. (1997b). User viewpoint modelling: Understanding and representing user viewpoints during requirements definition, *Information Systems Journal* 7: 213–239.
- Date, C. (1990a). EXISTS is not “exists”! (some logical flaws in SQL), in C. Date (ed.), *Relational Database Writings, 1985–1989*, Addison-Wesley, Reading, Massachusetts, chapter 13, pp. 339–356.
- Date, C. (1990b). What’s wrong with SQL?, in C. Date (ed.), *Relational Database Writings, 1985–1989*, Addison-Wesley, Reading, Massachusetts, chapter 12, pp. 325–337.
- Date, C. (1995). *An Introduction to Database Systems*, sixth edn, Addison-Wesley, Reading, Massachusetts.
- Date, C. and Darwen, H. (1993). *A Guide to the SQL Standard*, third edn, Addison-Wesley, Reading, Massachusetts.

- DiBattista, G., Garg, A., Liotta, G., Tamassia, R., Tassinari, E. and Vargiu, F. (1997). An experimental comparison of four graph drawing algorithms, *Computational Geometry — Theory and Applications* 7(5–6): 202–325.
- Diestel, R. (1997). *Graph Theory*, Graduate Texts in Mathematics, Springer-Verlag, New York.
- Duke, R., King, P., Rose, G. and Smith, G. (1991). The Object-Z specification language version 1, *Technical report TR 91-1*, Software Verification Research Centre, University of Queensland, Brisbane, Australia.
- Easterbrook, S. M. (1991a). *Elicitation of Requirements from Multiple Perspectives*, PhD thesis, Imperial College of Science Technology and Medicine, University of London, London.  
**URL:** <http://www.csee.wvu.edu/~easterbr/papers/1991/thesis.pdf>
- Easterbrook, S. M. (1991b). Handling conflict between domain descriptions with computer supported negotiation, *Knowledge Acquisition: An International Journal* 3(4): 255–289.  
**URL:** <ftp://ftp.cogs.susx.ac.uk/pub/reports/csrp/csrp202.ps.Z>
- Easterbrook, S. M., Finkelstein, A. C. W., Kramer, J. and Nuseibeh, B. A. (1994). Coordinating distributed ViewPoints: The anatomy of a consistency check, *Journal of Concurrent Engineering: Research and Applications* 2(3).  
**URL:** <ftp://ftp.cogs.susx.ac.uk/pub/reports/csrp/csrp333.ps.Z>;  
<ftp://dse.doc.ic.ac.uk/dse-papers/viewpoints/cera.ps.Z>;  
<http://www.csee.wvu.edu/~easterbr/papers/1994/csrp333.pdf>
- Easterbrook, S. M. and Nuseibeh, B. A. (1995). Managing inconsistencies in an evolving specification, *Second IEEE International Symposium on Requirements Engineering (RE'95)*, York, UK, pp. 48–55.  
**URL:** <ftp://ftp.cogs.susx.ac.uk/pub/reports/csrp/csrp358.ps.Z>;  
<ftp://dse.doc.ic.ac.uk/dse-papers/viewpoints/re95.ps.Z>;  
<http://www.csee.wvu.edu/~easterbr/papers/1995/csrp358.pdf>

- Easterbrook, S. M. and Nuseibeh, B. A. (1996). Using ViewPoints for inconsistency management, *Software Engineering Journal* **11**(1): 31–43.  
**URL:** <http://www.csee.wvu.edu/~easterbr/papers/1996/NASA-IVV-95-002.pdf>;  
<ftp://dse.doc.ic.ac.uk/dse-papers/viewpoints/sej95.ps.gz>
- Eastman, C., Jeng, T.-S., Assal, H., Cho, M. and Chase, S. (1995). *EDM-2 Reference Manual*, Center for Design and Communication, UCLA, Los Angeles, USA.
- Elmasri, R. and Navathe, S. B. (1994). *Fundamentals of Database Systems*, second edn, Benjamin/Cummings, Redwood City, California.
- Ernst, J. (1997). Introduction to CDIF, *Technical report*, Electronic Industries Association, CDIF Technical Group.  
**URL:** <http://www.eigroup.org/cdif/intro.html>
- Evergreen Software Tools (1995a). *EasyCASE<sup>®</sup> Database Engineer<sup>TM</sup> User's Guide*, v4.2, Evergreen Software Tools, Inc., Redmond, Washington.
- Evergreen Software Tools (1995b). *EasyCASE<sup>®</sup> Methodology Guide*, v4.2, Evergreen Software Tools, Inc., Redmond, Washington.
- Evergreen Software Tools (1995c). *EasyCASE<sup>®</sup> User's Guide*, v4.2, Evergreen Software Tools, Inc., Redmond, Washington.
- Finkelstein, A., Goedicke, M., Kramer, J. and Niskier, C. (1989). ViewPoint oriented software development: Methods and viewpoints in requirements engineering, in J. Bergstra and L. Feijs (eds), *Second Meteor Workshop on Methods for Formal Specification*, Vol. 490 of *Lecture Notes in Computer Science*, Springer-Verlag, Mierlo, The Netherlands, pp. 29–54.
- Finkelstein, A. and Sommerville, I. (1996). The viewpoints FAQ, *Software Engineering Journal* **11**(1): 2–4.  
**URL:** <ftp://cs.ucl.ac.uk/acwf/papers/viewfaq.ps.gz>
- Flanagan, D. (1997). *Java in a Nutshell*, The Java Series, second edn, O'Reilly, Sebastopol, California.



- Fosnight, E. and van Roessel, J. (1985). Vector data interfacing at the EROS Data Center; RIM to ARC/INFO and related interfaces, *Technical report*, EROS Data Center, Sioux Falls, South Dakota.
- Gallaire, H. and Minker, J. (1978). *Logic and Data Bases*, Plenum, New York.
- Gane, C. and Sarson, T. (1979). *Structured Systems Analysis: Tools and Techniques*, Prentice-Hall Software Series, Prentice-Hall, Englewood Cliffs, New Jersey.
- Gawkowski, J. A. and Mamrak, S. (1992). Toward a universal framework for data translation, *Technical report OSU-CISRC-11/92-TR31*, Computer and Information Science Research Center, The Ohio State University, Columbus, Ohio.  
**URL:** [〈ftp://ftp.cis.ohio-state.edu/pub/tech-report/1992/TR31.ps.gz〉](ftp://ftp.cis.ohio-state.edu/pub/tech-report/1992/TR31.ps.gz)
- Genesereth, M. and Fikes, R. (1992). *Knowledge Interchange Format Version 3.0 Reference Manual*, Computer Science Department, Stanford University, Stanford, California.
- Gosling, J. and McGilton, H. (1996). The Java language environment, *White paper*, Sun Microsystems, Inc., Palo Alto, California.  
**URL:** [〈ftp://ftp.javasoft.com/docs/papers/langenviron-pdf.zip〉](ftp://ftp.javasoft.com/docs/papers/langenviron-pdf.zip)
- Greenspan, S., Mylopoulos, J. and Borgida, A. (1994). On formal requirements modeling languages: RML revisited, in B. Fadini (ed.), *Sixteenth International Conference on Software Engineering*, IEEE Computer Society Press, Sorrento, Italy, pp. 135–148.
- Groff, J. R. and Weinberg, P. N. (1994). *LAN Times Guide to SQL*, Osborne McGraw-Hill, Berkeley, California.
- Grundy, J. C. (1993). *Multiple Textual and Graphical Views for Interactive Software Development Environments*, PhD thesis, Department of Computer Science, University of Auckland, Auckland, New Zealand.  
**URL:** [〈http://www.cs.auckland.ac.nz/~john-g/papers/MViews\\_thesis.ps.gz〉](http://www.cs.auckland.ac.nz/~john-g/papers/MViews_thesis.ps.gz)
- Grundy, J. C. (1998). Personal communication.
- Grundy, J. C. and Hosking, J. G. (1993a). Constructing multi-view editing environments using MViews, *1993 IEEE Symposium on Visual Languages*, IEEE CS Press,

Bergen, Norway, pp. 220–224.

**URL:** <http://www.cs.auckland.ac.nz/~john-g/papers/vl93.ps.gz>

Grundy, J. C. and Hosking, J. G. (1993b). The MViews framework for constructing multi-view editing environments, *New Zealand Journal of Computing* 4(2): 31–40.

**URL:** <http://www.cs.auckland.ac.nz/~john-g/papers/nzjc93.ps.gz>

Grundy, J. C. and Hosking, J. G. (1994). Constructing integrated software development environments with dependency graphs, *Working Paper 94/4*, Department of Computer Science, University of Waikato, Hamilton, New Zealand.

**URL:** <http://www.cs.auckland.ac.nz/~john-g/papers/MViews.ps.gz>

Grundy, J. C. and Hosking, J. G. (1995). Software environment support for integrated formal program specification and development, *1995 Asia-Pacific Software Engineering Conference (APSEC'95)*, IEEE CS Press, Brisbane, Australia, pp. 264–273.

**URL:** <http://www.cs.auckland.ac.nz/~john-g/papers/apsec95.ps.gz>

Grundy, J. C. and Hosking, J. G. (1996). Keeping free-edited textual and graphical views of information consistent, *Working Paper 96/4*, Department of Computer Science, University of Waikato, Hamilton, New Zealand.

**URL:** <http://www.cs.auckland.ac.nz/~john-g/papers/inconsistency.ps.gz>

Grundy, J. C. and Hosking, J. G. (1997). Constructing integrated software development environments with MViews, *International Journal of Applied Software Technology* 2(3/4): 133–160.

Grundy, J. C., Hosking, J. G. and Mugridge, W. B. (1996). Supporting flexible consistency management via discrete change description propagation, *Software — Practice and Experience* 26(9): 1053–1083.

**URL:** <http://www.cs.auckland.ac.nz/~john-g/papers/spe96.ps.gz>

Grundy, J. C., Mugridge, W. B. and Hosking, J. G. (1997a). Utilising past event histories in a process-centred software engineering environment, *1997 Australian Software Engineering Conference*, IEEE CS Press, Sydney, Australia, pp. 127–136.

**URL:** <http://www.cs.auckland.ac.nz/~john-g/papers/aswec97.ps.gz>

- Grundy, J. C., Mugridge, W. B. and Hosking, J. G. (1997b). A visual, Java-based componentware environment for constructing multi-view editing systems, *Second Component Users' Conference (CUC'97)*, SIGS Books, Munich.  
**URL:** <http://www.cs.auckland.ac.nz/~john-g/papers/cuc97.ps.gz>
- Grundy, J. C. and Venable, J. R. (1994). Providing integrated support for multiple development notations, *Working paper 94/17*, Department of Computer Science, University of Waikato, Hamilton, New Zealand.
- Grundy, J. C. and Venable, J. R. (1995a). Developing CASE tools which support integrated development notations, *Sixth Workshop on the Next Generation of CASE Tools (NGCT'95)*, Finland.  
**URL:** <http://www.cs.auckland.ac.nz/~john-g/papers/ngct95.ps.gz>
- Grundy, J. C. and Venable, J. R. (1995b). Providing integrated support for multiple development notations, *Seventh Conference on Advanced Information Systems Engineering (CAiSE'95)*, Vol. 932 of *Lecture Notes in Computer Science*, Springer-Verlag, Finland, pp. 255–268.  
**URL:** <http://www.cs.auckland.ac.nz/~john-g/papers/caise95.ps.gz>
- Grundy, J. C. and Venable, J. R. (1996). Towards an integrated environment for method engineering, *Method Engineering '96: IFIP WG 8.1/8.2 Working Conference on Principles of Method Construction and Tool Support*, McGraw-Hill, Atlanta.  
**URL:** <http://www.cs.auckland.ac.nz/~john-g/papers/me96.ps.gz>
- Hammer, M. and McLeod, D. (1981). Database description with SDM: A semantic database model, *ACM Transactions on Database Systems* **6**(3): 351–386.
- Holtzman, S. R. (1994). *Digital Mantras: The Languages of Abstract and Virtual Worlds*, MIT Press, Cambridge, Massachusetts.
- Hosking, J. G. and Grundy, J. C. (1995). Using change descriptions to maintain consistency across multiple representations, *Technical Report 109*, Department of Computer Science, University of Auckland, Auckland, New Zealand.  
**URL:** <http://www.cs.auckland.ac.nz/~techrep/TR109/doc.ps>

- Hosking, J. G., Mugridge, W., Amor, R. and Grundy, J. (1995). Keeping things consistent, *New Zealand Journal of Computing* **6**(1): 353–362.  
**URL:** <http://www.cs.auckland.ac.nz/~john-g/papers/nzcs95.ps.gz>
- Hsia, P., Samuel, J., Gao, J., Kung, D., Toyoshima, Y. and Chen, C. (1994). Formal approach to scenario analysis, *IEEE Software* **11**(2): 33–41.
- Hull, R. (1986). Relative information capacity of simple relational database schemata, *SIAM Journal on Computing* **15**(3): 856–886.
- Hull, R. and King, R. (1987). Semantic database modeling: Survey, applications, and research issues, *ACM Computing Surveys* **19**(3): 201–260.
- Informix Software (1996). Developing DataBlade<sup>®</sup> modules for INFORMIX<sup>®</sup>-Universal Server, *White paper*, Informix Software Corp., Menlo Park, California.  
**URL:** <http://www.informix.com/informix/whitepapers/databld/databld.htm>
- ISO-IEC (1987). Database Language SQL, *International Standard ISO/IEC 9075:1987*, International Organisation for Standardisation (ISO), Geneva, Switzerland.
- ISO-IEC (1989). Database Language SQL, *International Standard ISO/IEC 9075:1989*, International Organisation for Standardisation (ISO), Geneva, Switzerland.
- ISO-IEC (1992a). Database Language SQL, *International Standard ISO/IEC 9075:1992*, International Organisation for Standardisation (ISO), Geneva, Switzerland.
- ISO-IEC (1992b). Industrial automation systems and integration — Produce data representation and exchange, part II: The EXPRESS language reference manual, *Draft International Standard ISO DIS 10313-11*, International Organisation for Standardisation (ISO), Geneva, Switzerland.
- Jacobs, D. and Marlin, C. (1995). Software process representation to support multiple views, *International Journal of Software Engineering and Knowledge Engineering* **5**(4).
- Jacobson, I., Christerson, M., Jonsson, P. and Övergaard, G. (1992). *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, Reading, Massachusetts.

- Kahn, B. (1979). *A Structured Logical Data Base Design Methodology*, PhD thesis, Computer Science Department, University of Michigan, Ann Arbor, Michigan.
- Keeler, M. (1996). Database of all trades, *Database Programming & Design* **9**(11): 34–42.
- Klein, H. and Hirschheim, R. (1987). A comparative framework of data modelling paradigms and approaches, *The Computer Journal* **30**(1): 8–15.
- Kotonya, G. and Sommerville, I. (1996). Requirements engineering with viewpoints, *Software Engineering Journal* **11**(1): 5–18.
- Leite, J. and Freeman, P. A. (1991). Requirements validation through viewpoint resolution, *IEEE Transactions on Software Engineering* **17**(12): 1253–1269.
- Liu, S., Offutt, A. J., Ho-Stuart, C., Yong Sun and Ohba, M. (1998). SOFL: A formal engineering methodology for industrial applications, *IEEE Transactions on Software Engineering* **24**(1): 24–45.
- Lockhart, T. (1999). *PostgreSQL Programmer's Guide*, Postgres Global Development Group.  
**URL:** <http://www.postgresql.org/docs/programmer/index.html>
- Mamrak, S. A. and Barnes, J. (1994). Comparing tools and techniques for data translation, *Technical report OSU-CISRC-2/94-TR8*, Computer and Information Science Research Center, The Ohio State University, Columbus, Ohio.  
**URL:** <ftp://ftp.cis.ohio-state.edu/pub/tech-report/1994/TR08-DIR/>
- Martin, J. (1990). *Information Engineering, Book II: Planning and Analysis*, revised edn, Prentice-Hall, Englewood Cliffs, New Jersey.
- McLean, B., Medeiros, J. and Mount, P. T. (1997). Javapostgres95.  
**URL:** <http://www.retep.org.uk/postgres/JavaPostgres95-0.4.tar.gz>
- MetaCase Consulting (1993). *MetaEdit 1.0 Manual*, MetaCase Consulting Oy, Jyväskylä, Finland.
- Meyers, S. (1991). Difficulties in integrating multiview environments, *IEEE Software* **8**(1): 49–57.

- Miller, R. (1994). *Managing Structural Heterogeneity*, PhD thesis, Department of Computer Sciences, University of Wisconsin-Madison, Madison, Wisconsin.  
**URL:** <http://www.cs.toronto.edu/~miller/papers/dissertation.ps>
- Miller, R., Ioannidis, Y. and Ramakrishnan, R. (1993). The use of information capacity in schema integration and translation, *Nineteenth International Conference on Very Large Data Bases (VLDB)*, Dublin, Ireland, pp. 120–133.  
**URL:** <http://www.cs.toronto.edu/~miller/papers/MIR93b.ps>
- Miller, R., Ioannidis, Y. and Ramakrishnan, R. (1994a). Schema equivalence in heterogeneous systems: Bridging theory and practice, *Information Systems* **19**(1): 3–31.  
**URL:** <http://www.cs.toronto.edu/~miller/papers/MIR94b.ps>
- Miller, R., Ioannidis, Y. and Ramakrishnan, R. (1994b). Schema intension graphs: A formal model for the study of schema equivalence, *Technical report CS-TR-94-1185*, Department of Computer Sciences, University of Wisconsin-Madison, Madison, Wisconsin.  
**URL:** <http://www.cs.toronto.edu/~miller/papers/MIR93c.ps>
- Motro, A. (1987). Superviews: Virtual integration of multiple databases, *IEEE Transactions on Software Engineering* **SE-13**(7): 785–798.
- Muller, P.-A. (1997). *Instant UML*, Wrox Press, Birmingham.
- Mullery, G. (1979). CORE — A method for controlled requirements specification, *Fourth International Conference on Software Engineering*, IEEE Computer Society Press, Munich, Germany, pp. 126–135.
- National Institute of Standards and Technology (1993). Integration definition for information modeling (IDEF1X), *Federal Information Processing Standards Publication 184*, National Institute of Standards and Technology (NIST), Computer Systems Laboratory.  
**URL:** <http://www.sdct.itl.nist.gov/~ftp/idef/idef1x.rtf>
- Navathe, S. B. and Gadgil, S. G. (1982). A methodology for view integration in logical data base design, *Eighth International Conference on Very Large Data Bases*, Morgan Kaufmann, Los Altos, California, Mexico City, Mexico, pp. 142–164.

- Nuseibeh, B., Kramer, J. and Finkelstein, A. (1994). A framework for expressing the relationships between multiple views in requirements specification, *IEEE Transactions on Software Engineering* **20**(10): 760–773.  
**URL:** <ftp://cs.ucl.ac.uk/acwf/papers/tse94.icse.ps.gz>;  
<ftp://dse.doc.ic.ac.uk//dse-papers/viewpoints/tse94.icse.ps.Z>
- Object Design (1999). Welcome to ObjectStore, Web document, Object Design, Inc.  
**URL:** <http://www.odi.com/objectstore/>
- O'Brien, D. (1992). *Deft Editors and Utilities*, v4.2, Sybase, Inc., Emeryville, California.
- Ousterhout, J. K. (1998). Scripting: Higher level programming for the 21st century, *White paper*, Scriptics Corporation, Palo Alto, California.  
**URL:** <http://www.scriptics.com/people/john.ousterhout/scripting.html>
- Pascoe, R. and Penny, J. (1990). Construction of interfaces for the exchange of geographic data, *International Journal of Geographical Information Systems* **4**(2): 147–156.
- Pascoe, R. T. and Penny, J. P. (1995). Constructing interfaces between (and within) geographical information systems, *International Journal of Geographical Information Systems* **9**(3): 275–291.
- Price, C. (1995). Guidelines for implementing VML (view mapping language) multi-schema mapping specifications in C++, *Internal report*, Building Research Association of New Zealand, Inc., Wellington, New Zealand.
- Qian, X. (1995). Correct schema transformations, *Technical report SRI-CSL-95-08*, Computer Science Laboratory, SRI International, Menlo Park, California.  
**URL:** <http://www.csl.sri.com/reports/postscript/sri-csl-95-08.ps.gz>
- Rational Software Corporation (1997). *UML Summary, version 1.1*, Rational Software Corp., Santa Clara, California.
- Reiner, D. (1992). *Database design tools*, Benjamin/Cummings, Redwood City, California, chapter 15, pp. 411–454.
- Reiss, S. P. (1985). PECAN: Program development systems that support multiple views, *IEEE Transactions on Software Engineering* **SE-11**(3): 276–285.

- Reiss, S. P. (1990a). Connecting tools using message passing in the Field environment, *IEEE Software* 7(7): 57–66.
- Reiss, S. P. (1990b). Interacting with the FIELD environment, *Software — Practice and Experience* 20(S1): S1/89–S1/115.
- Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorenson, W. (1991). *Object-oriented Modeling and Design*, Prentice-Hall.
- Sallis, P., Tate, G. and MacDonell, S. (1995). *Software Engineering: Practice, Management, Improvement*, Addison-Wesley, Reading, Massachusetts.
- Serrano, J. A. (1994). User interfaces to data modelling systems, *Internal report*, Department of Computer Science, University of Glasgow, Glasgow, Scotland.  
**URL:** <http://www.dcs.gla.ac.uk/people/personal/serrano/links/UIDMS.PS>
- Shipman, D. (1981). The functional data model and the data language DAPLEX, *ACM Transactions on Database Systems* 6(1): 140–173.
- Smith, H. C. (1985). Database design: Composing fully normalized tables from a rigorous dependency diagram, *Communications of the ACM* 28(8): 826–838.
- Spaccapietra, S. and Parent, C. (1992). ERC+: An object based entity relationship approach, in P. Loucopoulos and R. Zicari (eds), *Conceptual Modelling, Databases and CASE: An Integrated View of Information Systems Development*, John Wiley.  
**URL:** <ftp://lbdsun.epfl.ch/pub/dbcase92.ps.Z>
- Stanger, N. and Pascoe, R. (1997a). Environments for viewpoint representations, in R. Galliers, S. Carlsson, C. Loebbecke, C. Murphy, H. Hansen and R. O’Callaghan (eds), *Fifth European Conference on Information Systems (ECIS’97)*, Vol. I, Cork Publishing, Cork, Ireland, pp. 367–382.
- Stanger, N. and Pascoe, R. (1997b). Exploiting the advantages of object-oriented programming in the implementation of a database design environment, *Information Science Discussion Paper 97/08*, Department of Information Science, University of Otago, Dunedin, New Zealand.  
**URL:** <http://divcom.otago.ac.nz/infosci/publctns/complete/papers/dp9708ns.zip>



- Stanger, N. and Pascoe, R. (1997c). Exploiting the advantages of object-oriented programming in the implementation of a database design environment, *Joint 1997 Asia Pacific Software Engineering Conference and International Computer Science Conference (APSEC'97/ICSC'97)*, IEEE Press, Hong Kong.  
**URL:** <http://divcom.otago.ac.nz/infosci/darc/publications/APSEC97.pdf>
- Su, S. and Fang, S. (1993). A neutral semantic representation for data model and schema translation, *Technical report TR-93-023*, University of Florida, Gainesville, Florida.  
**URL:** <ftp://ftp.cis.ufl.edu/cis/tech-reports/tr93/tr93-023.ps.Z>
- Su, S., Fang, S. and Lam, H. (1992). An object-oriented rule-based approach to data model and schema translation, *Technical report TR-92-015*, University of Florida, Gainesville, Florida.  
**URL:** <ftp://ftp.cis.ufl.edu/cis/tech-reports/tr92/tr92-015.ps.Z>
- Su, S., Krishnamurthy, V. and Lam, H. (1989). An object-oriented semantic association model (OSAM\*), in S. Kumara, A. Soyster and R. Kashyap (eds), *Artificial Intelligence: Manufacturing Theory and Practice*, Industrial Engineering and Management Press, Nocross, Georgia, pp. 463–493.
- Sun Microsystems (1997). Java demonstration applets and applications, Web document, Sun Microsystems, Inc.  
**URL:** <http://www.javasoft.com/products/jdk/1.1/docs/relnotes/demos.html>
- Sun Microsystems (1998a). Java™ platform 1.1.x core API specification, Web document, Sun Microsystems, Inc.  
**URL:** <http://www.javasoft.com/products/jdk/1.1/docs/api/packages.html>
- Sun Microsystems (1998b). The JDBC™ database access API, Web document, Sun Microsystems, Inc.  
**URL:** <http://java.sun.com/products/jdbc/index.html>
- Tamassia, R. (1985). New layout techniques for entity-relationship diagrams, in P. P. Chen (ed.), *Fourth International Conference on the Entity-Relationship Approach*, IEEE Computer Society Press/North Holland, Chicago, Illinois, pp. 304–311.

- Tittel, E. (1998). What's the point of XML?, *SunWorld* 12(2).  
**URL:** <http://www.sun.com/sunworldonline/swol-02-1998/swol-02-xml.html>
- Tsichritzis, D. and Klug, A. (eds) (1978). *The ANSI/SPARC/X3 DBMS framework*, AFIPS Press.
- Tsichritzis, D. and Lochovsky, F. (1982). *Data Models*, Prentice-Hall.
- Venable, J. R. (1993). *CoCoA: A Conceptual Data Modelling Approach for Complex Problem Domains*, PhD thesis, Thomas J. Watson School of Engineering and Applied Science, State University of New York at Binghamton, Binghamton, New York.
- Venable, J. R. and Grundy, J. C. (1995). Integrating and supporting entity relationship and object role models, *Fourteenth OO/ER Conference (OO-ER'95)*, Vol. 1021 of *Lecture Notes in Computer Science*, Springer-Verlag, Brisbane, Australia, pp. 318–328.  
**URL:** <http://www.cs.auckland.ac.nz/~john-g/papers/ooer95.ps.gz>
- Verheijen, G. and van Bekkum, J. (1982). NIAM: An information analysis method, in T. Olle, H. Sol and A. Verrijn-Stuart (eds), *Information Systems Design Methodologies: A Comparative Review*, North-Holland/IFIP, Amsterdam, The Netherlands, pp. 537–590.
- Visible Systems Corporation (1999). Database and application development — products, Web document, Visible Systems Corporation.  
**URL:** <http://www.visible.com/dataapp/daprods.html>

# Glossary

**CDIF** CASE Data Interchange Format. An extensible interchange format for CASE tools defined by the Electronic Industries Association (CDIF Technical Committee, 1994a). It allows the interchange of a wide variety of CASE data, including entity-relationship models, data flow models, state-transition models and project management models. See Sections 2.4.3 on page 30 and 3.4 on page 49.

**CoCoA** COmplex COvering Aggregation. A meta-modelling language defined by Venable (1993) for the purpose of defining representations. See Section 10.2 on page 280.

**Completeness (translation)** A property of translations that indicates whether a translation maps all constructs or properties of constructs from the source representation to the target representation, or only some of them. The completeness of a translation may be specified as either *complete* or *partial*. See Section 4.3.2 on page 75.

**Composition (translation)** A property of translations that indicates whether a translation may be decomposed into sub-translations of the same type. The composition of a translation may be specified as either *atomic* or *composite*. See Section 4.3.3 on page 76. (See also *type*.)

**Consistency (viewpoint)** The descriptions of a viewpoint must be consistent with each other, that is, they must agree in terms of the structure and semantics of the information that they describe. A viewpoint is internally consistent, so descriptions may not include information that disagrees with other descriptions in the viewpoint. Different viewpoints may however be inconsistent with each other.

**Construct** A component of a representation, such as an entity or a data flow. The nature of a construct is defined by its properties. The constructs of a representation

may be divided into technique-level constructs and scheme-level constructs. See Section 3.2.3 on page 46.

**DBMS** DataBase Management System.

**Description** An instantiation of a representation. Informally, a description can be thought of as analogous to a schema. Examples of descriptions include entity-relationship diagrams and data flow diagrams such as those found in Appendix C. See Section 3.2.2 on page 44.

**Description translation** The translation of a description from one representation to another. A description translation comprises a collection of rules that define the mappings between constructs of the two representations. See Section 4.3.1 on page 74.

**DFD** Data Flow Diagram.

**Direction (translation)** A property of a translation that specifies whether it may be applied in only one direction (*unidirectional*), or in both directions (*bidirectional*). See Section 4.3.4 on page 77.

**Element** An instantiation of a construct that forms a component of a description. For example, a Staff entity or the attribute name. See Section 3.2.3 on page 46.

**Element translation** The translation of a collection of one or more elements from one representation to another. Element translations are defined by rules that specify the mappings between construct properties in the source and target representations. See Section 4.3.1 on page 74.

**Enrichment** The process of providing extra information to a translation in order to improve its quality. Enrichment may occur before, during and after a translation. See Section 4.6 on page 91.

**ERD** Entity-Relationship Diagram.

**Exclusion (rule)** If rule  $r_i$  is applied to a collection of source elements, and as a consequence rule  $r_j$  cannot be applied to the same collection because it would produce

a conflicting result, then  $r_i$  is said to *exclude* the use of  $r_j$ . See Section 4.7.1 on page 98.

**Expressive overlap** The overlap in expressive power of two representations. Expressive overlap may fall into one of four categories: *disjoint* (no overlap), *intersecting* (partial overlap), *inclusive* (one representation contained by the other) and *equivalent* (the representations have identical expressive powers). See Section 3.5 on page 55.

**Expressive power** The boundary of what may be expressed using the constructs of a particular representation. In effect, it is the set of all possible descriptions that may be expressed using the representation. See Section 3.5 on page 55. (See also *relative information capacity*.)

**FDD** Functional Dependency Diagram.

**Heuristic** A form of rule that may sometimes produce a semantically inconsistent result. Informally, heuristics may be thought of as ‘rules of thumb’ that *usually* produce the correct result, but may not always do so. Also known as a *heuristic rule*; see Section 4.2 on page 68.

**Information capacity** (See *relative information capacity*.)

**inter\_class** A VML-S structure that is used to define element translations. See Section 7.2 on page 180.

**inter\_view** A VML-S structure that is used to define description translations. See Section 7.2 on page 180.

**Interfacing strategy** The strategy used to define translations between representations. In the individual interfacing strategy, translations are defined between every pair of representations ( $n(n - 1)/2$  translations for  $n$  representations). In the interchange format interfacing strategy, translations are only defined between each representation and a common interchange format ( $2n$  translations for  $n$  representations). See Section 2.4.3 on page 30.

**Invariant** A condition that must hold for the elements being processed by a rule. This condition may be applied to either the source elements, where it acts as a filter; or it may be applied to the target elements, where it acts as a constraint. See Section 4.2 on page 68.

**MDM** Multiple-Data-Model. A multiple representation modelling environment built by Atzeni and Torlone (1997). See Sections 2.2.2 on page 12 and 9.3.3 on page 258.

**MViews** A framework for building environments that support multiple textual and graphical views of an underlying model, developed by Grundy (1993). It has recently been reimplemented in Java (JViews). See Sections 2.2.1 on page 9 and 9.3.2 on page 255.

**ORECOM** Object-oriented Rule-based Extensible COre Model. An interchange format defined by Su and Fang (1993). See Sections 2.2.3 on page 13 and 9.3.4 on page 261.

**Performance (translation)** The number of distinct translation steps required to translate a description from one representation to another. The higher the number of steps, the worse the performance. See Section 2.4.2 on page 29.

**Perspective** An internally-consistent description of a real-world phenomenon from some particular point of view. Real-world phenomena may typically be viewed from several different perspectives simultaneously. See Section 2.3 on page 15. (See also *viewpoint*.)

**Property (construct)** Properties define the nature of constructs. For example, a construct representing a relational attribute might have the properties name, domain and relation, representing the name of the attribute, the domain it is drawn from and the relation in which it participates, respectively. See Section 3.2.3 on page 46.

**Quality (translation)** How well a translation deals with loss of information. In effect, how completely a translation maps the constructs of its source representation onto the constructs of its target representation. See Section 2.4.1 on page 28 and Chapter 8.

**Refinement** The process of combining several input descriptions to produce a single output description. Typically used to produce an implementation schema from a collection of source descriptions. See Section 10.5 on page 289.

**Relative information capacity** The relative information capacity of a schema determines the set of all possible instances of that schema. In effect, it is a measure of the information content of a schema. It is not an absolute measure; rather it provides a basis for comparing the information content of different schemas. See Section 8.2 on page 214. (See also *schema intension graph*.)

**Representation** The combination of a particular modelling technique and scheme in order to describe a viewpoint. In effect, it is a combination of constructs that may be used to describe real-world phenomena. Representations may be divided into informal, semi-formal and formal representations. Examples of representations include the entity-relationship approach + Chen ERD notation (semi-formal) and the relational model + SQL (formal). See Sections 2.3 on page 15 and 3.2.1 on page 42.

**Representation definition** A description of the constructs of a representation and the associations between them. See Section 3.4 on page 49.

**Rule** A mapping between a collection of source constructs/properties and a collection of target constructs/properties. Rules may be divided into technique-level and scheme-level rules. See Section 4.2 on page 68.

**Schema intension graph** A formal representation for describing schemas that may be used to compare the relative information capacities of schemas. See Section 8.2.1 on page 215.

**Scheme** The ‘specialised’ part of a representation, which usually includes the visual appearance of the abstract constructs of the representation. It can be thought of as analogous to a modelling notation. Examples include Martin ERD notation and relational calculus. See Section 3.2.1 on page 42.

**SGML** Standard Generalised Markup Language. A standard markup language for describing the logical structure of documents.

**SIG** (See *schema intension graph*.)

**Smith's method** A method defined by Smith (1985) for deriving a collection of fully normalised relations from a functional dependency diagram. See Appendix B.

**Source construct** A construct of the source representation of a translation. See Section 4.3.4 on page 77.

**SQL** Structured Query Language.

**STD** State/Transition Diagram.

**Subsumption (rule)** A rule  $r_i$  is *subsumed* by another rule  $r_s$  in a particular direction  $d$  if the source constructs of  $r_i$  are a subset ( $\subseteq$ ) of the source constructs of  $r_s$ , both rules may be applied in the direction  $d$ , and the invariants for both rules are not contradictory. See Section 4.7.1 on page 98.

**Subsumption/exclusion graph** A directed graph that describes the subsumptions and exclusions in a collection of rules for a particular translation direction. It may be used during the translation process to determine an appropriate order in which to evaluate rules against elements. See Section 4.7.1 on page 98.

**Swift** A simple prototype multiple representation modelling environment implemented in Java by the author. The name derives from two earlier CASE tools developed in-house at the Department of Information Science at the University of Otago. The first of these, named *Pronto*, was developed under the Mac OS in 1991 and used as a teaching tool for several years. A Windows version of the original tool, named *Rapid*, was implemented in 1994–95 and is still in use within the department. Swift continues the naming tradition. See Chapter 6.

**Target construct** A construct of the target representation of a translation. See Section 4.3.4 on page 77.

**Technique** The 'generic' part of a representation, which defines the basic constructs of the representation. It can be thought of as analogous to a modelling approach. Examples include the entity-relationship approach and dependency theory. See Section 3.2.1 on page 42.



**Trivial translation** A scheme translation that changes only the visual appearance of elements in a description, for example, changing bubbles in a Smith notation FDD into boxes in a Date notation FDD. See Section 4.3.1 on page 74.

**Type (translation)** A property of translations. For description translations, either *technique* (changes both technique and scheme) or *scheme* (changes only the scheme). For element translations, either *rule* or *heuristic*. See Section 4.3.1 on page 74.

**Viewpoint** A formalisation of a perspective. During requirements specification, several overlapping and possibly conflicting viewpoints may be elicited. These must be reconciled in some way in order to build a system. Viewpoints may be divided into developer and user viewpoints. See Section 2.3 on page 15.

**VML** View Mapping Language. A declarative translation specification language developed by Amor (1997). See Section 7.2 on page 180.

**VML-S** View Mapping Language — Swift. An extended version of VML defined by the author. See Chapter 7.



# Appendix A

## Notations and terminology

### A.1 Introduction

In this appendix are summarised the major notations that are used in this thesis, except for the modified Smith's notation for functional dependency diagrams, which is described in Appendix B. The original pre-VML notation for expressing translations is summarised in Section A.4. The detail of the CASE tool survey discussed in Chapter 9 is presented in Section A.5.

### A.2 Martin/EasyCASE ERD notation

Entity-relationship diagrams (ERDs) in this thesis are presented using a variant of Martin's (1990) ERD notation implemented by the EasyCASE CASE tool (Evergreen Software Tools, 1995b, Section 4.2.3). The notation is summarised in Tables A.1 and A.2 on the next page.

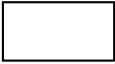
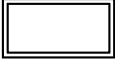



### A.3 Gane & Sarson/EasyCASE DFD notation

Data flow diagrams (DFDs) in this thesis are presented using a variant of Gane and Sarson's (1979) DFD notation implemented by the EasyCASE CASE tool (Evergreen Software Tools, 1995b, Section 3.4). The notation is summarised in Table A.3 on page 337.







### A.4 Original translation notation

The translation operator is denoted by the symbol  $\rightarrow$ . The same operator can be used to denote both *description translations*, that is, translating a description from one repre-

**Table A.1:** Summary of the Martin/EasyCASE ERD notation

| Graphic   | Name               | Description  |
|---|--------------------|--|
|  | Entity             | An entity represents an object or group of objects about which information is to be stored.  |
|  | Weak entity        | A weak entity represents an entity that can only exist when connected to some other entity.  |
|  | Associative entity | An associative entity typically represents a correlation table.  |
| •   | Mutual exclusivity | The mutual exclusivity symbol is used to represent a relationship between one entity and many other entities that are mutually exclusive. It is typically used to connect a supertype to its subtypes. |
|  | Supertype/subtype  | A supertype/subtype relationship links a single supertype entity with one or more subtype entities.  |
|  | Relationship       | A relationship represents an association between two entities. The cardinality of the relationship is denoted by various notations at each end of the relationship line — see Table A.2.               |

**Table A.2:** Martin/EasyCASE notation for relationship cardinalities

| Notation  | Cardinality        |
|---|--------------------|
|  | Unspecified        |
|  | One (1:1)          |
|  | None-or-one (0:1)  |
|  | Many (M:N)         |
|  | None-or-many (0:M) |
|  | One-or-many (1:M)  |

sensation to another, and *construct translations*, that is, translating a construct from one representation to another.



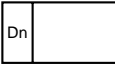
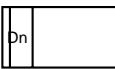
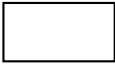





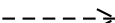
#### A.4.1 Description translations

If both operands of a translation are descriptions, this indicates the translation of a description from one representation to another. The translation of a description  $D_1$  in representation  $\mathfrak{R}_p(T_i, S_j)$  to a description  $D_2$  in representation  $\mathfrak{R}_q(T_k, S_l)$  is specified as:

$$D_1(V, T_i, S_j) \rightarrow D_2(V, T_k, S_l).$$

For example, translating a viewpoint description expressed using SQL into one us-

**Table A.3:** Summary of the Gane & Sarson/EasyCASE DFD notation

| Graphic   | Name                     | Description   |
|---|--------------------------|---|
|    | External entity          | An external entity represents a physical device or other system that is outside the scope of the system being defined, but provides information to and/or receives information from the system. |
|    | Repeated external entity | Represents an external entity that appears more than once in a diagram.   |
|    | Data store               | A data store stores the data flow output by a data process until it is required by some other data process.   |
|    | Repeated data store      | Represents a data store that appears more than once in a diagram.   |
|    | Resource store           | A resource store stores the contents of a resource flow until needed.   |
|    | Data process             | A data process transforms input data flows into output data flows.  |
|   | Multiple data process    | Denotes multiple instances of a process or concurrent processing.   |
|  | Split/merge              | Splits a composite flow into its component flows, merges component flows into composite flows, or allows a single flow to be directed to several targets.                                       |
|  | Interface                | Links a flow to a different DFD.  |
|  | Data flow                | A data flow represents the transfer of data between processes, data stores and external entities.   |
|  | Resource flow            | A resource flow represents the flow of physical resources such as goods, rather than data.  |

ing QUEL is denoted by:

$$D_1(V, \text{Relational}, \text{SQL}) \rightarrow D_2(V, \text{Relational}, \text{QUEL})$$

and translating a Chen-style ERD into a relational schema is denoted by:

$$D_1(V, E-R, \text{ERD}_{\text{Chen}}) \rightarrow D_2(V, \text{Relational}, \text{SQL}).$$

## A.4.2 Constructs

A *construct*  $C$  (of type  $\text{CTYPE}$ ) of representation  $\mathfrak{R}(T_i, S_j)$  is denoted by:

$$\mathfrak{R}(T_i, S_j) [C : \text{CTYPE}]$$

or, if  $\mathfrak{R}$  is clear, simply:

$$\mathfrak{R}[C : \text{CTYPE}].$$

The type CTYPE is analogous to the concept of a relational domain in that it specifies a pool of possible ‘instances’ from which  $C$  may be taken. The ‘:’ notation is used with similar meaning in both domain calculus (Date, 1995, p. 204) and Z (Brien and Nicholls, 1992, p. 6), that is, ‘ $C$  is an element of the set DOM’. The type of a construct is specified according to the constructs defined in the representation definition (see Appendix D).

Some examples of constructs are shown in Table A.4. Note that as shown in the last example,  $C$  may in general specify a set of constructs.

**Table A.4:** Examples of constructs

|  |  |
|--|--|
| $\mathfrak{R}_e(E\text{-}R, ERD_{Martin}) [E : \text{ENTITY}]$             | denotes an entity in a Martin notation E-R diagram                                   |
| $\mathfrak{R}_f(FuncDep, FDD_{Smith}) [s : \text{SVD}]$                    | denotes a single-valued dependency in a Smith notation functional dependency diagram |
| $\mathfrak{R}_r(Relational, SQL/92) [\{a_1, \dots, a_n\} : \text{COLUMN}]$ | denotes a collection of columns in an SQL/92 schema                                  |

### A.4.3 Construct translations

If both operands of a translation are constructs, this indicates the translation of a construct from one representation to another via a rule. Let  $\mathfrak{R}_s(T_i, S_j)$  and  $\mathfrak{R}_t(T_k, S_l)$  be two representations. The translation of construct  $C_1$  of  $\mathfrak{R}_s$  into construct  $C_2$  of  $\mathfrak{R}_t$  is denoted by:

$$\mathfrak{R}_s [C_1 : \text{STYPE}] \rightarrow \mathfrak{R}_t [C_2 : \text{TTYPE}]$$

where STYPE and TTYPE are construct types of  $\mathfrak{R}_s$  and  $\mathfrak{R}_t$  respectively. For example:

$$\mathfrak{R}_e(E\text{-}R, ERD_{Martin}) [m_k : \text{ATTRIBUTE}] \rightarrow \mathfrak{R}_f(FuncDep, FDD_{Smith}) [s_k : \text{ATTRIBUTE}]$$

denotes the translation of a Martin ERD attribute  $m_k$  into a Smith FDD attribute  $s_k$ .

Most construct translations are not this simple, however. In general, a construct translation will have a set of *preconditions* that apply to the source construct(s), and a set of *postconditions* that apply to the target construct(s). Some way is therefore required of specifying a set of axioms that a construct must fulfill. Axioms for a construct are denoted as follows:

$$\mathfrak{R}[C : \text{TYPE} \mid \{\text{axioms}\}]$$

A full construct translation with pre- and postconditions is thus denoted by:

$$\mathfrak{R}_s[C_s : \text{STYPE} \mid \{\text{preconditions}\}] \rightarrow \mathfrak{R}_t[C_t : \text{TTYPE} \mid \{\text{postconditions}\}]$$

For example, given  $\mathfrak{R}_e(E-R, ERD_{Martin})$  and  $\mathfrak{R}_r(Relational, SQL/92)$ , then:

$$\begin{aligned} \mathfrak{R}_e[a_k : \text{ATTRIBUTE} \mid \{(\exists \mathfrak{R}_e[E_k : \text{ENTITY}])(\mathfrak{R}_e[a_k] \in \mathfrak{R}_e[E_k])\}] \rightarrow \\ \mathfrak{R}_r[c_k : \text{COLUMN} \mid \{(\exists \mathfrak{R}_r[T_k : \text{TABLE}])(\mathfrak{R}_r[c_k] \in \mathfrak{R}_r[T_k])\}] \end{aligned}$$

denotes the translation of an attribute  $a_k$  of Martin ERD entity  $E_k$  into a column  $c_k$  of SQL/92 table  $T_k$ .

#### A.4.4 Heuristic construct translations

In order to distinguish clearly between rule-based construct translations and heuristic-based construct translations, the operator  $\dashrightarrow$  will be used to denote a heuristic construct translation. The remainder of the notation is unchanged.

#### A.4.5 Composite and atomic translations

All the examples of translations shown so far have been *atomic*, that is, they cannot be decomposed into a set of smaller translations of the same type (description or construct). If a translation can be decomposed into a collection of translations of the same type, then it is known as a *composite* translation, and is denoted by placing a small circle ( $\circ$ , representing the mathematical composition operator) over the translation operator, that is,  $\overset{\circ}{\rightarrow}$  or  $\overset{\circ}{\dashrightarrow}$ . Technically, all description translations could be considered a composite of several construct translations, but these were considered to be composite

translations, it would not then be possible to specify a series of description translations as a composite translation. That is,  $D_1 \rightarrow D_2 \rightarrow D_3$  could not be rewritten as  $D_1 \overset{\circ}{\rightarrow} D_3$ . Sub-translations are therefore limited to being of the same type as the super-translation.

## A.5 CASE tool survey

In this section, five ‘conventional’ CASE tools are surveyed to determine to the extent to which they facilitate the use of multiple representations and their support for representation translations. The tools surveyed were: Visible Systems’ EasyCASE, Visible Analyst and EasyER/EasyOBJECT; Sybase’s Deft; and MetaCase’s MetaEdit. EasyCASE and Deft were chosen because full versions were readily accessible; the remaining tools were chosen because comprehensive demonstration versions were available for download from the Internet.

The survey takes the following form:

1. A brief description of the tool will be given;
2. A list of the representations supported by the tool will be presented;
3. Any ability to facilitate the use of multiple representations will be identified; and
4. Any ability to translate descriptions from one representation to another will be identified.

### A.5.1 Visible Systems EasyCASE

EasyCASE (Visible Systems Corporation, 1999) is a complete CASE environment originally developed by Evergreen Software Tools, who merged with Visible Systems Corporation in 1997. EasyCASE is based around two main tools, which share a common data dictionary. The EasyCASE application itself (Evergreen Software Tools, 1995c) provides diagramming, high-level data dictionary access and reporting tools; while the *Database Engineer* (Evergreen Software Tools, 1995a) provides detailed manipulation of the data dictionary and control over the schema generation process.



EasyCASE supports a wide range of representations (Evergreen Software Tools, 1995b), as shown in Table A.5 on the next page. The common data dictionary allows different descriptions (using different representations) to share information, so EasyCASE partially facilitates the use of multiple representations.

EasyCASE supports many translations between representations:

- any supported E-R  $\leftrightarrow$  any supported relational (31 technique translations);
- any DFD  $\leftrightarrow$  any DFD (12 scheme translations); and
- some E-R  $\leftrightarrow$  some E-R (17 scheme translations).

As can be seen from the above, the only technique translations supported by EasyCASE are those between E-R and SQL representations, and the differences between these translations are small.

The scheme translations supported by EasyCASE are all trivial, as they can only be applied when the two representations have equivalent expressive powers, and result only in a change of notation. Scheme translations are effected by changing the notation type (scheme) of a description in the data dictionary. If the two representations do not have equivalent expressive powers, this can result in an error upon returning to the diagramming tool, but in some cases, source constructs that have no equivalent in the target representation are retained, resulting in a syntactically invalid description. This is particularly apparent with the E-R representations, where only seventeen translations out of a possible forty-two actually work. Some translations could be performed in one direction but not the other, despite the fact that all of the ERD schemes overlap sufficiently to perform translations in both directions. These are all significant indications that no actual translation is taking place.

## **A.5.2 Visible Systems Visible Analyst**

Visible Analyst (Visible Systems Corporation, 1999) is a CASE environment developed by Visible Systems Corporation. It comprises a single application that provides support for several representations (see Table A.6 on page 343). Although it has a data dictionary, it is not as integrated as the data dictionary in EasyCASE, as information cannot be shared across descriptions that are expressed using different representations.

**Table A.5: Representations supported by EasyCASE**

| Technique               | Scheme                                     | Notation                                  |
|-------------------------|--|---|
| Entity-relationship     | Chen ERD                                   | $\mathfrak{R}(E-R, ERD_{Chen})$           |
|                         | Martin ERD                                 | $\mathfrak{R}(E-R, ERD_{Martin})$         |
|                         | Bachman ERD                                | $\mathfrak{R}(E-R, ERD_{Bachman})$        |
|                         | Shlaer-Mellor ERD                          | $\mathfrak{R}(E-R, ERD_{ShlaerMellor})$   |
|                         | IDEF1X ERD                                 | $\mathfrak{R}(E-R, ERD_{IDEF1X})$         |
|                         | Merise ERD                                 | $\mathfrak{R}(E-R, ERD_{Merise})$         |
|                         | SSADM logical data structure diagram (LDS) | $\mathfrak{R}(E-R, LDS_{SSADM})$          |
|                         | Metrica 2 ERD                              | $\mathfrak{R}(E-R, ERD_{Metrica2})$       |
| Functional dependencies | Martin data model diagram (DMD)            | $\mathfrak{R}(FuncDep, DMD_{Martin})^a$   |
|                         | Bachman DMD                                | $\mathfrak{R}(FuncDep, DMD_{Bachman})^a$  |
| Data flow modelling     | Yourdon/DeMarco DFD                        | $\mathfrak{R}(DataFlow, DFD_{Y/DM})$      |
|                         | Gane & Sarson DFD                          | $\mathfrak{R}(DataFlow, DFD_{G\&S})$      |
|                         | Metrica 2 DFD                              | $\mathfrak{R}(DataFlow, DFD_{Metrica2})$  |
|                         | SSADM DFD                                  | $\mathfrak{R}(DataFlow, DFD_{SSADM})$     |
|                         | Yourdon/DeMarco transformation graph (TRG) | $\mathfrak{R}(DataFlow, TRG_{Metrica2})$  |
|                         | Gane & Sarson TRG                          | $\mathfrak{R}(DataFlow, TRG_{G\&S})$      |
| State-Event             | State transition diagram (STD)             | $\mathfrak{R}(StateEvent, STD)$           |
| Relational              | SQL (31 dialects) <sup>b</sup>             | $\mathfrak{R}(Relational, SQL)$           |
|                         | Microsoft Access <sup>c</sup>              | $\mathfrak{R}(Relational, MSAccess)$      |
| Data structure          | Data structure diagram (DSD)               | $\mathfrak{R}(DataStruct, DSD)$           |
| Structured design       | Yourdon/Constantine structure chart (STC)  | $\mathfrak{R}(Structured, STC_{Y/C})$     |
|                         | SSADM entity life history diagram (ELH)    | $\mathfrak{R}(Structured, ELH_{SSADM})^d$ |

**Notes on Table A.5:**

- <sup>a</sup> Data model diagrams are very similar to functional dependency diagrams (FDDs), although functional dependencies are not explicitly mentioned.
- <sup>b</sup> Strictly speaking, each dialect of SQL is a separate scheme, but they have been grouped here because of the sheer number of dialects supported. EasyCASE supports the following SQL dialects: ANSI SQL (probably SQL/89), ANSI SQL/2 (probably SQL/92), Microsoft Access (via ODBC), DB2, dBASE III Plus, dBASE IV, dBASE V, FoxPro 2.x, Informix 5/6, Ingres, InterBase 4, Netware SQL, Oracle 6/7, OS2 Extended Edition, Paradox 3/4.x/5, Progress 6/7, R:BASE SQL, Rdb/VMS, SQLBase 5.x/6, SQL Server, Sybase, Watcom SQL 3.2/4.0 and XDB 3/4.0.
- <sup>c</sup> Supports versions 1.1 and 2.0 via direct connection using the JET database engine.
- <sup>d</sup> Entity life history diagrams also appear to be at least partially linked to state-event modelling.

**Table A.6: Representations supported by Visible Analyst**

| Technique             | Scheme  | Notation  |
|-----------------------|---|---|
| Entity-relationship   | Unknown <sup>a</sup>  | $\mathfrak{R}(E-R, ERD_{VAW})$  |
| Data flow modelling   | Gane & Sarson DFD<br>Yourdon/DeMarco DFD                                      | $\mathfrak{R}(DataFlow, DFD_{G\&S})$<br>$\mathfrak{R}(DataFlow, DFD_{Y/DM})$    |
| Relational            | SQL (18 dialects) <sup>b</sup>  | $\mathfrak{R}(Relational, SQL)$   |
| Object modelling      | Class diagrams (OCD)  | $\mathfrak{R}(Object, OCD)$   |
| State-Event           | State transition diagram (STD)  | $\mathfrak{R}(StateEvent, STD)$   |
| Structured design     | Yourdon/Constantine structure chart (STC)<br>Functional decomposition diagram | $\mathfrak{R}(Structured, STC_{Y/C})$<br>$\mathfrak{R}(Structured, FuncDecomp)$ |
| HLL <sup>c</sup> code | COBOL<br>C  | $\mathfrak{R}(HLL, COBOL)$<br>$\mathfrak{R}(HLL, C)$                            |

**Notes on Table A.6:**

- <sup>a</sup> This ERD notation, while similar to other notations, appears to be peculiar to Visible Analyst.
- <sup>b</sup> The SQL dialects supported by Visible Analyst are: ANSI SQL/92, Microsoft Access, Centura SQL Base 5.0, DB2 2.3, Datacom:DB SQL, Informix-SQL, Ingres/SQL, Netware SQL 3.0, Oracle SQL 7.0, Rdb, SQL Server 4.X, SQL Server 6.X, SQL Server System 10, Teradata SQL 4.1.3, Unify SQL, Watcom SQL, XDB, dBase IV SQL 1.1 and a user-defined dialect.
- <sup>c</sup> 'HLL' = high-level language, that is, typical programming languages such as C++ or Pascal.

Visible Analyst only supports technique translations between E-R and relational representations (all bidirectional). It also supports the generation of C or COBOL code from a collection of source descriptions. No scheme translations are supported, which is expected given the lack of integration across descriptions in the data dictionary.

### A.5.3 Visible Systems EasyER/EasyOBJECT

EasyER/EasyOBJECT (Visible Systems Corporation, 1999) is a high-end CASE environment produced by Visible Systems Corporation. It comprises a single tool with an integrated data dictionary, and supports the set of representations shown in Table A.7 on the next page. The data dictionary works similarly to that in EasyCASE, so Ea-

**Table A.7: Representations supported by EasyER/EasyOBJECT**

| Technique           | Scheme                         | Notation                                |
|---------------------|--------------------------------|---|
| Entity-relationship | Microsoft Access 'ERD'         | $\mathfrak{N}(E-R, ERD_{Access})$       |
|                     | Martin ERD                     | $\mathfrak{N}(E-R, ERD_{Martin})$       |
|                     | Bachman ERD                    | $\mathfrak{N}(E-R, ERD_{Bachman})$      |
|                     | Shlaer-Mellor ERD              | $\mathfrak{N}(E-R, ERD_{ShlaerMellor})$ |
|                     | IDEF1X ERD                     | $\mathfrak{N}(E-R, ERD_{IDEF1X})$       |
|                     | 'Referenced arrowhead' ERD     | $\mathfrak{N}(E-R, ERD_{RefdArrow})$    |
|                     | 'Referencing arrowhead' ERD    | $\mathfrak{N}(E-R, ERD_{RefgArrow})$    |
| Relational          | SQL (45 dialects) <sup>a</sup> | $\mathfrak{N}(Relational, SQL)$         |
|                     | Microsoft Access <sup>b</sup>  | $\mathfrak{N}(Relational, MSAccess)$    |
| Object              | Coad/Yourdon                   | $\mathfrak{N}(Object, Coad/Yourdon)$    |
|                     | Rumbaugh OMT                   | $\mathfrak{N}(Object, Rumbaugh)$        |
|                     | Unified Modelling Language     | $\mathfrak{N}(Object, UML)$             |
|                     | (UML)                          |   |

**Notes on Table A.7:**

<sup>a</sup> The dialects include all those supported by EasyCASE, plus: DB2 v2.1, InterBase 4.1/4.2, Paradox 7, Progress 8, SQL Server 6.x, Sybase System 10/11, Visual dBASE 5.5 and Visual FoxPro 3.0/5.0.

<sup>b</sup> Supports versions 1.1, 2.0, 95 and 97 via direct connection using the JET database engine.

syER/EasyOBJECT partially facilitates the use of multiple representations.

EasyER/EasyOBJECT supports bidirectional technique translations between E-R and SQL representations and between object and SQL representations. It also appears to support technique translations between E-R and object representations, and scheme translations among the E-R representations and among the object representations. All the translations in these latter two groups are true translations, as some of them result in loss of information, but they are very simple translations in that they only involve direct one-to-one mappings between constructs. Source constructs that are not supported in the target representation are either ignored or changed into the nearest similar target construct, which may not always give the correct result. There is no capacity for mapping a collection of source constructs to a single target construct, or vice versa. In addition, these translations do not result in the generation of new descriptions; rather the existing description is altered in place.

## A.5.4 Sybase Deft

Deft (O'Brien, 1992) is a CASE environment produced by Sybase, Inc.<sup>1</sup> Deft comprises a collection of six tools that communicate via a common data dictionary, and supports a relatively small collection of representations, shown in Table A.8. The common data dictionary allows sharing of information across descriptions, so Deft partially facilitates the use of multiple representations in a manner similar to EasyCASE.

**Table A.8:** Representations supported by Deft

| Technique           | Scheme                                | Notation                                  |
|---------------------|---------------------------------------|---|
| Entity-relationship | Chen/Bachman ERD                      | $\mathfrak{R}(E-R, ERD_{C/B})$            |
|                     | Martin ERD                            | $\mathfrak{R}(E-R, ERD_{Martin})$         |
|                     | IRM ERD                               | $\mathfrak{R}(E-R, ERD_{IRM})$            |
| Data flow modelling | Gane & Sarson DFD                     | $\mathfrak{R}(DataFlow, DFD_{G\&S})$      |
|                     | Yourdon DFD                           | $\mathfrak{R}(DataFlow, DFD_{Yourdon})$   |
|                     | Process structure diagram (PSD)       | $\mathfrak{R}(DataFlow, PSD)$             |
| Relational          | SQL (11 dialects) <sup>a</sup>        | $\mathfrak{R}(Relational, SQL)$           |
|                     | Ingres QUEL (2 dialects) <sup>b</sup> | $\mathfrak{R}(Relational, QUEL_{Ingres})$ |

**Notes on Table A.8:**

<sup>a</sup> The SQL dialects supported by Deft are: DB2, 'Generic' (probably SQL/89), Informix, Ingres 5/6, Oracle 5/6/7, Rdb/VMS and Sybase 3/4.

<sup>b</sup> The QUEL dialects supported by Deft are Ingres 5 and Ingres 6.

Deft supports bidirectional technique translations between E-R and SQL representations only. The scheme translations supported by Deft are all trivial, as there is no difference in expressive power of the relevant representations. For example, the E-R representations differ only in the notation used to express cardinality and optionality, and may be 'translated' between merely by changing a display option in the preferences. A similar situation exists with the data flow modelling schemes.

### MetaCase's MetaEdit

MetaEdit (MetaCase Consulting, 1993) is a highly-customisable CASE tool developed by MetaCase Consulting Oy of Finland. The demonstration version of MetaEdit supports only three representations, but the full version supports many more (MetaCase

<sup>1</sup>Deft was discontinued as a supported product at the end of 1996.

Consulting, 1993), which are shown in Table A.9. It is also possible for developers to define additional representations within the tool. Descriptions in MetaEdit are completely separate from each other and there is no shared data dictionary. It is not possible to translate between any of the supported representations. That is, while MetaEdit supports multiple representations, it does not facilitate their use.

**Table A.9:** Representations supported by MetaEdit

| <b>Technique</b>    | <b>Scheme</b>                                   | <b>Notation</b>                           |
|---------------------|---|---|
| Entity-relationship | Chen ERD  | $\mathfrak{N}(E-R, ERD_{Chen})$           |
| Data flow modelling | Gane & Sarson DFD                               | $\mathfrak{N}(DataFlow, DFD_{G\&S})$      |
|                     | Booch OODA Process Diagram (OPD)                | $\mathfrak{N}(DataFlow, OPD_{Booch})$     |
|                     | Rumbaugh OMT DFD (RDD)                          | $\mathfrak{N}(DataFlow, DFD_{Rumbaugh})$  |
|                     | Yourdon SA/SD DFD (SAD)                         | $\mathfrak{N}(DataFlow, DFD_{Yourdon})$   |
| State event         | Booch OODA state transition diagram (STD)       | $\mathfrak{N}(StateEvent, STD_{Booch})$   |
|                     | Coad/Yourdon OOAD object state diagram (OSD)    | $\mathfrak{N}(StateEvent, OSD_{C/Y})$     |
|                     | Rumbaugh OMT state diagram (SD)                 | $\mathfrak{N}(StateEvent, SD_{Rumbaugh})$ |
|                     | Ward/Mellor real-time structured analysis (RTS) | $\mathfrak{N}(StateEvent, RTS_{W/M})$     |
| Object              | Booch OODA object diagram                       | $\mathfrak{N}(Object, Booch)$             |
|                     | Coad/Yourdon OO analysis and design             | $\mathfrak{N}(Object, Coad/Yourdon)$      |
|                     | Rumbaugh OMT class diagram                      | $\mathfrak{N}(Object, Rumbaugh)$          |
|                     | Welke Object-Property-Role-Relationship (OPRR)  | $\mathfrak{N}(Object, OPRR)$              |
| Structured design   | Yourdon structure chart (STC)                   | $\mathfrak{N}(Structured, STC_{Yourdon})$ |
|                     | Booch OODA module diagram (OMD)                 | $\mathfrak{N}(Structured, OMD_{Booch})$   |

# Appendix B

## Modifications to Smith's Method

### B.1 Introduction

A functional dependency diagram (FDD) is a means of graphically modelling the dependencies within a collection of attributes (Date, 1995). *Smith's method* (Smith, 1985) is a formal technique for deriving a set of normalised relations from an FDD. There is, however, a notable deficiency in the method as originally presented: the derivation of foreign keys. Smith's method does include two rules for deriving foreign keys, but there are two major problems with these rules:

1. They are incomplete. In all but the most trivial examples, there are usually several foreign keys that cannot be derived using the existing rules.
2. One of the two rules produces incorrect results in certain situations. Specifically, it can result in the derivation of a 'foreign key' that violates the relational definition of a foreign key.

In this appendix a new set of rules for deriving foreign keys from an FDD is described, along with some modifications to Smith's original FDD notation to assist in deriving foreign keys.

In Section B.2 Smith's original method is summarised. In Section B.3 the problems with deriving foreign keys are described in more detail. In Section B.4, modifications to Smith's FDD notation and a new set of foreign key derivation rules are proposed to solve these problems. In Section B.5 are presented examples of the new rules in use.

## B.2 Smith's Method — an overview

As previously stated, an FDD is a graphical representation of the functional dependencies within a collection of attributes. Date (1995) uses a different notation from that of Smith, but note that Date appears to use FDDs only to model the functional dependencies within a single relation, whereas Smith uses them as a database modelling tool. Smith derives his FDDs from a set of plain English *dependency-list statements*, such as the one shown in Figure B.1.

---

Anticipated design engineering work is organized into JOB\_NO engineering job numbers. Each JOB\_NO has one TYPE\_JOB (i.e., '1' = Basic Release, '2' = Sustaining, . . . ), one RESP\_ENGR responsible engineer (entered as an employee number), and one DUE\_DATE planned due date.

---

**Figure B.1:** Example of a dependency-list statement (Smith, 1985)

### B.2.1 The notation

The FDD notation used by Smith is as follows:

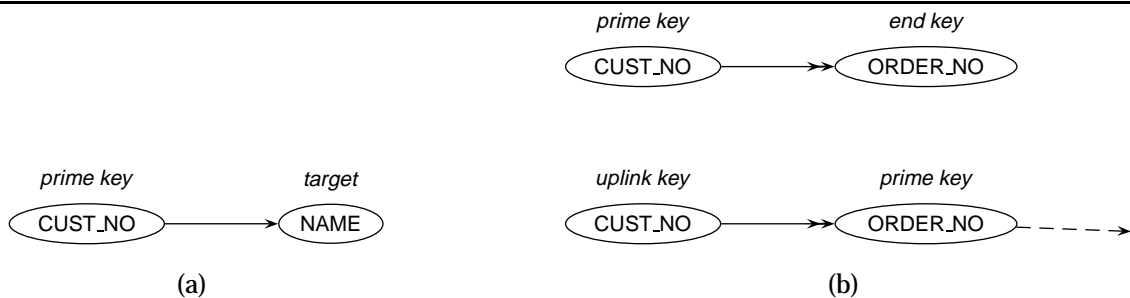
- Attributes (Smith refers to these as 'fields') are placed within *bubbles*. Multiple attributes may be placed within the same bubble to simplify the diagram (see Figure B.2).
- A single-valued dependence  $A \rightarrow B$  is represented by a single-headed arrow between the corresponding bubbles. The bubble at the start of the arrow is called a *prime-key bubble*, as shown in Figure B.3(a). The bubble at the end of the arrow is called a *target bubble*.
- A multivalued dependence  $C \twoheadrightarrow D$  is represented by a double-headed arrow between the corresponding bubbles. If the bubble at the end of the arrow is a prime key bubble, then the bubble at the start of the arrow is called an *uplink-key bubble*, otherwise it is a prime-key bubble. If the bubble at the end of the arrow is not a prime key or an uplink key, then it is known as an *end-key bubble*, as shown in Figure B.3(b).



- Attributes may be placed within more than one bubble. ‘Multibubbles’ are generally used to show the linkage of a chain of uplink-key, prime-key and end-key bubbles, as shown in Figure B.4(a) on the following page. Note that each bubble is independent of the others.
- *Domain flags* are used to tag attributes which belong to the same domain. For example, EMP\_NO and DEPT\_MGR both belong to the domain ‘employee number’, as shown in Figure B.4(b).



**Figure B.2:** (a) A bubble that contains a single attribute; (b), (c) bubbles that contain multiple attributes

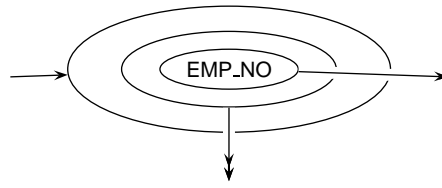


**Figure B.3:** (a) Single- and (b) multivalued dependencies

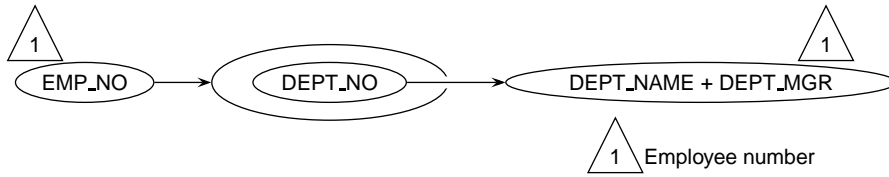
## B.2.2 Deriving a set of relations from an FDD

### Single-valued dependencies composed into relations

All target bubbles of the prime-key bubble, plus all uplink-key bubbles (if they exist) become the attributes of a single relation. The primary key of this table comprises the concatenation of all attributes within the prime-key bubble plus all attributes within uplink-key bubbles. See Figure B.5 on the next page for an example.

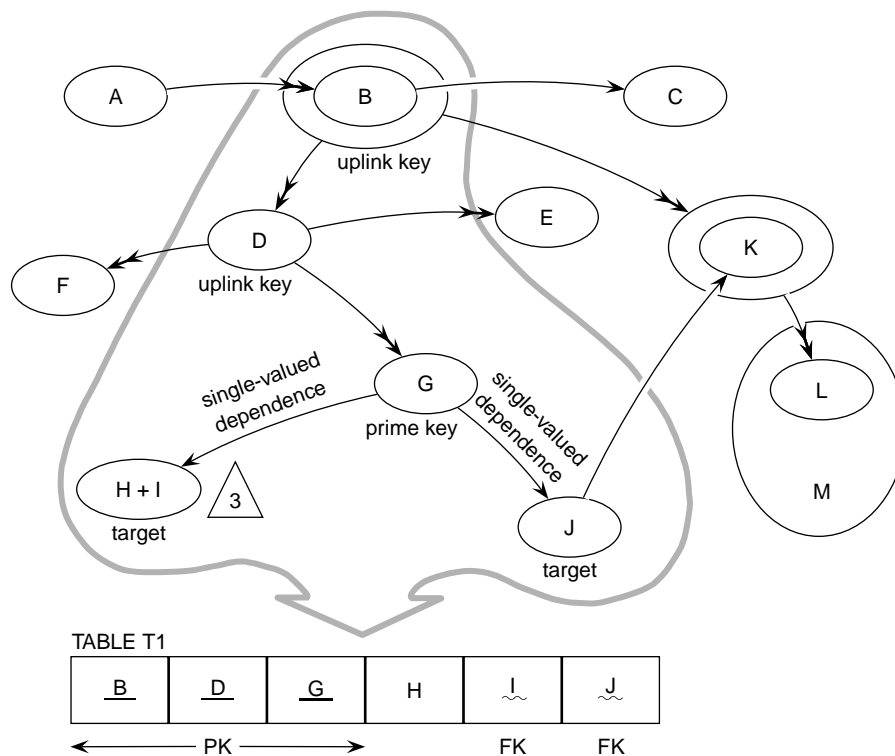


(a)



(b)

**Figure B.4:** (a) Multiple bubbles and (b) domain flags



**Figure B.5:** Deriving a relation from a single-valued dependency (Smith, 1985)

Attributes within a target bubble become foreign keys of the derived relation if they also function as a key bubble of any sort (the *target bubble rule*), or are tagged with a domain flag (the *domain flag rule*). These rules shall be revisited in Section B.3.

### End-key dependencies composed into relations

All attributes of the end-key bubble, its prime-key bubble and any/all uplink-key bubbles become the primary key of a single relation. See Figure B.6 for an example.

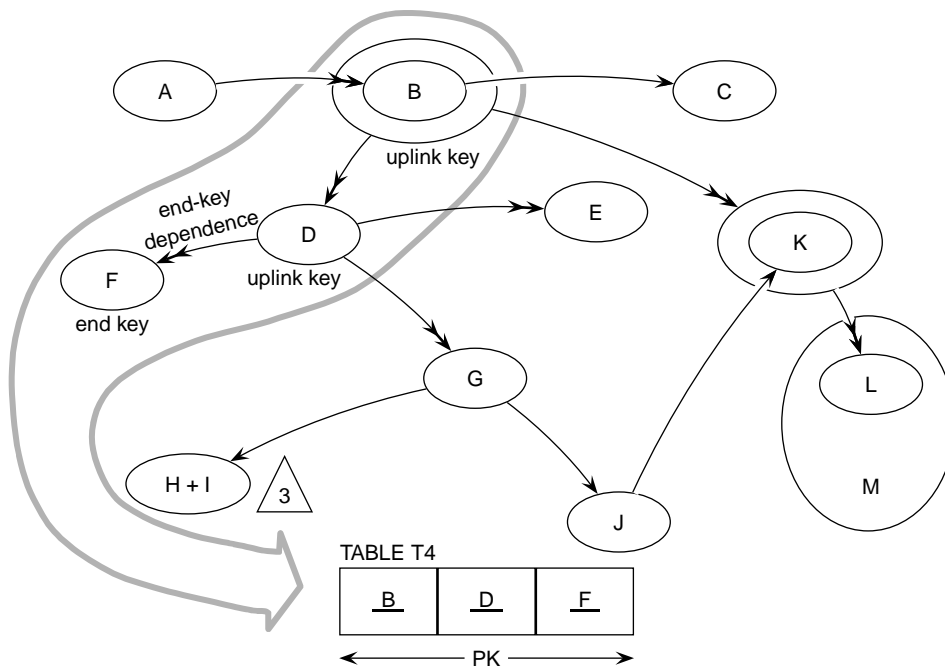


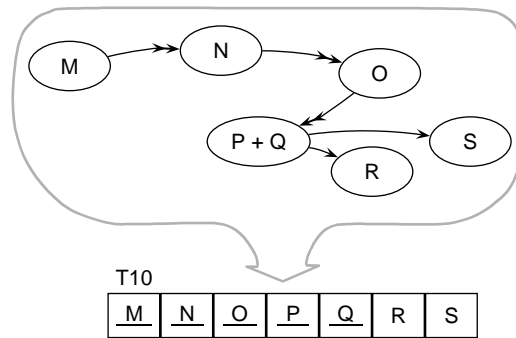
Figure B.6: Deriving a relation from an end-key dependency (Smith, 1985)

### Isolated bubbles composed into relations

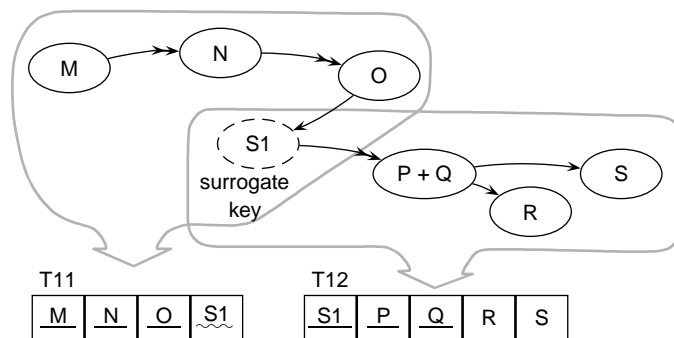
An *isolated bubble* is one that has no arrows pointing either to it or from it. All attributes within an isolated bubble become the primary key of a single relation.

### Practicable diagrams

Smith defines a *practicable dependency diagram* as one which does not result in relations with a primary key consisting of three or more attributes. To correct an impracticable FDD, the diagram is modified by adding *surrogate keys* (Date, 1995) to break the offending relation(s) into two or more sub-relations (Figure B.7 on the following page).



(a) Not practicable



(b) Practicable

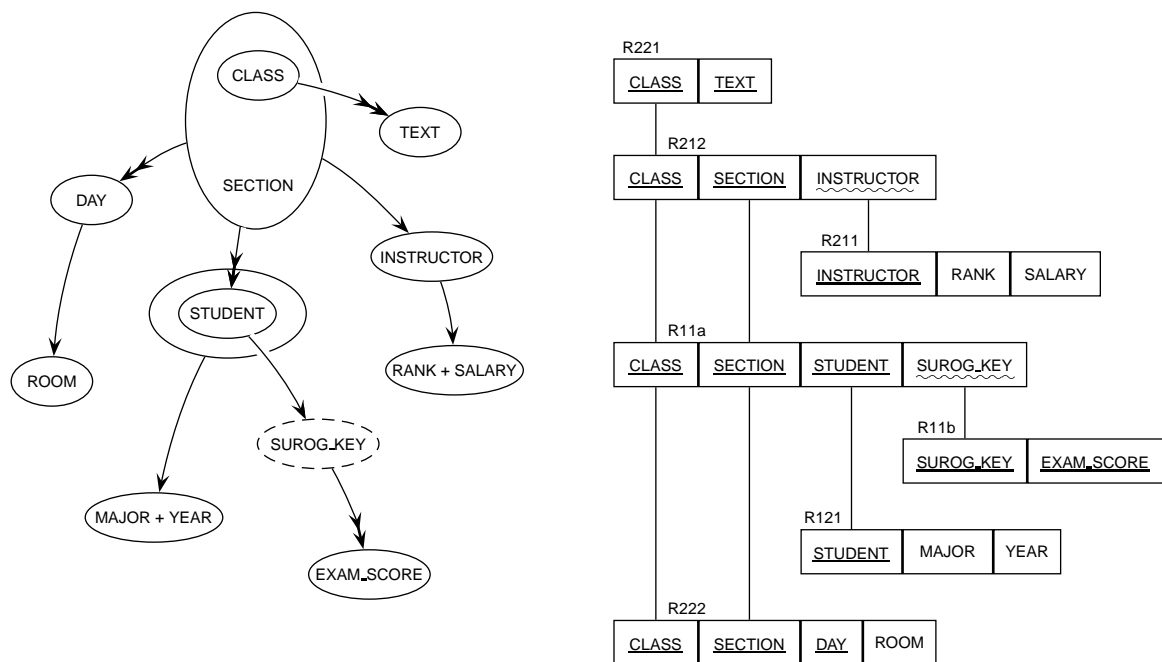
**Figure B.7:** Correcting an impracticable FDD (Smith, 1985)

### Other guidelines

Smith also states several other guidelines for designing FDDs which generally result in a 'better' design. These guidelines do not have any impact on the foreign key problem, so they are not described here.

## B.3 The problem with deriving foreign keys

The rules presented in the original method for deriving foreign key links are incomplete. For example, examining Smith's first example, shown in Figure B.8, it is obvious that CLASS + SECTION in relation R222 is a foreign key to CLASS + SECTION in relation R212. Using the existing foreign key derivation rules, however, there is no way to derive this foreign key. Similarly, STUDENT in R11a is a foreign key to STUDENT in R121, but cannot be derived from the diagram using the existing rules.



**Figure B.8:** Foreign keys that cannot be derived using the existing rules (Smith, 1985)

This is further exacerbated by the fact that target bubble rule can result in invalid foreign keys. This rule states that attributes within a target bubble become foreign keys of the resultant table if they also function as a key bubble. Applying this rule to Smith’s second example results in several attributes being indicated as foreign keys when they are not, such as those highlighted in Figure B.9 on the next page.

A foreign key is defined as an attribute (or set of attributes) whose value is identical to some candidate key in the database, or is null (Date, 1995; Elmasri and Navathe, 1994). Usually the candidate key is the primary key of the table in question. In Figure B.9, it can be seen that the so-called ‘foreign keys’ are not referencing candidate keys, rather they are referencing only part of the primary key of the referenced relation.

In addition, the domain flag rule states that attributes within a target bubble become foreign keys of that table if they are tagged with a domain flag. It is unclear from this rule as to which of the tagged bubbles should be used as the actual foreign key, and which should be used as the ‘target’ of the foreign key (that is, the attributes that the foreign key should reference), although this can usually be determined from the dependency-list statements.

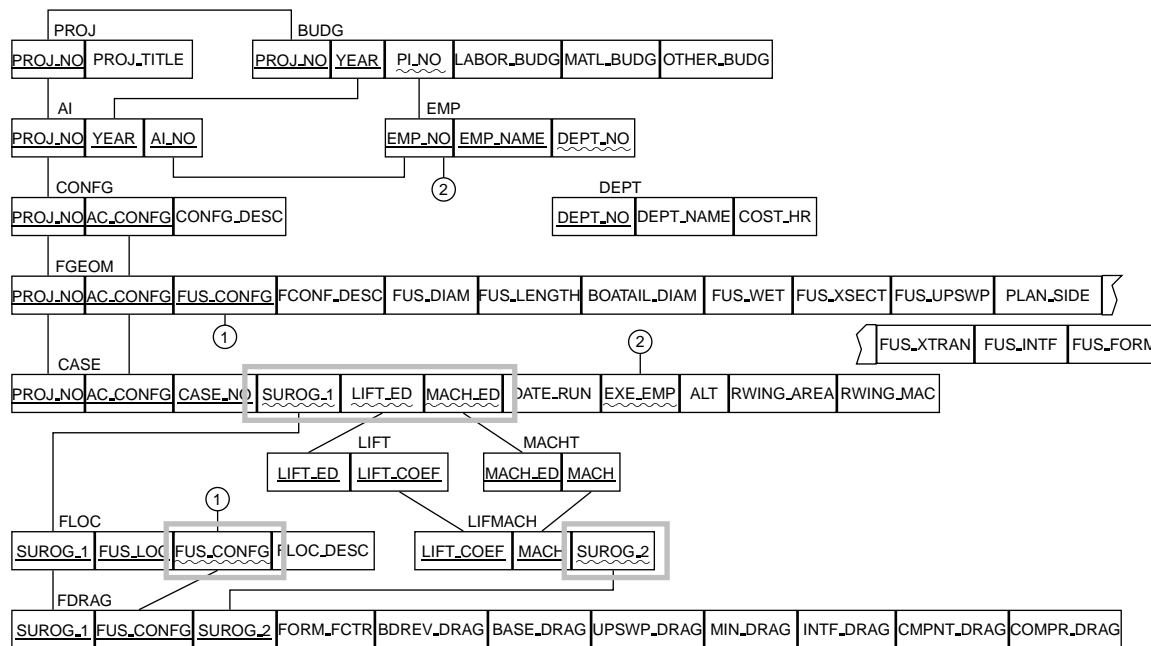


Figure B.9: Derivation of invalid foreign keys (Smith, 1985)

## B.4 The solution

The target bubble rule needs to be replaced. To facilitate this change, Smith's original terminology for bubble types must be altered. Smith uses the term *prime key* to denote the start bubble of a single-valued dependence, and sometimes the start bubble of a multivalued dependence. The term 'prime' really implies that the prime key attributes are the sole determinants of the target attributes. Thus it seems rather counter-intuitive that a prime-key bubble can be part of an uplink key chain, as it is no longer the sole determinant.

The following bubble terminology is therefore proposed:

**Single-key bubble:** any bubble that has no arrows pointing to it, and is the start bubble of a single-valued dependency.

**Target bubble:** the end bubble of a single-valued dependency.

**Multi-key bubble:** the start bubble of a multivalued dependency.

**End-key bubble:** any bubble that is not a multi-key bubble, and is the end bubble of a multivalued dependency.

**Isolated bubble:** A bubble with no attached dependencies (identical to Smith’s definition).

Bubbles may only be of one type. Smith did not enforce this restriction, for example, target bubbles could also be prime-key bubbles, although he did state that the multiple bubbles could be used to clarify such situations. Since the new notation requires every bubble to be of a single type, multiple bubbles become essential.

Having made these changes, it is now possible to replace the target bubble rule with the following:

**Key bubble rule:** Let  $B$  be a bubble of any type, and  $R_B$  be the derived relation to which this bubble contributes. If the attributes of  $B$  form the entire contents of a single-key bubble  $S$  ( $S \neq B$ , contributing to a derived relation  $R_S$ ), then the attributes contained by  $S$  become a foreign key of  $R_B$  that refers to  $R_S$ .

The domain flag rule could remain unchanged. As stated previously, any ambiguity as to where the foreign key should be placed can be resolved by examining the relevant dependency-list statements, but this does not lend itself very well to automation because important information is hidden from the translation. When translating a collection of domain flags into relational form, it is required to know which of the tagged attributes is the ‘target’ attribute for the purposes of generating the correct foreign key references. Smith makes no reference as to how this should be done, yet in his examples, the domain flags are somehow translated correctly. This is possibly because Smith expected the process to be carried out manually and used the dependency-list statements to resolve ambiguities, as suggested above. There is certainly no support in Smith’s notation for identifying the ‘target’ attribute, which makes the automation of this process somewhat difficult. In order to facilitate the automatic translation of domain flags, the author has introduced the notation shown in Figure B.10 to indicate that the tagged attribute is the ‘target’ attribute for that domain flag.



**Figure B.10:** ‘Target’ attribute domain flag notation

The domain flag rule can thus be redefined as:

**New domain flag rule:** Let  $B$  be a bubble of any type containing an attribute  $A$  that is tagged with a domain flag, and let  $R_B$  be the derived relation to which this bubble contributes. The domain flag is ‘targeted’ on another attribute  $D$  that is the sole attribute contained by a single-key bubble  $S$  ( $S \neq B$ , derived relation  $R_S$ ). Attribute  $A$  becomes a foreign key of  $R_B$  that refers to attribute  $D$  of  $R_S$ .

## B.5 Example — university marks

It was originally planned to use Smith’s examples to illustrate the new rules in action. However, upon closer examination, it was discovered that neither of the two examples are particularly useful. The first example (that of a University database) is not really complete enough illustrate the new rules. For example, a CLASS is defined as having a set of TEXTs, but this is the only attribute in the example which is directly dependent on CLASS. In reality, a CLASS would have several other dependent attributes, such as department, coordinator and number of points/credits. Because of these missing attributes, the example does not provide enough situations to show the new rules at work.

The second example (a drag prediction database), by contrast, is far too complex. In addition, it has so many complicated dependencies among attributes that it is arguable as to whether a relational implementation is the best solution.

Instead, the university marks example from Appendix C will be used. Applying Smith’s original foreign key rules to the FDD shown in Figure C.13, the following set of relations can be derived:

1. Staff(staff\_id, name, password)
2. Student(student\_id, name, password)
3. Element(element\_id, name, total\_mark, percent, date\_due, late\_penalty)
4. Assignment(assign\_id, element\_id, student\_id, staff\_id, date\_submitted, raw\_mark, comments)

*element\_id* is a foreign key to Element (target bubble rule)



*student\_id* is a foreign key to Student (target bubble rule)

*staff\_id* is a foreign key to Staff (target bubble rule)

5. Adjustment(assign\_id, adjustment\_no, reason, amount)  
*assign\_id* should be a foreign key to Assignment, but this cannot be derived because none of the bubbles containing *assign\_id* are target bubbles.
6. Question(question\_id, number, marks, guidelines, parent\_question)  
*parent\_question* is a foreign key to Question (domain flag rule)
7. Answer(answer\_id, question\_id, mark, comments, parent\_answer)  
*question\_id* is a foreign key to Question (target bubble rule)  
*parent\_answer* is a foreign key to Answer (domain flag rule)
8. Criterion(answer\_id, criterion\_name, mark, comments)  
*answer\_id* should be a foreign key to Answer, but this cannot be derived because none of the bubbles containing *answer\_id* are target bubbles.
9. Assign\_Answer(assign\_id, answer\_id)  
*assign\_id* should be a foreign key to Assignment and *answer\_id* should be a foreign key to Answer, but these cannot be derived because neither of the bubbles involved are target bubbles.
10. Element\_Question(element\_id, question\_id)  
*element\_id* should be a foreign key to Element and *question\_id* should be a foreign key to Question, but these cannot be derived because neither of the bubbles involved are target bubbles.

Using the new rules defined in Section B.4, the following set of relations can be derived:

1. Staff(staff\_id, name, password)
2. Student(student\_id, name, password)
3. Element(element\_id, name, total\_mark, percent, date\_due, late\_penalty)

4. Assignment(assign\_id, element\_id, student\_id, staff\_id, date\_submitted, raw\_mark, comments)  
*element\_id* is a foreign key to Element (key bubble rule)  
*student\_id* is a foreign key to Student (key bubble rule)  
*staff\_id* is a foreign key to Staff (key bubble rule)
5. Adjustment(assign\_id, adjustment\_no, reason, amount)  
*assign\_id* is a foreign key to Assignment (key bubble rule)
6. Question(question\_id, number, marks, guidelines, parent\_question)  
*parent\_question* is a foreign key to Question (domain flag rule)
7. Answer(answer\_id, question\_id, mark, comments, parent\_answer)  
*question\_id* is a foreign key to Question (key bubble rule)  
*parent\_answer* is a foreign key to Answer (domain flag rule)
8. Criterion(answer\_id, criterion\_name, mark, comments)  
*answer\_id* is a foreign key to Answer (key bubble rule)
9. Assign\_Answer(assign\_id, answer\_id)  
*assign\_id* is a foreign key to Assignment (key bubble rule)  
*answer\_id* is a foreign key to Answer (key bubble rule)
10. Element\_Question(element\_id, question\_id)  
*element\_id* is a foreign key to Element (key bubble rule)  
*question\_id* is a foreign key to Question (key bubble rule)

# Appendix C

## Example viewpoints

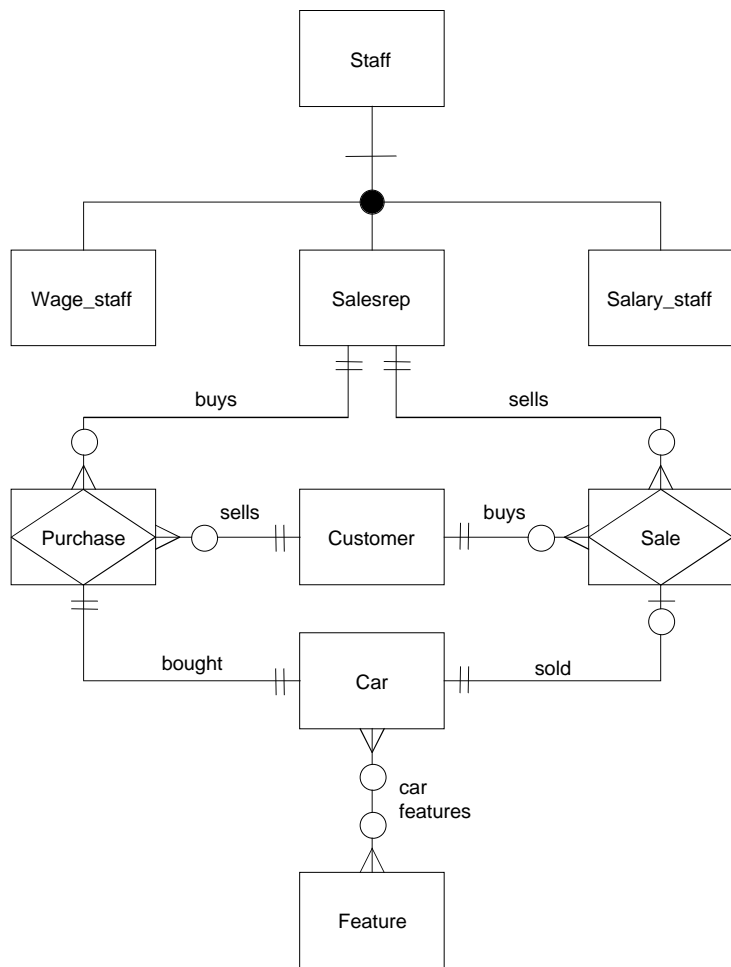
### C.1 Used cars viewpoint

This is a viewpoint of a small used cars dealership that purchases cars from and sells cars to customers. Purchases and sales are always for a single car and are dealt with exclusively by sales representatives. The business needs to know which sales representative was involved in a sale or purchase for commission purposes. Not all staff buy and sell cars; there are also accountants, mechanics, clerical staff and odd job staff. Sales representatives are paid on commission, while other staff are paid salary or wages. A list of non-standard features is kept for each car, such as air conditioning, air bags and alloy wheels. Some cars have several non-standard features and some none.

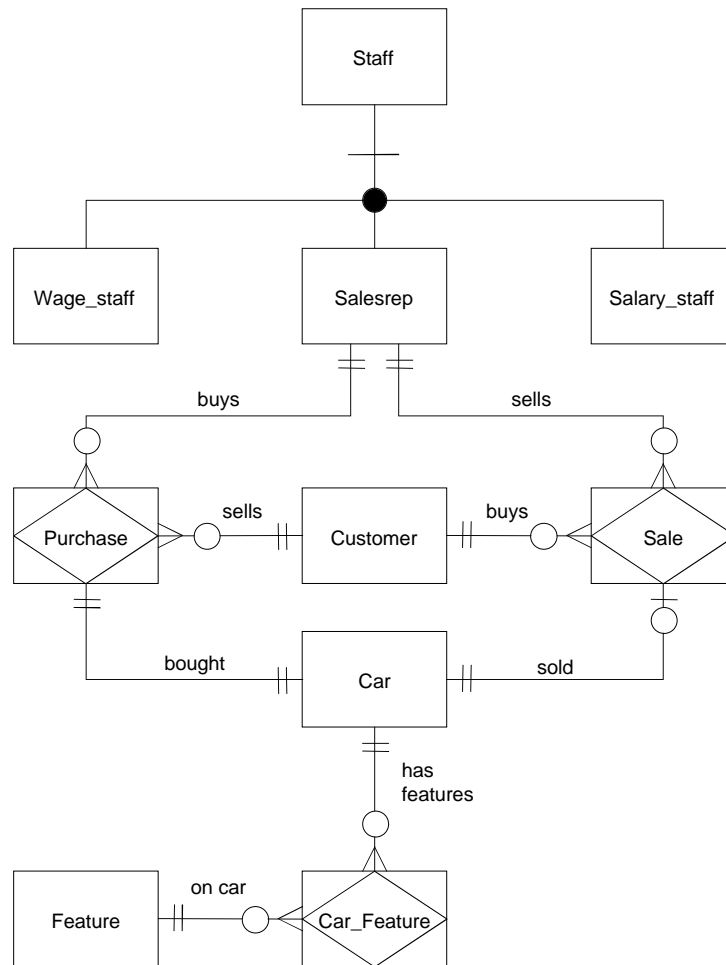
In Figure C.1 on the following page is shown one possible unnormalised E-R description of this business. The various types of staff (wage, salary and commission) have been modelled as subtypes of a general Staff entity. A normalised version of this ERD is shown in Figure C.2 on page 361.

In Figure C.3 on page 362 is shown a functional dependency diagram description of this business, based on the following set of dependencies:

- *ird\_number* → *name, address, phone*
- *wage\_staff\_id (ird\_number)* → *hourly\_rate, hours\_per\_week*
- *salary\_staff\_id (ird\_number)* → *salary*
- *salesrep\_id (ird\_number)* → *commission\_rate*
- *customer\_no* → *name, address, phone*

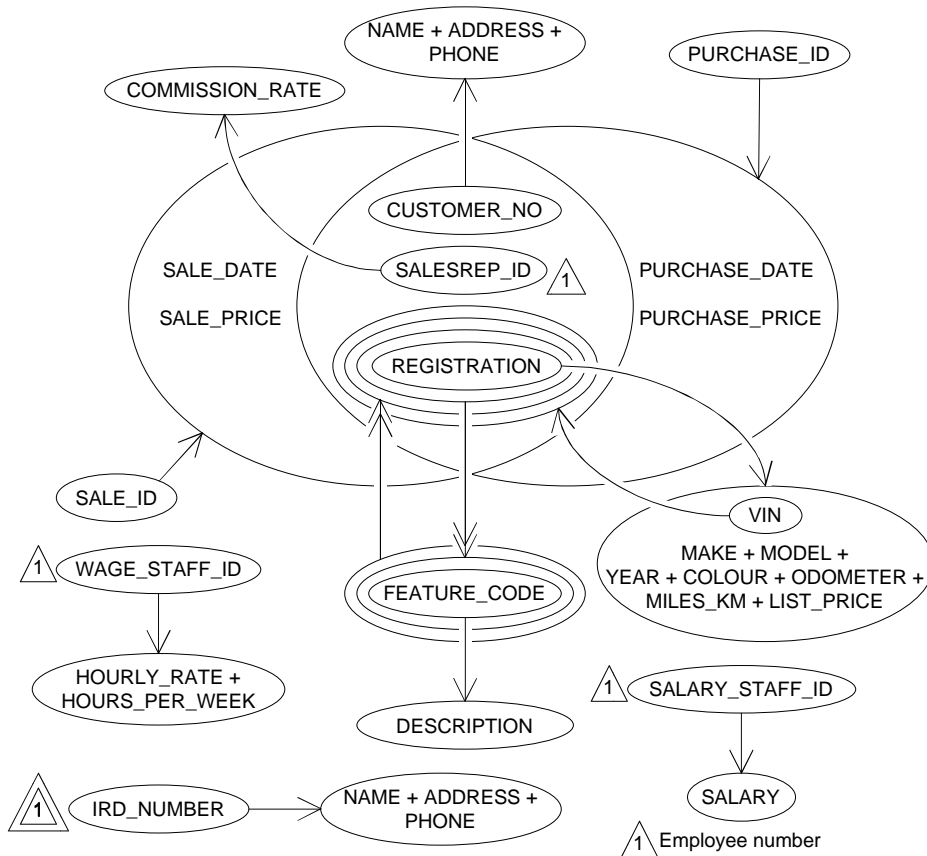


**Figure C.1:** E-R description of the used cars viewpoint (unnormalised)



**Figure C.2:** E-R description of the used cars viewpoint (normalised)

- *registration* → *vin*, *make*, *model*, *year*, *colour*, *odometer*, *miles\_km*, *list\_price*, *sale\_id*, *purchase\_id*
- *registration* → *feature\_code*
- *vin* → *registration*
- *feature\_code* → *description*
- *feature\_code* → *registration*
- *purchase\_id* → *purchase\_date*, *purchase\_price*, *customer\_no*, *salesrep\_id*, *registration*
- *sale\_id* → *sale\_date*, *sale\_price*, *customer\_no*, *salesrep\_id*, *registration*



**Figure C.3:** Functional dependency description of the used cars viewpoint

A complete SQL/92 description of this business is shown in Figure C.4, and a data flow description that models the sale process is shown in Figure C.5 on page 364.

---

```

create table staff
( ird_number char(7),
  name      char(80),
  address   char(80),
  phone     char(12),

  primary key (ird_number)
);

create table wage_staff
( wage_staff_id char(7),
  hourly_rate   smallint,
  hours_per_week integer(2),

  primary key (wage_staff_id),
  foreign key (wage_staff_id)
    references staff (ird_number)
);

create table salary_staff
( salary_staff_id char(7),
  salary          smallint,

  primary key (salary_staff_id),
  foreign key (salary_staff_id)
    references staff (ird_number)
);

create table salesrep
( salesrep_id char(7),
  commission_rate integer(2),

  primary key (salesrep_id),
  foreign key (salesrep_id)
    references staff (ird_number)
);

create table customer
( customer_no char(6),
  name      char(80),
  address   char(80),
  phone     char(12),

  primary key (customer_no)
);

create table car
( registration char(6),
  vin          char(20) not null unique,
  make        char(20),
  model       char(20),
  year        smallint,
  colour      char(20),
  odometer    integer,
  miles_km    char(1),
  list_price  integer,
  purchase_id char(6) not null unique,
  sale_id     char(6) unique,

  primary key (registration),
  foreign key (purchase_id)
    references purchase,
  foreign key (sale_id)
    references sale
);

create table feature
( feature_code char(6),
  description  char(80),

  primary key (feature_code)
);

create table car_feature
( feature_code char(6),
  registration char(6),

  primary key (feature_code, registration),
  foreign key (feature_code)
    references feature,
  foreign key (registration)
    references car
);

create table purchase
( purchase_id char(6),
  purchase_date date,
  purchase_price integer,
  customer_no char(6) not null,
  salesrep_id char(7) not null,
  registration char(6) not null unique,

  primary key (purchase_id),
  foreign key (customer_no)
    references customer,
  foreign key (salesrep_id)
    references salesrep,
  foreign key (registration)
    references car
);

create table sale
( sale_id char(6),
  sale_date date,
  sale_price integer,
  customer_no char(6) not null,
  salesrep_id char(7) not null,
  registration char(6) not null unique,

  primary key (sale_id),
  foreign key (customer_no)
    references customer,
  foreign key (salesrep_id)
    references salesrep,
  foreign key (registration)
    references car
);

```

---

**Figure C.4:** SQL/92 description of the used cars viewpoint

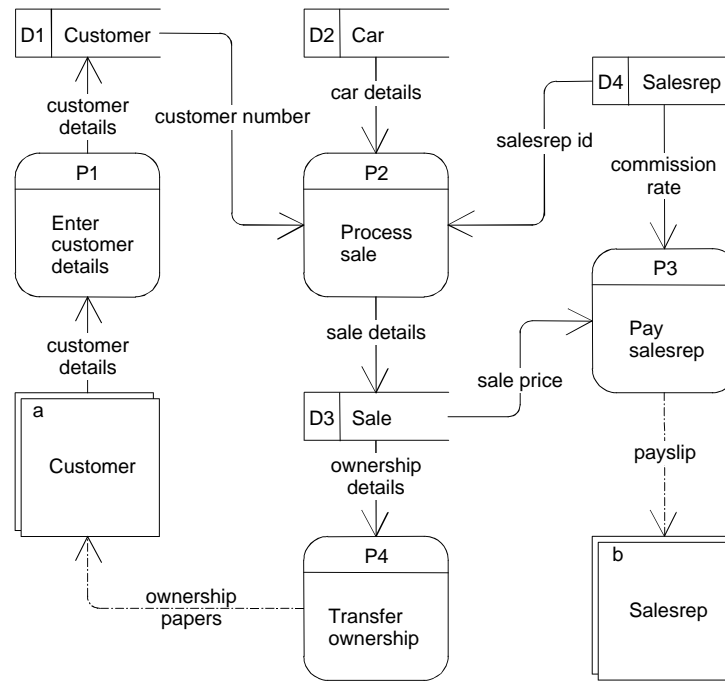


Figure C.5: Data flow description of the used cars viewpoint

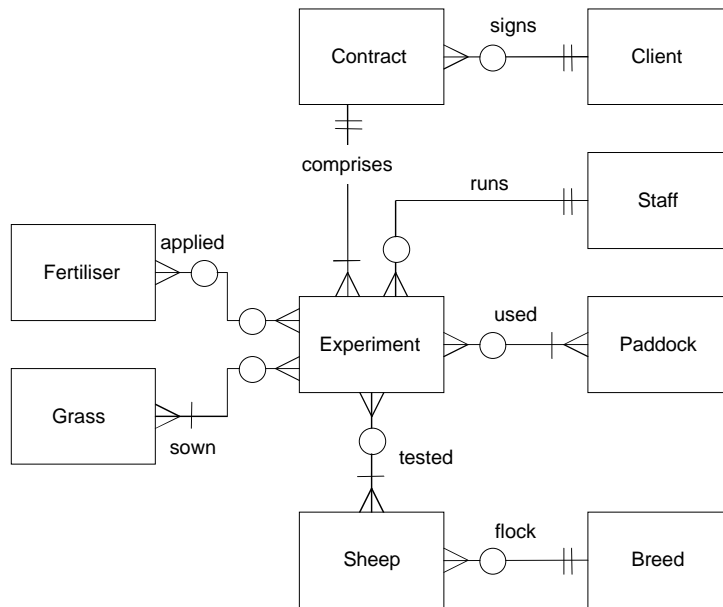
## C.2 Agricultural research institute viewpoint

This is a viewpoint of an agricultural research institute that carries out research under contract to fee paying external clients and the government. The data to be recorded for different contracts will vary depending on what the experimental parameters are, so a custom model is designed and implemented for each contract. For the purposes of this viewpoint, suppose there is a contract to evaluate the effect of a collection of fertilisers on the growth rate of various breeds of sheep grazing on various grass types.

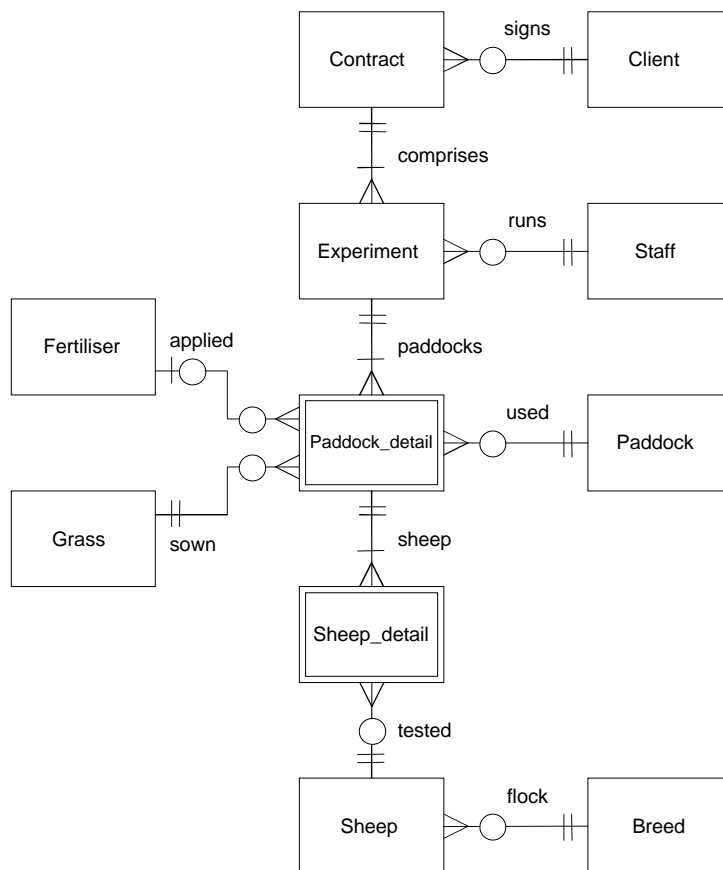
A series of experiments is to be carried out in which these parameters will be varied and the results measured as a weight change in the sheep. For a given experiment, each paddock is sown with a single grass type, a single fertiliser is applied and a flock of sheep of one breed are placed in the paddock. Fertilisers and grasses may be used in more than one paddock to test the effect of different pasture conditions, and some paddocks have no fertiliser applied as a control. Each experiment is run by a single scientist, and each contract is for a single client.

In Figure C.6 is shown one possible unnormalised E-R description of this viewpoint. This is a particularly interesting description because of the large number of many-to-many relationships associated with the Experiment entity. One possible nor-





**Figure C.6:** E-R description of the agricultural research institute viewpoint (unnormalised)



**Figure C.7:** E-R description of the agricultural research institute viewpoint (normalised)

malisation of Figure C.6 is shown in Figure C.7 on the preceding page, which assumes that sheep details are dependent on paddock details. (A different structure emerges if sheep details and paddock details are assumed to be independent.)

In Figure C.8 is shown a functional dependency description of the viewpoint, based on the following set of dependencies:

- *client\_id* → *name, address, category*
- *staff\_id* → *name, address, gender, dob, title, salary*
- *fertiliser\_id* → *name, supplier*
- *grass\_id* → *name*
- *paddock\_id* → *address, area, moisture, sunshine*
- *breed\_name* → *details, flock\_size*
- *sheep\_id* → *dob, gender, health, breed\_name*
- *contract\_id* → *details, sign\_date, finish\_date, fee, gst, client\_id*
- *experiment\_id* → *start\_date, finish\_date, staff\_id, contract\_id*
- *experiment\_id, paddock\_id* → *fertiliser\_id, grass\_id*
- *experiment\_id, paddock\_id, sheep\_id* → *start\_weight, finish\_weight*

The last two dependencies include the *embedded* multivalued dependencies (Date, 1995, p. 341) *experiment\_id* → *paddock\_id* and *experiment\_id, paddock\_id* → *sheep\_id* respectively. The relations corresponding to this set of dependencies are in at least fourth normal form.

An SQL/92 description of the viewpoint is shown in Figure C.9 on page 368, and a data flow description that models the experiment process is shown in Figure C.10 on page 369.



**Figure C.8:** Functional dependency description of the agricultural research institute viewpoint

---

```

create table staff
( staff_id char(10),
  name      char(20),
  address   char(20),
  gender    char(1),
  dob       date,
  title     char(15),
  salary    decimal(6,2),

  primary key (staff_id)
);

create table client
( client_id char(10),
  name      char(20),
  address   char(20),
  category  char(1),

  primary key (client_id)
);

create table fertiliser
( fertiliser_id char(10),
  name          char(20),
  supplier      char(20),

  primary key (fertiliser_id)
);

create table grass
( grass_id char(10),
  name     char(20),

  primary key (grass_id)
);

create table paddock
( paddock_id char(10),
  address    char(20),
  area       smallint,
  moisture   smallint,
  sunshine   smallint,

  primary key (paddock_id)
);

create table breed
( breed_name char(20),
  details    char(20),
  flock_size smallint,

  primary key (breed_name)
);

create table sheep
( sheep_id  char(10),
  breed_name char(20) not null,
  dob       date,
  gender    char(1),
  health    char(1),

  primary key (sheep_id),
  foreign key (breed_name)
    references breed
);

create table contract
( contract_id char(10),
  client_id   char(10) not null,
  details     char(20),
  fee         decimal(8,2),
  gst         decimal(8,2),
  sign_date   date not null,
  finish_date date,

  primary key (contract_id),
  foreign key (client_id)
    references client
);

create table experiment
( experiment_id char(10),
  contract_id   char(10) not null,
  staff_id      char(10) not null,
  finish_date   date,
  start_date    date,

  primary key (experiment_id),
  foreign key (contract_id)
    references contract,
  foreign key (staff_id)
    references staff
);

create table paddock_detail
( experiment_id char(10),
  paddock_id    char(10) not null,
  fertiliser_id char(10),
  grass_id      char(10) not null,

  primary key (experiment_id, paddock_id),
  foreign key (experiment_id)
    references experiment,
  foreign key (paddock_id)
    references paddock,
  foreign key (fertiliser_id)
    references fertiliser,
  foreign key (grass_id)
    references grass
);

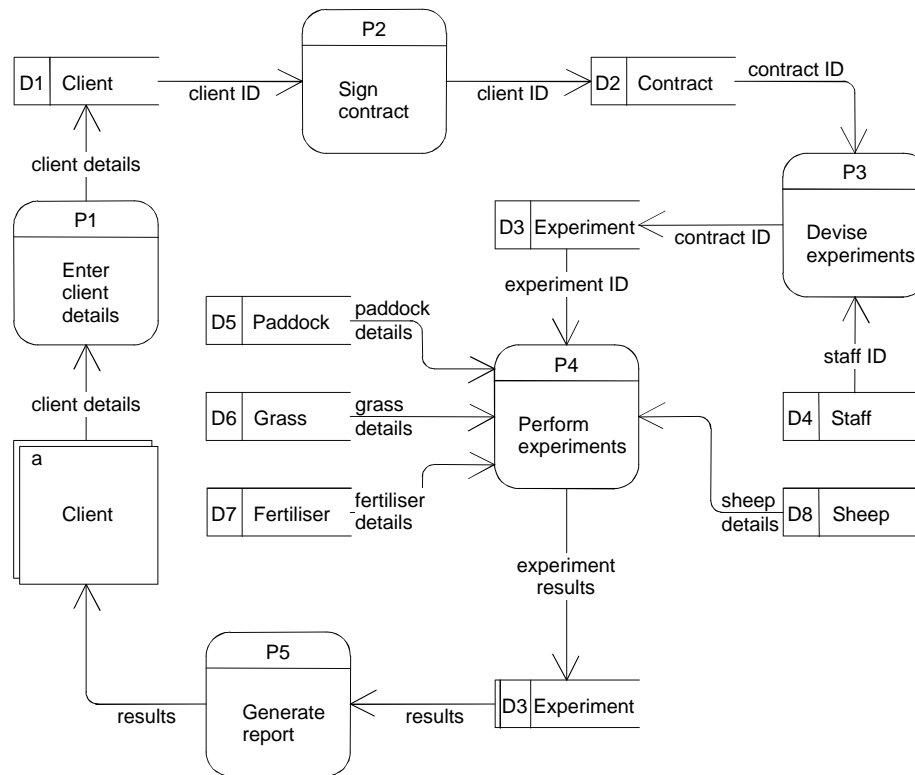
create table sheep_detail
( experiment_id char(10),
  paddock_id    char(10),
  sheep_id      char(10),
  finish_weight smallint,
  start_weight  smallint,

  primary key (experiment_id, paddock_id,
              sheep_id),
  foreign key (experiment_id)
    references experiment(experiment_id),
  foreign key (sheep_id)
    references sheep,
  foreign key (experiment_id, paddock_id)
    references paddock_detail
);

```

---

**Figure C.9: SQL/92 description of the agricultural research institute viewpoint**



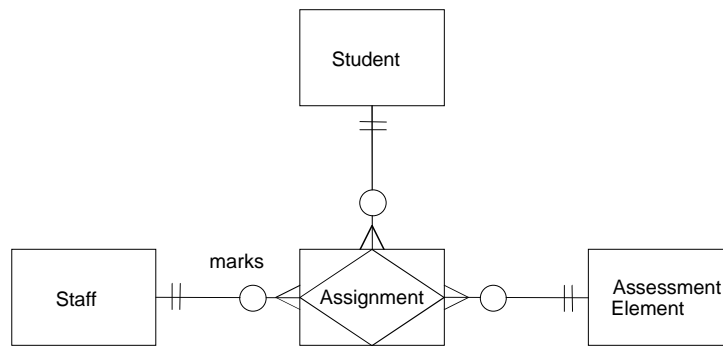
**Figure C.10:** Data flow description of the agricultural research institute viewpoint

### C.3 Assessment marks viewpoint

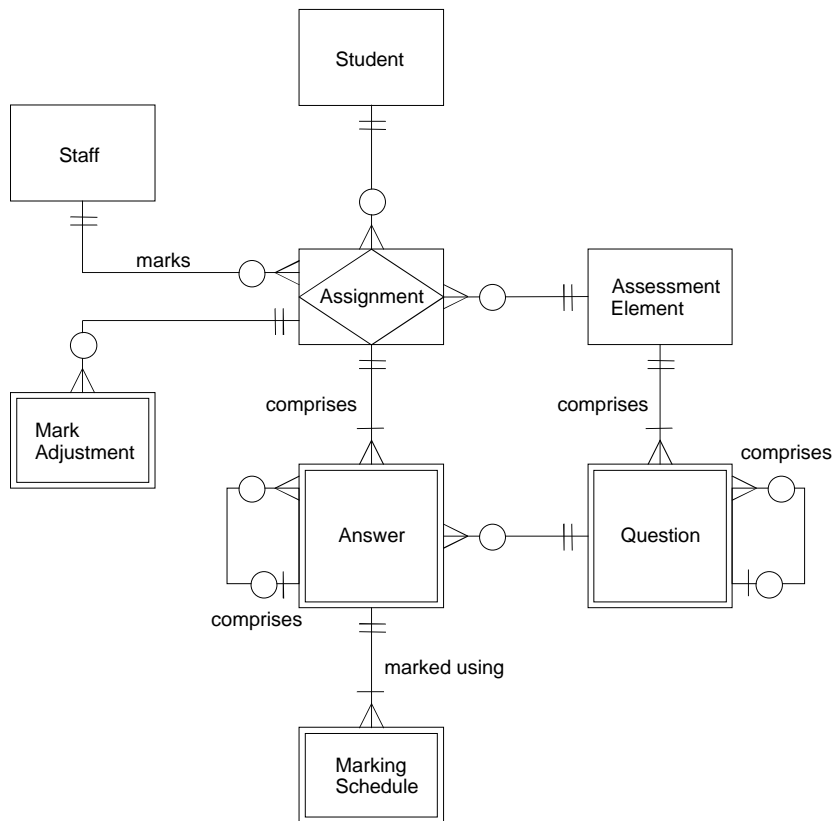
This is a viewpoint of an assessment marks recording database for a course. The final result for the course is determined by the results of a collection of assessment elements (such as practical exercises and examinations), each of which comprises a collection of questions. Each question may or may not comprise a collection of sub-questions. Students individually complete several assessment elements during the course, submitting each as an assignment, which is marked by a single staff member. As with assessment elements, an assignment comprises a collection of answers (corresponding to questions), which in turn comprise a collection of sub-answers.

The total mark for an assignment is broken down into a collection of marks for each individual answer. Each answer is marked according to a marking schedule that specifies a set of marking criteria and the marks allocation for each criterion. Results may be adjusted at a later date for reasons of illness or technical difficulties.

In Figure C.11 on the next page is shown one possible unnormalised E-R description of this viewpoint, which is shown in normalised form in Figure C.12.



**Figure C.11:** E-R description of the assessment marks viewpoint (unnormalised)



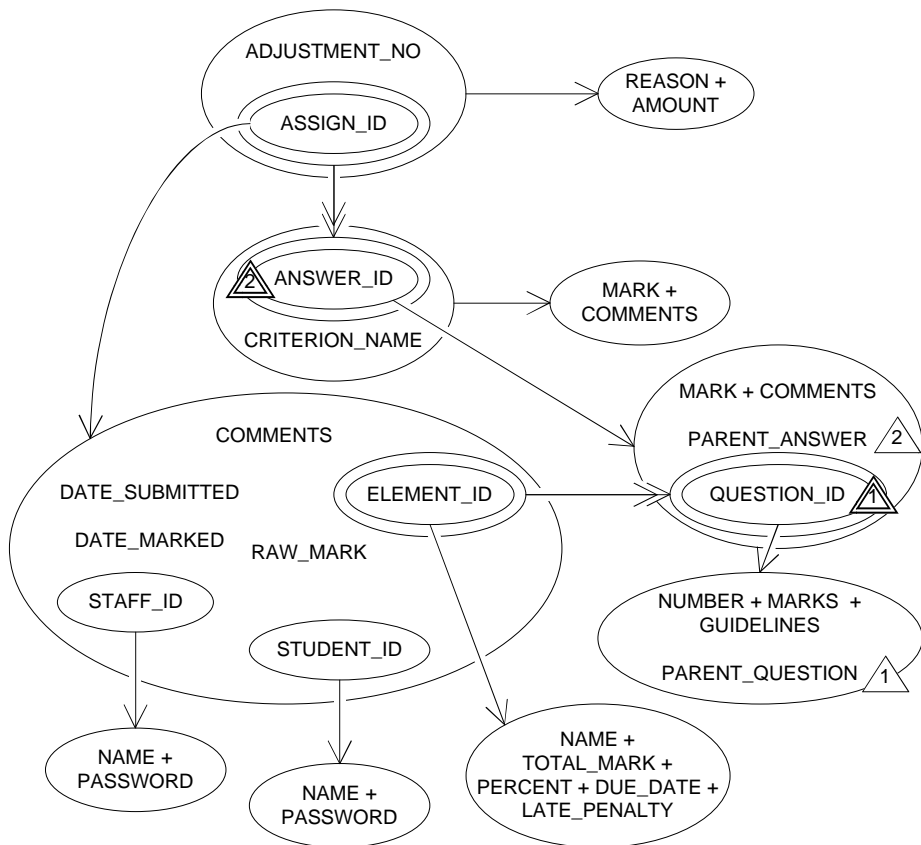
**Figure C.12:** E-R description of the assessment marks viewpoint (normalised)

In Figure C.13 on the following page is shown a functional dependency description of the viewpoint, based on the following set of dependencies:

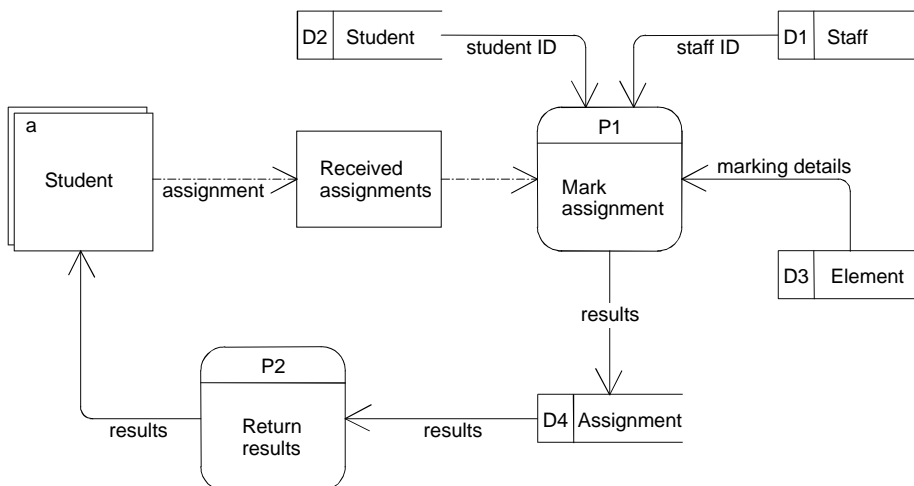
- *student\_id* → *name, password*
- *staff\_id* → *name, password*
- *element\_id* → *name, total\_mark, percent, due\_date, late\_penalty*
- *element\_id* → *question\_id*
- *question\_id* → *number, marks, guidelines, parent\_id (question\_id)*
- *assign\_id* → *date\_submitted, date\_marked, raw\_mark, comments, student\_id, staff\_id, element\_id*
- *assign\_id* → *answer\_id*
- *answer\_id* → *mark, comments, question\_id, parent\_id (answer\_id)*
- *assign\_id, adjustment\_no* → *reason, amount*
- *answer\_id, criterion\_name* → *mark, comments*

The last two functional dependencies include the embedded multivalued dependencies *assign\_id* → *adjustment\_no* and *answer\_id* → *criterion\_name* respectively. The relations corresponding to this set of dependencies are in at least fourth normal form.

A data flow description that models the marking process is shown in Figure C.14 on the next page, and an SQL/92 description of the viewpoint is shown in Figure C.15. Note that, as shown in Figure C.15 on page 373, the Question and Answer relations and entities have a unique identifier rather than a partial key, in order to facilitate questions having sub-questions and answers having sub-answers.



**Figure C.13:** Functional dependency description of the assessment marks viewpoint



**Figure C.14:** Data flow description of the assessment marks viewpoint



---

```

create table staff
( staff_id char(8),
  name      char(80),
  password  char(20),

  primary key (staff_id)
);

create table student
( student_id char(7),
  name       char(80),
  password   char(20),

  primary key (student_id)
);

create table element
( element_id integer,
  name       char(80),
  total_mark smallint,
  percent    smallint,
  due_date   date,
  late_penalty smallint,

  primary key (element_id)
);

create table question
( question_id integer,
  element_id  integer not null,
  number      char(5),
  marks       smallint,
  guidelines  char(500),
  parent_question integer,

  primary key (question_id),
  foreign key (element_id)
    references element,
  foreign key (parent_question)
    references question
);

create table assignment
( assign_id integer,
  element_id integer not null,
  student_id char(7) not null,
  staff_id   char(8) not null,
  date_submitted date,
  date_marked date,
  raw_mark   smallint,
  comments   char(500),

  primary key (assign_id),
  foreign key (element_id)
    references element,
  foreign key (student_id)
    references student,
  foreign key (staff_id)
    references staff
);

create table adjustment
( assign_id integer,
  adjustment_no integer,
  reason     char(200),
  amount     smallint,

  primary key (assign_id,
              adjustment_no),
  foreign key (assign_id)
    references assignment
);

create table answer
( answer_id integer,
  assign_id integer not null,
  question_id integer not null,
  mark       smallint,
  comments   char(500),
  parent_answer integer,

  primary key (answer_id),
  foreign key (assign_id)
    references assignment,
  foreign key (question_id)
    references question,
  foreign key (parent_answer)
    references answer
);

create table marking_schedule
( answer_id integer,
  criterion_name char(30),
  mark          smallint,
  comments      char(500),

  primary key (answer_id,
              criterion_name),
  foreign key (answer_id)
    references answer
);

```

---

**Figure C.15:** SQL/92 description of the assessment marks viewpoint



# Appendix D

## Technique and representation definitions

### D.1 Introduction

In this appendix are provided definitions of the techniques and representations used in this thesis. The following conventions are used:

- All definitions are expressed using a variant of Martin E-R notation as defined by the EasyCASE CASE tool (see also Appendix A Evergreen Software Tools, 1995b).
- Representations may specialise the constructs of a technique. New constructs and relationships in a representation definition are highlighted in **bold**.
- Constructs and relationships of a technique that are not used in a representation are drawn using dashed lines and/or *italics*.
- Sometimes a relationship may specialise another relationship in the definition. These relationships have the same name as the relationship they specialise, with the suffix '-s<sub>n</sub>', where *n* is a monotonically increasing integer. Thus, the first specialisation of the relationship contains is named contains-s<sub>1</sub>.
- A black dot (●) indicates mutual exclusivity and is usually used for generalisation hierarchies. It may also be used on relationships between entities (see the data flow modelling technique in Section D.5.1) to indicate that one of several different entities may participate in the relationship. The named section of the relationship indicates the entity that always participates.

## D.2 Entity-relationship technique

This technique is derived from Chen's (1976; 1977) definition of the E-R approach. The constructs of the approach outlined by Chen and shown in Figure D.1 are:

**ERENTITYTYPE:** An entity is a distinctly identifiable object. These entities may be classified into different types. In this respect, an entity type is analogous to an object class. An entity is an instance of a particular entity type. The term 'entity type' has fallen out of use, however, and 'entity' is often used to refer to what are strictly entity types.

**ERRELATIONSHIPTYPE:** A relationship is an association between a collection of entities. In a similar way to entities, relationships may also be classified into different types. Once again, the term 'relationship' has often replaced 'relationship type' in current usage.

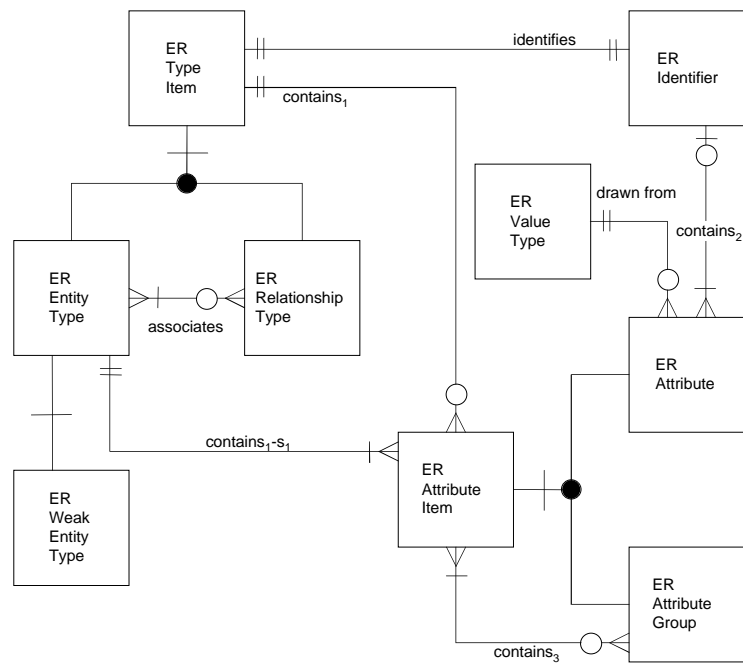
**ERATTRIBUTE:** Entities and relationships may have various properties, which are expressed in terms of attribute-value pairs, for example, ('age', 24). Attributes may be single valued or multivalued (a 'repeating group').

**ERVALUETYPE:** Values of attributes may be classified into different types. This is analogous to the concept of a domain of values.

**ERIDENTIFIER:** Both relationships and entities may be uniquely identified by relationship and entity identifiers respectively. An identifier comprises a collection of attribute-value pairs that collectively can uniquely identify the entity or relationship. This is analogous to the relational concept of a primary key.

**ERWEAKENTITYTYPE:** A weak entity type is an entity type that has one or both of the following two properties:

- its existence is dependent on another entity; and/or
- it cannot be uniquely identified by its own attributes and must be identified by its relationships with other entities.



**Figure D.1:** Definition of the E-R technique

These base constructs have been augmented with the following (described further below):

**ERTYPEITEM:** This is a generalisation of **ERENTITYTYPE** and **ERRELATIONSHIP-TYPE**.

**ERATTRIBUTEGROUP:** This construct allows for composite attributes, and may contain one or more attributes or attribute groups.

**ERATTRIBUTEITEM:** This is a generalisation of **ERATTRIBUTE** and **ERATTRIBUTE-GROUP**.

The properties of all these constructs are shown in Table D.1 on the next page.

The **ERENTITYTYPE** and **ERRELATIONSHIP-TYPE** constructs have similar properties, so the **ERTYPEITEM** construct has been introduced as a generalisation of these two constructs in order to simplify the structure of the technique definition.

The E-R approach as defined by Chen does not assume any particular implementation model and can thus in theory be used to build conceptual models for relational, network or object-oriented systems. In practice it is not expressive enough to deal with

**Table D.1: Construct properties for the E-R technique**

| CONSTRUCT<br>Property                          | 'Data type'              | Description  |
|--|--------------------------|--|
| ERTYPEITEM                                     |                          |  |
| name   | string                   | The name of the entity/relationship.   |
| attributes                                     | list(ERATTRIBUTEITEM)    | A list of component attribute items.   |
| identifier                                     | ERIDENTIFIER             | The identifier of the entity/relationship.   |
| ERENTITYTYPE (specialises ERTYPEITEM)          |                          |  |
| relationships                                  | list(ERRELATIONSHIPTYPE) | A (possible empty) list of relationships attached to the entity.   |
| ERWEAKENTITYTYPE (specialises ERENTITYTYPE)    |                          |  |
| dependentVia                                   | ERRELATIONSHIPTYPE       | The relationship that connects the weak entity to its 'parent' entity.                                     |
| ERRELATIONSHIPTYPE (specialises ERTYPEITEM)    |                          |  |
| entities                                       | list(ERENTITYTYPE)       | A list of entities that the relationship associates.   |
| cardinalities                                  | list(integer)            | The cardinality of each entity in the above list.  |
| existence_dependent                            | boolean                  | Is this an existence-dependent relationship (Chen, 1977, Section 3.3.1)?                                   |
| id_dependent                                   | boolean                  | Is this an ID-dependent relationship (Chen, 1977, Section 3.3.2)?  |
| ERVALUETYPE                                    |                          |  |
| name   | string                   | The name of the value type.  |
| datatype                                       | enumerator               | The data type of the value type (it is expected that these will be defined as an enumeration).             |
| size   | integer                  | The size of the values allowed in the value type, if applicable (that is, number of characters or digits). |
| dp   | integer                  | The number of decimal places, if applicable.   |
| attributes                                     | list(ERATTRIBUTE)        | A (possibly empty) list of attributes that are drawn from the value type.                                  |
| ERATTRIBUTEITEM                                |                          |  |
| name   | string                   | The name of the attribute/attribute group.   |
| containingItem                                 | ERTYPEITEM               | The entity/relationship that contains the element.   |
| attributeGroups                                | list(ERATTRIBUTEGROUP)   | A (possibly empty) list of attribute groups that the element appears in.                                   |
| repeating                                      | boolean                  | Is this a repeating element?   |
| ERATTRIBUTEGROUP (specialises ERATTRIBUTEITEM) |                          |  |
| attributeItems                                 | list(ERATTRIBUTEITEM)    | A list of attributes/attribute groups that comprise the attribute group.                                   |
| ERATTRIBUTE (specialises ERATTRIBUTEITEM)      |                          |  |
| valueType                                      | ERVALUETYPE              | The value type from which the attribute's values are drawn.  |
| identifier                                     | ERIDENTIFIER             | An optional entity identifier in which the attribute participates.   |
| ERIDENTIFIER                                   |                          |  |
| name   | string                   | The name of the identifier.  |
| identifiedItem                                 | ERTYPEITEM               | The entity or relationship that the identifier identifies.   |
| partial  | boolean                  | Is this a partial (non-unique) identifier?   |
| attributes                                     | list(ERATTRIBUTE)        | A list of attributes that comprise the identifier.   |

the full gamut of object-oriented modelling. In particular, it does not provide any notion of generalisation/specialisation, and no mention is made of composite attributes, that is, attributes that are made up of other attributes or objects. Composite attributes are not excluded by Chen’s definition, however, and have been catered for by the addition of the ERATTRIBUTEITEM and ERATTRIBUTEGROUP constructs. These allow the definition of unnormalised entities. Note that this does not provide ‘object’ capabilities, as an ERATTRIBUTEGROUP is not the same as an EREntityType. It is up to specific representations to define any object extensions to the basic technique.

### D.2.1 Martin ERD definition

The representation  $\mathfrak{R}_e(E-R, ERD_{Martin})$  corresponds to the E-R notation defined by Martin (1990). It extends the E-R technique with the following constructs (shown in Figure D.2):

MARTINREGULAREntity: A specialisation of the EREntityType construct.

MARTINWEAKEntity: A specialisation of the ERWEAKEntityType construct.

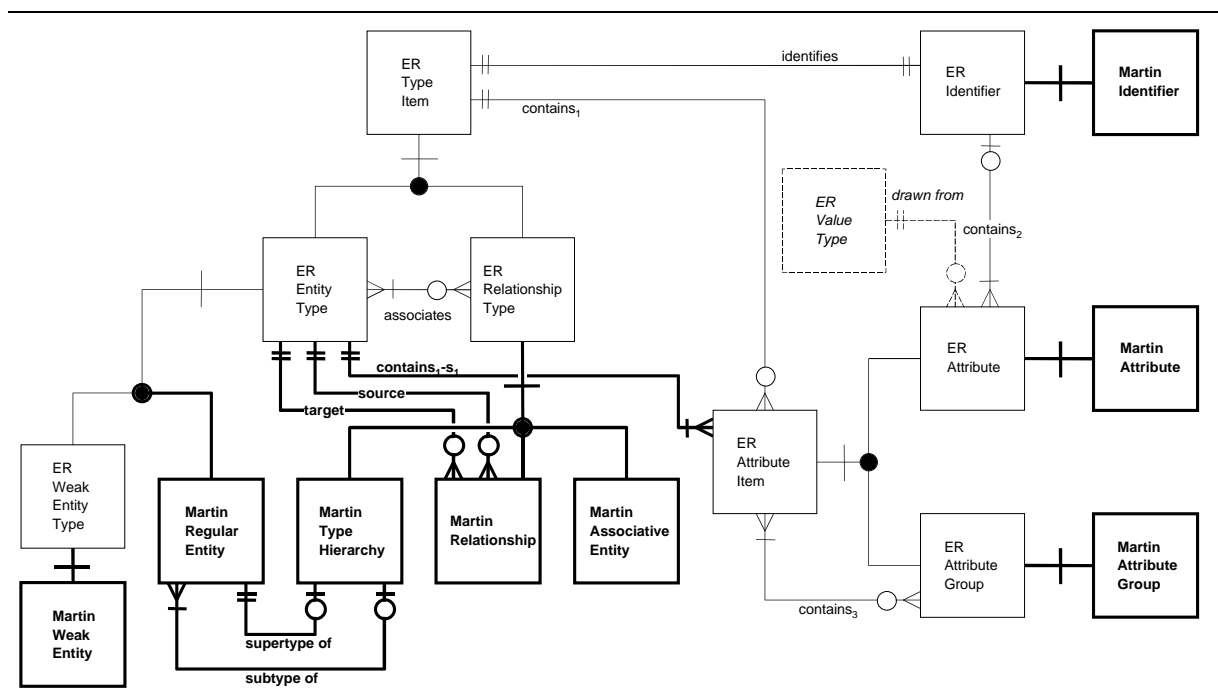


Figure D.2: Definition of the representation  $\mathfrak{R}_e(E-R, ERD_{Martin})$

**MARTINTYPEHIERARCHY:** This is a specialisation of the **ERRELATIONSHIP**TYPE construct that allows the definition of an inheritance hierarchy. It is effectively a generalisation association, which is why it is a specialisation of **ERRELATIONSHIP**TYPE.

**MARTINRELATIONSHIP:** This is a specialisation of the **ERRELATIONSHIP**TYPE construct that specifies a binary association between two entity and/or relationship types that are not **MARTINRELATIONSHIP** elements. An **ERRELATIONSHIP**TYPE may have degree greater than two. A **MARTINRELATIONSHIP** has no attributes.

**MARTINASSOCIATIVEENTITY:** A specialisation of the **ERRELATIONSHIP**TYPE construct.

**MARTINIDENTIFIER:** A specialisation of the **ERIDENTIFIER** construct.

**MARTINATTRIBUTE:** A specialisation of the **ERATTRIBUTE** construct.

**MARTINATTRIBUTEGROUP:** This construct is a specialisation of the **ERATTRIBUTEGROUP** construct.

The properties of these constructs are shown in Table D.2. Note that this representation does not support the **ERVALUE**TYPE construct.

One particular extension to take note of here is that **MARTINWEAKENTITY** constructs may be ‘embedded’ within other entities. This allows the definition of unnormalised entities that have relationships attached to groups of attributes within the entity, rather than directly to the entity itself. An example of a situation where this is useful is the Experiment entity in the agricultural research institute viewpoint described in Appendix C. The disadvantage of this approach is that there are two separate constructs for representing composite attributes: **MARTINATTRIBUTEGROUP** and **MARTINWEAKENTITY**.

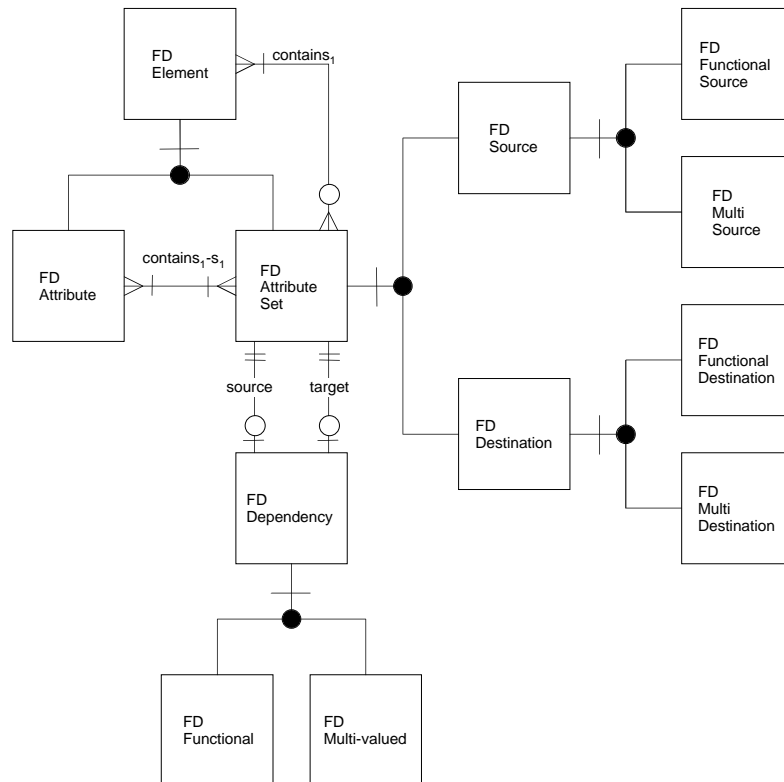
### **D.3 Functional dependency technique**

This technique derives from relational dependency theory (Codd, 1972a; Armstrong, 1974; Beeri et al., 1977), and comprises the base constructs shown in Figure D.3.



**Table D.2:** Construct properties of  $\mathfrak{R}_e(E-R, ERD_{Martin})$

| CONSTRUCT  | 'Data type'               | Description   |
|--|---------------------------|---|
| MARTINREGULARENTITY (specialises EREntityTYPE)           |                           |   |
| typeHierarchy  | MARTINTYPEHIERARCHY       | The type hierarchy in which the entity is the supertype, if applicable. |
| embeddedEntities   | list(MARTINWEAKENTITY)    | A (possibly empty) list of embedded weak entities.                      |
| MARTINWEAKENTITY (specialises ERWEAKENTITYTYPE)          |                           |   |
| embedded   | boolean                   | Is the weak entity embedded within another entity?                      |
| embeddedEntities   | list(MARTINWEAKENTITY)    | A (possibly empty) list of embedded weak entities.                      |
| MARTINASSOCIATIVEENTITY (specialises ERRELATIONSHIPTYPE) |                           |   |
| embeddedEntities   | list(MARTINWEAKENTITY)    | A (possibly empty) list of embedded weak entities.                      |
| MARTINRELATIONSHIP (specialises ERRELATIONSHIPTYPE)      |                           |   |
| source   | ERTYPEITEM                | The 'source' entity of the relationship.                                |
| target   | ERTYPEITEM                | The 'target' entity of the relationship.                                |
| srcCard  | integer                   | The cardinality of the source entity.                                   |
| srcOpt   | integer                   | The optionality of the source entity.                                   |
| dstCard  | integer                   | The cardinality of the target entity.                                   |
| dstOpt   | integer                   | The optionality of the target entity.                                   |
| MARTINTYPEHIERARCHY (specialises ERRELATIONSHIPTYPE)     |                           |   |
| supertype  | MARTINREGULARENTITY       | The supertype regular entity.   |
| subtypes   | list(MARTINREGULARENTITY) | The subtype regular entity.   |
| exclusive  | boolean                   | Are the subtypes mutually exclusive?                                    |
| MARTINATTRIBUTE (specialises ERATTRIBUTE)                |                           |   |
| MARTINATTRIBUTEGROUP (specialises ERATTRIBUTEGROUP)      |                           |   |
| MARTINIDENTIFIER (specialises ERIDENTIFIER)              |                           |   |



**Figure D.3:** Definition of the functional dependency technique

**FDATTRIBUTE:** An attribute of a relation (Date, 1995, p. 271).

**FDATTRIBUTESET:** A set of **FDATTRIBUTE** elements.

**FDDEPENDENCY:** A dependency of some sort between **FDATTRIBUTE** elements.

**FDFUNCTIONAL:** A functional dependency, that is  $A \rightarrow B$ .

**FDMULTIVALUED:** A multivalued dependency, that is  $C \twoheadrightarrow D$ .

These base constructs are augmented with the following:

**FDELEMENT:** This is a generalisation of **FDATTRIBUTE** and **FDATTRIBUTESET**.

**FDSOURCE:** A collection of attributes on the left-hand side of a dependency.

**FDTARGET:** A collection of attributes on the right-hand side of a dependency.

**FDFUNCTIONALSOURCE:** A collection of attributes on the left-hand side of a functional dependency.

**FDFUNCTIONALTARGET:** A collection of attributes on the right-hand side of a functional dependency.

**FDMULTISOURCE:** A collection of attributes on the left-hand side of a multivalued dependency.

**FDMULTITARGET:** A collection of attributes on the right-hand side of a multivalued dependency.

The properties of all these constructs are shown in Table D.3.

An **FDATTRIBUTESET** may contain both **FDATTRIBUTE** elements and other **FDATTRIBUTESET** elements, so the **FDELEMENT** construct has been defined as a generalisation of both the **FDATTRIBUTESET** and **FDATTRIBUTE** constructs in order to simplify the definition.

The **FDATTRIBUTESET** construct has been specialised into attribute sets that appear on the left-hand side of a dependency (**FDSOURCE**) and attribute sets that appear on the right-hand side of a dependency (**FDTARGET**). These have been further specialised into left- and right-hand sides of functional and multivalued dependencies respectively (**FDFUNCTIONALSOURCE**, **FDMULTISOURCE**, **FDFUNCTIONALTARGET**, and **FDMULTITARGET**).

**Table D.3: Construct properties of the functional dependency technique**

| CONSTRUCT<br>Property  | 'Data type'                      | Description  |
|--|----------------------------------|--|
| FDELEMENT<br>name  | string                           | The name of the element.   |
| FDATTRIBUTE (specialises FDELEMENT)<br>attributeSets               | list(FDATTRIBUTESET)             | A list of attribute sets in which the attribute participates.  |
| FDATTRIBUTESET (specialises FDELEMENT)<br>dependency<br>attributes | FDEPENDENCY<br>list(FDATTRIBUTE) | The dependency attached to the attribute set.<br>The attributes that comprise the attribute set.   |
| FDEPENDENCY<br>name<br>source<br>target                            | string<br>FDSOURCE<br>FDTARGET   | The name of the dependency.<br>The attribute set on the left-hand side of the dependency.<br>The attribute set on the right-hand side of the dependency. |
| FDSOURCE (specialises FDATTRIBUTESET)                              |                                  |  |
| FDFUNCTIONALSOURCE (specialises FDSOURCE)                          |                                  |  |
| FDMULTISOURCE (specialises FDSOURCE)                               |                                  |  |
| FDTARGET (specialises FDATTRIBUTESET)                              |                                  |  |
| FDFUNCTIONALTARGET (specialises FDTARGET)                          |                                  |  |
| FDMULTITARGET (specialises FDTARGET)                               |                                  |  |
| FDFUNCTIONAL (specialises FDEPENDENCY)                             |                                  |  |
| FDMULTIVALUED (specialises FDEPENDENCY)                            |                                  |  |

### D.3.1 Smith FDD definition

The representation  $\mathcal{R}_f(\text{FuncDep}, \text{FDD}_{\text{Smith}})$  corresponds to the variant of Smith's dependency notation described in Appendix B. It extends the functional dependency technique with the following constructs (shown in Figure D.4 on the following page):

**SSSINGLEVALUED:** A specialisation of the FDFUNCTIONAL construct.

**SSMULTIVALUED:** A specialisation of the FDMULTIVALUED construct.

**SSSINGLEKEYBUBBLE:** A specialisation of the FDFUNCTIONALSOURCE construct that corresponds to the single-key bubble construct of the notation.

**SSMULTIKEYBUBBLE:** A specialisation of the FDMULTISOURCE construct that corresponds to the multi-key bubble construct of the notation.

**SSTARGETBUBBLE:** A specialisation of the FDFUNCTIONALTARGET construct that corresponds to the target bubble construct of the notation.

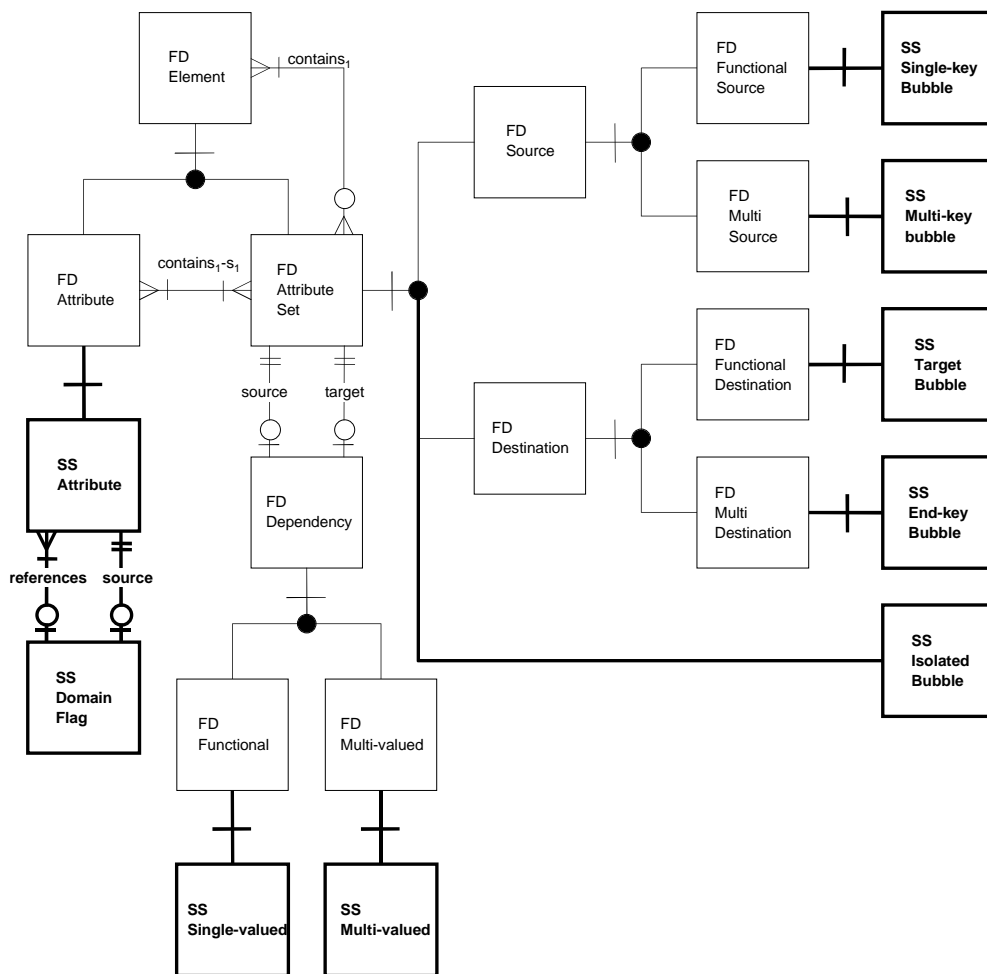
**SSENDKEYBUBBLE:** A specialisation of the FDMULTITARGET construct that corresponds to the end-key bubble construct of the notation.

**SSISOLATEDBUBBLE:** A specialisation of the **FDATTRIBUTESET** construct that corresponds to a set of attributes that participates in no dependencies.

**SSATTRIBUTE:** A specialisation of the **FDATTRIBUTE** construct.

**SSDOMAINFLAG:** This corresponds to the domain flag construct of the notation and represents a domain from which several attributes may be drawn.

The properties of these constructs are shown in Table D.4.



**Figure D.4:** Definition of the representation  $\mathcal{R}_f(\text{FuncDep}, \text{FDD}_{\text{Smith}})$

## D.4 Relational technique

The relational technique is derived from Codd's (1970) original definition of the relational model, now known as *RM/V1*, with a few enhancements from Date (1995).

**Table D.4:** Construct properties of  $\mathfrak{R}_f(\text{FuncDep}, \text{FDD}_{\text{Smith}})$

| CONSTRUCT<br>Property                              | 'Data type'       | Description   |
|--|-------------------|---|
| SSATTRIBUTE (specialises FDATTRIBUTE)              |                   |   |
| domainFlag   | SSDOMAINFLAG      | The domain flag attached to the attribute, if applicable.           |
| SSDOMAINFLAG                                       |                   |   |
| name   | string            | The name of the domain flag.  |
| description  | string            | The description of the domain flag.                                 |
| target   | SSATTRIBUTE       | The attribute that the domain flag references.                      |
| attributes   | list(SSATTRIBUTE) | A list of the other attributes that associate with the domain flag. |
| SSSINGLEKEYBUBBLE (specialises FDFUNCTIONALSOURCE) |                   |   |
| SSMULTIKEYBUBBLE (specialises FDMULTISOURCE)       |                   |   |
| SSENDKEYBUBBLE (specialises FDMULTITARGET)         |                   |   |
| SSTARGETBUBBLE (specialises FDFUNCTIONALTARGET)    |                   |   |
| SSISOLATEDBUBBLE (specialises FDATTRIBUTESET)      |                   |   |
| SSSINGLEVALUED (specialises FDFUNCTIONAL)          |                   |   |
| SSMULTIVALUED (specialises FDMULTIVALUED)          |                   |   |

This technique does not cover the more recent RM/V2 (Codd, 1990), or other variants of the relational model such as RM/T (Codd, 1979), as these have not been generally adopted (Date, 1995). It is expected that additional techniques will be defined to cover these variants of the relational model if required.

The relational technique comprises the following base constructs (Date, 1995), as shown in Figure D.5 on the next page:

**RMDOMAIN:** A named set of scalar values, all of the same type.

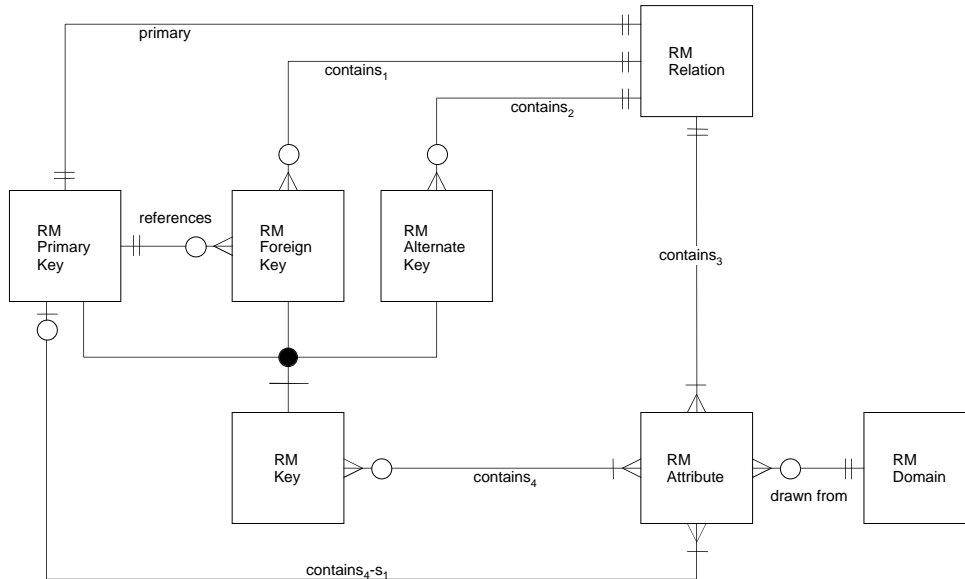
**RMRELATION:** A collection of tuples corresponding to a collection of RMATATTRIBUTE/RMDOMAIN pairs.

**RMATTRIBUTE:** A component of a relation that may hold data values. These values are drawn from a particular RMDOMAIN.

**RMPRIMARYKEY:** The designated unique identifier for an RMRELATION, comprising a collection of RMATATTRIBUTE elements.

**RMFOREIGNKEY:** A set of RMATATTRIBUTE elements that defines a link to the RMPRIMARYKEY of another (or the same) RMRELATION.

**RMALTERNATEKEY:** An alternative unique identifier for an RMRELATION that was not chosen as the RMPRIMARYKEY.



**Figure D.5:** Definition of the relational technique

These base constructs have been augmented with the RMKEY construct, which is a generalisation of the three ‘key’ constructs listed above. The properties of all these constructs are shown in Table D.5.

### D.4.1 SQL/92 definition

The representation  $\mathfrak{R}_r(\textit{Relational}, \textit{SQL}/92)$  corresponds to the SQL language as defined by the SQL/92 standard (ISO-IEC, 1992a; Date and Darwen, 1993). It extends the relational technique with the following constructs (shown in Figure D.6 on page 388):

**SQL92TABLE:** A specialisation of RMRELATION.

**SQL92COLUMN:** A specialisation of RMATTRIBUTE.

**SQL92DOMAIN:** A specialisation of RMDOMAIN. Note that not all SQL92COLUMN elements have a domain, as SQL/92 allows the direct specification of data types for a column.

**SQL92PRIMARYKEY:** A specialisation of RMPRIMARYKEY.

**SQL92FOREIGNKEY:** A specialisation of RMFOREIGNKEY.

**SQL92CONSTRAINT:** A constraint that may be applied to an SQL92COLUMN.

**Table D.5: Construct properties of the relational technique**

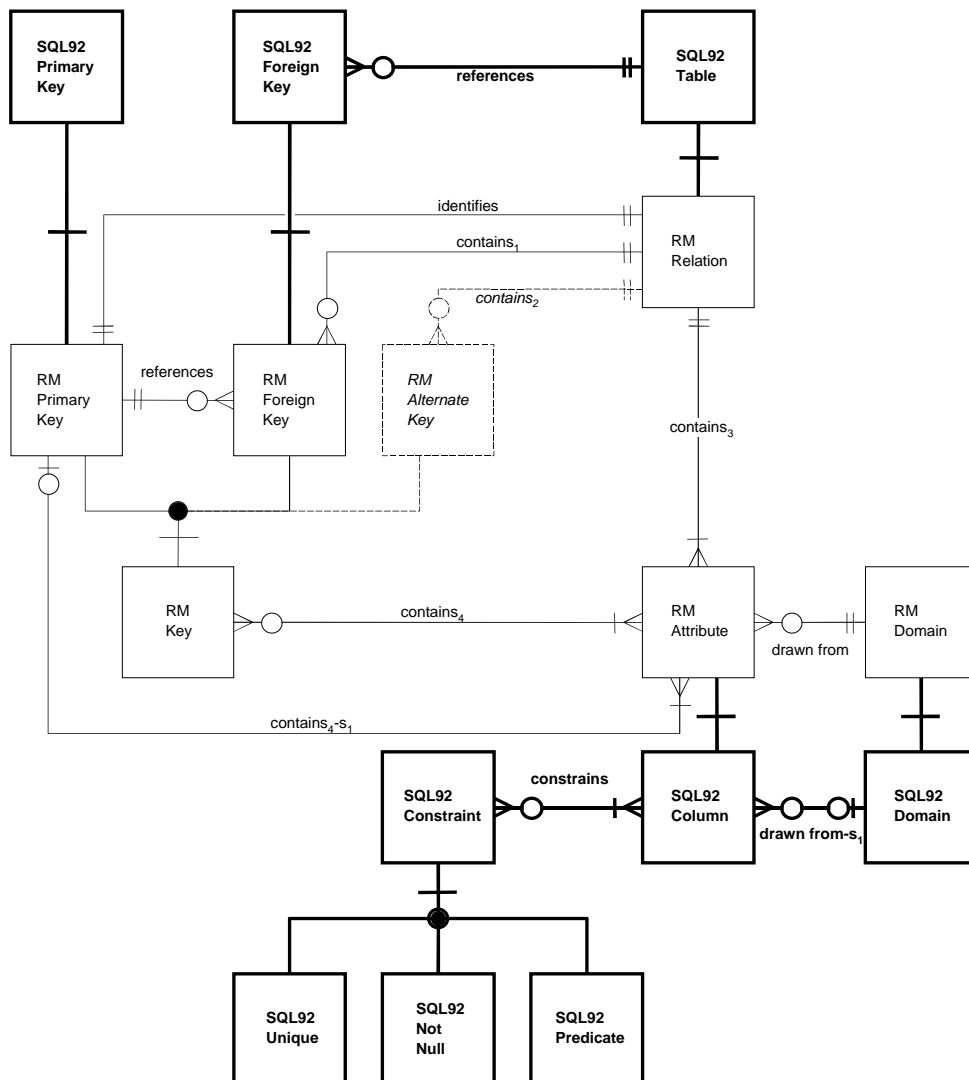
| <b>CONSTRUCT</b>                   | <b>Property</b> | <b>'Data type'</b>   | <b>Description</b>   |
|------------------------------------|-----------------|----------------------|--|
| RMDOMAIN                           | name            | string               | The name of the domain.  |
|                                    | datatype        | enumerator           | The data type of the domain.   |
|                                    | size            | integer              | The size of the values allowed in the domain, if applicable (that is, number of characters or digits). |
|                                    | dp              | integer              | The number of decimal places, if applicable.   |
|                                    | attributes      | list(RMATATTRIBUTE)  | A (possibly empty) list of attributes that are drawn from this domain.                                 |
| RMATTRIBUTE                        | name            | string               | The name of the attribute.   |
|                                    | domain          | RMDOMAIN             | The domain of the attribute.   |
|                                    | relation        | RMRELATION           | The relation that the attribute participates in.   |
|                                    | keys            | list(RMKEY)          | A (possibly empty) list of keys that the attribute participates in.                                    |
| RMRELATION                         | name            | string               | The name of the relation.  |
|                                    | primaryKey      | RMPRIMARYKEY         | The primary key of the relation.   |
|                                    | attributes      | list(RMATATTRIBUTE)  | A list of one or more attributes that comprise the relation.   |
|                                    | alternateKeys   | list(RMALTERNATEKEY) | A (possibly empty) list of alternate keys for the relation.  |
|                                    | foreignKeys     | list(RMFOREIGNKEY)   | A (possibly empty) list of foreign keys for the relation.  |
| RMKEY                              | name            | string               | The name of the key.   |
|                                    | relation        | RMRELATION           | The relation in which the key participates.  |
|                                    | attributes      | list(RMATATTRIBUTE)  | A list of one or more attributes that comprise the key.  |
| RMPRIMARYKEY (specialises RMKEY)   |                 |                      |  |
|                                    | refFKs          | list(RMFOREIGNKEY)   | A (possibly empty) list of foreign keys that reference the primary key.                                |
| RMFOREIGNKEY (specialises RMKEY)   |                 |                      |  |
|                                    | refPK           | RMPRIMARYKEY         | The primary key that the foreign key references.   |
| RMALTERNATEKEY (specialises RMKEY) |                 |                      |  |

**SQL92UNIQUE:** A specialisation of **SQL92CONSTRAINT**, corresponding to an SQL unique constraint.

**SQL92NOTNULL:** A specialisation of **SQL92CONSTRAINT**, corresponding to an SQL not null constraint.

**SQL92PREDICATE:** A specialisation of **SQL92CONSTRAINT** that corresponds to a general SQL constraint based on a predicate.

The properties of all these constructs are shown in Table D.6. Note that the **RMALTER-NATEKEY** construct is not used in this representation.



**Figure D.6:** Definition of the representation  $\mathfrak{R}_r(\text{Relational}, \text{SQL}/92)$



**Table D.6:** Construct properties of  $\mathfrak{R}_r$  (*Relational, SQL/92*)

| CONSTRUCT<br>Property                        | 'Data type'           | Description  |
|--|-----------------------|--|
| SQL92COLUMN (specialises RATTRIBUTE)         |                       |  |
| datatype                                     | enumerator            | The data type of the column.   |
| size   | integer               | The size of the values allowed in the column, if applicable (that is, number of characters or digits). |
| dp   | integer               | The number of decimal places, if applicable.   |
| constraints                                  | list(SQL92CONSTRAINT) | A (possibly empty) list of constraints in which the column appears.                                    |
| SQL92CONSTRAINT                              |                       |  |
| name   | string                | The name of the constraint.  |
| columns                                      | list(SQL92COLUMN)     | A list of columns that appear in the constraint.   |
| SQL92PREDICATE (specialises SQL92CONSTRAINT) |                       |  |
| predicateString                              | string                | The predicate for the constraint.  |
| SQL92TABLE (specialises RMRELATION)          |                       |  |
| refFKs                                       | list(SQL92FOREIGNKEY) | A (possibly empty) list of foreign keys that reference the table.                                      |
| SQL92FOREIGNKEY (specialises RMFOREIGNKEY)   |                       |  |
| refTable                                     | SQL92TABLE            | The table that the foreign key references.   |
| SQL92DOMAIN (specialises RMDOMAIN)           |                       |  |
| SQL92NOTNULL (specialises SQL92CONSTRAINT)   |                       |  |
| SQL92UNIQUE (specialises SQL92CONSTRAINT)    |                       |  |
| SQL92PRIMARYKEY (specialises RMPRIMARYKEY)   |                       |  |

## D.5 Data flow modelling technique

The data flow modelling technique is derived from data flow analysis using data flow diagrams (Gane and Sarson, 1979). It comprises the following base constructs (shown in Figure D.7 on the next page):

**DFFIELD:** A data field, such as a name, age or price.

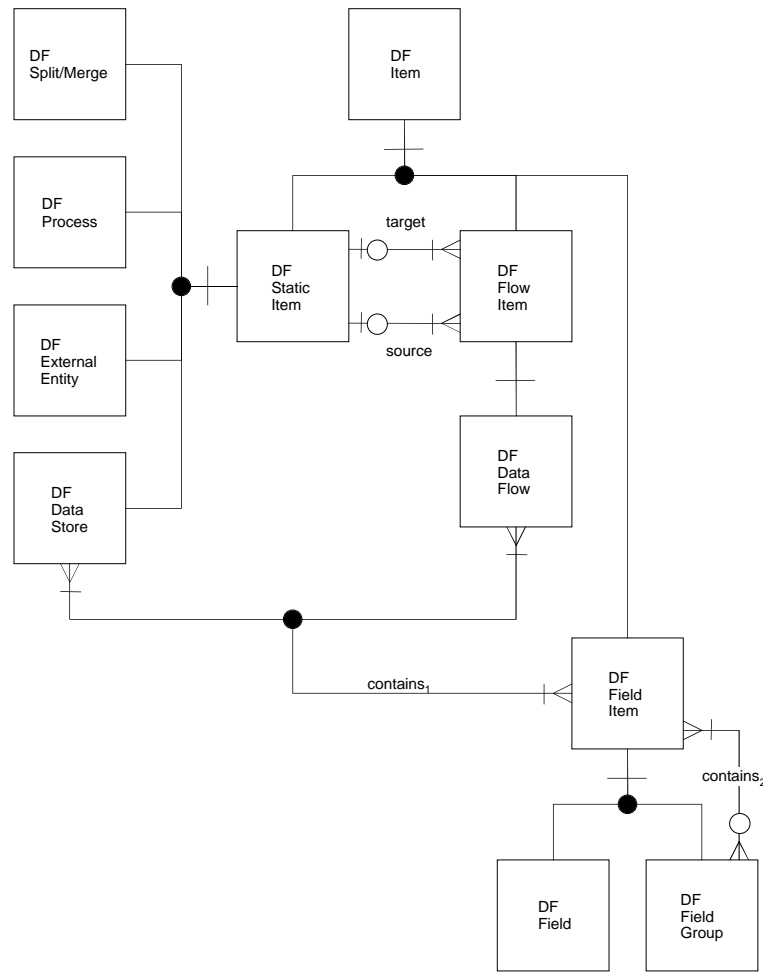
**DFFIELDGROUP:** A collection of data fields. **DFFIELDGROUP** elements may be nested within each other to produce an unnormalised structure.

**DFEXTERNALENTITY:** An external entity represents an object that is external to the viewpoint but acts as a source or target for data contained within the viewpoint.

**DFDATAFLOW:** The flow of data from a source to a destination.

**DFDATASTORE:** A place in which related data may be stored, such as a card file, relation or indexed data file.

**DFPROCESS:** A procedure that manipulates data in some way.



**Figure D.7:** Definition of the data flow modelling technique

**DFSPLITMERGE:** This allows a DFDATAFLOW element to be replicated into multiple identical DFDATAFLOW elements, or vice versa.

These base constructs have been augmented with the following:

**DFITEM:** This construct is a generalisation of the DFSTATICITEM, DFFLOWITEM and DFFIELDITEM constructs.

**DFSTATICITEM:** This is a generalisation of the DFDATASTORE, DFEXTERNALENTITY, DFPROCESS and DFSPLITMERGE constructs.

**DFFLOWITEM:** This is a generalisation of the DFDATAFLOW construct.

**DFFIELDITEM:** This is a generalisation of DFFIELD and DFFIELDGROUP.

The properties of all these constructs are shown in Table D.7.

**Table D.7: Construct properties of the data flow modelling technique**

| CONSTRUCT<br>Property  | 'Data type'                          | Description  |
|--|--------------------------------------|--|
| DFITEM<br>name   | string                               | The name of the element.   |
| DFFLOWITEM<br>source<br>destination  | DFSTATICITEM<br>DFSTATICITEM         | The source element of the flow.<br>The target element of the flow.   |
| DFDATAFLOW (specialises DFFLOWITEM)<br>fieldItems  | list(DFFIELDITEM)                    | The data that comprise the data flow.  |
| DFSTATICITEM<br>label<br>flows   | string<br>list(DFFLOWITEM)           | The label of the element, for example, 'P1' for a process.<br>A list of one or more attached flows.  |
| DFDATASTORE (specialises DFSTATICITEM)<br>fieldItems   | list(DFFIELDITEM)                    | The data that comprise the data store.   |
| DFFIELDITEM (specialises DFITEM)<br>containingItem<br>fieldGroups  | DFITEM<br>list(DFFIELDGROUP)         | The element that contains the field or field group.<br>A (possibly empty) list of field groups in which the element participates.  |
| DFFIELD (specialises DFFIELDITEM)<br>datatype<br>size<br><br>dp  | enumerator<br>integer<br><br>integer | The data type of the field.<br>The size of the values allowed in the field, if applicable (that is, number of characters or digits).<br>The number of decimal places, if applicable. |
| DFFIELDGROUP (specialises DFFIELDITEM)<br>fieldElements  | list(DFFIELDITEM)                    | A list of one or more fields/field groups that comprise the field group.   |
| DFPROCESS (specialises DFSTATICITEM)<br>DFSPLITMERGE (specialises DFSTATICITEM)<br>DFEXTERNALENTITY (specialises DFSTATICITEM) |                                      |  |

The DFFLOWITEM construct has been introduced as a generalisation of DFDATAFLOW because some data flow representations define additional types of flow. The DFFIELDITEM construct generalises both DFFIELD and DFFIELDGROUP, to allow DFFIELDGROUP elements to contain both DFFIELD elements and other DFFIELDGROUP elements. DFITEM is introduced as a generalisation of DFSTATICITEM, DFFLOWITEM and DFFIELDITEM for convenience.

### D.5.1 Gane & Sarson DFD definition

The representation  $\mathfrak{R}_d(DataFlow, DFD_{G\&S})$  corresponds to the data flow diagramming notation defined by Gane and Sarson (1979). It extends the data flow modelling technique with the following constructs (shown in Figure D.8 on the following page):

**GNSDATASTORE:** A specialisation of DFDATASTORE.

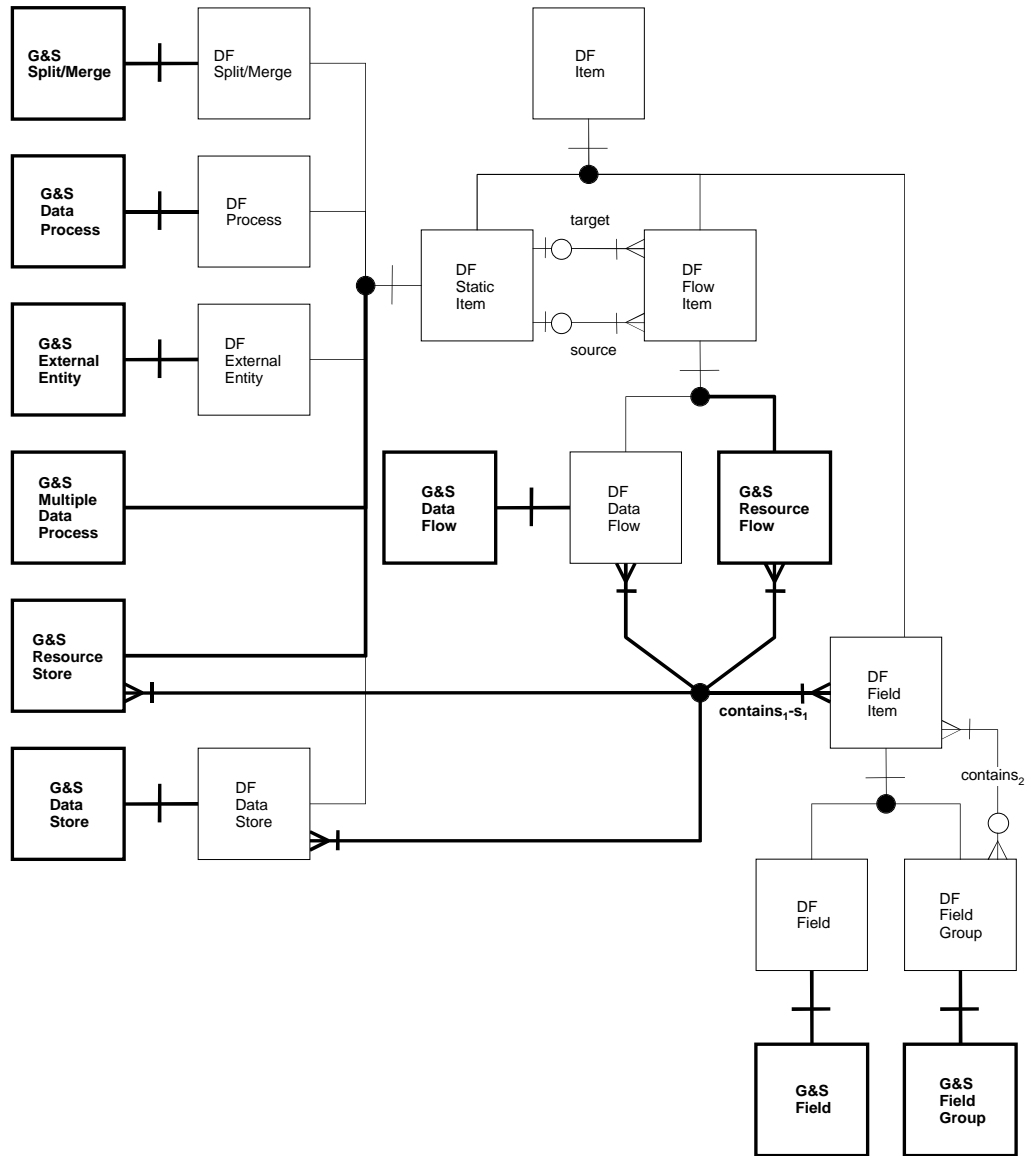


Figure D.8: Definition of the representation  $\mathfrak{R}_d(DataFlow, DFD_{G\&S})$

GNSDATAPROCESS: A specialisation of DFPROCESS.

GNSEXTERNALENTITY: A specialisation of DFEXTERNALENTITY.

GNSPLITMERGE: A specialisation of DFSPLITMERGE.

GNSDATAFLOW: A specialisation of DFDATAFLOW.

GNSFIELD: A specialisation of DFFIELD.

GNSFIELDGROUP: A specialisation of DFFIELDGROUP.

GNSRESOURCESTORE: A place in which physical objects may be stored, as opposed to data. A specialisation of DFSTATICITEM.

GNSMULTIPLEDATAPROCESS: A process that may be executed multiple times in parallel. A specialisation of DFSTATICITEM.

GNSRESOURCEFLOW: A flow that indicates the movement of physical objects, as opposed to data.

The properties of all these constructs are shown in Table D.8.

**Table D.8:** Construct properties of  $\mathfrak{R}_d(\text{DataFlow}, \text{DFD}_{G\&S})$

---

| CONSTRUCT   | Property   | 'Data type'       | Description   |
|---|------------|-------------------|---|
| GNSRESOURCEFLOW (specialises PMFLOWITEM)          | fieldItems | list(DFFIELDITEM) | A list of one or more field items that comprise the resource flow.  |
| GNSRESOURCESTORE (specialises PMSTATICITEM)       | fieldItems | list(DFFIELDITEM) | A list of one or more field items that comprise the resource store. |
| GNSDATAFLOW (specialises PMDATAFLOW)              |            |                   |   |
| GNSDATAPROCESS (specialises PMPROCESS)            |            |                   |   |
| GNSPLITMERGE (specialises PMSPLITMERGE)           |            |                   |   |
| GNSDATASTORE (specialises PMDATASTORE)            |            |                   |   |
| GNSEXTERNALENTITY (specialises PMEXTERNALENTITY)  |            |                   |   |
| GNSFIELD (specialises PMFIELD)                    |            |                   |   |
| GNSFIELDGROUP (specialises PMFIELDGROUP)          |            |                   |   |
| GNSMULTIPLEDATAPROCESS (specialises PMSTATICITEM) |            |                   |   |

---



# Appendix E

## Additional translations

This appendix contains full definitions of the translations  $\mathfrak{R}_f(\text{FuncDep}, \text{FDD}_{\text{Smith}}) \rightarrow \mathfrak{R}_e(\text{E-R}, \text{ERD}_{\text{Martin}})/\mathfrak{R}_f \leftarrow \mathfrak{R}_e$  and  $\mathfrak{R}_e(\text{E-R}, \text{ERD}_{\text{Martin}}) \rightleftharpoons \mathfrak{R}_d(\text{DataFlow}, \text{DFD}_{\text{G\&S}})$ . Discussion of the rules presented here may be found in Chapter 5. A full relative quality analysis for each translation is also presented here.

### E.1 $\mathfrak{R}_f \rightarrow \mathfrak{R}_e/\mathfrak{R}_f \leftarrow \mathfrak{R}_e$

In this translation, descriptions are translated between functional dependencies expressed using a functional dependency diagram in Smith notation and the entity-relationship approach expressed using Martin notation. This translation is complete in the forward direction and partial in the reverse direction. That is:

$$\begin{aligned} \mathfrak{R}_f(\text{FuncDep}, \text{FDD}_{\text{Smith}}) &\rightarrow \mathfrak{R}_e(\text{E-R}, \text{ERD}_{\text{Martin}}) \text{ and} \\ \mathfrak{R}_f(\text{FuncDep}, \text{FDD}_{\text{Smith}}) &\leftarrow \mathfrak{R}_e(\text{E-R}, \text{ERD}_{\text{Martin}}). \end{aligned}$$

Many of the rules of this translation are derived from a modified version of Smith's method for deriving a set of normalised relations from a functional dependency diagram (Smith, 1985, see also Appendix B).

In the forward direction, the input to this translation is a Smith functional dependency diagram with no dependency chains, that is, each bubble has only a single dependency attached to it. The output of the translation is a normalised ERD in at least fourth normal form. There are no restrictions in the reverse direction.

### E.1.1 Technique-level rules

$$\begin{aligned}
 \mathfrak{R}_f[\text{FDFUNCTIONALSOURCE}, \text{FDFUNCTIONAL}, \\
 \text{FDFUNCTIONALTARGET}] &\rightarrow \mathfrak{R}_e[\text{ERENTITYTYPE}] \\
 \mathfrak{R}_f[\text{FDFUNCTIONALSOURCE}, \text{FDFUNCTIONAL}, \\
 \text{FDFUNCTIONALTARGET}] &\leftarrow \mathfrak{R}_e[\text{ERENTITYTYPE}]
 \end{aligned}
 \tag{T1}$$

The functional dependency represented by the constructs on the left-hand side of the rule translates completely to an entity type of some sort. Thus, the functional dependency  $A \rightarrow BC$  translates to an entity type with attributes  $\{A, B, C\}$ . In the reverse direction, at least one of the entity type's attributes must not participate in the entity type's identifier.

$$\begin{aligned}
 \mathfrak{R}_f[\text{FDMULTISOURCE}, \text{FDMULTIVALUED}, \text{FDMULTITARGET}] &\rightarrow \\
 \mathfrak{R}_e[\text{ERENTITYTYPE}, \text{ERIDENTIFIER}] & \\
 \mathfrak{R}_f[\text{FDMULTISOURCE}, \text{FDMULTIVALUED}, \text{FDMULTITARGET}] &\leftarrow \\
 \mathfrak{R}_e[\text{ERENTITYTYPE}, \text{ERIDENTIFIER}] &
 \end{aligned}
 \tag{T2}$$

The multi-valued dependency represented by the constructs on the left-hand side of the rule translates completely to an entity type of some sort. The identifier of this entity type is determined by concatenating the attributes on both sides of the multivalued dependency (that is, the entity identifier comprises all the attributes of the entity type). The reverse translation is partial, and only applies when there are at most two attributes in the entity type, and there are no non-identifying attributes.

$$\begin{aligned}
 \mathfrak{R}_f[\text{FDATTRIBUTE}] &\rightarrow \mathfrak{R}_e[\text{ERATTRIBUTE}] \\
 \mathfrak{R}_f[\text{FDATTRIBUTE}] &\leftarrow \mathfrak{R}_e[\text{ERATTRIBUTE}]
 \end{aligned}
 \tag{T3}$$

An FD attribute translates completely to an E-R attribute, but only partially in the reverse direction.



$$\begin{aligned}
& \mathfrak{R}_f[\text{FDFUNCTIONALSOURCE}_1, \text{FDFUNCTIONALSOURCE}_2, \\
& \quad \text{FDFUNCTIONAL}_1, \text{FDFUNCTIONAL}_2, \\
& \quad \text{FDFUNCTIONALTARGET}_1, \text{FDFUNCTIONALTARGET}_2] \rightarrow \\
& \quad \mathfrak{R}_e[\text{ERRELATIONSHIPTYPE}, \text{ERENTITYTYPE}_1, \text{ERENTITYTYPE}_2] \\
& \mathfrak{R}_f[\text{FDFUNCTIONALSOURCE}_1, \text{FDFUNCTIONALSOURCE}_2, \\
& \quad \text{FDFUNCTIONAL}_1, \text{FDFUNCTIONAL}_2, \\
& \quad \text{FDFUNCTIONALTARGET}_1, \text{FDFUNCTIONALTARGET}_2] \leftarrow \\
& \quad \mathfrak{R}_e[\text{ERRELATIONSHIPTYPE}, \text{ERENTITYTYPE}_1, \text{ERENTITYTYPE}_2]
\end{aligned} \tag{T4}$$

The pair of dependencies  $A \rightarrow BPQ$  (the FD constructs with subscript ‘1’ above) and  $PQ \rightarrow RA$  (the FD constructs with subscript ‘2’ above) translate completely to a pair of entity types that are connected by a relationship type with no attributes. In effect,  $PQ$  and  $A$  act as links between the two entity types. The optionalities of the relationship element are indeterminate in this rule — specialisations of this rule must define them appropriately.

$$\begin{aligned}
& \mathfrak{R}_f[\text{FDFUNCTIONALSOURCE}_1, \text{FDFUNCTIONALSOURCE}_2, \text{FDFUNCTIONAL}_1, \\
& \quad \text{FDFUNCTIONAL}_2, \text{FDFUNCTIONALTARGET}_1, \text{FDFUNCTIONALTARGET}_2, \\
& \quad \text{FDMULTISOURCE}_1, \text{FDMULTISOURCE}_2, \text{FDMULTIVALUED}_1, \\
& \quad \text{FDMULTIVALUED}_2, \text{FDMULTITARGET}_1, \text{FDMULTITARGET}_2] \leftarrow \\
& \quad \mathfrak{R}_e[\text{ERRELATIONSHIPTYPE}, \text{ERENTITYTYPE}_1, \text{ERENTITYTYPE}_2]
\end{aligned} \tag{T5}$$

A many-to-many relationship type between two entity types translates partially to a set of dependencies similar to the following:

$$\begin{aligned}
& A \rightarrow BC \quad (\text{from ERENTITYTYPE}_1) \\
& PQ \rightarrow R \quad (\text{from ERENTITYTYPE}_2) \\
& \left. \begin{array}{l} A \twoheadrightarrow PQ \\ PQ \twoheadrightarrow A \end{array} \right\} (\text{from ERRELATIONSHIPTYPE})
\end{aligned}$$

$$\begin{aligned} \mathfrak{R}_f[\text{FDFUNCTIONALSOURCE}] &\rightarrow \mathfrak{R}_e[\text{ERIDENTIFIER}] \\ \mathfrak{R}_f[\text{FDFUNCTIONALSOURCE}] &\leftarrow \mathfrak{R}_e[\text{ERIDENTIFIER}] \end{aligned} \quad (\text{T6})$$

The left-hand side of a functional dependency translates completely to a non-partial entity identifier, but only partially in the reverse direction.

$$\begin{aligned} \mathfrak{R}_f[\text{FDFUNCTIONALSOURCE}, \text{FDFUNCTIONAL}, \text{FDFUNCTIONALTARGET}] &\leftarrow \\ \mathfrak{R}_e[\text{ERWEAKENTITYTYPE}, \text{ERRELATIONSHIPTYPE}, \text{ERENTITYTYPE}] & \end{aligned} \quad (\text{T7})$$

A weak entity type that is dependent on another entity type via a relationship type translates partially to a functional dependency. The exact nature of the dependency depends on whether or not the identifier of the weak entity type is partial or not, and also on how many relationships are attached to the weak entity type. These conditions are specified by specialisations of this rule.

### E.1.2 Scheme-level rules

$$\begin{aligned} \mathfrak{R}_f[\text{SSSINGLEKEYBUBBLE}, \text{SSSINGLEVALUED}, \\ \text{SSTARGETBUBBLE}] &\rightarrow \mathfrak{R}_e[\text{MARTINREGULARENTITY}] \\ \mathfrak{R}_f[\text{SSSINGLEKEYBUBBLE}, \text{SSSINGLEVALUED}, \\ \text{SSTARGETBUBBLE}] &\leftarrow \mathfrak{R}_e[\text{MARTINREGULARENTITY}] \end{aligned} \quad (\text{S1})$$

This rule is a specialisation of technique-level rule T1, and translates a functional dependency into a regular entity, as illustrated in Figure E.1.



**Figure E.1:** Translating a single-valued dependency to and from a Martin ERD (rule S1)

$$\begin{aligned} \mathfrak{R}_f[\text{SSMULTIKEYBUBBLE}, \text{SSMULTIVALUED}, \text{SENDKEYBUBBLE}] &\rightarrow \\ \mathfrak{R}_e[\text{MARTINREGULARENTITY}, \text{MARTINIDENTIFIER}] & \\ \mathfrak{R}_f[\text{SSMULTIKEYBUBBLE}, \text{SSMULTIVALUED}, \text{SENDKEYBUBBLE}] &\leftarrow \\ \mathfrak{R}_e[\text{MARTINREGULARENTITY}, \text{MARTINIDENTIFIER}] & \end{aligned} \quad (\text{S2})$$

This rule is a specialisation of technique-level rule T2, and translates a multi-valued dependency into a regular entity and an associated entity identifier, as illustrated in Figure E.2. The entity must have no non-key attributes, so the attributes of the regular entity must be the same as those of the identifier.



**Figure E.2:** Translating a multivalued dependency to and from a Martin ERD (rule S2)

$$\begin{aligned} \mathfrak{R}_f [\text{SSSINGLEKEYBUBBLE}] &\rightarrow \mathfrak{R}_e [\text{MARTINIDENTIFIER}] \\ \mathfrak{R}_f [\text{SSSINGLEKEYBUBBLE}] &\leftarrow \mathfrak{R}_e [\text{MARTINIDENTIFIER}] \end{aligned} \tag{S3}$$

A single-key bubble translates completely to a non-partial entity identifier (STAFF\_ID in Figure E.3), but only partially in the reverse direction. This rule is a specialisation of technique-level rule T6.



**Figure E.3:** Translating a single-key bubble between an FDD and an ERD (rule S3)

$$\begin{aligned} \mathfrak{R}_f [\text{SSISOLATEDBUBBLE}] &\rightarrow \mathfrak{R}_e [\text{MARTINREGULARENTITY}, \text{MARTINIDENTIFIER}] \\ \mathfrak{R}_f [\text{SSISOLATEDBUBBLE}] &\leftarrow \mathfrak{R}_e [\text{MARTINREGULARENTITY}, \text{MARTINIDENTIFIER}] \end{aligned} \tag{S4}$$

An isolated bubble translates completely to a regular entity (Order\_Ship in Figure E.4 — note that this example is not drawn from one of the example viewpoints). The entity has no non-key attributes. The reverse translation is partial, and only applies when the number of attributes in the entity is not two (see Section 5.4 on page 135).

$$\begin{aligned} \mathfrak{R}_f [\text{SSATTRIBUTE}] &\rightarrow \mathfrak{R}_e [\text{MARTINATTRIBUTE}] \\ \mathfrak{R}_f [\text{SSATTRIBUTE}] &\leftarrow \mathfrak{R}_e [\text{MARTINATTRIBUTE}] \end{aligned} \tag{S5}$$

This rule is a specialisation of technique-level rule T3, and is effectively identical to that rule.



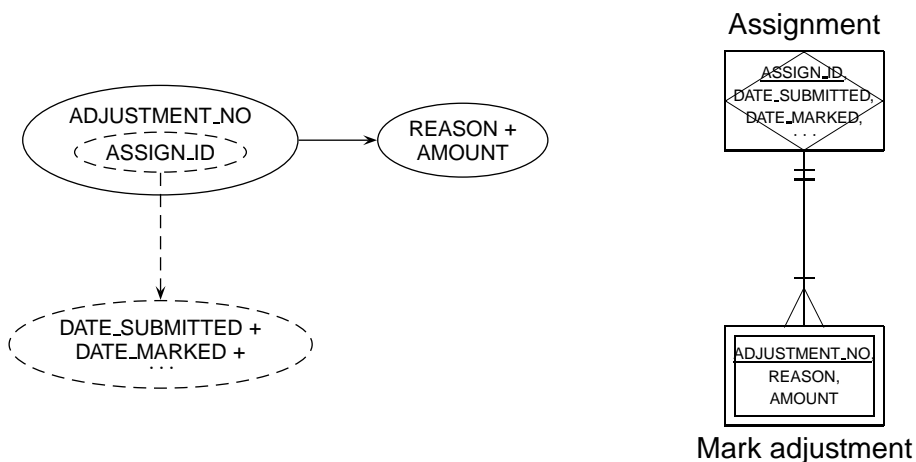
**Figure E.4:** Translating an isolated bubble to and from a Martin ERD (rule S4)

$$\mathfrak{R}_f[\text{SSSINGLEKEYBUBBLE}, \text{SSSINGLEVALUED}, \text{SSTARGETBUBBLE}] \leftarrow \mathfrak{R}_e[\text{MARTINASSOCIATIVEENTITY}] \quad (\text{S6})$$

An associative entity translates partially to a functional dependency (cf. rule S1).

$$\mathfrak{R}_f[\text{SSSINGLEKEYBUBBLE}, \text{SSSINGLEVALUED}, \text{SSTARGETBUBBLE}] \leftarrow \mathfrak{R}_e[\text{MARTINWEAKENTITY}, \text{MARTINRELATIONSHIP}, \text{ERTYPEITEM}] \quad (\text{S7})$$

This rule is a specialisation of technique-level rule T1. The weak entity (Mark Adjustment in Figure E.5) may be embedded or non-embedded, has no associated relationships other than the dependent relationship to the ‘parent’ entity (Assignment), and has an identifier (partial or unique). The weak entity translates to a single-valued dependency as shown in Figure E.5. The attributes of the single-key bubble are generated by concatenating the identifying attributes of both the weak entity and the parent entity. The attributes of the target bubble correspond to the non-identifying attributes of the weak entity.

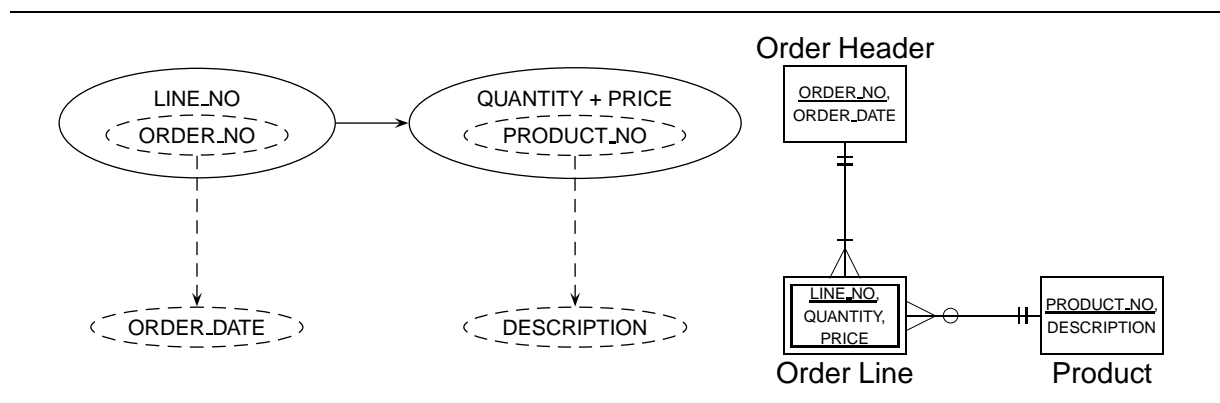


**Figure E.5:** Translating a weak entity to an FDD (rule S7)

$$\mathfrak{R}_f[\text{SSSINGLEKEYBUBBLE}, \text{SSSINGLEVALUED}, \text{SSTARGETBUBBLE}] \leftarrow \quad (\text{S8})$$

$$\mathfrak{R}_e[\text{MARTINWEAKENTITY}, \text{MARTINRELATIONSHIP}, \text{ERTYPEITEM}]$$

This rule is identical to rule S7, except that the weak entity (Order Line in Figure E.6) has one or more associated relationships. The identifying attributes of the attached entities are included in the target bubble. The only restriction on the relationships is that the end attached to the associated entity may not be ‘many’. (Note that the example in Figure E.6 is not drawn from the example viewpoints in Appendix C.)

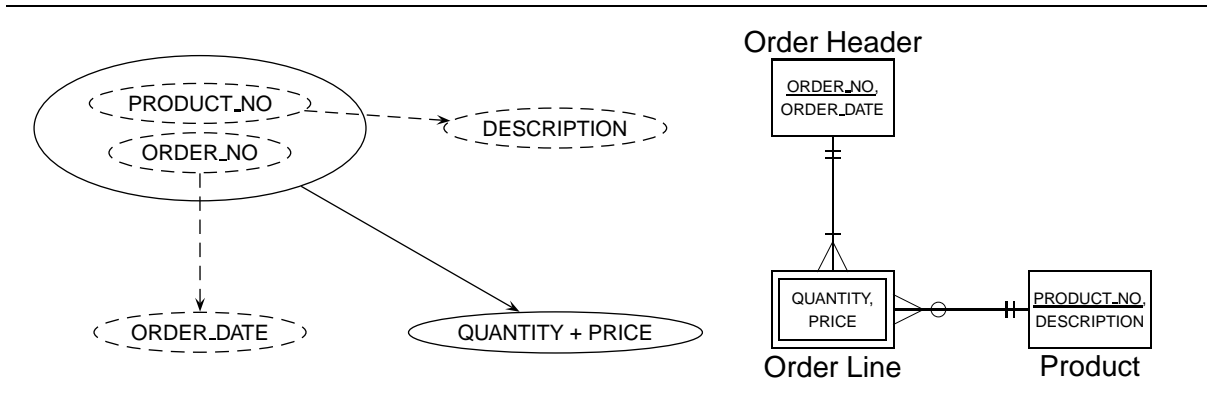


**Figure E.6:** Translating a weak entity to an FDD (rule S8)

$$\mathfrak{R}_f[\text{SSSINGLEKEYBUBBLE}, \text{SSSINGLEVALUED}, \text{SSTARGETBUBBLE}] \leftarrow \quad (\text{S9})$$

$$\mathfrak{R}_e[\text{MARTINWEAKENTITY}, \text{MARTINRELATIONSHIP}, \text{ERTYPEITEM}]$$

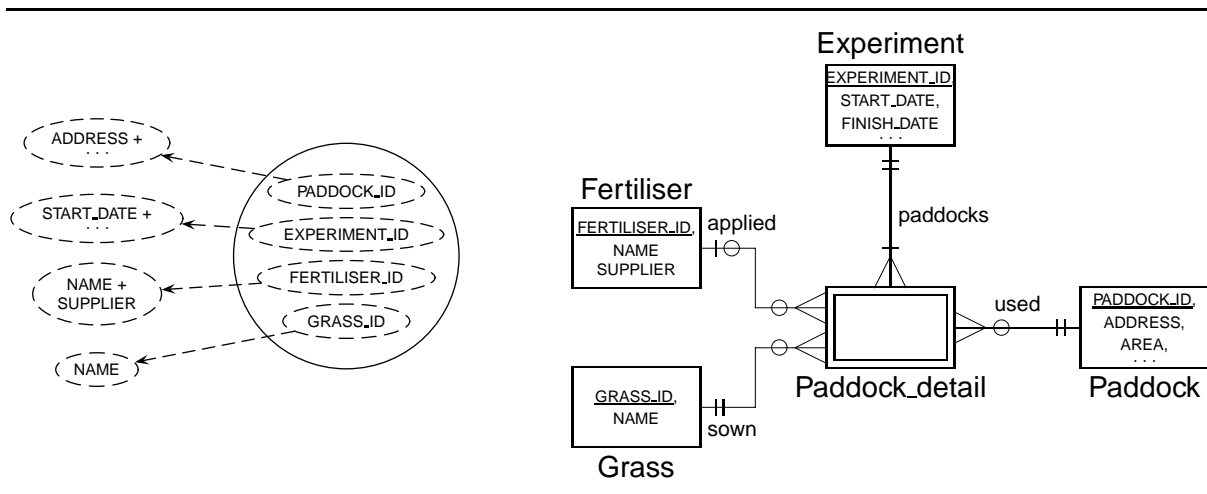
This rule is identical to rule S8, except that the weak entity has no identifier. The identifying attributes of the associated entities are included in the single-key bubble. The use of this rule is illustrated in Figure E.7 on the following page (this example is a modification of that shown in Figure E.6).



**Figure E.7:** Translating a weak entity to an FDD (rule S9)

$$\mathfrak{R}_f[\text{SSISOLATEDBUBBLE}] \leftarrow \mathfrak{R}_e[\text{MARTINWEAKENTITY}, \text{MARTINRELATIONSHIP}, \text{ERTYPEITEM}] \quad (\text{S10})$$

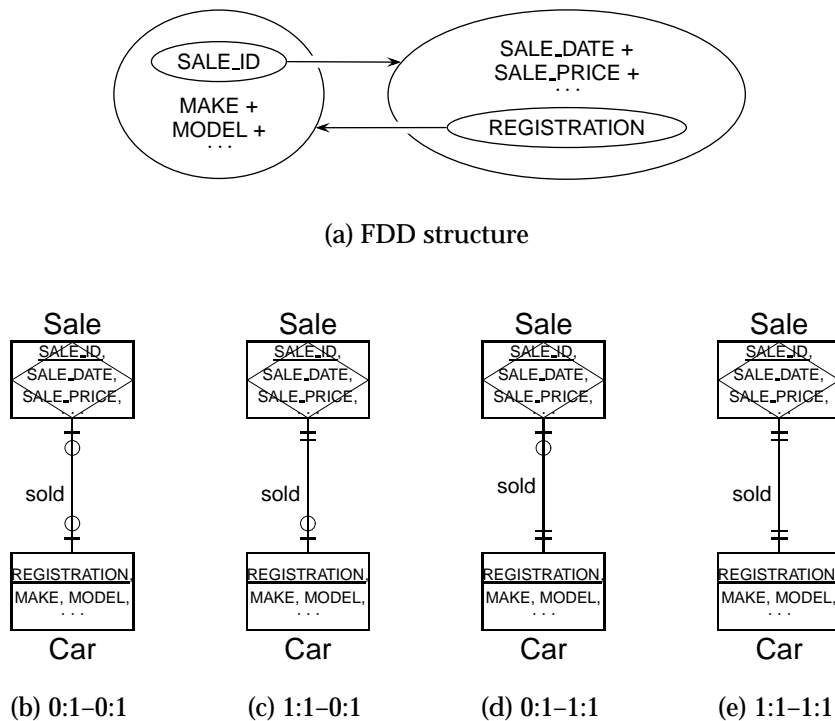
This rule is similar to rule S9, except that the weak entity contains no independent attributes, that is, the only ‘attributes’ are those implied by the attached relationships. This structure translates to an isolated bubble, as shown in Figure E.8. The identifying attributes of the associated entities are included in the isolated bubble.



**Figure E.8:** Translating a weak entity to an FDD (rule S10)

$$\begin{aligned}
& \mathfrak{R}_f[\text{SSSINGLEKEYBUBBLE}_1, \text{SSSINGLEKEYBUBBLE}_2, \text{FDSINGLEVALUED}_1, \\
& \quad \text{FDSINGLEVALUED}_2, \text{SSTARGETBUBBLE}_1, \text{SSTARGETBUBBLE}_2] \rightarrow \\
& \quad \mathfrak{R}_e[\text{MARTINRELATIONSHIP}, \text{ERTYPEITEM}_1, \text{ERTYPEITEM}_2] \\
& \mathfrak{R}_f[\text{SSSINGLEKEYBUBBLE}_1, \text{SSSINGLEKEYBUBBLE}_2, \text{FDSINGLEVALUED}_1, \\
& \quad \text{FDSINGLEVALUED}_2, \text{SSTARGETBUBBLE}_1, \text{SSTARGETBUBBLE}_2] \leftarrow \\
& \quad \mathfrak{R}_e[\text{MARTINRELATIONSHIP}, \text{ERTYPEITEM}_1, \text{ERTYPEITEM}_2]
\end{aligned} \tag{S11}$$

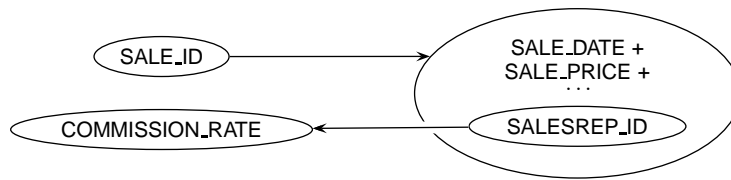
This rule is a specialisation of technique-level rule T4. The structure shown in Figure E.9(a) translates completely to the structure shown in Figure E.9(e). In the reverse direction, the four E-R structures shown in Figures E.9(b)–E.9(e) all map partially to the FDD structure shown.



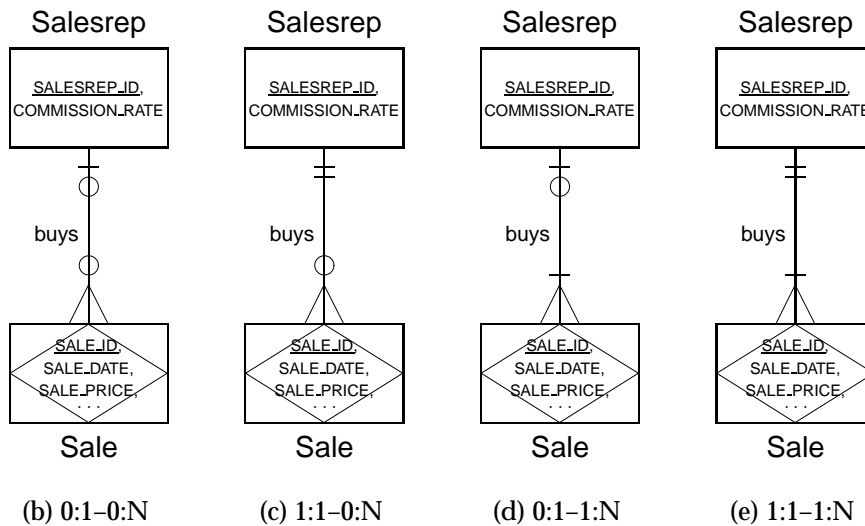
**Figure E.9:** Translating one-to-one relationships between an ERD and an FDD (rule S11)

$$\begin{aligned}
& \mathfrak{R}_f[\text{SSSINGLEKEYBUBBLE}_1, \text{SSSINGLEKEYBUBBLE}_2, \text{FDSINGLEVALUED}_1, \\
& \quad \text{FDSINGLEVALUED}_2, \text{SSTARGETBUBBLE}_1, \text{SSTARGETBUBBLE}_2] \rightarrow \\
& \quad \mathfrak{R}_e[\text{MARTINRELATIONSHIP}, \text{ERTYPEITEM}_1, \text{ERTYPEITEM}_2] \tag{S12} \\
& \mathfrak{R}_f[\text{SSSINGLEKEYBUBBLE}_1, \text{SSSINGLEKEYBUBBLE}_2, \text{FDSINGLEVALUED}_1, \\
& \quad \text{FDSINGLEVALUED}_2, \text{SSTARGETBUBBLE}_1, \text{SSTARGETBUBBLE}_2] \leftarrow \\
& \quad \mathfrak{R}_e[\text{MARTINRELATIONSHIP}, \text{ERTYPEITEM}_1, \text{ERTYPEITEM}_2]
\end{aligned}$$

This rule is a specialisation of technique-level rule T4. The structure shown in Figure E.10(a) translates completely to the structure shown in Figure E.10(e). In the reverse direction, the four E-R structures shown in Figures E.10(b)–E.10(e) all map partially to the structure shown in Figure E.10(a).



(a) FDD structure

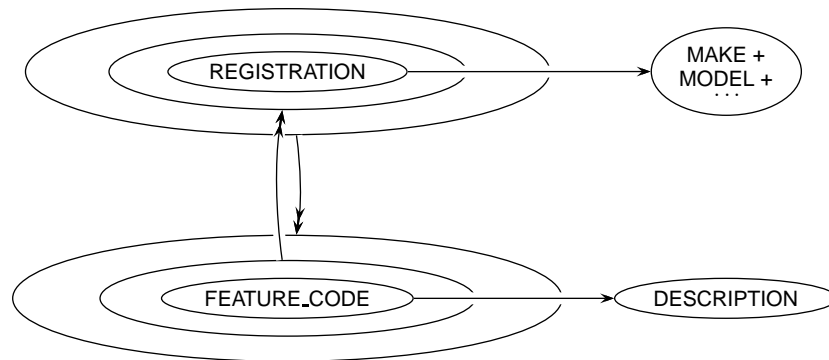


**Figure E.10:** Translating one-to-many relationships between an ERD and an FDD (rule S12)

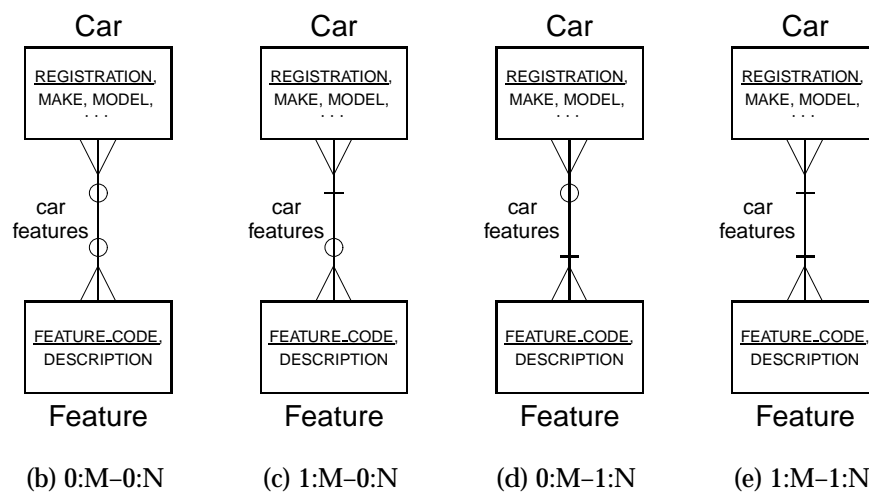


$$\begin{aligned}
& \mathfrak{R}_f [\text{SSINGLEKEYBUBBLE}_1, \text{SSINGLEKEYBUBBLE}_2, \text{FDSINGLEVALUED}_1, \\
& \text{FDSINGLEVALUED}_2, \text{SSTARGETBUBBLE}_1, \text{SSTARGETBUBBLE}_2, \\
& \text{SSMULTIKEYBUBBLE}_1, \text{SSMULTIKEYBUBBLE}_2, \text{FDMULTIVALUED}_1, \quad (\text{S13}) \\
& \text{FDMULTIVALUED}_2, \text{SENDKEYBUBBLE}_1, \text{SENDKEYBUBBLE}_2] \leftarrow \\
& \mathfrak{R}_e [\text{MARTINRELATIONSHIP}, \text{ERTYPEITEM}_1, \text{ERTYPEITEM}_2]
\end{aligned}$$

This rule is a specialisation of technique-level rule T5, and is effectively identical to that rule. The four many-to-many relationship structures shown in Figure E.11(b)–E.11(e) all translate partially to the structure shown in Figure E.11(a).



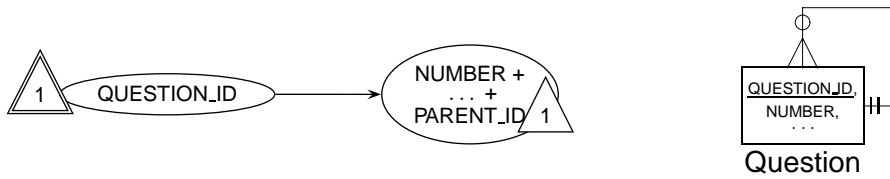
(a) FDD structure



**Figure E.11:** Translating many-to-many relationships from an ERD to an FDD (rule S13)

$$\mathfrak{R}_f[\text{FDATTRIBUTESET}, \text{SSDOMAINFLAG}, \text{SSSINGLEKEYBUBBLE}, \text{SSATTRIBUTE}_1, \text{SSATTRIBUTE}_2] \rightarrow \mathfrak{R}_e[\text{MARTINRELATIONSHIP}] \quad (\text{S14})$$

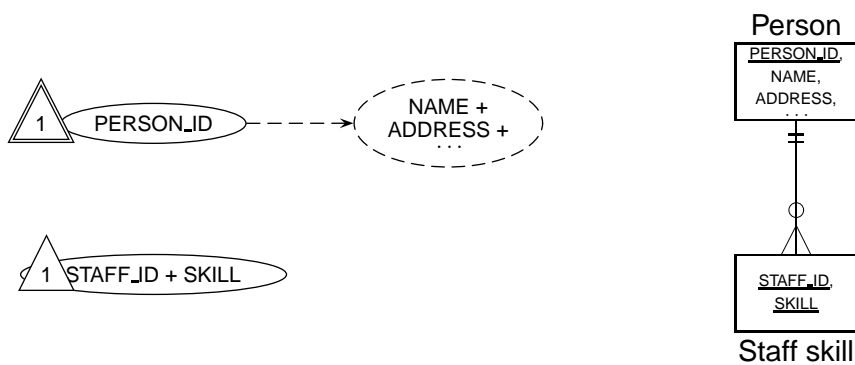
Consider a non-isolated bubble containing an attribute (PARENT.ID in Figure E.12) that is tagged with a domain flag. This domain flag references some other attribute (QUESTION.ID in Figure E.12) in a single-key bubble. This configuration translates completely to a 1:1-0:N relationship from the entity contributed to by the single-key bubble (Question), to the entity contributed to by the non-isolated bubble (Question).



**Figure E.12:** Translating a domain flag from an FDD to an ERD (rule S14)

$$\mathfrak{R}_f[\text{SSISOLATEDBUBBLE}, \text{SSDOMAINFLAG}, \text{SSSINGLEKEYBUBBLE}, \text{SSATTRIBUTE}_1, \text{SSATTRIBUTE}_2] \rightarrow \mathfrak{R}_e[\text{MARTINRELATIONSHIP}] \quad (\text{S15})$$

This rule is identical to rule S14, except that isolated bubbles are allowed, as shown in Figure E.13.

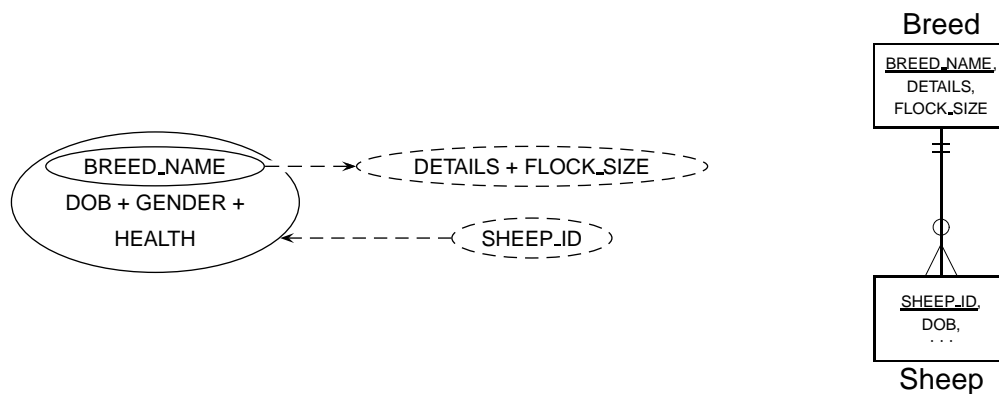


**Figure E.13:** Translating a domain flag from an FDD to an ERD (rule S15)

$$\mathfrak{R}_f[\text{SSSINGLEKEYBUBBLE}, \text{FDATTRIBUTESET}] \rightarrow \tag{S16}$$

$$\mathfrak{R}_e[\text{ERTYPEITEM}_1, \text{ERTYPEITEM}_2, \text{MARTINRELATIONSHIP}]$$

Consider a single-key bubble whose attributes are contained by some other non-isolated bubble, as illustrated by the BREED\_NAME single-key bubble in Figure E.14. This can be translated to a 1:1–0:N relationship from the entity contributed to by the single-key bubble (Breed), to the entity contributed to by the non-isolated bubble (Sheep).



**Figure E.14:** Translating a contained single-key bubble from an FDD to an ERD (rule S16)

$$\mathfrak{R}_f[\text{SSISOLATEDBUBBLE}, \text{FDATTRIBUTESET}] \rightarrow \tag{S17}$$

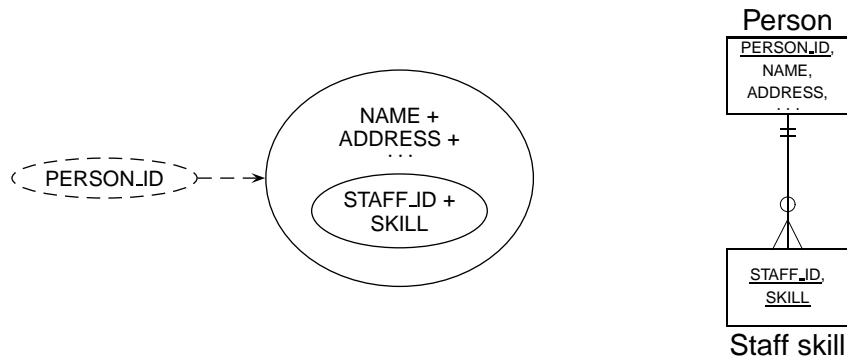
$$\mathfrak{R}_e[\text{ERTYPEITEM}_1, \text{ERTYPEITEM}_2, \text{MARTINRELATIONSHIP}]$$

This rule is identical to rule S16, except that the contained bubble (STAFF\_ID + SKILL in Figure E.15 on the next page) is an isolated bubble.

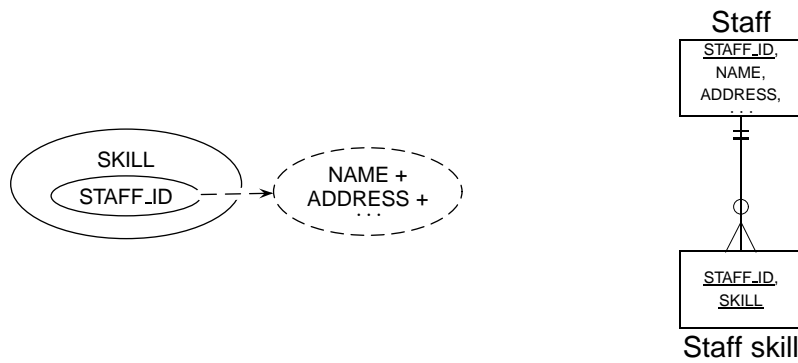
$$\mathfrak{R}_f[\text{SSSINGLEKEYBUBBLE}, \text{SSISOLATEDBUBBLE}] \rightarrow \tag{S18}$$

$$\mathfrak{R}_e[\text{ERTYPEITEM}_1, \text{ERTYPEITEM}_2, \text{MARTINRELATIONSHIP}]$$

This rule is identical to rule S16, except that the containing bubble (STAFF\_ID + SKILL in Figure E.16 on the following page) is an isolated bubble.



**Figure E.15:** Translating a contained isolated bubble from an FDD to an ERD (rule S17)



**Figure E.16:** Translating a contained single-key bubble from an FDD to an ERD (rule S18)

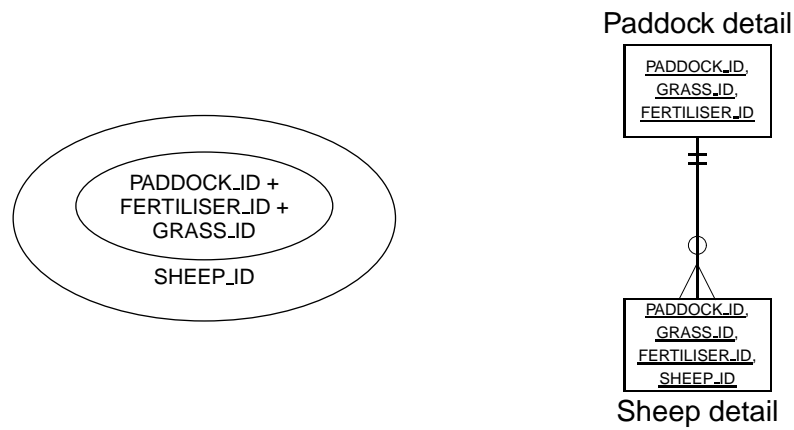
$$\mathfrak{R}_f[\text{SSISOLATEDBUBBLE}_1, \text{SSISOLATEDBUBBLE}_2] \rightarrow \mathfrak{R}_e[\text{ERTYPEITEM}_1, \text{ERTYPEITEM}_2, \text{MARTINRELATIONSHIP}] \quad (\text{S19})$$

This rule is identical to rule S17, except that the containing bubble is also an isolated bubble (see Figure E.17).

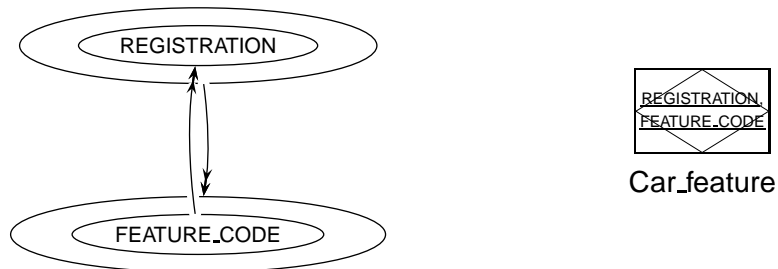
### E.1.3 Heuristics

$$\mathfrak{R}_f[\text{SSMULTIKEYBUBBLE}_1, \text{SSMULTIKEYBUBBLE}_2, \text{SSMULTIVALUED}_1, \text{SSMULTIVALUED}_2, \text{SSENDKEYBUBBLE}_1, \text{SSENDKEYBUBBLE}_1] \rightarrow \mathfrak{R}_e[\text{MARTINASSOCIATIVEENTITY}, \text{MARTINIDENTIFIER}] \quad (\text{H1})$$

The FDD structure shown in Figure E.18 can be translated completely to an associative entity.



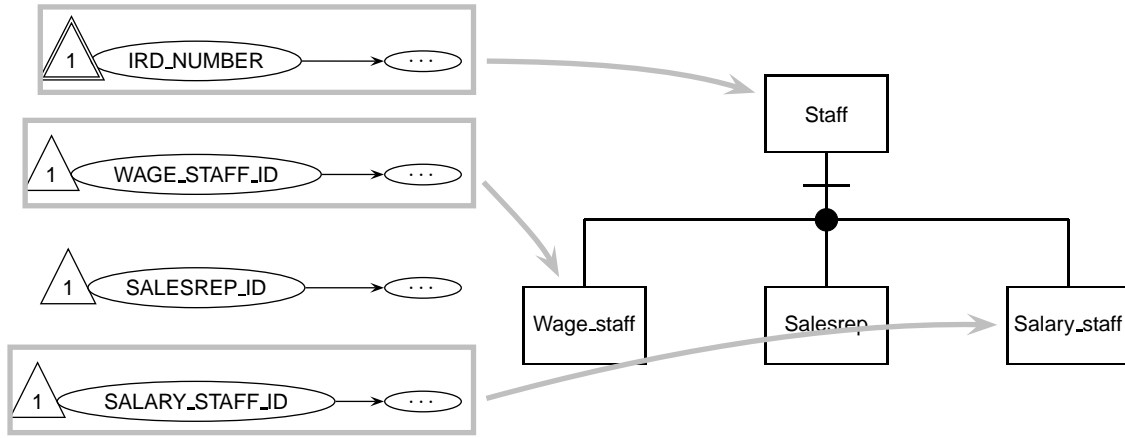
**Figure E.17:** Translating a contained isolated bubble from an FDD to an ERD (rule S19)



**Figure E.18:** Translating a circular multi-valued dependency from an FDD to an ERD (heuristic H1)

$$\begin{aligned}
 & \mathfrak{R}_f[\text{FDATTRIBUTESET}_1, \dots, \text{FDATTRIBUTESET}_n, \\
 & \quad \text{SSDOMAINFLAG}, \text{SSSINGLEKEYBUBBLE}, \\
 & \quad \text{SSATTRIBUTE}_a, \text{SSATTRIBUTE}_1, \dots, \text{SSATTRIBUTE}_n] \rightarrow \\
 & \quad \mathfrak{R}_e[\text{MARTINTYPEHIERARCHY}]
 \end{aligned}
 \tag{H2}$$

Consider a collection of non-multi-key bubbles (SALESREP\_ID, SALARY\_STAFF\_ID and WAGE\_STAFF\_ID in Figure E.19 on the next page), that each contain an attribute tagged with the same domain flag. This domain flag references another attribute (IRD\_NUMBER) in another single-key bubble. If the number of referencing attributes is greater than two, this can be translated completely to a type hierarchy between the entity contributed to by the single-key bubble (Staff — the supertype) and the entities contributed to by the non-multi-key bubbles (Wage\_staff, Salesrep and Salary\_staff — the subtypes).



**Figure E.19:** Deriving a type hierarchy from an FDD (heuristic H2)

### E.1.4 Expressive overlap

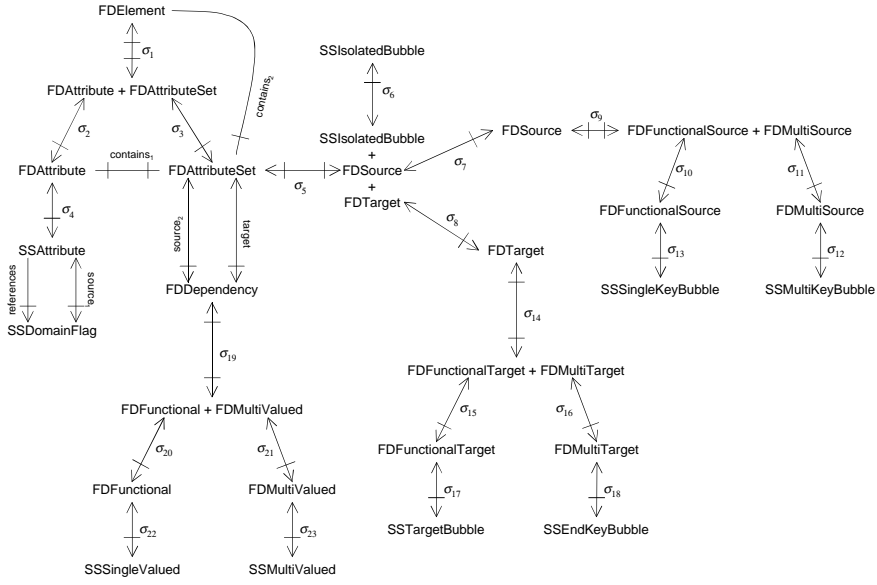
In Figure E.20 are shown schema intension graphs for the representations  $\mathfrak{R}_f(\text{FuncDep}, \text{FDD}_{\text{Smith}})$  and  $\mathfrak{R}_e(E\text{-}R, \text{ERD}_{\text{Martin}})$ . Applying the following set of SIG transformations to the SIG for  $\mathfrak{R}_f$  produces the SIG subgraph shown in Figure E.21:

1. Create new node  $\text{FDAttribute}'$  and associated bijective selection edge  $\sigma_4'$ .
2. Create new node  $\text{FDAttribute}''$  and associated bijective selection edge  $\sigma_4''$ .
3. Move edge  $\text{contains}_1$  across selection edges  $\sigma_4'$  and  $\sigma_4''$ , and edges  $\sigma_3$  and  $\sigma_1$ .
4. Remove the surjectivity annotation from selection edge  $\sigma_4'$ .
5. Move edge references across selection edge  $\sigma_4$ .

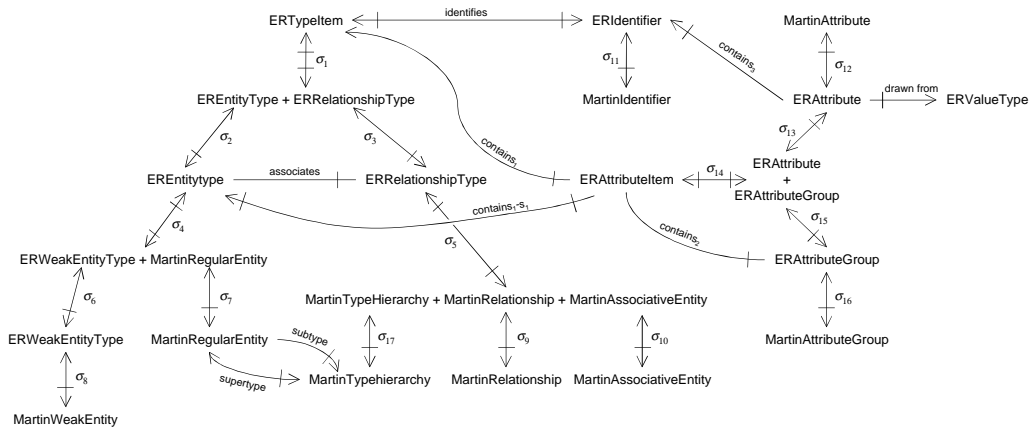
This transformed subgraph is isomorphic to a subgraph of the SIG for  $\mathfrak{R}_e$ , so the two representations have intersecting expressive powers. The expressive overlap is indicated by the blue shaded areas in Figure E.22 on page 412.

### E.1.5 Relative quality

In Figure E.23 on page 413 are shown the tagged SIGs for both representations in both directions of the translation. Blue indicates constructs and edges tagged as a result of appearing in rules, and green indicates constructs and edges tagged as a result of appearing in heuristics. The results of the relative quality measurement are summarised in Table E.1 on page 412.

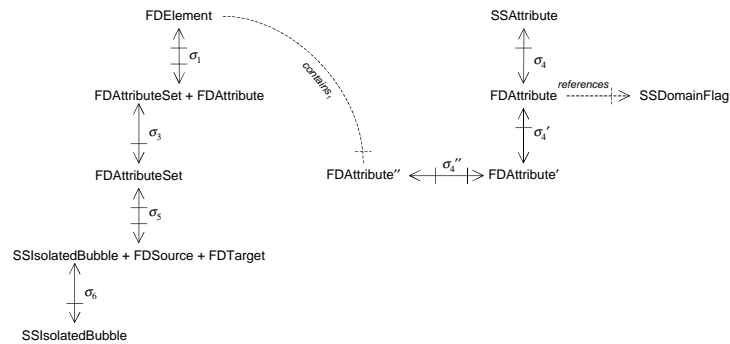


(a)  $\mathfrak{R}_f(\text{FuncDep}, \text{FDD}_{\text{Smith}})$

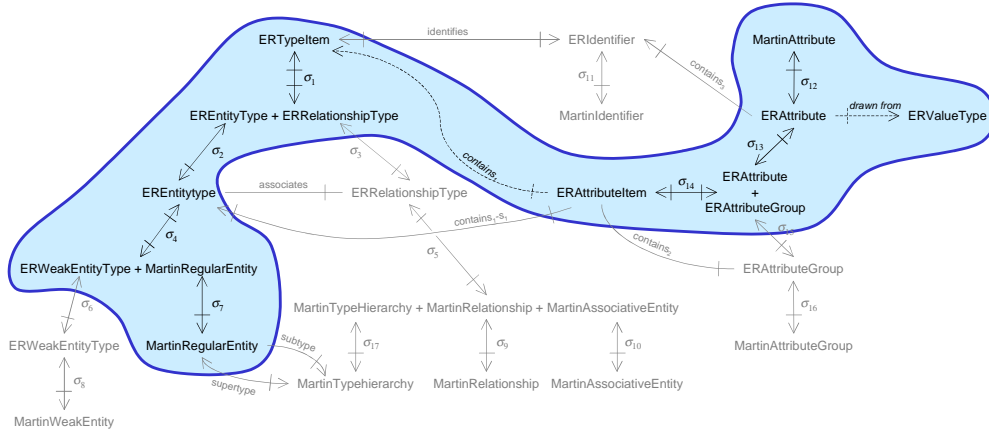


(b) representation  $\mathfrak{R}_e(E-R, \text{ERD}_{\text{Martin}})$

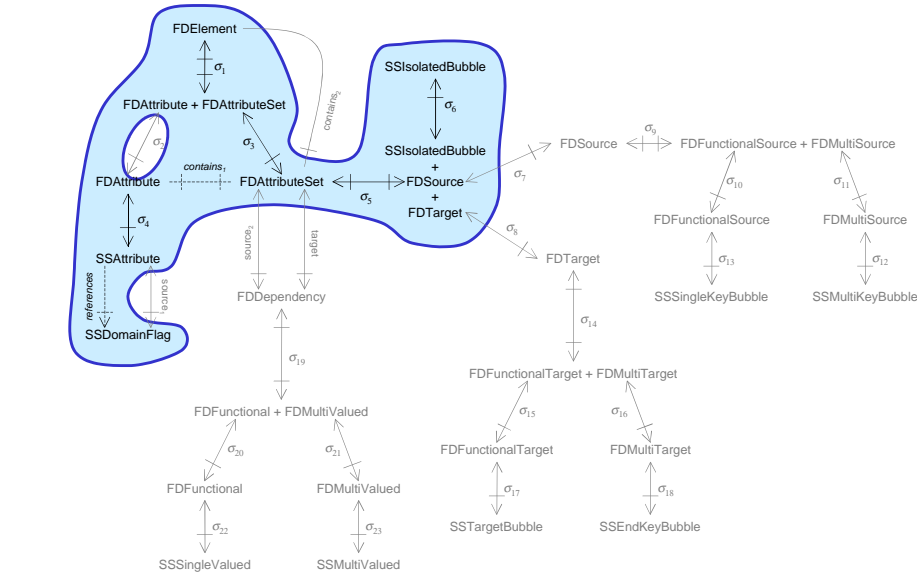
**Figure E.20:** SIGs for  $\mathfrak{R}_f(\text{FuncDep}, \text{FDD}_{\text{Smith}})$  and  $\mathfrak{R}_e(E-R, \text{ERD}_{\text{Martin}})$



**Figure E.21:** Transformed SIG for  $\mathfrak{R}_f(\text{FuncDep}, \text{FDD}_{\text{Smith}})$



(a)  $\mathfrak{R}_e(E-R, ERD_{Martin})$



(b)  $\mathfrak{R}_f(FuncDep, FDD_{Smith})$

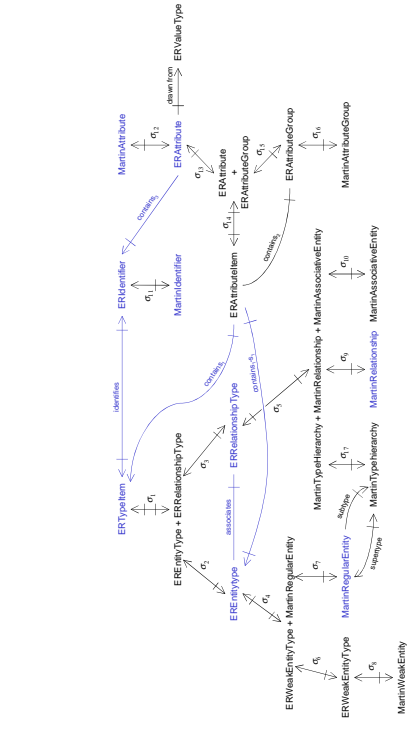
Figure E.22: Expressive overlap for  $\mathfrak{R}_e(E-R, ERD_{Martin})$  and  $\mathfrak{R}_f(FuncDep, FDD_{Smith})$

Table E.1: Relative quality measurements for the  $\mathfrak{R}_f \rightarrow \mathfrak{R}_e / \mathfrak{R}_f \leftarrow \mathfrak{R}_e$  translation

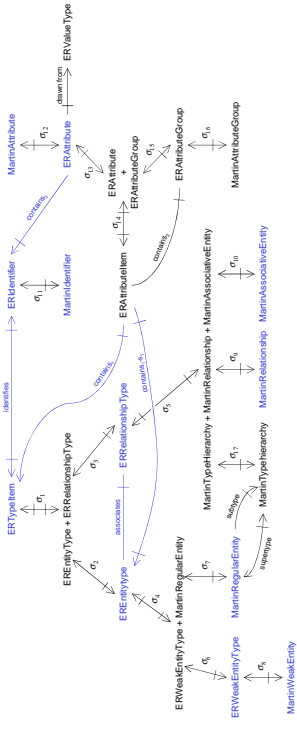
SIG for  $\mathfrak{R}_f(FuncDep, FDD_{Smith})$  has 21 construct nodes and 6 non- $\sigma$  edges.  
 SIG for  $\mathfrak{R}_e(E-R, ERD_{Martin})$  has 17 construct nodes and 9 non- $\sigma$  edges.

|                    | Direction                                   | Tagged source constructs |       | Tagged source edges |        | Tagged target constructs |       | Tagged target edges |       |
|--------------------|---|--------------------------|-------|---------------------|--------|--------------------------|-------|---------------------|-------|
|                    |   | #                        | %     | #                   | %      | #                        | %     | #                   | %     |
| Without heuristics | $\mathfrak{R}_f \rightarrow \mathfrak{R}_e$ | 17                       | 80.95 | 6                   | 100.00 | 9                        | 52.94 | 5                   | 55.56 |
|                    | $\mathfrak{R}_e \rightarrow \mathfrak{R}_f$ | 12                       | 70.59 | 5                   | 55.56  | 15                       | 71.43 | 4                   | 66.67 |
| With heuristics    | $\mathfrak{R}_f \rightarrow \mathfrak{R}_e$ | 17                       | 80.95 | 6                   | 100.00 | 11                       | 64.71 | 7                   | 77.78 |
|                    | $\mathfrak{R}_e \rightarrow \mathfrak{R}_f$ | 12                       | 70.59 | 5                   | 55.56  | 15                       | 71.43 | 4                   | 66.67 |

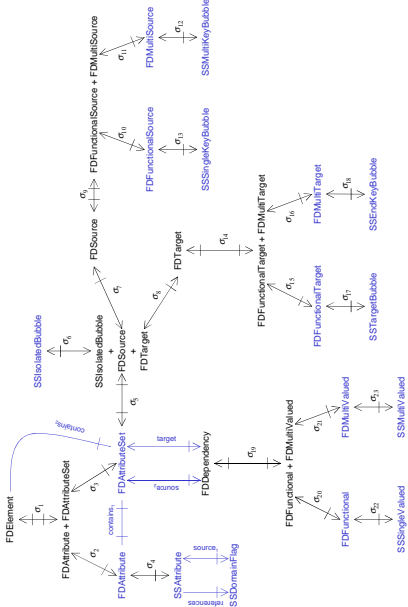




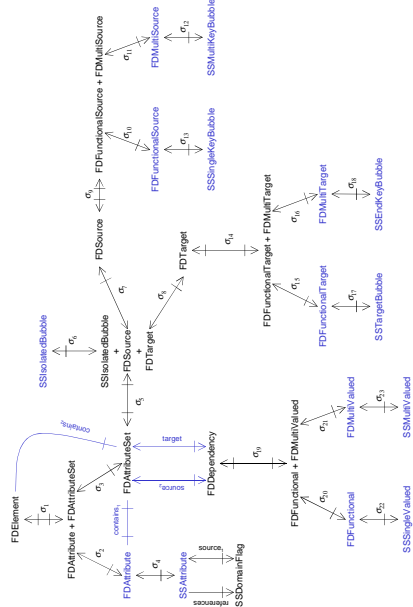
(b)  $\mathfrak{X}_e(E-R, ERDMartin)$  ( $\rightarrow$ )



(d)  $\mathfrak{X}_e(E-R, ERDMartin)$  ( $\leftarrow$ )



(a)  $\mathfrak{X}_f(FuncDep, FDDSmith)$  ( $\leftrightarrow$ )



(c)  $\mathfrak{X}_f(FuncDep, FDDSmith)$  ( $\leftarrow$ )

Figure E.23: Relative quality analysis for  $\mathfrak{X}_e \rightarrow \mathfrak{X}_f / \mathfrak{X}_e \leftarrow \mathfrak{X}_f$

## E.2 $\mathfrak{R}_e \rightleftharpoons \mathfrak{R}_d$

In this translation, descriptions are translated between an entity-relationship description expressed in Martin notation and a data flow diagram expressed in Gane and Sarson notation (Gane and Sarson, 1979), that is:

$$\mathfrak{R}_e(E-R, ERD_{Martin}) \rightleftharpoons \mathfrak{R}_d(DataFlow, DFD_{G\&S}).$$

There are no restrictions on this translation in either direction.

### E.2.1 Technique-level rules

$$\mathfrak{R}_e[ERENTITYTYPE] \rightarrow \mathfrak{R}_d[DFDATASTORE] \quad (T1)$$

A entity of any sort translates partially to a data store.

$$\mathfrak{R}_e[ERATTRIBUTEITEM] \rightleftharpoons \mathfrak{R}_d[DFFIELDITEM] \quad (T2)$$

An attribute (attribute group) translates partially to a field (field group), and vice versa.

### E.2.2 Scheme-level rules

$$\mathfrak{R}_e[MARTINREGULARENTITY] \rightleftharpoons \mathfrak{R}_d[GNSDATASTORE] \quad (S1)$$

This rule is a specialisation of technique-level rule T1 for regular entities, and is otherwise identical to that rule.

$$\mathfrak{R}_e[MARTINASSOCIATIVEENTITY] \rightarrow \mathfrak{R}_d[GNSDATASTORE] \quad (S2)$$

An associative entity translates partially to a data store.

$$\mathfrak{R}_e[MARTINWEAKENTITY] \rightarrow \mathfrak{R}_d[GNSDATASTORE] \quad (S3)$$

This rule is a specialisation of technique-level rule T1 for non-embedded weak entities and is identical to that rule except that it is unidirectional in the forward direction.

$$\mathfrak{R}_e [\text{MARTINATTRIBUTE}] \rightleftharpoons \mathfrak{R}_r [\text{GNSFIELD}] \quad (\text{S4})$$

This is a specialisation of technique-level rule T2, and is otherwise identical to that rule.

$$\mathfrak{R}_e [\text{MARTINATTRIBUTEGROUP}] \rightleftharpoons \mathfrak{R}_r [\text{GNSFIELDGROUP}] \quad (\text{S5})$$

This is a specialisation of technique-level rule T2 for attribute groups, and is effectively identical to that rule.

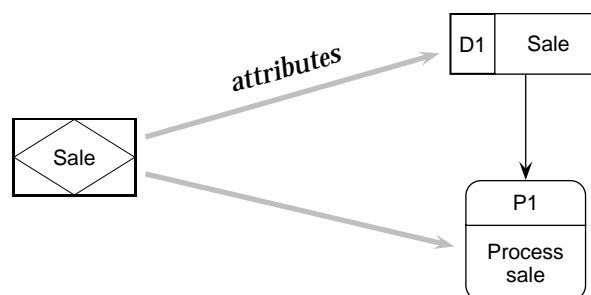
$$\mathfrak{R}_e [\text{MARTINWEAKENTITY}] \rightarrow \mathfrak{R}_d [\text{GNSFIELDGROUP}] \quad (\text{S6})$$

An embedded weak entity translates partially to a field group.

### E.2.3 Heuristics

$$\mathfrak{R}_e [\text{MARTINASSOCIATIVEENTITY}] \rightarrow \mathfrak{R}_d [\text{GNSDATAPROCESS, GNSDATASTORE, GNSDATAFLOW}] \quad (\text{H1})$$

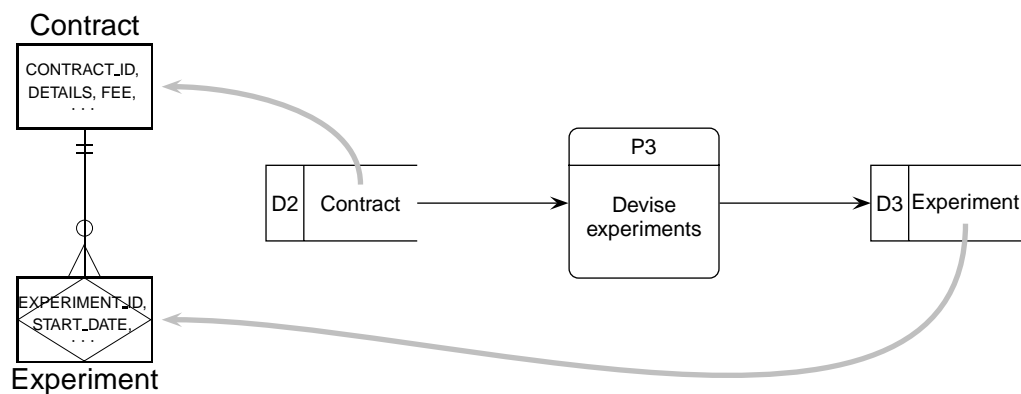
An associative entity (Sale in Figure E.24) can often represent some kind of activity, such as a transaction or an experiment (Campbell, 1992). Such an activity can imply a partial translation to a data process (Process sale in Figure E.24) and a data store (Sale), linked by a data flow.



**Figure E.24:** Translating an associative entity that represents an activity into a data process and data store (heuristic H1)

$$\begin{aligned}
& \mathfrak{R}_e[\text{MARTINRELATIONSHIP}, \text{MARTINREGULARENTITY}_1, \\
& \quad \text{MARTINREGULARENTITY}_2] \nrightarrow + \\
& \mathfrak{R}_d[\text{GNSDATAPROCESS}, \text{GNSDATASTORE}_1, \text{GNSDATASTORE}_2, \\
& \quad \text{GNSDATAFLOW}_1, \text{GNSDATAFLOW}_2]
\end{aligned}
\tag{H2}$$

Consider two data stores (Contract and Experiment in Figure E.25) and a data process (Devise experiments), with data flows running from one data store to the process, and from the process to the other data store. This can be translated to two regular entities (Contract and Experiment in Figure E.25) connected by a 1:1-0:N relationship.



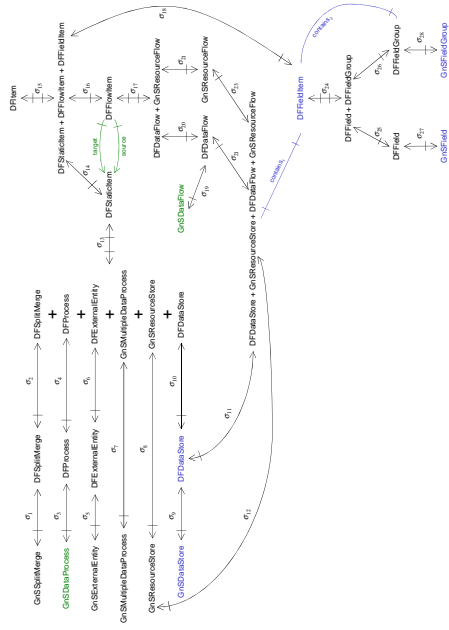
**Figure E.25:** Translating data flows into a relationship (heuristic H2)

## E.2.4 Expressive overlap

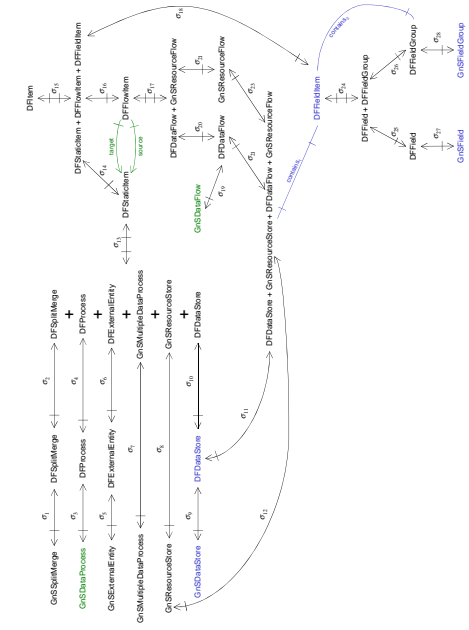
See Section 8.3.1 on page 227.

## E.2.5 Relative quality

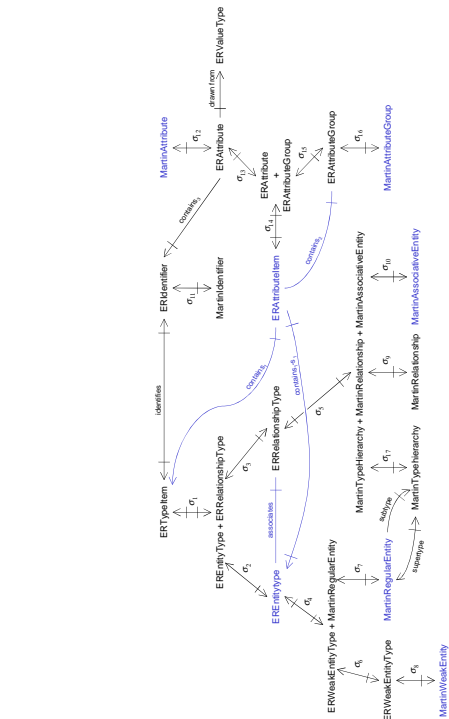
In Figure E.26 are shown the tagged SIGs for both representations in both directions of the translation. Blue indicates constructs and edges tagged as a result of appearing in rules, and green indicates constructs and edges tagged as a result of appearing in heuristics. The results of the relative quality measurement are summarised in Table E.2 on page 418.



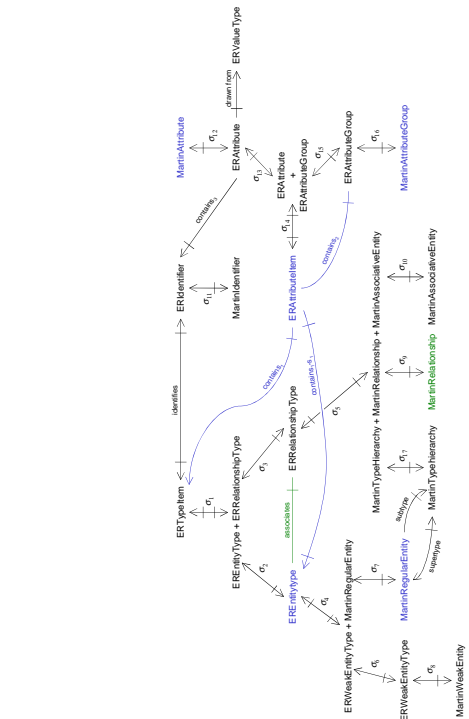
(a)  $\mathfrak{R}_e(E-R, ERD_{Martin}) (\dashv)$



(b)  $\mathfrak{R}_d(DataFlow, DFD_{C\&S}) (\dashrightarrow)$



(c)  $\mathfrak{R}_e(E-R, ERD_{Martin}) (\dashv)$



(d)  $\mathfrak{R}_d(DataFlow, DFD_{C\&S}) (\dashrightarrow)$

Figure E.26: Relative quality analysis for  $\mathfrak{R}_e \equiv \mathfrak{R}_d$

**Table E.2:** Relative quality measurements for the  $\mathfrak{R}_e \rightleftharpoons \mathfrak{R}_d$  translation

---

SIG for  $\mathfrak{R}_e(E-R, ERD_{Martin})$  has 17 construct nodes and 9 non- $\sigma$  edges.

SIG for  $\mathfrak{R}_f(DataFlow, DFD_{G\&S})$  has 21 construct nodes and 4 non- $\sigma$  edges.

|                           |   | Tagged source constructs |       | Tagged source edges |        | Tagged target constructs |       | Tagged target edges |        |
|---------------------------|---|--------------------------|-------|---------------------|--------|--------------------------|-------|---------------------|--------|
|                           | Direction                                   | #                        | %     | #                   | %      | #                        | %     | #                   | %      |
| <b>Without heuristics</b> | $\mathfrak{R}_e \rightarrow \mathfrak{R}_d$ | 7                        | 41.18 | 4                   | 44.44  | 5                        | 23.81 | 2                   | 50.00  |
|                           | $\mathfrak{R}_d \rightarrow \mathfrak{R}_e$ | 5                        | 23.81 | 2                   | 50.00  | 5                        | 29.41 | 3                   | 33.33  |
| <b>With heuristics</b>    | $\mathfrak{R}_e \rightarrow \mathfrak{R}_d$ | 7                        | 41.18 | 4                   | 44.44  | 7                        | 33.33 | 4                   | 100.00 |
|                           | $\mathfrak{R}_d \rightarrow \mathfrak{R}_e$ | 7                        | 33.33 | 4                   | 100.00 | 6                        | 35.29 | 4                   | 44.44  |

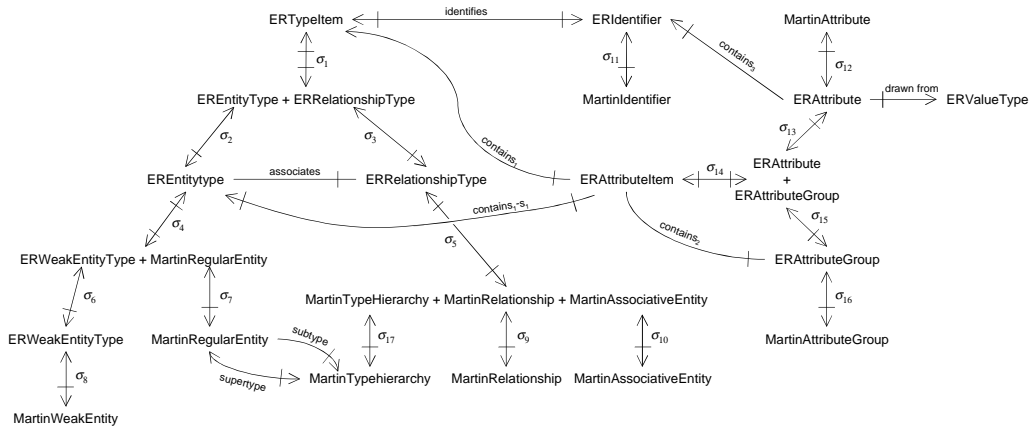
---

## E.3 $\mathfrak{R}_e \rightleftharpoons \mathfrak{R}_r$

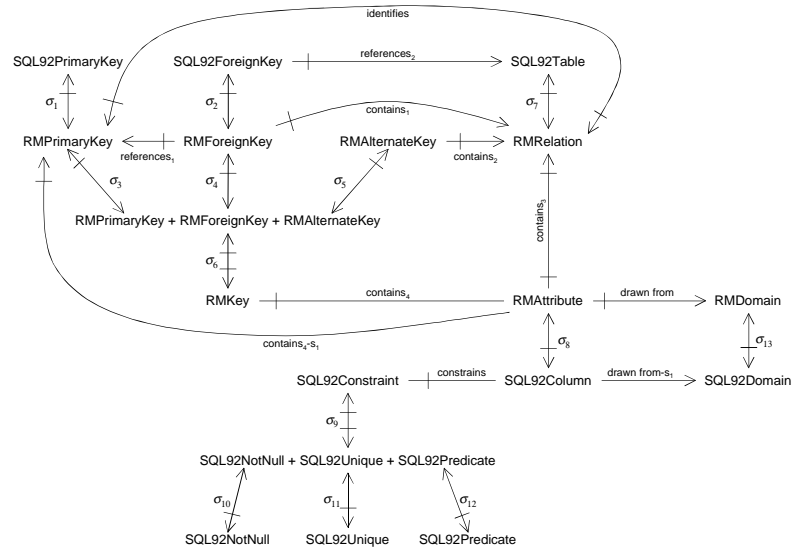
### E.3.1 Expressive overlap

In Figure E.27 are shown the schema intension graphs for the representations  $\mathfrak{R}_e(E-R, ERD_{Martin})$  and  $\mathfrak{R}_r(Relational, SQL/92)$ . Applying the following set of SIG transformations to the SIG for  $\mathfrak{R}_r$  results in the SIG shown in Figure E.28:

1. Remove the surjectivity annotation from edge  $contains_1$ .
2. Create new node  $RMAttribute'$  and associated bijective selection edge  $\sigma_8'$ .
3. Create new node  $RMAttribute''$  and associated bijective selection edge  $\sigma_8''$ .
4. Move edge  $contains_3$  across selection edges  $\sigma_8'$  and  $\sigma_8''$ .
5. Remove the surjectivity annotation from selection edge  $\sigma_8'$ .
6. Create new node  $RMForeignKey'$  and associated bijective selection edge  $\sigma_2'$ .
7. Move selection edge  $\sigma_2$  across selection edge  $\sigma_2'$ .
8. Create new node  $RMRelation'$  and associated bijective selection edge  $\sigma_7'$ .
9. Create new node  $RMRelation''$  and associated bijective selection edge  $\sigma_7''$ .
10. Create new node  $RMRelation'''$  and associated bijective selection edge  $\sigma_7'''$ .
11. Move selection edge  $\sigma_7$  across selection edges  $\sigma_7'$ ,  $\sigma_7''$  and  $\sigma_7'''$ .
12. Move edge  $contains_1$  across selection edges  $\sigma_7'$  and  $\sigma_7''$ .

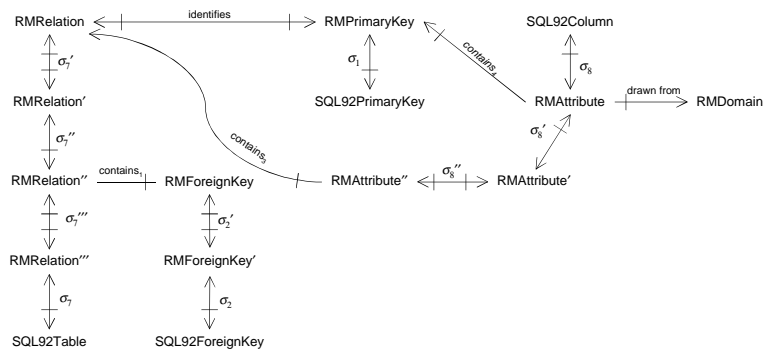


(a)  $\mathfrak{R}_e(E-R, ERD_{Martin})$



(b)  $\mathfrak{R}_r(Relational, SQL/92)$

**Figure E.27:** SIGs for  $\mathfrak{R}_e(E-R, ERD_{Martin})$  and  $\mathfrak{R}_r(Relational, SQL/92)$

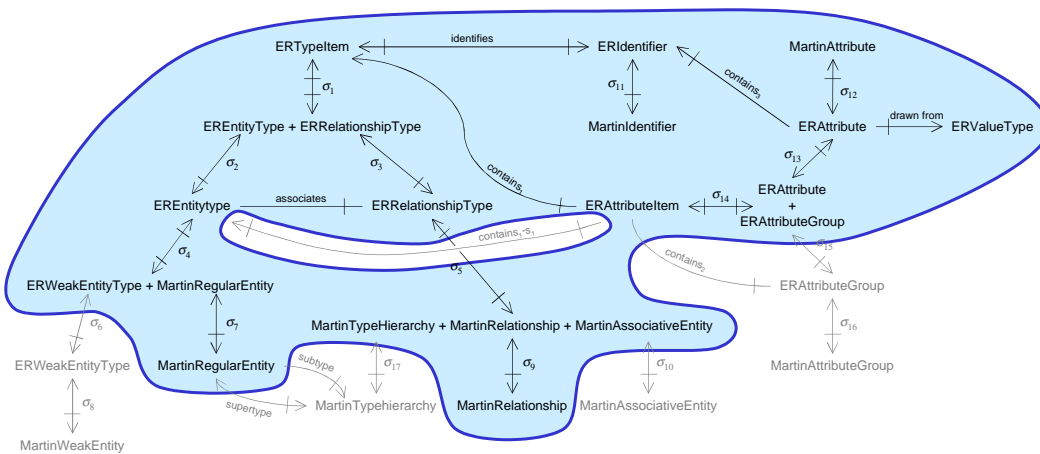


**Figure E.28:** Transformed SIG for  $\mathfrak{R}_r(Relational, SQL/92)$

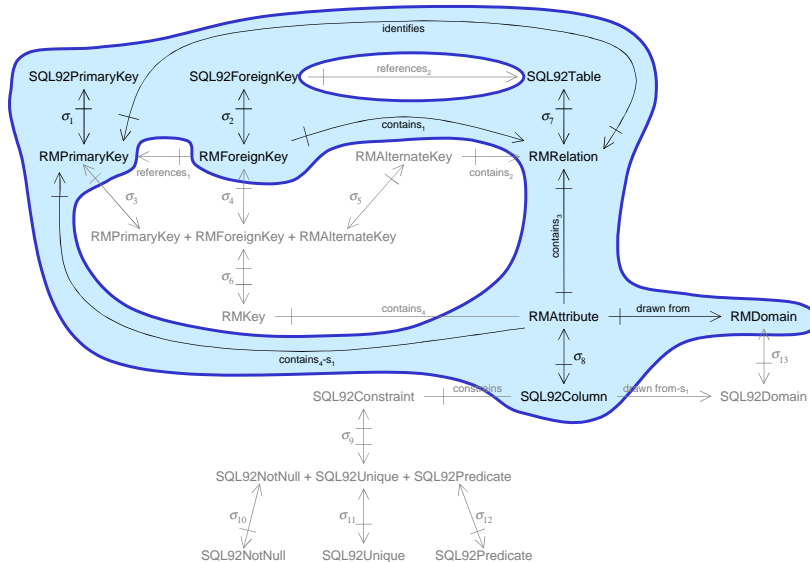
13. Remove the totality annotation from selection edge  $\sigma_7''$ .

14. Remove the functionality annotation from edge  $\text{contains}_1$ .

This transformed SIG corresponds to the subgraphs enclosed by the blue shaded areas in Figure E.29.



(a)  $\mathfrak{X}_e(E-R, ERD_{Martin})$



(b)  $\mathfrak{X}_r(Relational, SQL/92)$

Figure E.29: Expressive overlap for  $\mathfrak{X}_e(E-R, ERD_{Martin})$  and  $\mathfrak{X}_r(Relational, SQL/92)$



### E.3.2 Relative quality

In Figure E.30 on the next page are shown the tagged SIGs for both representations in both directions of the translation. Blue indicates constructs and edges tagged as a result of appearing in rules, and green indicates constructs and edges tagged as a result of appearing in heuristics. The results of the relative quality measurement are summarised in Table E.3.

**Table E.3:** Relative quality measurements for the  $\mathfrak{R}_e \rightleftharpoons \mathfrak{R}_r$  translation

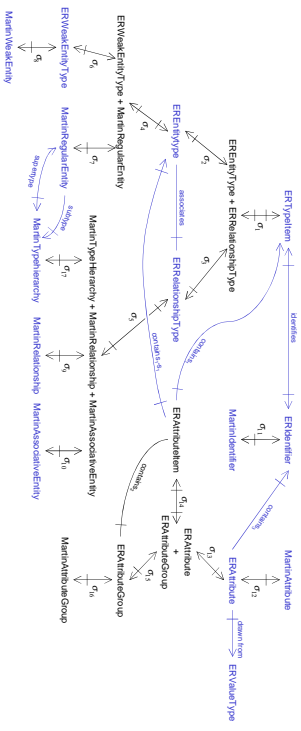
---

SIG for  $\mathfrak{R}_e(E-R, ERD_{Martin})$  has 17 construct nodes and 9 non- $\sigma$  edges.

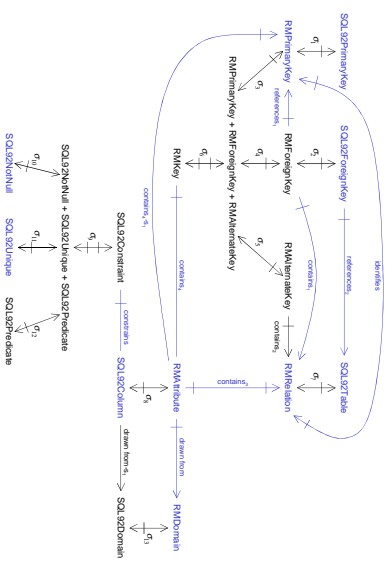
SIG for  $\mathfrak{R}_r(Relational, SQL/92)$  has 16 construct nodes and 11 non- $\sigma$  edges.

|                           | Direction                                   | Tagged source constructs |       | Tagged source edges |       | Tagged target constructs |       | Tagged target edges |       |
|---------------------------|---|--------------------------|-------|---------------------|-------|--------------------------|-------|---------------------|-------|
|                           |   | #                        | %     | #                   | %     | #                        | %     | #                   | %     |
| <b>Without heuristics</b> | $\mathfrak{R}_e \rightarrow \mathfrak{R}_r$ | 14                       | 82.35 | 8                   | 88.89 | 10                       | 62.50 | 9                   | 81.82 |
|                           | $\mathfrak{R}_r \rightarrow \mathfrak{R}_e$ | 11                       | 68.75 | 10                  | 90.91 | 10                       | 58.82 | 8                   | 88.89 |
| <b>With heuristics</b>    | $\mathfrak{R}_e \rightarrow \mathfrak{R}_r$ | 14                       | 82.35 | 8                   | 88.89 | 10                       | 62.50 | 9                   | 81.82 |
|                           | $\mathfrak{R}_r \rightarrow \mathfrak{R}_e$ | 11                       | 68.75 | 10                  | 90.91 | 10                       | 58.82 | 8                   | 88.89 |

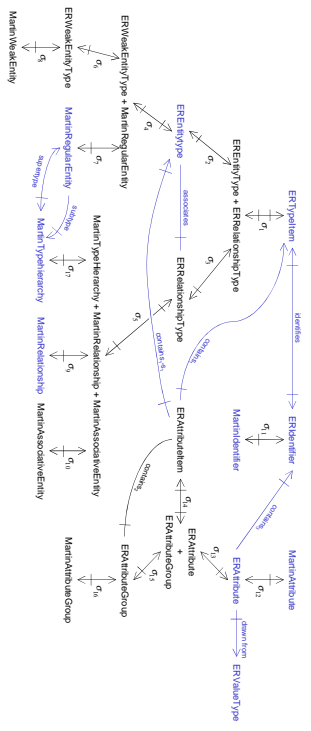
---



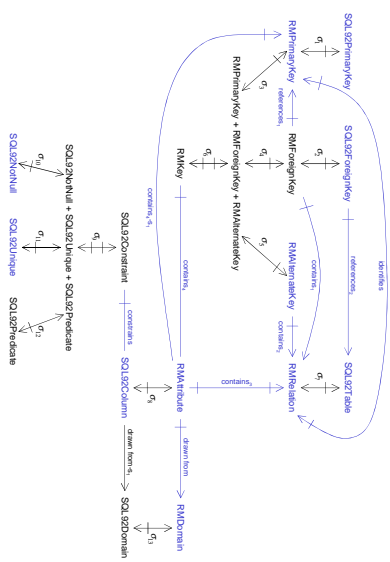
(a)  $\mathfrak{R}_e(E-R, ERD_{Martin})$  ( $\rightarrow$ )



(b)  $\mathfrak{R}_r(Relational, SQL/92)$  ( $\rightarrow$ )



(c)  $\mathfrak{R}_e(E-R, ERD_{Martin})$  ( $\leftarrow$ )



(d)  $\mathfrak{R}_r(DataFlow, SQL/92)$  ( $\leftarrow$ )

Figure E.30: Relative quality analysis for  $\mathfrak{R}_e \rightleftharpoons \mathfrak{R}_r$

# Appendix F

## VML-S syntax and specifications

### F.1 VML-S BNF syntax definition

```

    <alias_id> ::= <simple_id>
    <alias_name> ::= <alias_id> [ [ <list_id> { , <list_id> } ] ]
    <and_op> ::= ,
    <attribute_id> ::= <simple_id>
    | <variable_id>
    <attribute_name> ::= (( <class_name> | <alias_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>
    <class_id> ::= <simple_id>
    <class_list> ::= [ [ <class_list_name> { , <class_list_name> } ] ]
    <class_list_name> ::= group( <class_name> ) [ <alias_name> ]
    | <class_name> [ <alias_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>
    <direction> ::= direction( <dir_map_op> )
```

```

⟨equivalences_def⟩ ::= equivalences( ⟨equivalent⟩ { , ⟨equivalent⟩ } )
  ⟨equivalent⟩ ::= ⟨expression⟩ ⟨map_op⟩ ⟨expression⟩
                | map_to_from( ⟨predicate⟩ , ⟨predicate⟩ )
                | [⟨dir_map_op⟩] ⟨bijection⟩
                | [⟨dir_map_op⟩] ⟨predicate⟩
  ⟨excludes⟩ ::= excludes( ⟨inter_class_id⟩ [⟨dir_map_op⟩]
                          { , ⟨inter_class_id⟩ [⟨dir_map_op⟩] } )
  ⟨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⟩ )
                 ::= | ⟨inter_class_id⟩
  ⟨initialiser⟩ ::= ⟨expression⟩ = ⟨expression⟩
                 | ⟨method⟩
                 | ⟨predicate⟩
  ⟨initialisers_def⟩ ::= initialisers( ⟨initialiser⟩ { , ⟨initialiser⟩ } )
  ⟨inter_class_def⟩ ::= inter_class( ⟨class_list⟩ , ⟨class_list⟩
                                   [ , ⟨label⟩ ] [ , ⟨inherits⟩ ] [ , ⟨direction⟩ ]
                                   [ , ⟨excludes⟩ ] [ , ⟨invariants_def⟩ ]
                                   [ , ⟨equivalences_def⟩ ] [ , ⟨initialisers_def⟩ ] ) .
  ⟨inter_class_id⟩ ::= ⟨simple_id⟩
  ⟨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⟩ } )
  ⟨bijection⟩ ::= bijection( ⟨bijection_expr⟩ , ⟨bijection_expr⟩ )
  ⟨bijection_expr⟩ ::= ⟨class_name⟩
                    | ⟨attribute_name⟩
                    | ⟨invariant_simple_expr⟩

```

```

    <label> ::= label( <inter_class_id> )
<list_expression> ::= [ <expression> { , <expression> } ]
    <list_id> ::= <integer_literal>
                | <attribute_value>
    <mapping> ::= <inter_view_def> { <inter_class_def> }
    <map_type> ::= complete
                | partial
    <method> ::= <method_head> @ <method_tail>
    <method_head> ::= [ [ <model_id> : ] <class_id> . ] <method_attribute_ref>
                | [ <model_id> : ] <class_id>
                | ε
    <method_tail> ::= <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(

```

```

    <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> }
    <dir_map_op> ::= => | <=> | <=>
    <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>
    <map_op> ::= = | <dir_map_op>
    <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>
    <relOp> ::= <= | >= | <> | < | > | = | \= | =: =
    <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>

```

## F.2 Example of a complete VML-S translation

In this section, the translation  $\mathfrak{R}_f(\text{FuncDep}, \text{FDD}_{\text{Smith}}) \leftarrow \mathfrak{R}_e(\text{E-R}, \text{ERD}_{\text{Martin}})$  will be demonstrated using the source description  $D_1(\text{V}_{\text{agri}}, \text{E-R}, \text{ERD}_{\text{Martin}})$  of the agricultural viewpoint (see Figure C.8 on page 367). This description comprises the following elements:

- MARTINREGULARENTITY elements Contract, Client, Staff, Experiment, Grass, Fertiliser, Paddock, Sheep and Breed.
- One MARTINIDENTIFIER element for each of these regular entities (ContractID, ClientID, . . . , BreedID).
- A large number of MARTINATTRIBUTE elements, including:
  - client\_id, name, address and category in Client; and
  - breed\_name, details and flock\_size in Sheep.

The remainder are not listed here for brevity.

- Two embedded MARTINWEAKENTITY elements, representing repeating groups:

- Paddock\_detail, embedded within Experiment, with no explicit attributes and no identifier; and
  - Sheep\_detail, embedded within Paddock\_detail, containing attributes finish\_weight and start\_weight, and with no identifier.
- The eight MARTINRELATIONSHIP elements from Figure C.8 on page 367 (comprises, signs, . . . , flock), with the following modifications:
    - the applied, sown and used relationships connect to the embedded weak entity Paddock\_detail; and
    - the tested relationship connects to the embedded weak entity Sheep\_detail.

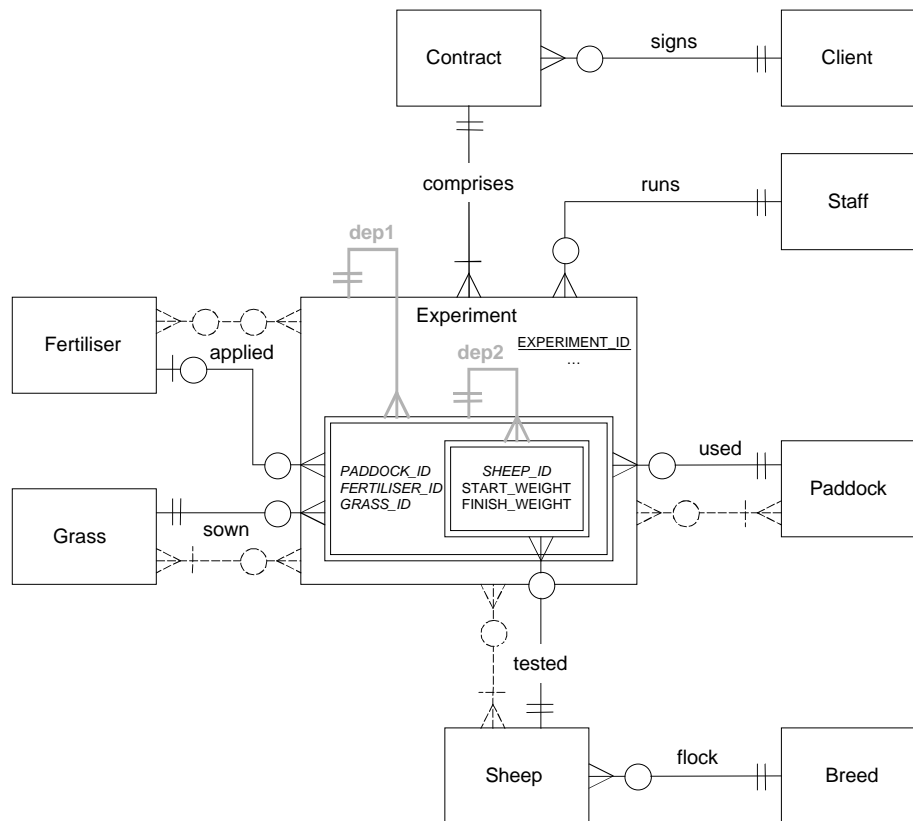
The embedded weak entities mean that the Experiment entity has a complex internal structure, as shown in Figure F.1. The dashed relationship lines correspond to the relationships as they would appear normally, that is, when the diagram is not ‘exploded’ (Figure C.8 on page 367). The thick gray relationship lines represent the implicit dependent relationship between an embedded weak entity and the entity that contains it. It is assumed here that these are automatically generated by the modelling environment when an embedded weak entity is created. The attributes in italics are implied by the attached relationships.

The translation process described in Chapters 4 and 7 takes the approach of iterating through all elements of a description, and attempting to find rules which may be applied to each element. This process, which was described in Algorithm 7.1 on page 204, will now be demonstrated. Suppose the first element chosen for translation is the regular entity Sheep. The rules for this translation (summarised in Table 5.4 on page 136) are examined to find all rules that meet the following two conditions:

1. they include the MARTINREGULARENTITY construct or one of its generalisations among their source constructs; and
2. they appear in the subsumption/exclusion graph for the reverse direction (see Figure 5.20 on page 139).

Applying these two conditions gives the following list of potentially applicable rules: {S1, S2, S4, S7, S8, S9, S10, S11, S12, S13}. The subsumption/exclusion graph is again





**Figure F.1:** Complex internal structure of the Experiment entity

consulted to determine an appropriate order for evaluating these rules, which is: {S2, S1, S4, S7, S8, S9, S10, S11, S12, S13}.

Next, the translation algorithm iterates through this list of rules, calling INITIALELEMENTGROUPS (Algorithm 7.4 on page 209) for each rule in turn. The result of this algorithm is an initial list of elements for the rule, using Sheep as the key element; these lists are shown for each rule in Table F.1 on the next page. The lists are then passed to GENERATECOMBINATIONS (Algorithm 7.5 on page 210), which tests the invariants of the rule against the initial list of elements, producing the filtered element combinations shown in Table F.2 on the next page. The complete combination generation process for rule S12 is shown in Table F.3 on page 431. If the set of filtered element combinations for a rule is non-empty, the rule is applied to each combination of elements in the set, resulting in the generation of appropriate target elements. Thus, rule S1 is applied to the Sheep element, and rule S12 is applied to the collection of elements {Sheep, Breed, flock} to produce the target elements are shown on the next page in Figure F.2(b). The key element (Sheep) is shown in bold in Figure F.2(a).

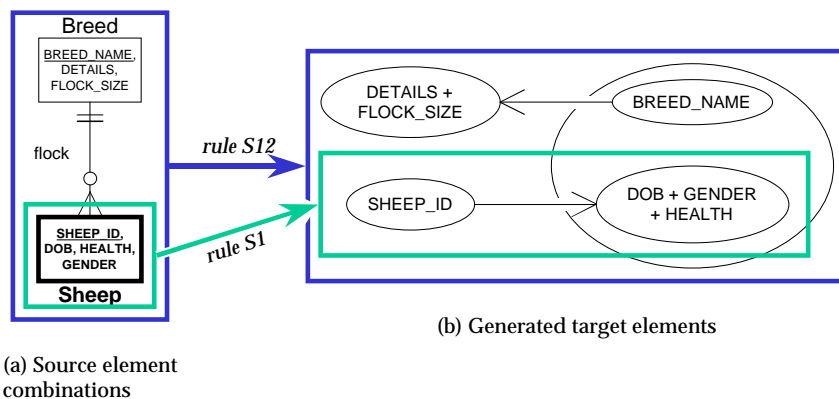
**Table F.1: Initial element lists for the Sheep entity**

| Rule(s) | Element lists   |
|---------|---|
| S2      | {{Sheep}, {ContractID, ClientID, ... , BreedID}}  |
| S1      | {{Sheep}}   |
| S4      | {{Sheep}}   |
| S7-S10  | {{Paddock_detail, Sheep_detail}, {signs, runs, ... , flock}, {Sheep}}                                       |
| S11-S13 | {{signs, runs, ... , flock}, {{{Sheep, Sheep}, {Sheep, Contract}, ... , {Paddock, Sheep}, {Breed, Sheep}}}} |

**Table F.2: Filtered element combinations for the Sheep entity**

| Rule(s) | Element combinations        |
|---------|-----------------------------|
| S2      | {}                          |
| S1      | {{Sheep}}                   |
| S4      | {}                          |
| S7-S10  | {}                          |
| S11     | {}                          |
| S12     | {{flock, {{Breed, Sheep}}}} |
| S13     | {}                          |

Suppose the next element chosen for translation is the embedded weak entity Paddock\_detail. Five rules may potentially be applied to this element: {S7, S8, S9, S10}. The initial element lists for each of these rules are identical: {{Paddock\_detail}, {signs, runs, comprises, applied, sown, used, tested, flock, dep1, dep2}, {Client, Contract, Staff, Paddock, Grass, Fertiliser, Experiment, Sheep, Breed, Paddock\_detail, Sheep\_detail}}. Passing these to GENERATECOMBINATIONS produces empty element combinations for rules S7-S9, and the filtered element combination {{Paddock\_detail, dep1, Experiment}} for rule S10. As a result, only rule S10 can be applied, producing the new



**Figure F.2: Translating elements associated with the Sheep element**

**Table F.3: Generating element combinations for rule S12**

**Rule S12, key element Sheep.**

**Rule header:**

```
[ssinglekeybubble[2], sssinglevalued[2], sstargetbubble[2]], [martinrelationship, erentitytype[2]]
```

**Invariants ('native' and inherited):**

```
martinrelationship.srcCard = 1
martinrelationship.dstCard > 1
martinrelationship.source = eretypeitem[1]
martinrelationship.target = eretypeitem[2]
\+eretypeitem[@class('martinweakentity')
subset(sssinglekeybubble[2].attributes, sstargetbubble[1].attributes)
member(erentitytype[1], errelationshipstype.entities[])
member(erentitytype[2], errelationshipstype.entities[])
member(errelationshipstype, erentitytype[1].relationships)
member(errelationshipstype, erentitytype[2].relationships)
count(errelationshipstype.attributes[]) = 0
fdfunctionalsource[1].dependency = fdfunctional[1]
fdfunctionaltarget[1].dependency = fdfunctional[1]
fdfunctional[1].source = fdfunctionalsource[1]
fdfunctional[1].destination = fdfunctionaltarget[1]
fdfunctionalsource[2].dependency = fdfunctional[2]
fdfunctionaltarget[2].dependency = fdfunctional[2]
fdfunctional[2].source = fdfunctionalsource[2]
fdfunctional[2].destination = fdfunctionaltarget[2]
```

**Initial element groups:** {{{Sheep, Sheep}, {Sheep, Breed}, {Breed, Sheep}, {Sheep, Sheep\_detail}, {Sheep\_detail, Sheep}, {Sheep, Paddock\_detail}, {Paddock\_detail, Sheep}, {Sheep, Grass}, {Grass, Sheep}, {Sheep, Fertiliser}, {Fertiliser, Sheep}, {Sheep, Paddock}, {Paddock, Sheep}, {Sheep, Experiment}, {Experiment, Sheep}, {Sheep, Staff}, {Staff, Sheep}, {Sheep, Contract}, {Contract, Sheep}, {Sheep, Client}, {Client, Sheep}}}, {signs, runs, comprises, applied, sown, used, tested, flock, dep1, dep2}}

R = {}

**First element set:** {{{Sheep, Sheep}, {Sheep, Breed}, . . . , {Sheep, Client}, {Client, Sheep}}}

**Invariants relating purely to ERTYPEITEM elements or generalisations thereof:**

```
\+eretypeitem[@class('martinweakentity')
```

**Applied to first element set:** R = {{{Sheep, Sheep}, {Sheep, Breed}, {Breed, Sheep}, {Sheep, Grass}, {Grass, Sheep}, {Sheep, Fertiliser}, {Fertiliser, Sheep}, {Sheep, Paddock}, {Paddock, Sheep}, {Sheep, Experiment}, {Experiment, Sheep}, {Sheep, Staff}, {Staff, Sheep}, {Sheep, Contract}, {Contract, Sheep}, {Sheep, Client}, {Client, Sheep}}}

**Invariants relating to all constructs in R:**

```
\+eretypeitem[@class('martinweakentity')
```

**Applied to R results in no change to R.**

**Second element set:** {signs, runs, comprises, applied, sown, used, tested, flock}

**Invariants relating purely to MARTINRELATIONSHIP elements or generalisations thereof:**

```
martinrelationship.srcCard = 1
martinrelationship.dstCard > 1
count(errelationshipstype.attributes[]) = 0
```

**Applied to second element set results in no change to second set.**

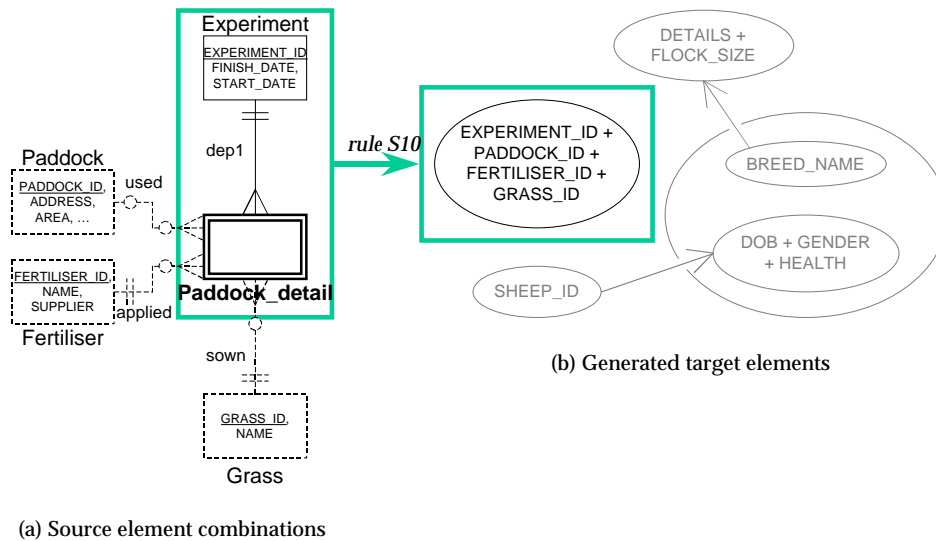
**Combine second element set with R:** R = {{{signs, {Sheep, Sheep}, {Sheep, Breed}, {Breed, Sheep}, {Sheep, Grass}, {Grass, Sheep}, {Sheep, Fertiliser}, {Fertiliser, Sheep}, {Sheep, Paddock}, {Paddock, Sheep}, {Sheep, Experiment}, {Experiment, Sheep}, {Sheep, Staff}, {Staff, Sheep}, {Sheep, Contract}, {Contract, Sheep}, {Sheep, Client}, {Client, Sheep}}}, {runs, {Sheep, Sheep}, {Sheep, Breed}, {Breed, Sheep}, {Sheep, Grass}, {Grass, Sheep}, {Sheep, Fertiliser}, {Fertiliser, Sheep}, {Sheep, Paddock}, {Paddock, Sheep}, {Sheep, Experiment}, {Experiment, Sheep}, {Sheep, Staff}, {Staff, Sheep}, {Sheep, Contract}, {Contract, Sheep}, {Sheep, Client}, {Client, Sheep}}}, {comprises, {Sheep, Sheep}, {Sheep, Breed}, {Breed, Sheep}, {Sheep, Grass}, {Grass, Sheep}, {Sheep, Fertiliser}, {Fertiliser, Sheep}, {Sheep, Paddock}, {Paddock, Sheep}, {Sheep, Experiment}, {Experiment, Sheep}, {Sheep, Staff}, {Staff, Sheep}, {Sheep, Contract}, {Contract, Sheep}, {Sheep, Client}, {Client, Sheep}}}, {tested, {Sheep, Sheep}, {Sheep, Breed}, {Breed, Sheep}, {Sheep, Grass}, {Grass, Sheep}, {Sheep, Fertiliser}, {Fertiliser, Sheep}, {Sheep, Paddock}, {Paddock, Sheep}, {Sheep, Experiment}, {Experiment, Sheep}, {Sheep, Staff}, {Staff, Sheep}, {Sheep, Contract}, {Contract, Sheep}, {Sheep, Client}, {Client, Sheep}}}, {applied, {Sheep, Sheep}, {Sheep, Breed}, {Breed, Sheep}, {Sheep, Grass}, {Grass, Sheep}, {Sheep, Fertiliser}, {Fertiliser, Sheep}, {Sheep, Paddock}, {Paddock, Sheep}, {Sheep, Experiment}, {Experiment, Sheep}, {Sheep, Staff}, {Staff, Sheep}, {Sheep, Contract}, {Contract, Sheep}, {Sheep, Client}, {Client, Sheep}}}, {sown, {Sheep, Sheep}, {Sheep, Breed}, {Breed, Sheep}, {Sheep, Grass}, {Grass, Sheep}, {Sheep, Fertiliser}, {Fertiliser, Sheep}, {Sheep, Paddock}, {Paddock, Sheep}, {Sheep, Experiment}, {Experiment, Sheep}, {Sheep, Staff}, {Staff, Sheep}, {Sheep, Contract}, {Contract, Sheep}, {Sheep, Client}, {Client, Sheep}}}, {used, {Sheep, Sheep}, {Sheep, Breed}, {Breed, Sheep}, {Sheep, Grass}, {Grass, Sheep}, {Sheep, Fertiliser}, {Fertiliser, Sheep}, {Sheep, Paddock}, {Paddock, Sheep}, {Sheep, Experiment}, {Experiment, Sheep}, {Sheep, Staff}, {Staff, Sheep}, {Sheep, Contract}, {Contract, Sheep}, {Sheep, Client}, {Client, Sheep}}}, {dep1, {Sheep, Sheep}, {Sheep, Breed}, {Breed, Sheep}, {Sheep, Grass}, {Grass, Sheep}, {Sheep, Fertiliser}, {Fertiliser, Sheep}, {Sheep, Paddock}, {Paddock, Sheep}, {Sheep, Experiment}, {Experiment, Sheep}, {Sheep, Staff}, {Staff, Sheep}, {Sheep, Contract}, {Contract, Sheep}, {Sheep, Client}, {Client, Sheep}}}, {dep2, {Sheep, Sheep}, {Sheep, Breed}, {Breed, Sheep}, {Sheep, Grass}, {Grass, Sheep}, {Sheep, Fertiliser}, {Fertiliser, Sheep}, {Sheep, Paddock}, {Paddock, Sheep}, {Sheep, Experiment}, {Experiment, Sheep}, {Sheep, Staff}, {Staff, Sheep}, {Sheep, Contract}, {Contract, Sheep}, {Sheep, Client}, {Client, Sheep}}}}}

**Invariants relating to all constructs in R:**

```
martinrelationship.source = eretypeitem[1]
martinrelationship.target = eretypeitem[2]
member(erentitytype[1], errelationshipstype.entities[])
member(erentitytype[2], errelationshipstype.entities[])
member(errelationshipstype, erentitytype[1].relationships)
member(errelationshipstype, erentitytype[2].relationships)
```

**Applied to R gives:** R = {{flock, {{Breed, Sheep}}}}

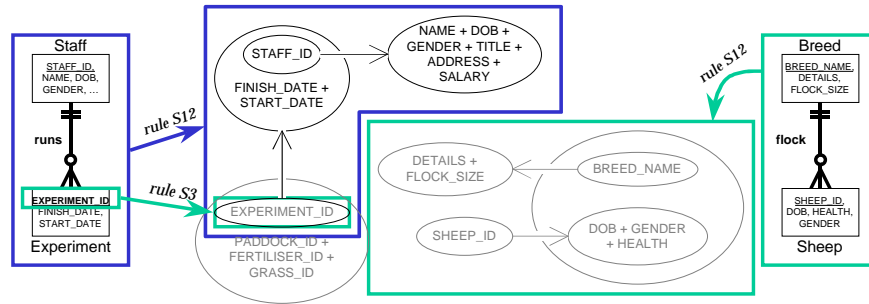
target elements indicated in Figure F.3(b). The existing elements created by previous mapping are drawn in gray, while the newly created elements are drawn in black. The dashed elements in Figure F.3(a) provide important information to rule S10, but are not explicitly mentioned in the rule header, and therefore do not form part of the source element combination.



**Figure F.3:** Translating elements associated with the Paddock\_detail element

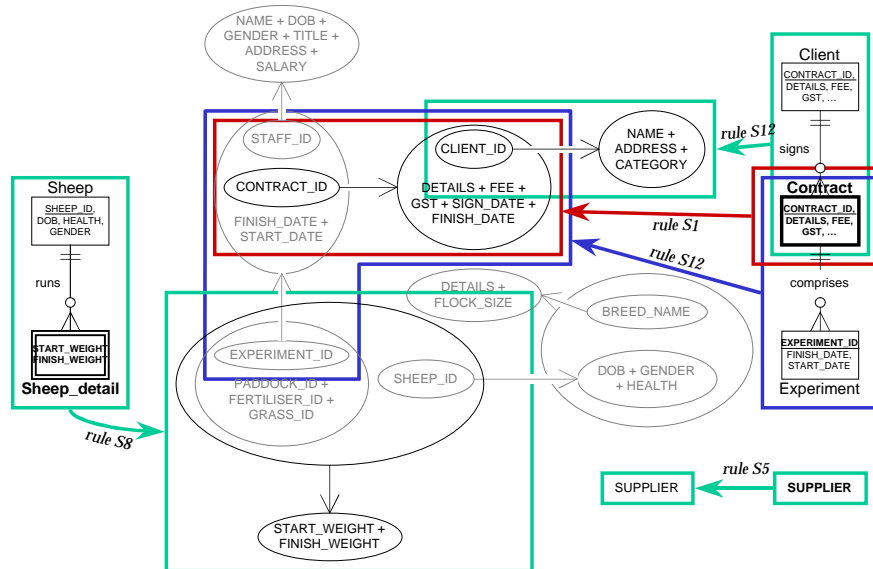
From this point, the translation might proceed as follows:

1. The MARTINRELATIONSHIP element runs is chosen, to which the rules S7–S13 may potentially be applied. Further processing produces the filtered element combination  $\{\{\{\text{Staff, Experiment}\}, \text{runs}\}\}$  for rule S12 and empty sets for all the other rules. Rule S12 is applied as shown in Figure F.4.
2. The MARTINRELATIONSHIP element flock is chosen, which can be treated similarly to the runs element above. When rule S12 is applied to the source element combination  $\{\{\{\text{Breed, Sheep}\}, \text{flock}\}\}$ , it is found that the appropriate target elements already exist, so these are re-used instead of generating new elements, as shown in Figure F.4.
3. The MARTINIDENTIFIER element EXPERIMENT\_ID is mapped by rule S3 to a SSSINGLEKEYBUBBLE element, as shown in Figure F.4.



**Figure F.4:** Further mappings in the example translation: I

4. The MARTINREGULARENTITY element Contract can be mapped by rule S1, with the element combination  $\{\{\text{Contract}\}\}$ , and by rule S12, with the element combinations  $\{\{\text{Client, signs, Contract}\}, \{\text{Contract, comprises, Experiment}\}\}$ . This is shown in Figure F.5.
5. The MARTINATTRIBUTE element SUPPLIER is mapped by rule S5 to the SSATTRIBUTE element SUPPLIER, as shown in Figure F.5.
6. The embedded weak entity Sheep\_detail is mapped by rule S8 to an appropriate collection of target elements, as shown in Figure F.5.



**Figure F.5:** Further mappings in the example translation: II

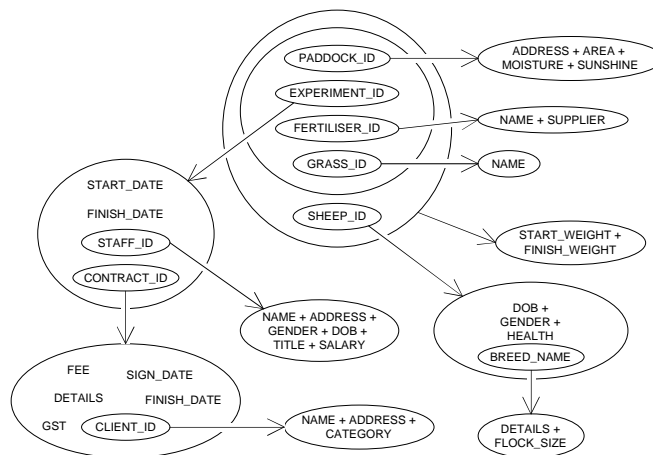
The remaining mappings are similar to those detailed above. All of the mappings in this example are summarised in Table F.4 on the following page. The final description produced by the translation is shown in Figure F.6 on the next page.

**Table F.4:** Summary of mappings in the example translation

| CONSTRUCT           | Elements   | Rule(s) applied                                    |
|---------------------|--|--|
| MARTINATTRIBUTE     | staff_id, name, address, . . . , start_weight  | S5   |
| MARTINRELATIONSHIP  | signs, comprises, runs, flock<br>dep1 (applied, sown, used) <sup>b</sup><br>dep2 (tested) <sup>b</sup> | S12/S12 <sup>a</sup><br>S10<br>S8                  |
| MARTINREGULARENTITY | Client, Contract, Staff, Sheep, Breed<br>Fertiliser, Grass, Paddock<br>Experiment                      | S1, S12/S12 <sup>a</sup><br>S1<br>S1, S10, S12/S12 |
| MARTINWEAKENTITY    | Paddock_detail<br>Sheep_detail   | S10, S8<br>S8                                      |
| MARTINIDENTIFIER    | ClientID, StaffID, . . . , BreedID   | S3   |

**Notes on Table F.4:**

- <sup>a</sup> Which rule is applied depends on the ‘orientation’ of the relationship.
- <sup>b</sup> The relationships in parentheses implicitly take part in the rule, but are not explicitly mentioned in the rule header.



**Figure F.6:** Final FDD description produced by the example translation

It is interesting to compare the FDD in Figure F.6 with the manually-created FDD shown in Figure C.8 on page 367. The main difference is in the structures that represent the repeating groups. In the ‘manual’ FDD, the attributes FERTILISER\_ID and GRASS\_ID are in the target bubble of the single-key bubble containing PADDOCK\_ID and EXPERIMENT\_ID, whereas in the generated FDD, all of these attributes are contained within a single isolated bubble. This is probably because the translation does not ‘know’ the exact semantics of the repeating groups. Such knowledge would be difficult to encode within the translation because it would change for each case.

## F.3 Full VML-S translation specifications

### F.3.1 $\mathfrak{R}_e(E-R, ERD_{Martin}) \iff \mathfrak{R}_r(Relational, SQL/92)$

```
inter_view(er_martinerd, read_write, relational_sql92, read_write, partial).
```

```
/* Technique-level rules */
/* T1: EREntity <=> RMRelation */
inter_class
([erentitytype], [rmrelation],
 label (t1_entity_relation),
 invariants
 ( \+erentitytype.attributes[]@class('erattributegroup'),
   erentitytype.attributes[]=>repeating = false,
   \+erentitytype@class('erweakentitytype')
 ),
 equivalences
 ( erentitytype.name = rmrelation.name,
   erentitytype.identifier = rmrelation.primaryKey,
   erentitytype.attributes[] = rmrelation.attributes[]
 )
).

/* T2: ERRelationshipType =>> RMRelation */
inter_class
([errelationshipstype], [rmrelation],
 label (t2_relationship_relation),
 direction (=>>),
 invariants
 ( count(errelationshipstype.attributes[]) > 0,
   errelationshipstype.attributes[]=>repeating = false,
   \+errelationshipstype.attributes[]@class('erattributegroup')
 ),
 equivalences
 ( errelationshipstype.name = rmrelation.name,
   errelationshipstype.attributes[] = rmrelation.attributes[],
   errelationshipstype.identifier = rmrelation.identifier
 )
).

/* T3: ERAttribute <=> RMAttribute */
inter_class
([erattribute], [rmattribute],
 label (t3_attribute_attribute),
 invariants
 ( erattribute.repeating = false ),
 equivalences
 ( erattribute.name = rmattribute.name,
   erattribute.valueType = rmattribute.domain,
   erattribute.containingItem = rmattribute.relation,
   bijection(erattribute.identifier, rmattribute.keys[]@class('rmprimaryKey'))
 )
).

/* T4: ERIdentifier <=> RMPrimaryKey */
inter_class
inter_class
([eridentifier], [rmprimaryKey],
 label (t4_id_pk),
 invariants
 ( eridentifier.partial = false ),
 equivalences
 ( eridentifier.name = rmprimaryKey.name,
   eridentifier.identifiedItem = rmprimaryKey.relation,
   eridentifier.attributes[] = rmprimaryKey.attributes[]
 )
).

/* T5: ERWeakEntityType + EREntityType + ERRelationshipType =>> RMRelation + RMPrimaryKey */
inter_class
([erweakentitytype, erentitytype, errelationshipstype], [rmrelation, rmprimaryKey],
 label (t5_weak_relation),
 direction (=>>),
 invariants
 ( ( errelationshipstype.id_dependent = true;
   errelationshipstype.existence_dependent = true
 ),
   \+erweakentitytype.attributes[]@class('erattributegroup'),
   erweakentitytype.attributes[]=>repeating = false
   \+erentitytype.attributes[]@class('erattributegroup'),
   erentitytype.attributes[]=>repeating = false,
   \+errelationshipstype.attributes[]@class('erattributegroup'),
   errelationshipstype.attributes[]=>repeating = false,
   erweakentitytype.dependentVia = errelationshipstype,
   member(errelationshipstype, erweakentitytype.relationships),
   member(errelationshipstype, erentitytype.relationships),
   member(erentitytype, errelationshipstype.entities),

```

```

    member(erweakentitytype, errelationshipstype.entities),
    rmrelation.primaryKey = rmprimarykey,
    rmprimarykey.relation = rmrelation
  ),
  equivalences
  ( erweakentitytype.name = rmrelation.name,
    erweakentitytype.attributes[] = rmrelation.attributes[],
    build_weak_pk(erentitytype, erweakentitytype, [], rmprimarykey)
  )
).

/* T6: ERIdentifier <=> RMAternateKey */
inter_class
( [eridentifier], [rmalternatekey],
  label (t6_id_ak),
  direction (<=>),
  invariants
  ( eridentifier.partial = false ),
  equivalences
  ( eridentifier.name = rmalternatekey.name,
    eridentifier.identifiedItem = rmalternatekey.relation,
    eridentifier.attributes[] = rmalternatekey.attributes[]
  )
).

/* T7: ERValueType <=> RMDomain */
inter_class
( [ervaluetype], [rmdomain],
  label (t7_valuetype_domain),
  equivalences
  ( ervaluetype.name = rmdomain.name,
    ervaluetype.datatype = rmdomain.datatype,
    ervaluetype.size = rmdomain.size,
    ervaluetype.dp = rmdomain.dp,
    ervaluetype.attributes[] = rmdomain.attributes[]
  )
).

/* Scheme-level rules */

/* S1: MartinRegularEntity <=> SQL92Table */
inter_class
( [martinregularentity], [sql92table],
  label (s1_regular_table),
  inherits (t1_entity_relation),
  invariants
  ( count(martinregularentity.embeddedEntities[]) = 0 )
).

/* S2: MartinAssociativeEntity =>> SQL92Table */
inter_class
( [martinassociativeentity], [sql92table],
  label (s2_assoc_table),
  inherits (t2_relationship_relation),
  direction (=>>),
  invariants
  ( count(martinassociativeentity.embeddedEntities[]) = 0 )
).

/* S3: MartinAttribute <=> SQL92Column */
inter_class
( [martinattribute], [sql92column],
  label (s3_attr_column),
  inherits (t3_attribute_attribute)
).

/* S4: MartinIdentifier <=> SQL92PrimaryKey */
inter_class
( [martinidentifier], [sql92primarykey],
  label (s4_id_pk),
  inherits (t4_id_pk)
).

/* Common definitions for one-to-one relationship rules. */
inter_class
( [martinrelationship, ertypeitem[2]], [sql92foreignkey[2]],
  label (relationship_11_base),
  invariants
  ( martinrelationship.srcCard = 1,
    martinrelationship.dstCard = 1,
    martinrelationship.source = ertypeitem[1],
    martinrelationship.target = ertypeitem[2],
    sql92foreignkey[1].refTable = sql92foreignkey[2].relation,
    sql92foreignkey[2].refTable = sql92foreignkey[1].relation,
    sql92foreignkey[1].refPK = sql92foreignkey[2].relation=>primaryKey,
    sql92foreignkey[2].refPK = sql92foreignkey[1].relation=>primaryKey,
    sql92foreignkey[1].columns[] = sql92foreignkey[2].relation=>primaryKey=>columns[],
    sql92foreignkey[2].columns[] = sql92foreignkey[1].relation=>primaryKey=>columns[]
  ),

```



```

equivalences
( =>> append('_fk', ertypeitem[2].name, sql92foreignkey[1].name),
  =>> append('_fk', ertypeitem[1].name, sql92foreignkey[2].name),
  =>> combine_attr(ertypeitem[1], ertypeitem[2].identifier, sql92foreignkey[2].relation),
  =>> combine_attr(ertypeitem[2], ertypeitem[1].identifier, sql92foreignkey[1].relation)
),
initialisers
( <=<= append(ertypeitem[2].name, ertypeitem[1].name, martinrelationship.name) )
).

/* Common definitions for one-to-many relationship rules. */
inter_class
( [martinrelationship, ertypeitem[2]], [sql92foreignkey],
  label (relationship_1N_base),
  invariants
  ( martinrelationship.source = ertypeitem[1],
    martinrelationship.target = ertypeitem[2],
    martinrelationship.srcCard = 1,
    martinrelationship.dstCard > 1
  ),
  equivalences
  ( martinrelationship.name = sql92foreignkey.name,
    =>> combine_attr(ertypeitem[2], ertypeitem[1].identifier, sql92foreignkey.relation),
    ertypeitem[1] = sql92foreignkey.refTable,
    ertypeitem[1].identifier = sql92foreignkey.refPK,
    ertypeitem[1].identifier=>attributes[] = sql92foreignkey.attributes[]
  ),
  initialisers
  ( martinrelationship.dstOpt = 0,
    martinrelationship.dstCard = 2
  )
).

/* Common definitions for many-to-one relationship rules. */
inter_class
( [martinrelationship, ertypeitem[2]], [sql92foreignkey],
  label (relationship_M1_base),
  direction (=>>),
  invariants
  ( martinrelationship.source = ertypeitem[1],
    martinrelationship.target = ertypeitem[2],
    martinrelationship.srcCard > 1,
    martinrelationship.dstCard = 1
  ),
  equivalences
  ( martinrelationship.name = sql92foreignkey.name,
    =>> combine_attr(ertypeitem[1], ertypeitem[2].identifier, sql92foreignkey.relation),
    ertypeitem[2] = sql92foreignkey.refTable,
    ertypeitem[2].identifier = sql92foreignkey.refPK,
    ertypeitem[2].identifier=>attributes[] = sql92foreignkey.attributes[]
  ),
  initialisers
  ( martinrelationship.srcOpt = 0,
    martinrelationship.srcCard = 2
  )
).

/* S5: MartinRelationship + ERTypeItem (1:1) + ERTypeItem (*:N) <=>
 *      SQL92ForeignKey + SQL92NotNull
 */
inter_class
( [martinrelationship, ertypeitem[2]], [sql92foreignkey, sql92nonnull],
  label (s5_rel_11xN),
  inherits (relationship_1N_base),
  excludes (s10_rel_01xN <<=),
  invariants
  ( martinrelationship.srcOpt = 1,
    sql92foreignkey.columns[] = sql92nonnull.columns[]
  )
).

/* S6: MartinRelationship + ERTypeItem (1:1) + ERTypeItem (1:1) <=>
 *      2 SQL92ForeignKey + 2 SQL92Unique + 2 SQL92NotNull
 */
inter_class
( [martinrelationship, ertypeitem[2]], [sql92foreignkey[2], sql92unique[2], sql92nonnull[2]],
  label (s6_rel_1111),
  inherits (relationship_11_base),
  excludes (h1_id_unique <<=, s5_rel_11xN <<=, s7_rel_0111 <<=,
    s9_typehierarchy_fk <<=, s10_rel_01xN <<=, s12_rel_0101 <<=),
  invariants
  ( martinrelationship.srcOpt = 1,
    martinrelationship.dstOpt = 1,
    sql92foreignkey[1].columns[] = sql92unique[1].columns[],
    sql92foreignkey[2].columns[] = sql92unique[2].columns[],
    sql92foreignkey[1].columns[] = sql92nonnull[1].columns[],
    sql92foreignkey[2].columns[] = sql92nonnull[2].columns[]
  )
).

```

```

/* S7: MartinRelationship + ERTypeItem(0:1) + ERTypeItem(1:1) <=>
 *      2 SQL92ForeignKey + 2 SQL92Unique + SQL92NotNull
 */
inter_class
( [martinrelationship, ertypeitem[2]], [sql92foreignkey[2], sql92unique[2], sql92nonnull],
  label (s7_rel_0111),
  inherits (relationship_11_base),
  excludes (hl_id_unique <<=, s5_rel_11xN <<=, s9_typehierarchy_fk <<=,
            s10_rel_01xN <<=, s12_rel_0101 <<=),
  invariants
  ( martinrelationship.srcOpt = 0,
    martinrelationship.dstOpt = 1,
    sql92foreignkey[1].columns[] = sql92unique[1].columns[],
    sql92foreignkey[2].columns[] = sql92unique[2].columns[],
    sql92foreignkey[1].columns[] = sql92nonnull.columns[]
  )
).

/* S8: MartinRelationship + ERTypeItem (*:M) + ERTypeItem (*:N) <=>
 *      SQL92Table + SQL92PrimaryKey + 2 SQL92ForeignKey
 */
inter_class
( [martinrelationship, ertypeitem[2]], [sql92table, sql92primarykey, sql92foreignkey[2]],
  label (s8_rel_xMxN),
  excludes (s1_regular_table <<=, s4_id_pk <<=, s10_rel_01xN <<=),
  invariants
  ( martinrelationship.srcCard > 1,
    martinrelationship.dstCard > 1,
    martinrelationship.source = ertypeitem[1],
    martinrelationship.target = ertypeitem[2],
    sql92table.primaryKey = sql92primarykey,
    sql92table.attributes[] = sql92primarykey.attributes[],
    member(sql92foreignkey[], sql92table.foreignKeys),
    sql92foreignkey[1].relation = sql92table,
    append(sql92foreignkey[2].columns, sql92foreignkey[1].columns, sql92primarykey.attributes)
  ),
  equivalences
  ( martinrelationship.name = sql92table.name,
    =>> append('_fk', ertypeitem[1].name, sql92foreignkey[1].name),
    =>> append('_fk', ertypeitem[2].name, sql92foreignkey[2].name),
    =>> combine_attr(ertypeitem[1].identifier, ertypeitem[2].identifier, sql92table),
    ertypeitem[1] = sql92foreignkey[1].refTable,
    ertypeitem[2] = sql92foreignkey[2].refTable,
    ertypeitem[1].identifier = sql92foreignkey[1].refPK,
    ertypeitem[2].identifier = sql92foreignkey[2].refPK,
    ertypeitem[1].identifier=>attributes[] = sql92foreignkey[1].columns[],
    ertypeitem[2].identifier=>attributes[] = sql92foreignkey[2].columns[]
  ),
  initialisers
  ( martinrelationship.srcCard = 2,
    martinrelationship.dstCard = 2,
    martinrelationship.srcOpt = 0,
    martinrelationship.dstOpt = 0
  )
).

/* S9: MartinTypeHierarchy + 2 MartinRegularEntity =>> SQL92ForeignKey + SQL92Unique + SQL92NotNull
 */
inter_class
( [martintypehierarchy, martinidentifier, martinregularentity parent, martinregularentity child],
  [sql92foreignkey, sql92unique, sql92nonnull],
  label (s9_typehierarchy_fk),
  excludes (hl_id_unique <<=, s5_rel_11xN <<=, s10_rel_01xN <<=),
  invariants
  ( parent.subtypes = martintypehierarchy,
    martintypehierarchy.supertype = parent,
    member(child, martintypehierarchy.subtypes),
    child.supertype = martintypehierarchy,
    parent.identifier = martinidentifier,
    martinidentifier.entity = parent,
    sql92foreignkey.attributes = sql92unique.columns,
    sql92foreignkey.attributes = sql92nonnull.columns
  ),
  equivalences
  ( =>> append('_fk', child.name, sql92foreignkey.name),
    =>> combine_attr(child, martinidentifier, sql92foreignkey.relation),
    martintypehierarchy.supertype = sql92foreignkey.refTable,
    martinidentifier = sql92foreignkey.refPK,
    martinidentifier.attributes[] = sql92foreignkey.attributes[]
  )
).

/* S10: MartinRelationship + ERTypeItem (0:1) + ERTypeItem (*:N) <=> SQL92ForeignKey */
inter_class
( [martinrelationship, ertypeitem[2]], [sql92foreignkey],
  label (s10_rel_01xN),
  inherits (relationship_1N_base),
  invariants ( martinrelationship.srcOpt = 0 )
).

```

```

/* S11: MartinWeakEntity + EREntityType + MartinRelationship =>>
*
*      SQL92Table + SQL92PrimaryKey
*/
inter_class
( [martinweakentity, erentitytype, martinrelationship], [sql92table, sql92primaryKey],
  label (s11_weak_table),
  inherits (t5_weak_relation),
  direction (=>>),
  invariants
  ( count(martinweakentity.embeddedEntities[]) = 0,
    martinweakentity.embedded = false,
    martinrelationship.source = erentitytype,
    martinrelationship.target = martinweakentity,
    martinrelationship.srcCard = 1,
    martinrelationship.srcOpt = 1,
    martinrelationship.dstCard > 1
  )
).

/* S12: MartinRelationship + ERTypeItem (0:1) + ERTypeItem (0:1) <=>
*
*      2 SQL92ForeignKey + 2 SQL92Unique
*/
inter_class
( [martinrelationship, ertypeitem[2]], [sql92foreignkey[2], sql92unique[2]],
  label (s12_rel_0101),
  inherits (relationship_11_base),
  excludes (h1_id_unique <=<, s10_rel_01xN <=<=),
  invariants
  ( martinrelationship.srcOpt = 0,
    martinrelationship.dstOpt = 0,
    sql92foreignkey[1].columns[] = sql92unique[1].columns[],
    sql92foreignkey[2].columns[] = sql92unique[2].columns[]
  )
).

/* Heuristics */

/* H1: MartinIdentifier <=<= SQL92Unique */
inter_class
( [martinidentifier], [sql92unique],
  label (h1_id_unique),
  direction (<=<=),
  invariants
  ( martinidentifier.partial = false )
  equivalences
  ( martinidentifier.name = sql92unique.name,
    martinidentifier.attributes[] = sql92unique.columns[],
    martinidentifier.identifiedItem = sql92unique.columns[]=>relation
  )
).

/* Auxiliary functions -- not tested! */

/* Combine two list of ERAttributes to give a list of SQL92Columns for an SQL92Table */
combine_attr(Item1, Item2, Table) :-
  Item1@attributes(A1),
  Item2@attributes(A2),
  append(A2, A1, Atemp),
  map_attr(Atemp, TAttr),
  Table@attributes := TAttr.

map_attr([], []).
map_attr([E|Er], [T|Tr]) :-
  map_attr(Er, Tr),
  E@name(Name),
  E@datatype(Type),
  E@size(Size),
  E@dp(DP),
  sql92column@create(T),
  T@name := Name,
  T@datatype := Type,
  T@size := Size.

/* Take the primary key of an EREntityType and combine it with the key elements of a
* MartinWeakEntity, giving an SQL92PrimaryKey.
*/
build_weak_pk(Entity, Weak, CK) :-
  Entity@primaryKey(PK),
  Weak@primaryKey(WPK),
  PK@attributes(PAttr),
  WPK@attributes(WAttr),
  append(WAttr, PAttr, Atemp),
  map_attr(Atemp, CKAttr),
  CK@attributes := CKAttr.

```

## F.3.2 $\mathfrak{R}_f(\text{FuncDep}, \text{FDD}_{\text{Smith}}) \rightarrow \mathfrak{R}_e(E-R, \text{ERD}_{\text{Martin}}) / \mathfrak{R}_f \leftarrow \mathfrak{R}_e$

```

inter_view(funcdep_smithfdd, read_write, er_martinerd, read_write, partial).

/* Technique-level rules */

/* T1: FDFunctionalSource + FDFunctional + FDFunctionalTarget <=> EREntityType */
inter_class
( [fdfunctionalsource, fdfunctional, fdfunctionaltarget], [erentitytype],
  label (t1_fd_entity),
  invariants
  ( fdfunctionalsource = fdfunctionalsource,
    fdfunctional.destination = fdfunctionaltarget,
    fdfunctionalsource.dependency = fdfunctional,
    fdfunctionaltarget.dependency = fdfunctional,
    count(erentitytype.attributes[]) = \= - count(erentitytype.identifier=>attributes[])
  ),
  equivalences
  ( fdfunctional.name = erentitytype.name,
    fdfunctionalsource = erentitytype.identifier,
    =>> get_fd_attr(fdfunctionalsource.attributes, fdfunctionaltarget.attributes, erentitytype.attributes),
    <<= get_nonkey(fdfunctionaltarget.attributes, erentitytype.attributes)
  ),
  initialisers
  ( <<= append('target', erentitytype.name, fdfunctionaltarget.name),
    <<= append('source', erentitytype.name, fdfunctionalsource.name)
  )
).

/* T2: FDMultiSource + FDMultiValued + FDMultiTarget <=> EREntityType + ERIdentifier */
inter_class
( [fdmultisource, fdmultivalued, fdmultitarget], [erentitytype, eridentifier],
  label (t2_mvd_entity),
  invariants
  ( fdmultivalued.source = fdmultisource,
    fdmultivalued.destination = fdmultitarget,
    fdmultisource.dependency = fdmultivalued,
    fdmultitarget.dependency = fdmultivalued,
    \+erentitytype@class('erweakentitytype'),
    count(erentitytype.attributes[]) = 2,
    count(eridentifier.attributes[]) = 2, % no non-key attributes
    eridentifier.partial = false,
    eridentifier.entity = erentitytype,
    erentitytype.identifier = eridentifier
  ),
  equivalences
  ( fdmultivalued.name = erentitytype.name,
    <<= fdmultisource.attributes[1] = erentitytype.attributes[1],
    <<= fdmultitarget.attributes[2] = erentitytype.attributes[2]
  ),
  initialisers
  ( <<= append('target', erentitytype.name, fdmultitarget.name),
    <<= append('source', erentitytype.name, fdmultisource.name),
    =>> append(fdmultitarget.attributes, fdmultisource.attributes, erentitytype.attributes)
  )
).

/* T3: FDAttribute <=> ERAttribute */
inter_class
( [fdattribute], [erattribute],
  label (t3_attr_attr),
  equivalences
  ( fdattribute.name = erattribute.name )
).

/* T4: 2 FDFunctionalSource + 2 FDSingleValued + 2 FDFunctionalTarget <=>
 *      ERRelationshipType + 2 EREntityType
 */
inter_class
( [fdfunctionalsource[2], fdfunctional[2], fdfunctionaltarget[2]], [errelationshiptype, erentitytype[2]],
  label (t4_relationship),
  invariants
  ( member(erentitytype[1], errelationshiptype.entities()),
    member(erentitytype[2], errelationshiptype.entities()),
    member(errelationshiptype, erentitytype[1].relationships),
    member(errelationshiptype, erentitytype[2].relationships),
    count(errelationshiptype.attributes[]) = 0,
    fdfunctionalsource[1].dependency = fdfunctional[1],
    fdfunctionaltarget[1].dependency = fdfunctional[1],
    fdfunctional[1].source = fdfunctionalsource[1],
    fdfunctional[1].destination = fdfunctionaltarget[1],
    fdfunctionalsource[2].dependency = fdfunctional[2],
    fdfunctionaltarget[2].dependency = fdfunctional[2],
    fdfunctional[2].source = fdfunctionalsource[2],
    fdfunctional[2].destination = fdfunctionaltarget[2]
  ),
  equivalences
  ( fdfunctional[1].name = erentitytype[1].name,
    fdfunctional[2].name = erentitytype[2].name,

```

```

    fdfunctionalsource[1] = erentitytype[1].identifier,
    => get_fd_attr(fdfunctionalsource[1].attributes, fdfunctionaltarget[1].attributes, erentitytype[1].attributes),
    <=< get_nonkey(fdfunctionaltarget[1].attributes, erentitytype[1].attributes),
    fdfunctionalsource[2] = erentitytype[2].identifier,
    => get_fd_attr(fdfunctionalsource[2].attributes, fdfunctionaltarget[2].attributes, erentitytype[2].attributes),
    <=< get_nonkey(fdfunctionaltarget[2].attributes, erentitytype[2].attributes)
  ),
  initialisers
  (
    <=< append('target', erentitytype[1].name, fdfunctionaltarget[1].name),
    <=< append('source', erentitytype[1].name, fdfunctionalsource[1].name),
    <=< append('target', erentitytype[2].name, fdfunctionaltarget[2].name),
    <=< append('source', erentitytype[2].name, fdfunctionalsource[2].name)
  )
)
).

/* T5: 2 FDFunctionalSource + 2 FDFunctional + 2 FDFunctionalTarget +
 *      2 FDMultiSource + 2 FDMultiValued + 2 FDMultiTarget <=<=
 *      ERRelationshipType (*:M-*:N) + 2 EREntityType
 */
([fdfunctionalsource[2], fdfunctional[2], fdfunctionaltarget[2],
 fdmultisource[2], fdmultivalued[2], fdmultitarget[2]],
 [errelationshipstype, erentitytype[2]],
 label (t5_many_to_many),
 direction (<=<=),
 invariants
  (
    errelationshipstype.source = erentitytype[1],
    errelationshipstype.target = erentitytype[2],
    fdfunctionalsource[1].dependency = fdfunctional[1],
    fdfunctionaltarget[1].dependency = fdfunctional[1],
    fdfunctional[1].source = fdfunctionalsource[1],
    fdfunctional[1].target = fdfunctionaltarget[1],
    fdfunctionalsource[2].dependency = fdfunctional[2],
    fdfunctionaltarget[2].dependency = fdfunctional[2],
    fdfunctional[2].source = fdfunctionalsource[2],
    fdfunctional[2].target = fdfunctionaltarget[2],
    fdmultisource[1].dependency = fdmultivalued[1],
    fdmultitarget[1].dependency = fdmultivalued[1],
    fdmultivalued[1].source = fdmultisource[1],
    fdmultivalued[1].target = fdmultitarget[1],
    fdmultisource[2].dependency = fdmultivalued[2],
    fdmultitarget[2].dependency = fdmultivalued[2],
    fdmultivalued[2].source = fdmultisource[2],
    fdmultivalued[2].target = fdmultitarget[2],
    fdfunctionalsource[1].attributes = fdmultisource[1].attributes,
    fdmultisource[1].attributes = fdmultitarget[2].attributes,
    fdfunctionalsource[2].attributes = fdmultisource[2].attributes,
    fdmultisource[2].attributes = fdmultitarget[1].attributes
  ),
  equivalences
  (
    fdfunctional[1].name = erentitytype[1].name,
    fdfunctional[2].name = erentitytype[2].name,
    fdfunctionalsource[1] = erentitytype[1].identifier,
    fdfunctionalsource[2] = erentitytype[2].identifier,
    get_nonkey(fdfunctionaltarget[1].attributes, erentitytype[1].attributes),
    get_nonkey(fdfunctionaltarget[2].attributes, erentitytype[2].attributes)
  ),
  initialisers
  (
    append('target', erentitytype[1].name, fdfunctionaltarget[1].name),
    append('source', erentitytype[1].name, fdfunctionalsource[1].name),
    append('target', erentitytype[2].name, fdfunctionaltarget[2].name),
    append('source', erentitytype[2].name, fdfunctionalsource[2].name)
  )
)
).

/* T6: FDFunctionalSource <=> ERIdentifier */
inter_class
([fdfunctionalsource], [eridentifier],
 label (t6_lhs_id),
 invariants
  (
    eridentifier.partial = false
  ),
  equivalences
  (
    fdfunctionalsource.name = eridentifier.name,
    fdfunctionalsource.attributes[] = eridentifier.attributes[]
  )
)
).

/* T7: FDFunctionalSource + FDFunctional + FDFunctionalTarget <=<=
 *      ERWeakEntityType + ERRelationshipType + EREntityType
 */
inter_class
([fdfunctionalsource, fdfunctional, fdfunctionaltarget], [erweakentitytype, errelationshipstype, erentitytype],
 label (t7_weak),
 direction (<=<=),
 invariants
  (
    fdfunctionalsource.dependency = fdfunctional,
    fdfunctionaltarget.dependency = fdfunctional,
    fdfunctional.source = fdfunctionalsource,
    fdfunctional.target = fdfunctionaltarget,
    member(erweakentitytype, errelationshipstype.entities),

```

```

    member(erentitytype, errelationship.entities),
    member(errelationship.type, erweakentitytype.relationships),
    member(errelationship.type, erentitytype.relationships),
    ( errelationship.type.id_dependent = true;
      errelationship.type.existence_dependent = true
    ),
    erweakentitytype.dependentVia = errelationship.type
  )
).

/* Scheme-level rules */

/* S1: SSSingleKeyBubble + SSSingleValued + SSTargetBubble <=> MartinRegularEntity */
inter_class
( [sssinglekeybubble, sssinglevalued, sstargetbubble], [martinregularentity],
  label (s1_fd_regular),
  inherits (t1_fd_entity)
).

/* S2: SSMultiKeyBubble + SSMultiValued + SSendKeyBubble <=> MartinRegularEntity + MartinIdentifier */
inter_class
( [ssmultikeybubble, ssmultivalued, ssendkeybubble], [martinregularentity, martinidentifier],
  label (s2_mvd_regular),
  inherits (t2_mvd_entity)
).

/* S3: SSSingleKeyBubble <=> MartinIdentifier */
inter_class
( [sssinglekeybubble], [martinidentifier],
  label (s3_sk_id),
  inherits (t6_lhs_id)
).

/* S4: SSIsoatedBubble <=> MartinRegularEntity */
inter_class
( [ssisolatedbubble], [martinregularentity, martinidentifier],
  label (s4_isolated_regular),
  invariants
  ( martinregularentity.identifier = martinidentifier,
    martinidentifier.entity = martinregularentity,
    count(martinregularentity.attributes[]) =\= 2,
    martinregularentity.attributes[] = martinidentifier.attributes[]
  ),
  equivalences
  ( ssisolatedbubble.name = martinregularentity.name,
    ssisolatedbubble.attributes[] = martinregularentity.attributes[],
    ssisolatedbubble.attributes[] = martinidentifier.attributes[]
  )
).

/* S5: SSAttribute <=> MartinAttribute */
inter_class
( [ssattribute], [martinattribute],
  label (s5_attr_attr),
  inherits (t3_attr_attr)
).

/* S6: SSSingleKeyBubble + SSSingleValued + SSTargetBubble <=< MartinAssociativeEntity
inter_class
( [sssinglekeybubble, sssinglevalued, sstargetbubble], [martinassociativeentity],
  label (s6_fd_assoc),
  direction (<<=),
  invariants
  ( sssinglevalued.source = sssinglekeybubble,
    sssinglevalued.destination = sstargetbubble,
    sssinglekeybubble.dependency = sssinglevalued,
    sstargetbubble.dependency = sssinglevalued,
    count(martinassociativeentity.attributes[]) =\= count(martinassociativeentity.identifier=>attributes[])
  ),
  equivalences
  ( sssinglevalued.name = martinassociativeentity.name,
    sssinglekeybubble = martinassociativeentity.identifier,
    => get_fd_attr(sssinglekeybubble.attributes, sstargetbubble.attributes, erentitytype.attributes),
    <<= get_nonkey(fdfunctionaltarget.attributes, martinassociativeentity.attributes)
  ),
  initialisers
  ( <<= append('target', martinassociativeentity.name, sstargetbubble.name),
    <<= append('source', martinassociativeentity.name, sssinglekeybubble.name)
  )
).

/* S7: SSSingleKeyBubble + SSSingleValued + SSTargetBubble <=<
 *      MartinWeakEntity (key/no rel/attr) + dependent MartinRelationship + ERTypeItem
 */
inter_class
( [sssinglekeybubble, sssinglevalued, sstargetbubble], [martinweakentity, martinrelationship, ertypeitem],
  label (s7_weak_key_attr),
  inherits (t7_weak),
  direction (<<=),

```

```

invariants
( count(martinweakentity.identifier=>attributes[]) > 0,
  count(martinweakentity.attributes[]) > 0,
  count(martinweakentity.relationships[]) <= 1,
  martinrelationship.srcOpt = 1,
  martinrelationship.srcCard = 1,
  martinrelationship.dstCard > 1,
  martinrelationship.source = ertypeitem,
  martinrelationship.target = erweakentitytype
),
equivalences
( build_weak_pk(ertypeitem, martinweakentity, [], sssinglekeybubble),
  get_nonkey(sstargetbubble.attributes, martinweakentity.attributes)
),
initialisers
( martinrelationship.dstOpt = 0,
  martinrelationship.dstCard = 2
)
).

/* S8: SSSingleKeyBubble + SSSingleValued + SSTargetBubble <=
 *      MartinWeakEntity (key/rel/attr) + dependent MartinRelationship + ERTypeItem
 */
inter_class
( [sssingkeybubble, sssinglevalued, sstargetbubble], [martinweakentity, martinrelationship, ertypeitem],
  label (s8_weak_key_attr_rel),
  inherits (t7_weak),
  direction (<=),
  invariants
( count(martinweakentity.identifier=>attributes[]) > 0,
  count(martinweakentity.attributes[]) > 0,
  count(martinweakentity.relationships[]) >= 1,
  martinrelationship.srcOpt = 1,
  martinrelationship.srcCard = 1,
  martinrelationship.dstCard > 1,
  martinrelationship.source = ertypeitem,
  martinrelationship.target = erweakentitytype
),
equivalences
( build_weak_pk(ertypeitem, martinweakentity, [], sssinglekeybubble),
  get_nonkey(sstargetbubble.attributes, martinweakentity.relationships, martinweakentity.attributes)
),
initialisers
( martinrelationship.dstOpt = 0,
  martinrelationship.dstCard = 2
)
).

/* S9: SSSingleKeyBubble + SSSingleValued + SSTargetBubble <=
 *      MartinWeakEntity (no key/rel/attr) + dependent MartinRelationship + ERTypeItem
 */
inter_class
( [sssingkeybubble, sssinglevalued, sstargetbubble], [martinweakentity, martinrelationship, ertypeitem],
  label (s9_weak_rel_attr),
  inherits (t7_weak),
  direction (<=),
  invariants
( count(martinweakentity.identifier=>attributes[]) = 0,
  count(martinweakentity.attributes[]) > 0,
  count(martinweakentity.relationships[]) >= 1,
  martinrelationship.srcOpt = 1,
  martinrelationship.srcCard = 1,
  martinrelationship.dstCard > 1,
  martinrelationship.source = ertypeitem,
  martinrelationship.target = erweakentitytype
),
equivalences
( build_weak_pk(ertypeitem, [], martinweakentity.relationships, sssinglekeybubble),
  get_nonkey(sstargetbubble.attributes, martinweakentity.attributes)
),
initialisers
( martinrelationship.dstOpt = 0,
  martinrelationship.dstCard = 2
)
).

/* S10: SSIsoatedBubble <= MartinWeakEntity (no key/rel/no attr) + dependent MartinRelationship + ERTypeItem */
inter_class
( [ssisolatedbubble], [martinweakentity, martinrelationship, ertypeitem],
  label (s10_weak_rel),
  direction (<=),
  invariants
( member(martinweakentity, martinrelationship.entities),
  member(ertypeitem, martinrelationship.entities),
  member(martinrelationship, martinweakentity.relationships),
  member(martinrelationship, ertypeitem.relationships),
  count(martinweakentity.relationships[]) >= 1
)
)

```

```

    ( martinrelationship.id_dependent = true;
      martinrelationship.existence_dependent = true
    ),
    martinweakentity.dependentVia = martinrelationship,
    count(martinweakentity.identifier=>attributes[]) = 0,
    count(martinweakentity.attributes[]) = 0,
    martinrelationship.srcOpt = 1,
    martinrelationship.srcCard = 1,
    martinrelationship.dstCard > 1,
    martinrelationship.source = ertypeitem,
    martinrelationship.target = erweakentitytype
  ),
  equivalences
  (
    ( ssisolatedbubble.name = martinweakentity.name,
      build_weak_pk(ertypeitem, [], martinweakentity.relationships, ssisolatedbubble)
    ),
  ),
  initialisers
  (
    ( martinrelationship.dstOpt = 0,
      martinrelationship.dstCard = 2
    )
  )
).

/* S11: 2 SSSingleKeyBubble + 2 SSSingleValued + 2 SSTargetBubble <=> MartinRelationship (*:1-*:1) + 2 non-weak ERTypeItem
*/
inter_class
(
  [ sssinglekeybubble[2], sssinglevalued[2], sstargetbubble[2]],
  [ martinrelationship, ertypeitem[2]],
  label (s11_rel_11),
  inherits (t4_relationship),
  invariants
  (
    ( martinrelationship.srcCard = 1,
      martinrelationship.dstCard = 1,
      martinrelationship.source = ertypeitem[1],
      martinrelationship.target = ertypeitem[2],
      \+ertypeitem[]@class('martinweakentity'),
      subset(sssinglekeybubble[1].attributes, sstargetbubble[2].attributes),
      subset(sssinglekeybubble[2].attributes, sstargetbubble[1].attributes)
    )
  ),
  initialisers
  (
    ( martinrelationship.srcOpt = 0,
      martinrelationship.dstOpt = 0
    )
  )
).

/* S12: 2 SSSingleKeyBubble + 2 SSSingleValued + 2 SSTargetBubble <=> MartinRelationship (*:1-*:N) + 2 non-weak ERTypeItem
*/
inter_class
(
  [ sssinglekeybubble[2], sssinglevalued[2], sstargetbubble[2]], [ martinrelationship, erentitytype[2]],
  label (s12_rel_1N),
  inherits (t4_relationship),
  invariants
  (
    ( martinrelationship.srcCard = 1,
      martinrelationship.dstCard > 1,
      martinrelationship.source = ertypeitem[1],
      martinrelationship.target = ertypeitem[2],
      \+ertypeitem[]@class('martinweakentity'),
      subset(sssinglekeybubble[2].attributes, sstargetbubble[1].attributes)
    )
  ),
  initialisers
  (
    ( martinrelationship.srcOpt = 0,
      martinrelationship.dstOpt = 0,
      martinrelationship.dstCard = 2
    )
  )
).

/* S13: 2 SSSingleKeyBubble + 2 SSSingleValued + 2 SSTargetBubble +
*       2 SSMultiKeyBubble + 2 SSMultiValued + 2 SSendKeyBubble <=< MartinRelationship (*:M-*:N) + 2 non-weak ERTypeItem
*/
inter_class
(
  [ sssinglekeybubble[2], sssinglevalued[2], sstargetbubble[2],
    ssmultikeybubble[2], ssmultiValued[2], ssendkeybubble[2]],
  [ martinrelationship, martinentity[]],
  label (s13_rel_mn),
  inherits (t5_many_to_many),
  direction (<<=),
  invariants
  (
    ( errelationshipiptype.srcCard > 1,
      errelationshipiptype.dstCard > 1,
      martinrelationship.source = ertypeitem[1],
      martinrelationship.target = ertypeitem[2],
      \+ertypeitem[]@class('martinweakentity')
    )
  ),
  initialisers
  (
    ( martinrelationship.srcOpt = 0,
      martinrelationship.dstOpt = 0,
      martinrelationship.srcCard = 2,
      martinrelationship.dstCard = 2
    )
  )
).

```



```

/* S14: FDAttributeSet + SSDomainFlag + SSSingleKeyBubble + 2 SSAttribute => MartinRelationship */
inter_class
( [fdattributeset, ssdomainflag, sssinglekeybubble, ssattribute[2]], [martinrelationship],
  label (s14_domain_flag), direction (=>),
  invariants
  ( \+(fdattributeset@class('ssisolatedbubble')),
    member(ssattribute[1], fdattributeset.attributes),
    member(ssattribute[2], sssinglekeybubble.attributes),
    member(fdattributeset, ssattribute[1].attributeSets),
    member(sssinglekeybubble, ssattribute[2].attributeSets),
    ssdomainflag.source = ssattribute[2],
    member(ssattribute[1], ssdomainflag.attributes),
    ssattribute[1].dfSource = ssdomainflag,
    ssattribute[2].dfReference = ssdomainflag,
    martinrelationship.srcOpt = 1,
    martinrelationship.srcCard = 1,
    martinrelationship.dstCard > 1
  ),
  equivalences
  ( ssdomainflag.name = martinrelationship.name,
    fdattributeset.dependency = martinrelationship.destination,
    sssinglekeybubble.dependency = martinrelationship.source
  ),
  initialisers
  ( martinrelationship.dstCard = 2,
    martinrelationship.dstOpt = 0
  )
).

/* S15: SSIsoatedBubble + SSDomainFlag + SSSingleKeyBubble + 2 SSAttribute => MartinRelationship */
inter_class
( [ssisolatedbubble, ssdomainflag, sssinglekeybubble, ssattribute[2]], [martinrelationship],
  label (s15_domain_flag_iso),
  direction (=>),
  invariants
  ( member(ssattribute[1], ssisolatedbubble.attributes),
    member(ssattribute[2], sssinglekeybubble.attributes),
    member(ssisolatedbubble, ssattribute[1].attributeSets),
    member(sssinglekeybubble, ssattribute[2].attributeSets),
    ssdomainflag.source = ssattribute[2],
    member(ssattribute[1], ssdomainflag.attributes),
    ssattribute[1].dfSource = ssdomainflag,
    ssattribute[2].dfReference = ssdomainflag,
    martinrelationship.srcOpt = 1,
    martinrelationship.srcCard = 1,
    martinrelationship.dstCard > 1
  ),
  equivalences
  ( ssdomainflag.name = martinrelationship.name,
    ssisolatedbubble = martinrelationship.destination,
    sssinglekeybubble.dependency = martinrelationship.source
  ),
  initialisers
  ( martinrelationship.dstCard = 2,
    martinrelationship.dstOpt = 0
  )
).

/* S16: SSSingleKeyBubble + non-isolated FDAttributeSet => 2 ERTYPEITEM + MartinRelationship (1:1-0:N) */
inter_class
( [sssinglekeybubble, fdattributeset], [ertypeitem[2], martinrelationship],
  label (s16_contained_sk),
  direction (=>),
  invariants
  ( subset(sssinglekeybubble.attributes, fdattributeset.attributes),
    \+(fdattributeset@class('ssisolatedbubble')),
    martinrelationship.source = ertypeitem[1],
    martinrelationship.destination = ertypeitem[2],
    member(martinrelationship, ertypeitem[1].relationships),
    martinrelationship.srcOpt = 1,
    martinrelationship.srcCard = 1,
    martinrelationship.dstCard > 1,
    martinrelationship.source = ertypeitem[1],
    martinrelationship.target = ertypeitem[2]
  ),
  equivalences
  ( sssinglekeybubble.dependency = ertypeitem[1],
    fdattributeset.dependency = ertypeitem[2]
  ),
  initialisers
  ( martinrelationship.dstCard = 2,
    martinrelationship.dstOpt = 0,
    append(fdattributeset.dependency=>name, sssinglekeybubble.dependency=>name, martinrelationship.name)
  )
).

```

```

/* S17: SSIsolatedBubble + non-isolated FDAttributeSet =>
*      2 ERTypeItem + MartinRelationship (1:1-0:N)
*/
inter_class
( [ssisolatedbubble, fdattributeset], [ertypeitem[2], martinrelationship],
  label (s17_contained_iso),
  direction (=>),
  invariants
  ( subset(ssisolatedbubble.attributes, fdattributeset.attributes),
    \+(fdattributeset@class('ssisolatedbubble')),
    martinrelationship.source = ertypeitem[1],
    martinrelationship.destination = ertypeitem[2],
    member(martinrelationship, ertypeitem[1].relationships),
    martinrelationship.srcOpt = 1,
    martinrelationship.srcCard = 1,
    martinrelationship.dstCard > 1,
    martinrelationship.source = ertypeitem[1],
    martinrelationship.target = ertypeitem[2]
  ),
  equivalences
  ( ssisolatedbubble = ertypeitem[1],
    fdattributeset.dependency = ertypeitem[2],
  ),
  initialisers
  ( martinrelationship.dstCard = 2,
    martinrelationship.dstOpt = 0,
    append(fdattributeset.dependency=>name, ssisolatedbubble.name, martinrelationship.name)
  )
).

/* S18: SSSingleKeyBubble + SSIsolatedBubble => 2 ERTypeItem + MartinRelationship (1:1-0:N) */
inter_class
( [sssinglekeybubble, ssisolatedbubble], [ertypeitem[2], martinrelationship],
  label (s18_isolated_sk),
  direction (=>),
  invariants
  ( subset(sssinglekeybubble.attributes, ssisolatedbubble.attributes),
    martinrelationship.source = ertypeitem[1],
    martinrelationship.destination = ertypeitem[2],
    member(martinrelationship, ertypeitem[1].relationships),
    martinrelationship.srcOpt = 1,
    martinrelationship.srcCard = 1,
    martinrelationship.dstCard > 1,
    martinrelationship.source = ertypeitem[1],
    martinrelationship.target = ertypeitem[2]
  ),
  equivalences
  ( sssinglekeybubble.dependency = ertypeitem[1],
    ssisolatedbubble = ertypeitem[2]
  ),
  initialisers
  ( martinrelationship.dstCard = 2,
    martinrelationship.dstOpt = 0,
    append(ssisolatedbubble.name, sssinglekeybubble.dependency=>name, martinrelationship.name)
  )
).

/* S19: 2 SSIsolatedBubble => 2 ERTypeItem + MartinRelationship (1:1-0:N) */
inter_class
( [ssisolatedbubble[2]], [ertypeitem[2], martinrelationship],
  label (s19_isolated_iso),
  direction (=>),
  invariants
  ( subset(ssisolatedbubble[2].attributes, ssisolatedbubble[1].attributes),
    martinrelationship.source = ertypeitem[1],
    martinrelationship.destination = ertypeitem[2],
    member(martinrelationship, ertypeitem[1].relationships),
    martinrelationship.srcOpt = 1,
    martinrelationship.srcCard = 1,
    martinrelationship.dstCard > 1,
    martinrelationship.source = ertypeitem[1],
    martinrelationship.target = ertypeitem[2]
  ),
  equivalences
  ( ssisolatedbubble[2] = ertypeitem[1],
    ssisolatedbubble[1] = ertypeitem[2]
  ),
  initialisers
  ( martinrelationship.dstCard = 2,
    martinrelationship.dstOpt = 0,
    append(ssisolatedbubble[1].name, ssisolatedbubble[2].name, martinrelationship.name)
  )
).

```

```

/* Heuristics */

/* H1: 2 x SSMultiKeyBubble + 2 x SSMultiValued + 2 x SSendKeyBubble =>
 *      MartinAssociativeEntity + MartinIdentifier
 */
inter_class
( [ssmultikeybubble[2], ssmultivalued[2], ssendkeybubble[2]],
  [martinassociativeentity, martinidentifier],
  label (hl_2_multi),
  direction (=>>),
  invariants
  ( martinidentifier.entity = martinassociativeentity,
    martinassociativeentity.identifier = martinidentifier,
    martinidentifier.partial = false,
    martinidentifier.attributes[] = martinassociativeentity.attributes[],
    ssmultikeybubble[1].dependency = ssmultivalued[1],
    ssendkeybubble[1].dependency = ssmultivalued[1],
    ssmultivalued[1].source = ssmultikeybubble[1],
    ssmultivalued[1].destination = ssendkeybubble[1],
    ssmultikeybubble[2].dependency = ssmultivalued[2],
    ssendkeybubble[2].dependency = ssmultivalued[2],
    ssmultivalued[2].source = ssmultikeybubble[2],
    ssmultivalued[2].destination = ssendkeybubble[2],
    ssmultikeybubble[1].attributes = ssendkeybubble[2].attributes,
    ssmultikeybubble[2].attributes = ssendkeybubble[1].attributes
  ),
  equivalences
  ( get_fd_attr(ssmultikeybubble[1].attributes, ssmultikeybubble[2].attributes, martinassociativeentity.attributes)
  ),
  initialisers
  ( append(ssmultivalued[2].name, ssmultivalued[1].name, martinassociativeentity.name) )
).

/* H2: n FDAttributeSet + SSDomainFlag + SSSingleKeyBubble +
 *      "parent" SSAttribute + n "child" SSAttribute => MartinTypeHierarchy
 */
inter_class
( [fdattributeset[], ssdomainflag, sssinglekeybubble, ssattribute parent, ssattribute[] children],
  [martintypehierarchy],
  label (h2_subtypes),
  direction (=>>),
  invariants
  ( count(children[]) > 2,
    \+(fdattributeset[]@class('ssmultikeybubble')),
    member(children[], fdattributeset[].attributes),
    member(parent, sssinglekeybubble.attributes),
    member(fdattributeset[], children[].attributeSets),
    member(sssinglekeybubble, sssinglekeybubble.attributeSets),
    ssdomainflag.target = parent,
    member(children[], ssdomainflag.attributes),
    parent.domainFlag = ssdomainflag,
    children[].domainFlag = ssdomainflag
  ),
  equivalences
  ( ssdomainflag.name = martintypehierarchy.name,
    parent.dependency = martintypehierarchy.supertype,
    children[].dependency = martintypehierarchy.subtypes[]
  ),
  initialisers
  ( martintypehierarchy.exclusive = true )
).

/* Auxiliary functions -- not tested! */

/* Given an EREntity, determine whether there are any non-key ERAttributes. */
nonkey_exist(Entity) :-
  Entity@primaryKey(PK),
  Entity@attributes(Attr),
  PK@attributes(PKAttr),
  length(Attr) =\= length(PKAttr).

/* Combine the FDAttributes from two FDAttributeSets and put them into an EREntity. */
get_fd_attr([], _).
get_fd_attr([A|Ar], Entity) :-
  get_fd_attr(Ar, Entity),
  A@name(Name),
  erattribute@create(Ea),
  Ea@name := Name,
  Entity@attributes(EAttr),
  append([Ea], EAttr, NewEAttr),
  Entity@attributes := NewEAttr.
get_fd_attr(A1, A2, Entity) :-
  get_fd_attr(A1, Entity),
  get_fd_attr(A2, Entity).

```

```

/* Given an EREntity, map its ERAttributes to an FDAttributeSet.
* OR: Given an EREntity + a group of ERRelationships, map the ERAttributes of
* the EREntity + the key ERAttributes of each of the EREntities attached to the
* ERRelationships.
* Note this is only really useful in the reverse direction (FDD <= ERD).
*/
get_nonkey(_, []).
get_nonkey([Fa|FR], [Ea|Er]) :-
    get_nonkey(Fr, Er),
    A@name(Name),
    erattribute@create(Ea),
    Ea@name := Name,
    Entity@attributes(EAttr),
    append([Eal], EAttr, NewEAttr),
    Entity@attributes := NewEAttr.

```

### F.3.3 $\mathfrak{R}_e(E-R, ERD_{Martin}) \rightleftharpoons \mathfrak{R}_d(DataFlow, DFD_{G\&S})$

```
inter_view(er_martinerd, read_write, process_gnsdfd, read_write, partial).
```

```

/* Technique-level rules */

/* T1: EREntityType <=> DFDataStore */
inter_class
( [erentitytype], [dfdatastore],
  label (t1_entity_datastore),
  equivalences
  ( erentitytype.name = dfdatastore.name,
    erentitytype.attributes[] = dfdatastore.fieldItems[]
  )
).

/* T2: ERAttributeItem <=> DFFieldItem */
inter_class
( [erattributeitem], [dffielditem],
  label (t2_attr_field),
  equivalences
  ( erattributeitem.name = dffielditem.name,
    erattributeitem.containingItem = dffielditem.containingItem
  )
).

/* Scheme-level rules */

/* S1: MartinRegularEntity <=> GnSDataStore */
inter_class
( [martinregularentity], [gnsdatastore],
  label (s1_regular_datastore),
  inherits (t1_entity_datastore)
).

/* S2: MartinAssociativeEntity =>> GnSDataStore */
inter_class
( [martinassociativeentity], [gnsdatastore],
  label (s2_assoc_datastore),
  direction (=>>),
  equivalences
  ( martinassociativeentity.name = gnsdatastore.name,
    martinassociativeentity.attributes[] = gnsdatastore.fieldItems[]
  )
).

/* S3: non-embedded MartinWeakEntity =>> GnSDataStore */
inter_class
( [martinweakentity], [gnsdatastore],
  label (s3_weak_datastore),
  inherits (t1_entity_datastore),
  direction (=>>),
  invariants
  ( embedded = false )
).

/* S4: MartinAttribute <=> GnSField */
inter_class
( [martinattribute], [gnsfield],
  label (s4_attr_field),
  inherits (t2_attr_field)
).

/* S5: MartinAttributeGroup <=> GnSFieldGroup */
inter_class
( [martinattributegroup], [gnsfieldgroup],
  label (s5_attrgroup_fieldgroup),
  inherits (t2_attr_field),
  equivalences
  ( martinattributegroup.attributeItems[] = gnsfieldgroup.fieldItems[] )
).

```

```

/* S6: embedded MartinWeakEntity <=> GnSFieldGroup */
inter_class
( [martinweakentity], [gnsfieldgroup],
  label (s6_weak_datastore),
  direction (=>>),
  invariants
  ( embedded = true ),
  equivalences
  ( martinweakentity.name = gnsfieldgroup.name,
    martinweakentity.attributes[] = gnsfieldgroup.fieldItems[]
  )
).

/* Heuristics */

/* H1: MartinAssociativeEntity =>> GnSDataProcess + GnSDataStore + GnSDataFlow */
inter_class
( [martinassociativeentity], [gnsdataprocess, gnsdatastore, gnsdataflow]
  label (h1_assoc_process),
  direction (=>>),
  equivalences
  ( martinassociativeentity = gnsdatastore ),
  initialisers
  ( gnsdataflow.source = gnsdatastore,
    gnsdataflow.destination = gnsdataprocess
  )
).

/* H2: MartinRelationship + 2 MartinRegularEntity <=< GnSDataProcess + 2 GnSDataStore + 2 GnSDataFlow */
inter_class
( [martinrelationship, martinregularentity[2]], [gnsdataprocess, gnsdatastore[2], gnsdataflow[2]]
  label (h2_rel_process),
  direction (<<=),
  invariants
  ( martinrelationship.source = martinregularentity[1],
    martinrelationship.destination = martinregularentity[2],
    member(martinrelationship, martinregularentity[1].relationships),
    member(martinrelationship, martinregularentity[2].relationships),
    martinregularentity.srcOpt = 1,
    martinregularentity.srcCard = 1,
    martinregularentity.dstOpt = 0,
    martinregularentity.dstCard > 1,
    gnsdataflow[1].source = gnsdatastore[1],
    gnsdataflow[1].destination = gnsdataprocess,
    gnsdataflow[2].source = gnsdataprocess,
    gnsdataflow[2].destination = gnsdatastore[2],
    member(gnsdataflow[1], gnsdatastore[1].flows),
    member(gnsdataflow[1], gnsdataprocess),
    member(gnsdataflow[2], gnsdatastore[2].flows),
    member(gnsdataflow[2], gnsdataprocess)
  ),
  equivalences
  ( martinregularentity[] = gnsdatastore[] ),
  initialisers
  ( martinregularentity.dstCard = 2 )
).

```



# Appendix G

## The Swift repository

### G.1 Glossary of representation tags

In this section are listed the identifying tags used by Swift's description modelling unit to distinguish different types of repository items (such as representations, constructs, and so on). Tags have been included for all the representations used in the thesis.

#### G.1.1 $\mathcal{R}_f(\text{FuncDep}, \text{FDD}_{\text{Smith}})$

This representation is implemented by classes in the package `swift.repn.fdepsmit`. The core class is `SmithFDD`.




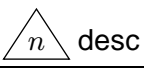
#### General

| Item             | Description                         | Tag    |
|------------------|-------------------------------------|--------|
| Technique        | functional dependencies             | 'fdep' |
| Scheme           | Smith functional dependency diagram | 'smit' |
| Description type | functional dependency diagram       | 'fdd_' |

#### Connectors

| Construct      | Graphic                      | Description              | Tag    |
|----------------|------------------------------|--------------------------|--------|
| SSSINGLEVALUED | $\longrightarrow$            | single valued dependency | 'svdp' |
| SSMULTIVALUED  | $\longrightarrow\rightarrow$ | multi valued dependency  | 'mvdp' |

## Symbols

| Construct         | Graphic  | Swift class                   | Tag    |
|-------------------|--|-------------------------------|--------|
| SSATTRIBUTE       | Attr   | Attribute                     | 'attr' |
| SSSINGLEKEYBUBBLE |   | Bubble <sup>a</sup>           | 'bubl' |
| SSTARGETBUBBLE    |  |                               |        |
| SSMULTIKEYBUBBLE  |  |                               |        |
| SSENDKEYBUBBLE    |  |                               |        |
| SSIISOLATEDBUBBLE |  |                               |        |
| SSDOMAINFLAG      | <br> | DomainFlag <sup>b</sup>       | 'df1g' |
|                   |  | DomainFlagSource <sup>b</sup> | 'dfsr' |
| n/a               |   | DomainFlagDesc                | 'dfds' |

### Notes:

- <sup>a</sup> The different kinds of bubbles are distinguished by an internal variable of the Bubble class called bubbleKind. This is set to one of the values 'skey' (single-key), 'mkey' (multi-key), 'targ' (target), 'ekey' (end-key), 'ckey' (chain key — used internally) or 'isol' (isolated).
- <sup>b</sup> These will probably be combined at some stage.

### G.1.2 $\mathcal{R}_e(E-R, ERD_{Martin})$

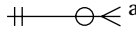


This representation is implemented by classes in the package `swift.repn.erm_mrtc`. The core class is `MartinERDc`.

### General

| Item             | Description  | Tag    |
|------------------|--|--------|
| Technique        | entity-relationship modelling                      | 'erm_' |
| Scheme           | Martin entity-relationship diagram ('crow's feet') | 'mrtc' |
| Description type | data flow diagram                                  | 'erd_' |



## Connectors

| Construct           | Graphic  | Swift class                | Tag    |
|---------------------|--|----------------------------|--------|
| MARTINRELATIONSHIP  |   | Relationship               | 'rshp' |
| MARTINTYPEHIERARCHY | <br> | SubtypeParent <sup>b</sup> | 'stpp' |
|                     |  | SubtypeChild <sup>b</sup>  | 'stpc' |

### Notes:

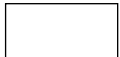

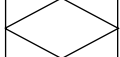
<sup>a</sup> Appearance varies. Source and destination flags specify cardinality and optionality as follows:

- flag[1]: true = many;
- flag[2]: true = optional;
- flag[3]: true = mandatory;
- flag[3]: true = one.

Flags may be used in combination where it makes sense. There are separate flags for optional/mandatory and many/one to allow for relationships with unknown optionality and cardinality.

<sup>b</sup> These will probably be combined at some stage.

## Symbols

| Construct               | Graphic   | Swift class       | Tag    |
|-------------------------|---|-------------------|--------|
| MARTINREGULARENTITY     |  | RegularEntity     | 'rent' |
| MARTINWEAKENTITY        |  | WeakEntity        | 'went' |
| MARTINASSOCIATIVEENTITY |  | AssociativeEntity | 'aent' |
| MARTINIDENTIFIER        | none  | n/a <sup>a</sup>  | n/a    |
| MARTINATTRIBUTE         | none  | n/a <sup>b</sup>  | n/a    |
| MARTINATTRIBUTEGROUP    | none  | n/a <sup>c</sup>  | n/a    |

### Notes:

- <sup>a</sup> This is effectively subsumed by the class swift.repn.Constraint.
- <sup>b</sup> This is dealt with by the class swift.repn.Attribute.
- <sup>c</sup> This is dealt with by the class swift.repn.Record.


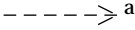
### G.1.3 $\mathcal{R}_d(DataFlow, DFD_{G\&S})$

This representation is implemented by classes in the package `swift.repn.procgnsn`. The core class is `GaneSarsonDFD`.

#### General

| Item             | Description                     | Tag    |
|------------------|---------------------------------|--------|
| Technique        | data flow modelling             | 'proc' |
| Scheme           | Gane & Sarson data flow diagram | 'gnsn' |
| Description type | data flow diagram               | 'dfd_' |


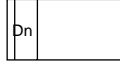





#### Connectors

| Construct       | Graphic   | Swift class  | Tag    |
|-----------------|---|--------------|--------|
| GNSDATAFLOW     |  | DataFlow     | 'dflo' |
| GNSRESOURCEFLOW |  | ResourceFlow | 'rflo' |

#### Notes:

<sup>a</sup> Arrow points from source item to destination item.

#### Symbols

| Construct                 | Graphic   | Swift class      | Tag    |
|---------------------------|---|------------------|--------|
| GNSEXTERNALENTITY[]       |  | ExternalEntity   | 'eent' |
| GNSDATASTORE[]            |  | DataStore        | 'dstr' |
| GNSRESOURCESTORE          |  | ResourceStore    | 'rstr' |
| GNSDATAPROCESS            |  | DataProcess      | 'proc' |
| GNSMULTIPLIEDATAPROCESS   |  | MultiDataProcess | 'mprc' |
| GNSSPLITMERGE             |  | SplitMerge       | 'spmg' |
| GNSINTERFACE <sup>a</sup> |  | Interface        | 'intf' |

#### Notes:

<sup>a</sup> Automatically generated if you start or terminate a data flow outside any symbol.

## G.1.4 $\mathfrak{R}_r$ (*Relational, SQL/92*)

This representation is implemented by classes in the package `swift.repn.relmsq92`. The core class is `SQL92`.

### General

| Item             | Description      | Tag    |
|------------------|------------------|--------|
| Technique        | relational model | 'relm' |
| Scheme           | ANSI/ISO SQL/92  | 'sq92' |
| Description type | SQL schema       | 'sqls' |

### Connectors

| Construct       | Graphic  | Swift class | Tag |
|-----------------|--|-------------|-----|
| SQL92FOREIGNKEY | foreign key (<columns><br>references <table> (<columns>) | n/a         | n/a |

### Symbols

| Construct       | Graphic   | Swift class | Tag    |
|-----------------|---|-------------|--------|
| SQL92TABLE      | create table <tablename><br>(<column-defs>);              | Table       | 'tabl' |
| SQL92COLUMN     | <colname> <datatype><br>[<constraints>]                   | n/a         | n/a    |
| SQL92DOMAIN     | create domain <domainname><br><datatype> [<constraints>]; | n/a         | n/a    |
| SQL92PRIMARYKEY | primary key (<columns>)                                   | n/a         | n/a    |
| SQL92CONSTRAINT | none  | n/a         | n/a    |
| SQL92UNIQUE     | unique (<columns>)  | n/a         | n/a    |
| SQL92NOTNULL    | not null  | n/a         | n/a    |
| SQL92PREDICATE  | check (<predicate>)                                       | n/a         | n/a    |

## G.2 Full repository schema

### G.2.1 repository.sql

```
-----  
--  
-- "Static" classes contain information that is always available and rarely  
-- changes. "Non-static" classes contain information that varies on a regular  
-- basis. "Abstract" never have data stored in them.  
--  
-----  
  
-----  
-- Class S_REPOSITORY_ITEM [abstract]  
-- Generic "root class" for all "non-static" metadata-storing classes in the  
-- repository.  
--  
-- deleted: whether or not this item has been marked for deletion  
--  
create table s_repository_item  
( deleted bool  
);  
  
-----  
-- Class S_VIEWPOINT  
-- Contains viewpoint information.  
--  
-- vname: name of viewpoint  
-- creator: username of viewpoint creator  
--  
create table s_viewpoint  
( vname text,  
  creator text  
) inherits (s_repository_item);  
  
-----  
-- Class S_REPRESENTATION [static]  
-- Contains information about representations.  
--  
-- technique: representation technique, e.g., 'erm' for ER modelling  
-- techname:  technique name, e.g., 'Entity-relationship modelling'  
-- scheme:    representation scheme, e.g., 'mrtc' for Martin  
-- schemename: scheme name, e.g., 'Martin ERD with crows feet'  
-- javaclass: oid of Java class file for the representation?  
--            (currently name of Java class)  
--  
create table s_representation  
( technique char4,  
  techname text,  
  scheme char4,  
  schemename text,  
  javaclass text  
);  
  
-----  
-- Class S_TRANSLATION [static]  
-- Contains information about translations.  
--  
-- source:    source representation OID  
-- destination: destination representation OID  
-- javaclass: oid of Java class file for the translation?  
--            (currently name of Java class)  
--  
create table s_translation  
( source oid,  
  destination oid,  
  javaclass text  
);  
  
-----  
-- Class S_CONSTRUCT_GLOSSARY [static]  
-- Contains information about particular construct types.  
--  
-- representation: oid of appropriate representation  
-- ctype:          type of construct, e.g., 'rshp' for relationship  
-- fullname:      full name of the construct type, e.g., 'E-R relationship'  
--  
create table s_construct_glossary  
( representation oid,  
  ctype char4,  
  fullname text  
);
```

```

-----
-- Class S_DATATYPE [static]
-- Contains information about particular data types.
--
-- tcode: 4-character code identifying the data type
-- tname: name of the data type, e.g., 'fixed'
-- tdesc: description of the data type, e.g., 'fixed-point integer'
--
create table s_datatype
( tcode char4,
  tname char(20),
  tdesc text
);

-----
-- Class S_DESCRIPTION
-- Contains description information.
--
-- dname: name of description
-- dtype: type of description, e.g., 'erd' for ERD
-- representation: oid of appropriate representation
-- parent_id: oid of the parent description, if applicable (null if none)
--
create table s_description
( dname text,
  dtype char4,
  representation oid,
  parent_id oid,
  viewpoint_id oid
) inherits (s_repository_item);

-----
-- Class S_CONSTRUCT [abstract]
-- Generic construct information.
--
-- cname: name of construct
-- ctype: type of construct, e.g., 'enty' for ERD entity (varies depending
-- on representation used -- subclasses implicitly specify the
-- 'generic type' of the construct)
-- flags: miscellaneous construct features, where applicable (varies depending
-- on representation used)
--
create table s_construct
( cname text,
  ctype char4,
  flags int4
) inherits (s_repository_item);

-----
-- Class S_GRAPHIC_CONSTRUCT [abstract]
-- Generic information specific to "graphic" constructs.
--
-- label: text label to be displayed (use cname if empty)
-- desc_id: oid of enclosing description (note graphic constructs only appear on a single
-- description at any given time)
--
create table s_graphic_construct
( label text,
  desc_id oid
) inherits (s_construct);

-----
-- Class S_CONNECTOR
-- Contains information about connectors (relationships, data flows, etc).
--
-- src_flags: misc flags for the source end of the connector (e.g., for an
-- ER relationship, this stores cardinality and optionality info)
-- dst_flags: misc flags for the destination end of the connector
-- src_port: "port" to connect to on the source symbol (1-16)
-- dst_port: "port" to connect to on the destination symbol (1-16)
-- cpath: list of points that form the connector's path (unimplemented)
--
create table s_connector
( src_flags int4,
  dst_flags int4,
  src_port int2,
  dst_port int2,
  cpath point[]
) inherits (s_graphic_construct);

-----
-- Class S_SYMBOL
-- Contains information about symbols (entities, data stores, processes, etc).
--
-- bounds: bounding rectangle
--
create table s_symbol
( bounds box
) inherits (s_graphic_construct);

```

```

-----
-- Class S_TEXTBLOCK
-- Contains information about text blocks.
--
-- bounds: bounding rectangle
-- value: the actual text
--
create table s_textblock
( bounds box,
  value text
) inherits (s_graphic_construct);

-----
-- Class S_DICTIONARY_CONSTRUCT [abstract]
-- Generic information specific to "dictionary" constructs.
--
-- viewpoint_id: oid of enclosing viewpoint (i.e., dictionary constructs do not span
--               viewpoints)
--
create table s_dictionary_construct
( viewpoint_id oid
) inherits (s_construct);

-----
-- Class S_RECORD
-- Contains information about records of attributes.
--
create table s_record
(
) inherits (s_dictionary_construct);

-----
-- Class S_ATTRIBUTE
-- Contains information about attributes.
--
-- datatype: oid of the data type or domain of the attribute
-- size:     size of the attribute, where applicable (null if not)
-- dp:      number of decimal points, where applicable
--
create table s_attribute
( datatype oid,
  size smallint,
  dp smallint
) inherits (s_dictionary_construct);

-----
-- Class S_DOMAIN
-- Contains information about domains.
--
-- datatype: oid of the data type of the domain
-- size:     size of the domain, where applicable (null if not)
--
create table s_domain
( datatype oid,
  size smallint,
  dp smallint
) inherits (s_dictionary_construct);

-----
-- Class S_CONSTRAINT
-- Contains information about generic constraints. Includes keys (primary, foreign
-- and alternate).
--
-- contype: the type of the constraint, e.g., 'uniq' for unique
--
create table s_constraint
( contype char4
) inherits (s_dictionary_construct);

-----
-- Class S_DEFINITION
-- Contains textual definitions of other dictionary constructs.
--
-- value: the text of the definition
create table s_definition
( value text
) inherits (s_dictionary_construct);

-----
-- Class S_CONSTRUCT_LINK
-- Links any constructs with any other construct. Useful for implementing associations
-- between graphic constructs and dictionary constructs, attributes and keys, attributes
-- and record, etc, etc, etc.
--
-- Construct IDs are generally inserted such that parent is the "parent" construct.
-- Where there is no "parent", construct IDs may be inserted in arbitrary order.
-- Link types are as follows:
--

```

```

-- Type Description
-- graf graphic construct contains graphic construct
-- grdc graphic construct associated with dictionary construct
-- rcrd record contains record
-- ratr record contains attribute
-- atcn attribute has constraint
-- dmcn domain has constraint
-- defn dictionary construct has definition
-- pkey record has primary key
-- fkey record contains foreign key
-- refn foreign key references record
-- key_ key contains attribute
-- csrc connector source symbol
-- cdst connector destination symbol
--
-- parent: oid the first ("parent") construct
-- child: oid of second ("child") construct
-- description: oid of the description in which the constructs are linked
-- linktype: type of link, see above
-- flags: link-specific flags
--
create table s_construct_link
( parent oid,
  child oid,
  description oid,
  linktype char4,
  flags int4
) inherits (s_repository_item);

-----
-- Class S_KEY_LINK
-- Links records, attributes and keys.
--
-- linkto: oid of the associated record
--
create table s_key_link
( linkto oid
) inherits (s_construct_link);

-----
-- Class S_EVENT_LOG
-- Records operations on repository constructs.
--
-- when: timestamp of when the operation occurred
-- what: what the operation was
-- affects: the oid of the object that is affected by the operation
-- event: the oid of the event that triggered this event (if applicable)
--
create table s_event_log
( when abstime,
  what char4,
  affects oid,
  event oid
);

```

## G.2.2 functions.sql

```

create function GET_VIEWPOINT (text) returns oid
as 'select oid from s_viewpoint where vname = $1'
language 'sql';

create function GET_DESCRIPTION (text, oid) returns oid
as 'select oid from s_description where dname = $1 and viewpoint_id = $2'
language 'sql';

create function GET_SYMBOL (text, oid) returns oid
as 'select oid from s_symbol where cname = $1 and desc_id = $2'
language 'sql';

create function GET_CONNECTOR (text, oid) returns oid
as 'select oid from s_connector where cname = $1 and desc_id = $2'
language 'sql';

create function GET_REPRESENTATION (char4, char4) returns oid
as 'select oid from s_representation where technique = $1 and scheme = $2'
language 'sql';

create function GET_DATATYPE (char4) returns oid
as 'select oid from s_datatype where tcode = $1'
language 'sql';

create function GET_DOMAIN (text, oid) returns oid
as 'select oid from s_domain where cname = $1 and viewpoint_id = $2'
language 'sql';

```

```

create function GET_ATTRIBUTE (text, oid) returns oid
  as 'select oid from s_attribute where cname = $1 and viewpoint_id = $2'
  language 'sql';

create function GET_CONSTRAINT (text, oid) returns oid
  as 'select oid from s_constraint where cname = $1 and viewpoint_id = $2'
  language 'sql';

create function GET_DEFINITION (text, oid) returns oid
  as 'select oid from s_definition where cname = $1 and viewpoint_id = $2'
  language 'sql';

create function GET_RECORD (text, oid) returns oid
  as 'select oid from s_record where cname = $1 and viewpoint_id = $2'
  language 'sql';

```

## G.2.3 staticdata.sql

```

-----
-- REPRESENTATIONS
--
-- Smith FDD
insert into s_representation (technique, techname, scheme, schemename, javaclass)
values ('fdep', 'Functional dependencies', 'smit',
       'Smith functional dependency diagram', 'SmithFDD');

insert into s_construct_glossary
values (GET_REPRESENTATION('fdep','smit'), 'dflg', 'domain flag');

insert into s_construct_glossary
values (GET_REPRESENTATION('fdep','smit'), 'attr', 'attribute');

insert into s_construct_glossary
values (GET_REPRESENTATION('fdep','smit'), 'bubl', 'bubble');

insert into s_construct_glossary
values (GET_REPRESENTATION('fdep','smit'), 'dfds', 'domain flag descriptor');

insert into s_construct_glossary
values (GET_REPRESENTATION('fdep','smit'), 'dfsr', 'domain flag source');

insert into s_construct_glossary
values (GET_REPRESENTATION('fdep','smit'), 'svdp', 'single-valued dependency');

insert into s_construct_glossary
values (GET_REPRESENTATION('fdep','smit'), 'mvdp', 'multi-valued dependency');

-- Martin ERD (crow's feet)
insert into s_representation (technique, techname, scheme, schemename, javaclass)
values ('erm', 'Entity-relationship approach', 'mrtd',
       'Martin entity-relationship diagram (crow's feet)', 'MartinERDc');

insert into s_construct_glossary
values (GET_REPRESENTATION('erm','mrtd'), 'rent', 'regular entity');

insert into s_construct_glossary
values (GET_REPRESENTATION('erm','mrtd'), 'aent', 'associative entity');

insert into s_construct_glossary
values (GET_REPRESENTATION('erm','mrtd'), 'went', 'weak entity');

insert into s_construct_glossary
values (GET_REPRESENTATION('erm','mrtd'), 'mexc', 'mutual exclusivity');

insert into s_construct_glossary
values (GET_REPRESENTATION('erm','mrtd'), '!mxc', 'non-mutual exclusivity');

insert into s_construct_glossary
values (GET_REPRESENTATION('erm','mrtd'), 'rshp', 'relationship');

insert into s_construct_glossary
values (GET_REPRESENTATION('erm','mrtd'), 'stpp', 'subtype');

insert into s_construct_glossary
values (GET_REPRESENTATION('erm','mrtd'), 'stpc', 'subtype');

-- Gane & Sarson DFD
insert into s_representation (technique, techname, scheme, schemename, javaclass)
values ('proc', 'Process modelling', 'gnsn', 'Gane & Sarson data flow diagram', 'GaneSarsonDFD');

insert into s_construct_glossary
values (GET_REPRESENTATION('proc','gnsn'), 'eent', 'external entity');

insert into s_construct_glossary
values (GET_REPRESENTATION('proc','gnsn'), 'een2', 'repeated external entity');

insert into s_construct_glossary
values (GET_REPRESENTATION('proc','gnsn'), 'proc', 'data process');

```



```

insert into s_construct_glossary
values (GET_REPRESENTATION('proc','gnsn'),'mprc','multiple data process');

insert into s_construct_glossary
values (GET_REPRESENTATION('proc','gnsn'),'dstr','data store');

insert into s_construct_glossary
values (GET_REPRESENTATION('proc','gnsn'),'dst2','repeated data store');

insert into s_construct_glossary
values (GET_REPRESENTATION('proc','gnsn'),'rstr','resource store');

insert into s_construct_glossary
values (GET_REPRESENTATION('proc','gnsn'),'spmng','split/merge');

insert into s_construct_glossary
values (GET_REPRESENTATION('proc','gnsn'),'intf','interface');

insert into s_construct_glossary
values (GET_REPRESENTATION('proc','gnsn'),'dflo','data flow');

insert into s_construct_glossary
values (GET_REPRESENTATION('proc','gnsn'),'rflo','resource flow');

-- ANSI SQL/92
insert into s_representation (technique, techname, scheme, schemename, javaclass)
values ('rela', 'Relational model', 'a/92', 'ANSI SQL/92', 'ANSISQL92');

-----
-- TRANSLATIONS
--
insert into s_translation values (GET_REPRESENTATION('fdep', 'smit'),
                                GET_REPRESENTATION('erm_', 'mrta'), 'fdepsmit.erm_mrta');
insert into s_translation values (GET_REPRESENTATION('fdep', 'smit'),
                                GET_REPRESENTATION('erm_', 'mrta'), 'fdepsmit.erm_mrta');

insert into s_translation values (GET_REPRESENTATION('erm_', 'mrta'),
                                GET_REPRESENTATION('proc', 'gnsn'), 'erm_mrta.procgnsn');
insert into s_translation values (GET_REPRESENTATION('erm_', 'mrta'),
                                GET_REPRESENTATION('proc', 'gnsn'), 'erm_mrta.procgnsn');

-----
-- DATA TYPES
--
insert into s_datatype values ('int_', 'integer', 'integer');

insert into s_datatype values ('fixd', 'fixed', 'fixed-point decimal number');

insert into s_datatype values ('date', 'date', 'date');

insert into s_datatype values ('bool', 'boolean', 'boolean');

insert into s_datatype values ('char', 'char', 'fixed-length character string');

insert into s_datatype values ('vchr', 'varchar', 'variable-length character string');

insert into s_datatype values ('flot', 'float', 'single precision floating point number');

insert into s_datatype values ('dbl_', 'double', 'double precision floating point number');

```



# Appendix H

## Swift class hierarchy

In this appendix is presented the complete class hierarchy of Swift in graphical form. The notation used is a slightly modified version of that used in *Java in a Nutshell* (Flanagan, 1997, Chapter 17); a key to the notation may be found in Figure H.1 on the next page.

The class hierarchy is presented package-by-package. The packages are presented in the following order:

**swift.dd:** classes related to repository access — see Figure H.2 on the following page.

**swift.event:** classes related to event handling — see Figure H.3 on the next page.

**swift.model:** classes for manipulating viewpoints and descriptions — see Figure H.4 on the following page.

**swift.repn:** classes for manipulating representations — see Figure H.5 on page 465.

**swift.repn.erm\_mrtc:** classes that implement  $\mathfrak{R}_e(E-R, ERD_{Martin})$  — see Figure H.6 on page 465.

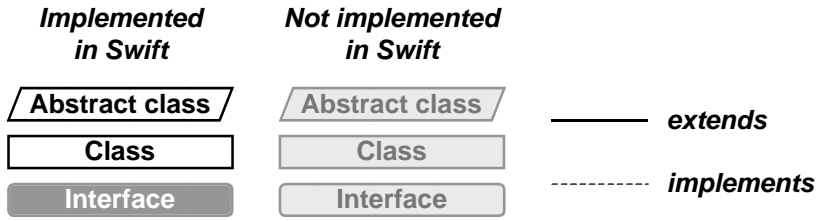
**swift.repn.fdepsmit:** classes that implement  $\mathfrak{R}_f(FuncDep, FDD_{Smith})$  — see Figure H.7 on page 466.

**swift.repn.procgnsn:** classes that representation  $\mathfrak{R}_d(DataFlow, DFD_{G\&S})$  — see Figure H.8 on page 467.

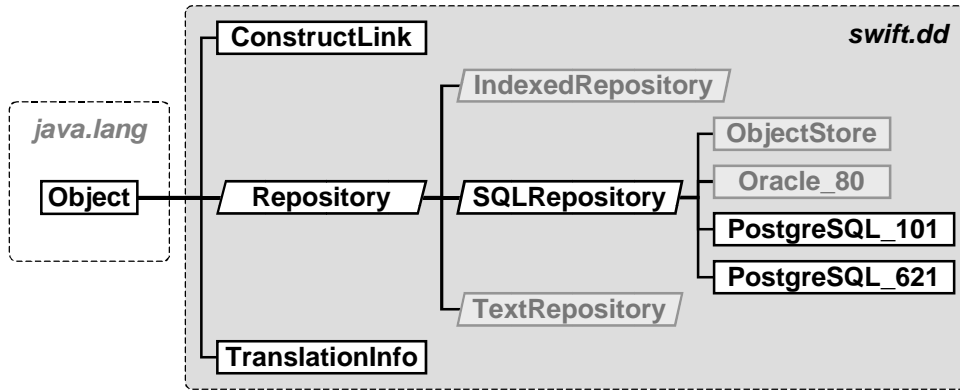
**swift.trans.\*:** classes that implement translations — see Figure H.9 on page 467.

**swift.ui:** classes that implement Swift's user interface — see Figure H.10 on page 468.

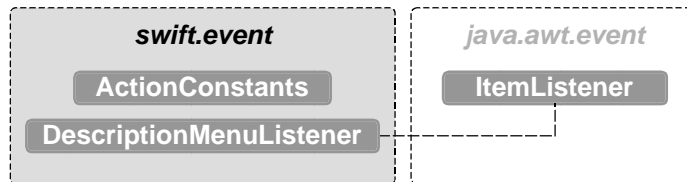
**swift.util:** miscellaneous utility classes — see Figure H.11 on page 468.



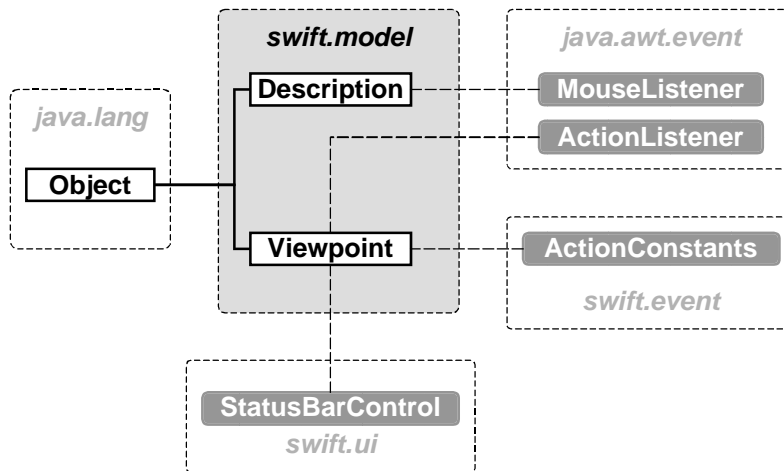
**Figure H.1:** Key for interpreting class diagrams



**Figure H.2:** Swift class hierarchy: swift.dd package



**Figure H.3:** Swift class hierarchy: swift.event package



**Figure H.4:** Swift class hierarchy: swift.model package

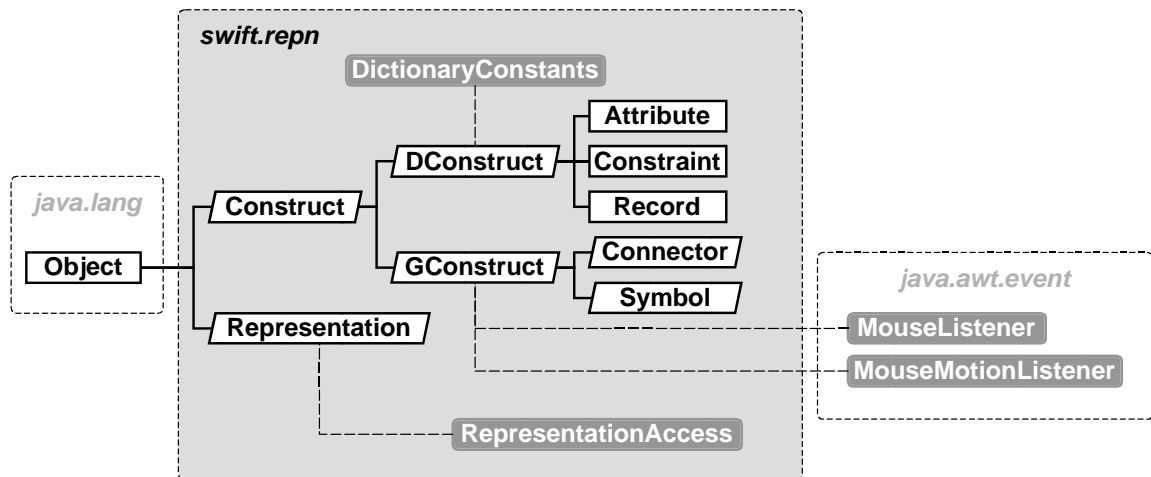


Figure H.5: Swift class hierarchy: swift.repn package

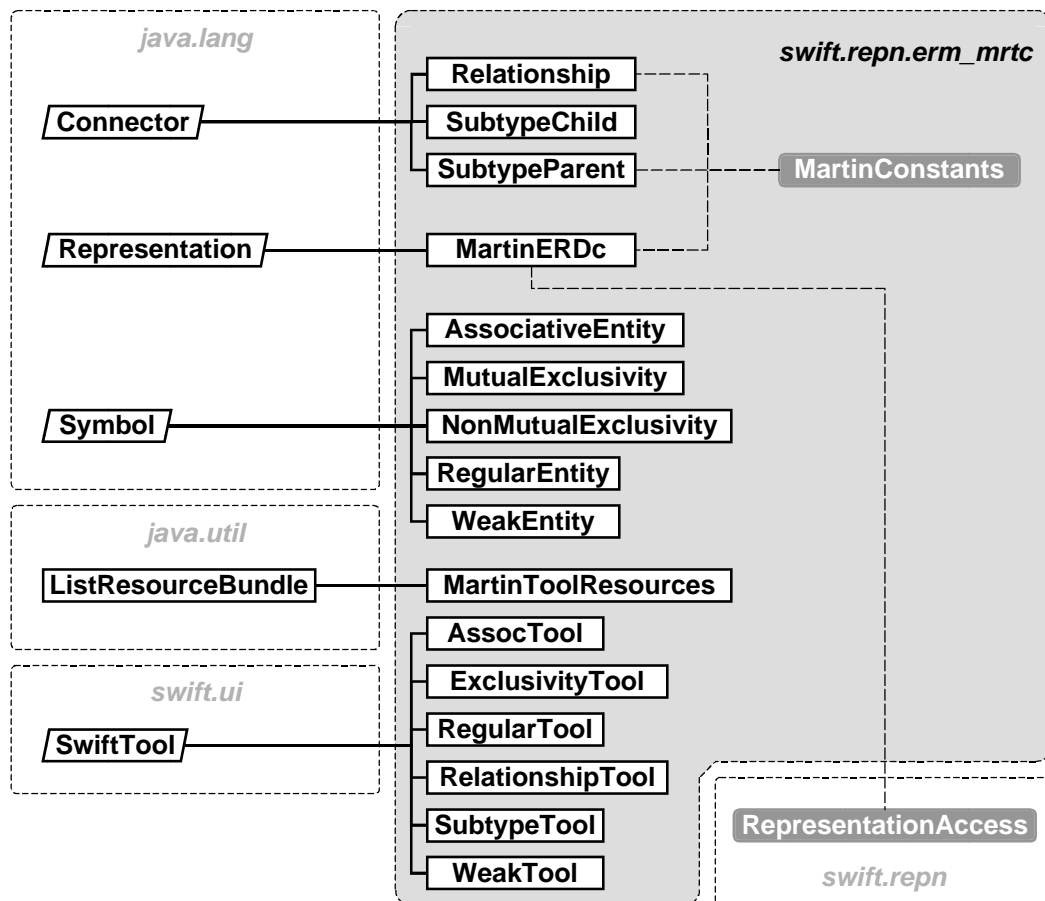


Figure H.6: Swift class hierarchy: swift.repn.erm\_mrtc package

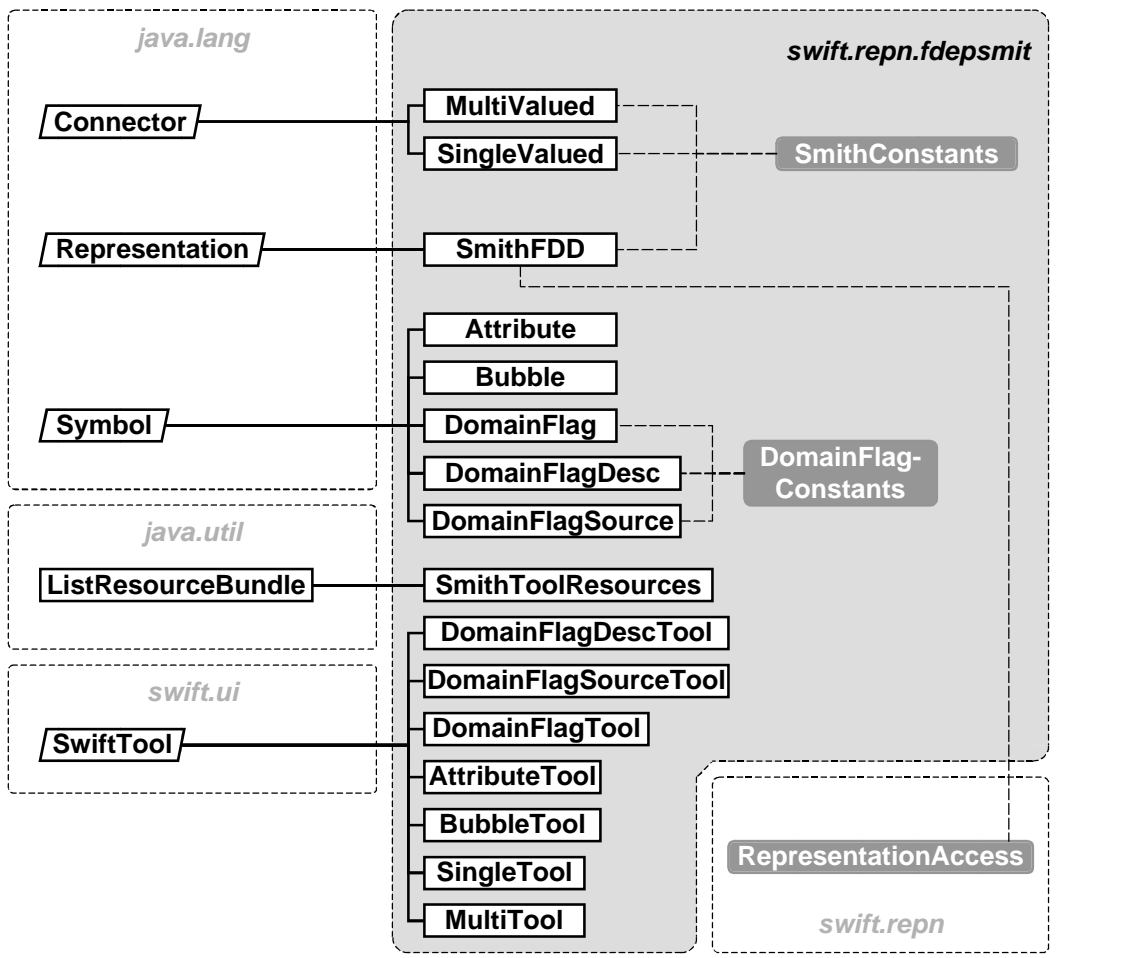


Figure H.7: Swift class hierarchy: `swift.repn.fdepsmit` package

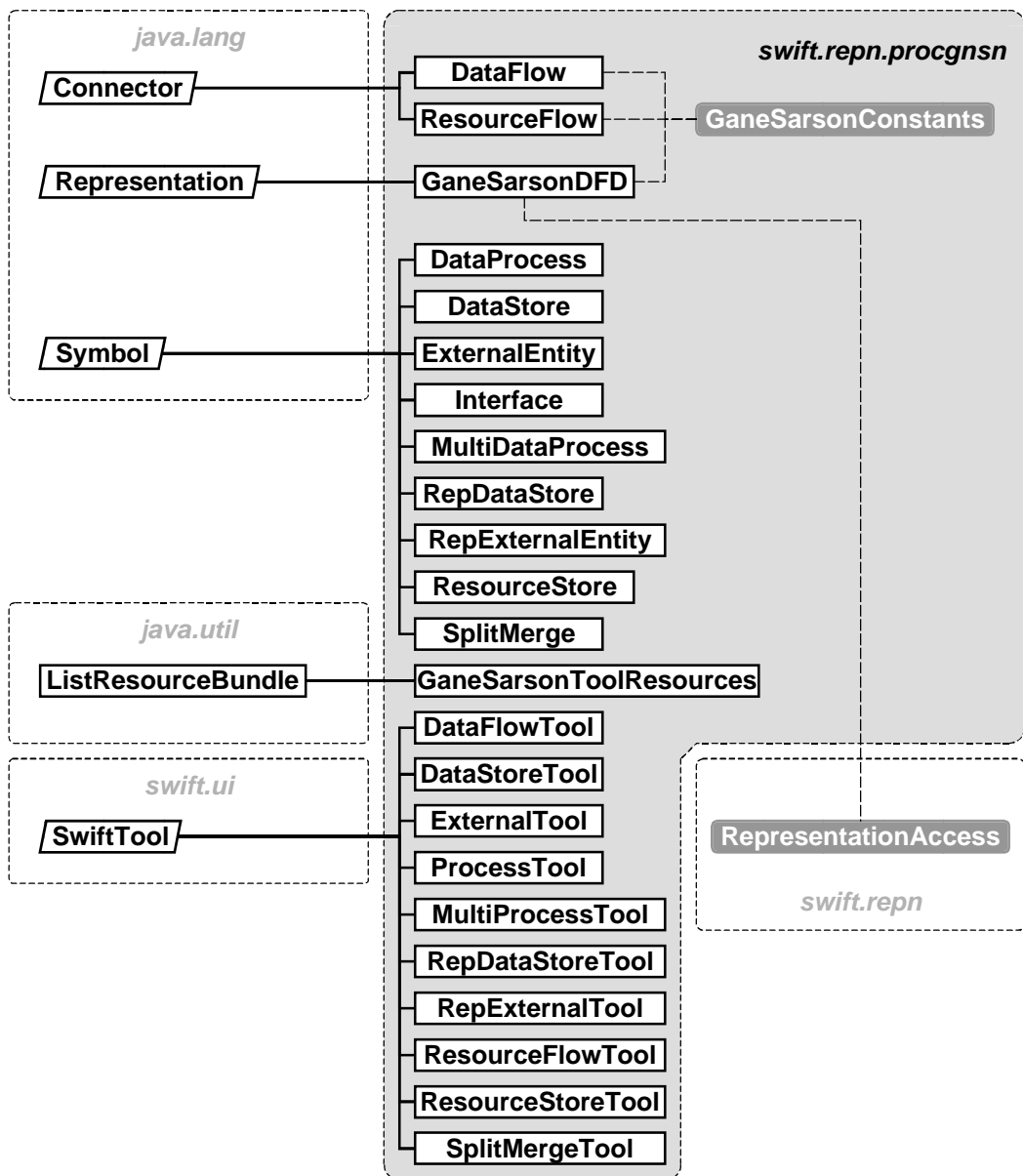


Figure H.8: Swift class hierarchy: `swift.repn.procgnsn` package

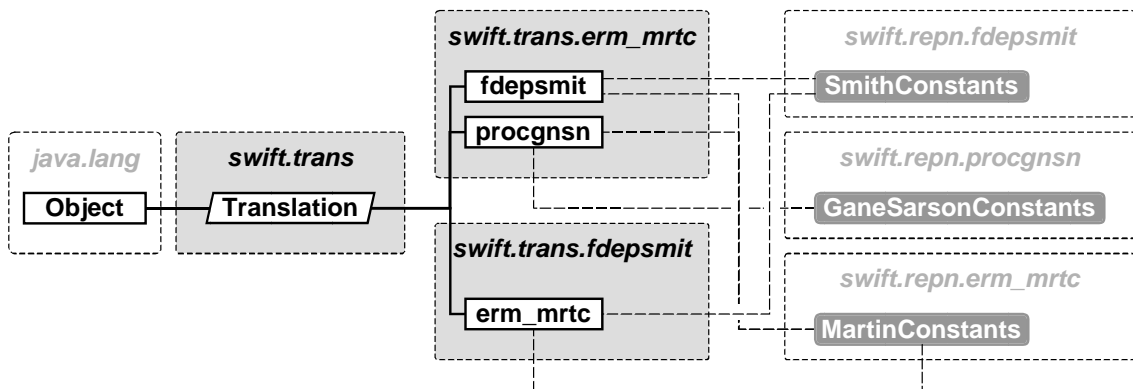
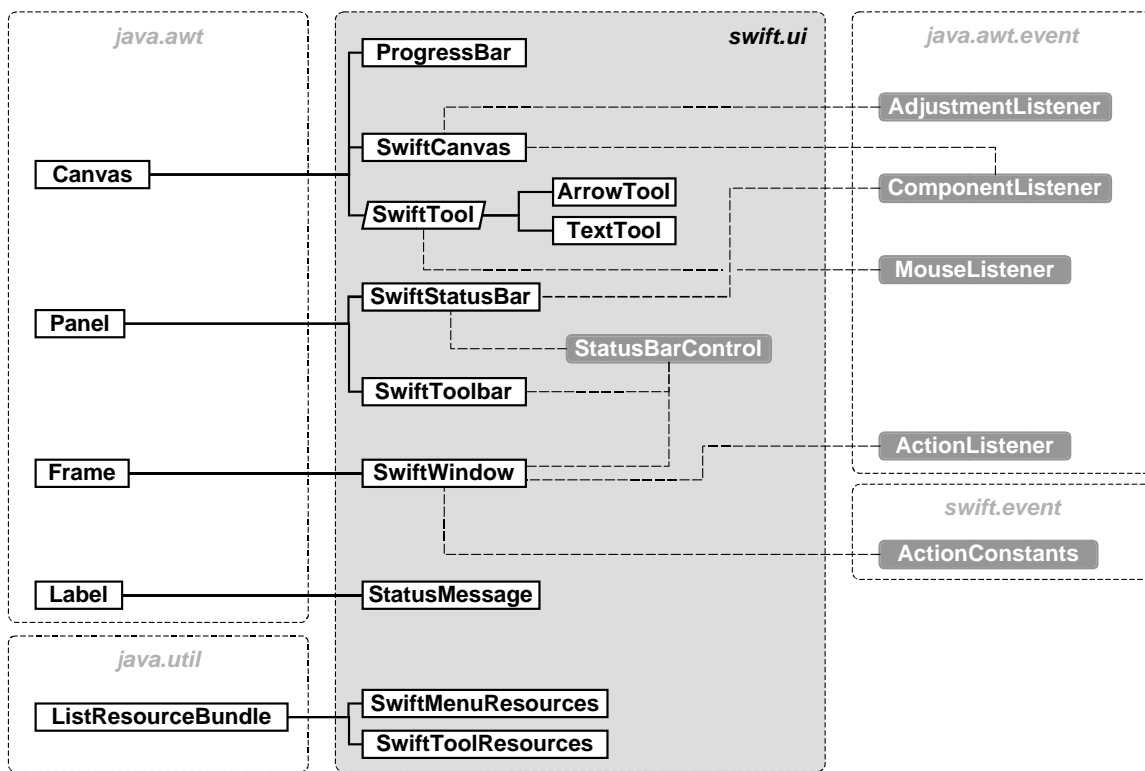
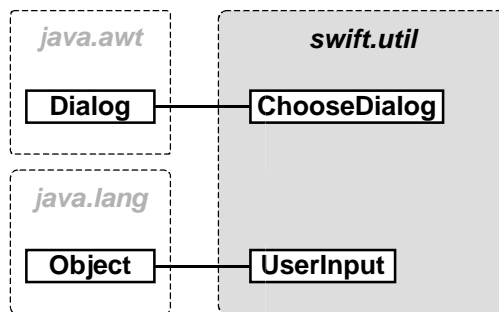


Figure H.9: Swift class hierarchy: `swift.trans` packages



**Figure H.10:** Swift class hierarchy: `swift.ui` package



**Figure H.11:** Swift class hierarchy: `swift.util` package