

# UC Irvine

## ICS Technical Reports

### Title

Object-oriented views: a novel approach for tool integration in design environments  
(dissertation)

### Permalink

<https://escholarship.org/uc/item/4k94r844>

### Author

Rundensteiner, Elke A.

### Publication Date

1992

Peer reviewed

UNIVERSITY OF CALIFORNIA  
IRVINE

Object-Oriented Views: A Novel Approach For Tool  
Integration in Design Environments  
Technical Report #92-83

DISSERTATION

submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in Information and Computer Science

by

Elke Angelika Rundensteiner

Dissertation Committee:

Professor Lubomir Bic, Chair

Professor Daniel D. Gajski

Professor Nikil Dutt



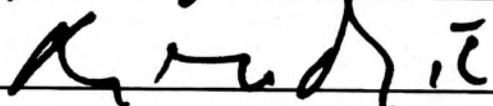
1992

©1992

ELKE ANGELIKA RUNDENSTEINER

ALL RIGHTS RESERVED

The dissertation of Elke Angelika Rundensteiner is approved,  
and is acceptable in quality and form for  
publication on microfilm:

  
\_\_\_\_\_  
  
\_\_\_\_\_  
  
\_\_\_\_\_  
Committee Chair

University of California, Irvine

1992

Dedication

To my parents

Christa and Hans Rundensteiner

and

to my husband

Frank E. Lovering

for their love, encouragement and limitless patience.

# Contents

List of Figures . . . . .	ix
Acknowledgments . . . . .	xv
Curriculum Vitae . . . . .	xvi
Abstract . . . . .	xxiv
<b>Chapter 1 Introduction . . . . .</b>	<b>1</b>
1.1 Engineering Design Process . . . . .	1
1.2 Engineering Design Environments . . . . .	3
1.3 Integrated Engineering Design Environments . . . . .	5
1.4 A Three-Layered Model of Design Object Management . . . . .	7
1.4.1 Design Process Management . . . . .	9
1.4.2 Design Entity Management . . . . .	10
1.4.3 Design Data Management . . . . .	11
1.4.4 Status of Design Object Management . . . . .	13
1.5 Approaches towards Tool Integration . . . . .	13
1.5.1 The File-Based Approach . . . . .	14
1.5.2 The (Traditional) Database Approach . . . . .	16
1.5.3 The Object Server Approach . . . . .	18
1.6 Tool Integration Using the View-Based Object Server Approach . . . . .	20
1.7 Problem Definition . . . . .	23
1.8 Road Map of the Dissertation . . . . .	28
<b>Chapter 2 Related Work on Database Views . . . . .</b>	<b>30</b>
2.1 Relational Views . . . . .	30
2.2 Extending Relational Views Using Abstract Data Types . . . . .	36
2.3 Maintaining Multiple Representations . . . . .	36
2.4 The Derivation of Virtual Classes . . . . .	37

2.5	Multiple Protocol Approaches . . . . .	38
2.6	Schema Virtualization . . . . .	38
2.7	Miscellaneous Topics . . . . .	39
<b>Chapter 3</b>	<b>Object-Oriented Concepts . . . . .</b>	<b>41</b>
3.1	The Object Model . . . . .	41
3.2	Type Hierarchy and Type Relationships . . . . .	42
3.3	Class Hierarchy and Class Relationships . . . . .	44
3.4	View Schema Definition . . . . .	48
3.5	The Validity of the View Generalization Hierarchy . . . . .	51
3.6	The Closure of the View Property Decomposition Hierarchy . . . . .	52
<b>Chapter 4</b>	<b>The <i>MultiView</i> Methodology . . . . .</b>	<b>55</b>
4.1	The Basic Features of <i>MultiView</i> . . . . .	55
4.2	Realization of <i>MultiView</i> . . . . .	59
<b>Chapter 5</b>	<b>Class Customization Using Object Algebra . . . . .</b>	<b>63</b>
5.1	Dual Aspects of a Class: Types and Sets . . . . .	63
5.2	The Object Algebra Operators . . . . .	64
5.2.1	The Hide Operator . . . . .	64
5.2.2	The Refine Operator . . . . .	66
5.2.3	The Select Operator . . . . .	68
5.2.4	The Union Operator . . . . .	69
5.2.5	The Intersection Operator . . . . .	71
5.2.6	The Difference Operator . . . . .	72
5.3	The Macro Object-Algebra Operators . . . . .	76
5.3.1	The Hide* Macro Operator . . . . .	77
5.3.2	The Refine* Macro Operator . . . . .	79
5.3.3	The Select* Macro Operator . . . . .	80
5.3.4	The Union* Macro Operator . . . . .	82
5.3.5	The Intersection* Macro Operator . . . . .	82
5.3.6	The Difference* Macro Operator . . . . .	84
<b>Chapter 6</b>	<b>Set Operators in Object-Oriented Databases . . . . .</b>	<b>87</b>
6.1	Basic Issues . . . . .	87
6.2	Conventional Set Theory . . . . .	89
6.3	Characteristics of Object-Oriented Data Models . . . . .	90

6.3.1	Entities and Classes . . . . .	91
6.3.2	Class Derivation Operations . . . . .	95
6.3.3	Characteristics of Properties . . . . .	96
6.3.4	Class Relationships . . . . .	97
6.4	Set Operations in Object-Oriented Databases . . . . .	100
6.4.1	Motivation . . . . .	100
6.4.2	Difference Operations . . . . .	103
6.4.3	Union Operations . . . . .	106
6.4.4	Intersection Operations . . . . .	109
6.4.5	Symmetric Difference Operations . . . . .	110
6.5	Examples of Class Derivation . . . . .	114
6.6	An Analysis of Set Operations and the Class Relationships of Resulting Derived Classes . . . . .	118
6.7	Related Research on Set Operators . . . . .	122
6.8	Conclusions . . . . .	125
<b>Chapter 7 Automatic Schema Integration . . . . .</b>		<b>126</b>
7.1	Towards One Comprehensive Global Schema . . . . .	126
7.1.1	Motivation . . . . .	126
7.1.2	Towards a Simple Classification Algorithm of Virtual Classes .	129
7.1.3	Manual Placement Versus Automatic Classification . . . . .	131
7.1.4	Road Map for the Rest of the Chapter . . . . .	132
7.2	Two Problems of Schema Integration . . . . .	132
7.2.1	The Type Inheritance Problem . . . . .	133
7.2.2	The <i>is-a</i> Incompatibility Problem . . . . .	135
7.3	A Solution to the Type Inheritance Problem . . . . .	137
7.3.1	The Type Closure and Class Closure Properties . . . . .	137
7.3.2	Using the Closure Property for Class Integration . . . . .	139
7.3.3	Minimizing the Generation of Intermediate Classes . . . . .	143
7.3.4	Interconnecting Intermediate Classes . . . . .	147
7.3.5	Class Integration After Type Preparation . . . . .	150
7.3.6	The Type Hierarchy Preparation Algorithm . . . . .	151
7.4	The Class Placement Algorithm . . . . .	156
7.4.1	The General Class Placement Algorithm . . . . .	156
7.4.2	Computing The Direct Parents Set . . . . .	165
7.4.3	Computing The Direct Children Set . . . . .	173



7.4.4	Summary . . . . .	183
7.5	Putting Everything Together: An Algorithm for Class Integration . . .	183
7.6	A Solution to the Subset/Subtype Incompatibility Problem . . . . .	186
7.7	Customizing the Classification Algorithm for the Object Algebra . . .	191
<b>Chapter 8</b>	<b>The View Definition Language . . . . .</b>	<b>196</b>
8.1	Introduction to the View Definition Language . . . . .	196
8.2	Commands for View Schema Creation and Deletion . . . . .	197
8.3	View Schema Manipulation Commands . . . . .	198
8.4	Examples of View Schema Definition . . . . .	199
<b>Chapter 9</b>	<b>Algorithms for Automatic View Generation . . . . .</b>	<b>202</b>
9.1	Motivation and Problem Definition . . . . .	202
9.2	View Generation for Global Schemata with Single Inheritance . . . .	203
9.3	View Generation for Global Schemata with Multiple Inheritance . . .	209
<b>Chapter 10</b>	<b>Algorithms for Enforcing View Consistency . . . . .</b>	<b>225</b>
10.1	The View Validity Checker . . . . .	225
10.2	The View Closure Checker . . . . .	229
10.2.1	Basic Concepts . . . . .	229
10.2.2	Closed-View Generation: Algorithm and Examples . . . . .	231
10.2.3	Correctness and Complexity of the Algorithm . . . . .	235
<b>Chapter 11</b>	<b>The View Independence of <i>MultiView</i> . . . . .</b>	<b>239</b>
11.1	The View Independence Concept . . . . .	239
11.2	Proving <i>MultiView</i> View Independent . . . . .	241
11.2.1	Preservation of View Classes . . . . .	241
11.2.2	Preservation of View <i>is-a</i> Relationships . . . . .	245
<b>Chapter 12</b>	<b>Database Support for Behavioral Synthesis: A Potpourri.</b>	<b>246</b>
12.1	An Introduction to Database Support for Behavioral Synthesis . . .	246
12.2	Related Research in Object Management for Behavioral Synthesis . .	248
12.3	A Unified Behavioral Design Object Model . . . . .	250
12.3.1	The Behavioral Design Object Model . . . . .	250
12.3.2	The Behavioral Design Data Representation Models . . . . .	254
12.3.3	The Behavioral Design Entity Graph Model . . . . .	261
12.4	A Textual Exchange Format for Design Data . . . . .	264

12.4.1	Introduction . . . . .	264
12.4.2	Other Work Related to Textual Exchange Formats . . . . .	266
12.4.3	BDEF: The Behavioral Design Data Exchange Format . . . . .	267
12.4.4	BDEF Examples . . . . .	273
12.4.5	BDEF Implementation . . . . .	278
12.5	Design Exploration Using Design Transformations . . . . .	279
12.5.1	Design Exploration . . . . .	279
12.5.2	The Collection of Design Transformations . . . . .	280
12.5.3	A Comprehensive Example of Design Transformations . . . . .	291
12.6	Design Tasks in a Behavioral Synthesis Environment . . . . .	294
<b>Chapter 13</b>	<b>Specifying Behavioral Design Views Using <i>MultiView</i></b> . . . . .	<b>304</b>
13.1	An Introduction to Design Views . . . . .	304
13.2	Design Views for the Floorplanning Design Task . . . . .	305
13.2.1	The Floorplanning Design Task . . . . .	305
13.2.2	Grouping of Objects . . . . .	307
13.2.3	Removing of Structure Manipulation Operators . . . . .	309
13.2.4	Customizing the View with Application-Specific Functions . . . . .	315
13.2.5	Increasing the Consistency of the Design Data . . . . .	317
13.2.6	Customizing Using Derived Attributes . . . . .	320
13.3	Design Views for the Binding Design Task . . . . .	324
13.4	Design Views for the Timing Constraints in a Control Flow Graph . . . . .	332
13.4.1	The Graph Compiler View . . . . .	333
13.4.2	Simplying The Timing Information in the Control Flow Graph . . . . .	337
13.4.3	Removing the Timing Information from the Control Flow Graph . . . . .	343
13.4.4	Three Design Views of a Control Flow Graph . . . . .	347
13.5	Views For Simplification of the Data Flow Graph Model . . . . .	347
13.6	Concluding Remarks . . . . .	359
<b>Chapter 14</b>	<b>Conclusions and Future Work</b> . . . . .	<b>362</b>
14.1	Contributions . . . . .	362
14.1.1	Database Contributions . . . . .	362
14.1.2	Computer-Aided Design Related Contributions . . . . .	366
14.2	Future Work . . . . .	368
14.2.1	Future Database Work . . . . .	368
14.2.2	Future Work Related to Computer-Aided Design . . . . .	371

# List of Figures

1.1	A Simple View of a Design Process. . . . .	1
1.2	A CAD Environment. . . . .	4
1.3	An Integrated CAD Framework. . . . .	6
1.4	A Three-Layered Object Model for Design Environments. . . . .	8
1.5	Different Tool Integration Schemes. . . . .	15
1.6	Comparison of the Relational and the Object-Oriented Data Models. . . . .	24
1.7	An Object-Oriented View for the Floorplanning Design Task. . . . .	26
2.1	A CAD Example Using Relational Views. . . . .	31
2.2	An Example of the Update Ambiguity of Relational Views. . . . .	34
3.1	Examples of Base, Global and View Schemata. . . . .	50
3.2	Examples of the View Schema Validity Criteria. . . . .	52
3.3	Examples of Closed and Non-Closed Views. . . . .	54
4.1	An Example of the <i>MultiView</i> Approach. . . . .	60
4.2	Centralized versus Distributed Realization of <i>MultiView</i> . . . . .	62
5.1	Syntax, Semantics and Class Relationships for the Object Algebra Operators. . . . .	65
5.2	An Example of the <b>hide</b> Operator. . . . .	66
5.3	An Example of the <b>refine</b> Operator. . . . .	68
5.4	An Example of the <b>select</b> Operator. . . . .	69
5.5	An Example of the <b>union</b> Operator. . . . .	70
5.6	An Example of the <b>intersect</b> Operator. . . . .	72
5.7	An Example of the <b>diff</b> Operator. . . . .	73
5.8	Examples of Class Derivation and Integration Using Object Algebra. . . . .	74
5.9	Examples of Class Derivation and Integration (cont.). . . . .	75
5.10	The <b>hide*</b> Macro Operator. . . . .	79
5.11	The <b>refine*</b> Macro Operator. . . . .	80
5.12	The <b>selection*</b> Macro Operator. . . . .	81

5.13	The <b>union*</b> Macro Operator. . . . .	83
5.14	The <b>intersection*</b> Macro Operator. . . . .	84
5.15	The <b>difference*</b> Macro Operator. . . . .	85
6.1	Conventional Set Operators and Resulting Subset Relationships. . .	90
6.2	Inheritance of Property Characteristics for a Difference Class. . . . .	105
6.3	Derived Class Created by the Difference Operation. . . . .	106
6.4	Inheritance of Property Characteristics for a Union Class. . . . .	108
6.5	Derived Class Created by the Union Operation. . . . .	108
6.6	Inheritance of Property Characteristics for a Intersection Class. . . . .	110
6.7	Derived Class Created by the Intersection Operation. . . . .	111
6.8	Inheritance of Property Characteristics for a Symmetric Difference Class. . . . .	113
6.9	Derived Class Created by the Symmetric Difference Operation. . .	113
6.10	Examples of Extracting Set Operations. . . . .	115
6.11	Examples of Set Operations Creating <i>Is-A</i> Incompatible Classes. .	116
6.12	Compositions of Set and Type Relationships. . . . .	118
6.13	Six Cases of Type Descriptions. . . . .	119
6.14	Set Operations and Resulting Set, Type and Class Relationships. . .	121
6.15	Set Relationships of Derived Classes. . . . .	123
7.1	Complete Classification Versus Partial Classification. . . . .	127
7.2	The Type Inheritance Problem. . . . .	134
7.3	The <i>Is-a</i> Incompatibility Problem. . . . .	136
7.4	Lowest Common Supertype of Two Types in a Type Hierarchy. . .	138
7.5	The Necessity of Creating Intermediate Classes. . . . .	140
7.6	Partitioning of the Schema Graph $G$ Using the Equivalence Relation	145
7.7	An Equivalence Group $G_i$ Forms a Connected Subgraph of $G$ . . . .	147
7.8	Connecting Intermediate Classes with Classes in the Schema Graph.	149
7.9	The Type-Hierarchy-Preparation Algorithm. . . . .	152
7.10	An Example of Class Hierarchy Preparation. . . . .	154
7.11	The Class-Placement Algorithm. . . . .	158
7.12	An Example of the Class-Placement Algorithm. . . . .	160
7.13	Search Space for Class Placement. . . . .	161
7.14	Removal of Redundant Edges During Class Placement. . . . .	164
7.15	An Algorithm for Computing the $DIRECT-PARENTS_{VC}$ Set. . . . .	168
7.16	An Example of the Find-Direct-Parents Algorithm. . . . .	169

7.17	An Algorithm for Computing the DIRECT-CHILDREN <sub>VC</sub> Set for Single Inheritance. . . . .	173
7.18	An Example of the Find-Direct-Children1 Algorithm. . . . .	174
7.19	An Algorithm for Computing the DIRECT-CHILDREN <sub>VC</sub> Set for Multiple Inheritance. . . . .	176
7.20	An Example of the Find-Direct-Children2 Algorithm. . . . .	178
7.21	An Algorithm for Removing all Indirect Subclasses from the DIRECT-CHILDREN <sub>VC</sub> Set. . . . .	180
7.22	An Example of the Remove-Redundant-Classes Algorithm. . . . .	181
7.23	An Algorithm for Finding All Direct-Children for Multiple Inheritance. . . . .	182
7.24	The Complete Class Integration Algorithm. . . . .	184
7.25	An Example of Complete Classification (Identical Set Contents). . . . .	185
7.26	An Example of Complete Classification (Distinct Set Contents). . . . .	187
7.27	The <i>is-a</i> Incompatibility Problem with VC's Type in the Schema. . . . .	189
7.28	An Example of Solving the <i>is-a</i> Incompatibility Problem. . . . .	190
7.29	Class Integration Customized for each Object Algebra Operator. . . . .	192
7.30	Linear Class Integration Time for the <b>refine</b> Operator. . . . .	193
8.1	The BNF Syntax of the View Definition Language. . . . .	197
8.2	An Example of View Specification. . . . .	200
9.1	The Edge-Creation Algorithm. . . . .	205
9.2	Example Snapshots of the Edge-Creation Algorithm. . . . .	206
9.3	Redundant and Required Edges. . . . .	207
9.4	An Example of Creating Redundant View <i>Is-A</i> Arcs. . . . .	210
9.5	The Edge-Reduction Algorithm for Removal of Redundant Arcs. . . . .	211
9.6	An Example of Removing Redundant Edges Using the Edge-Reduction Algorithm. . . . .	213
9.7	The View-Generation1 Algorithm. . . . .	214
9.8	Example of the View-Generation1 Algorithm for Creating a Valid View Schema. . . . .	215
9.9	The View Schema Creation Algorithm for DAGs. . . . .	217
9.10	The View-Generation2 Algorithm. . . . .	219
9.11	An Example of the View-Generation2 Algorithm for Creating a Valid Schema. . . . .	220
10.1	The View-Consistency Checking Algorithm. . . . .	227

10.2	Preparing the Consistency-Status Matrix for Consistency Checking.	228
10.3	The Closed-View Generation Algorithm.	232
10.4	Examples of the Closed-View Generation Algorithm.	234
11.1	Two Approaches for Type Determination of a View Class.	242
12.1	The BDDDB Design Object Model (BDOM).	251
12.2	Utilization of BDOM.	253
12.3	VHDL Specification of a Programmable Up-and-Down Counter.	257
12.4	State Information for the Programmable Up-and-Down Counter.	257
12.5	ECDFG Representation of the Up-and-Down Counter.	258
12.6	ACG Representation of the Up-and-Down Counter.	260
12.7	Schema for the Behavioral Design Entity Graph Model.	262
12.8	Behavioral Design Entity Graph Example.	265
12.9	Our Approach	267
12.10	Graphical Depiction of Textual Format Syntax Rules.	269
12.11	An Example Using the Textual Format Rules (BDEF).	271
12.12	Graphical Representation of a Data Flow Graph.	274
12.13	A BDEF Description of a Data Flow Graph.	275
12.14	Graphical Depiction of a Timing Constraint in a DFG.	276
12.15	BDEF Description of the Timing Constraint in a DFG.	277
12.16	VHDL Description of Conditional Statements.	284
12.17	The Condition-Decompose Design Transformation.	284
12.18	Conditional VHDL Descriptions.	286
12.19	The Control-To-Data-Flow Design Transformation.	286
12.20	The Condition-Nest Design Transformation.	289
12.21	ECDFG Graphs for the Counter Example.	292
12.22	Operation-Based BIF Tables for Counter ECDFG Graphs.	293
12.23	DFG with Nested Conditions for the Counter Example.	295
12.24	A BIF Table for the DFG with Nested Conditions.	295
12.25	DFG with One Complex Condition for the Counter Example.	296
12.26	A BIF Table for the DFG with One Complex Condition.	296
12.27	A VHDL Design Synthesis Environment	297
12.28	A Classification of Behavioral Synthesis Design Tasks.	299
12.29	Design Flow Information.	300
12.30	Design Tool Interactions with BDDDB.	302

13.1	The Component Graph Portion of the Global Model. . . . .	306
13.2	The Global Schema after Integration of the <b>union</b> class <b>FP1</b> . . . .	308
13.3	Partitioning the Global Schema into Equivalence Groups Using the <b>hide</b> class <b>FP2</b> . . . . .	311
13.4	Integrating <b>FP2</b> into the Global Schema: Generation and Integration of Intermediate Classes and Placement of <b>FP2</b> . . . . .	312
13.5	Selection of View Classes for the Floorplanning1 View. . . . .	313
13.6	Generation of the Floorplanning1 View. . . . .	314
13.7	Correcting Floorplanning1 View for Closure. . . . .	314
13.8	Modifying the Floorplanning1 View by Adding the <b>FP3</b> Class. . .	317
13.9	Pin Positioning Relative to Height and Width of Components. . . .	318
13.10	Floorplanning2 View Derived by Adding Consistency in Pin Movement to Floorplanning1 View. . . . .	321
13.11	Adjusting the Global Model for the Floorplanning2 View. . . . .	322
13.12	The Global Model after the Creation of Two Floorplanning Views. . .	323
13.13	The Binding Design Task. . . . .	325
13.14	The Global Schema Before Constructing the Binding View. . . . .	326
13.15	Inserting the <b>CompB</b> Class into the Global Schema. . . . .	327
13.16	Inserting the <b>Dfop1</b> Class into the Global Schema. . . . .	328
13.17	Inserting the modified <b>Dfop1</b> Class into the Global Schema. . . . .	329
13.18	Inserting the <b>DfopB</b> Class into the Global Schema. . . . .	330
13.19	Selecting View Classes for the Binding View. . . . .	331
13.20	The Binding View Schema. . . . .	332
13.21	A Global Schema with Control Flow and Timing Constraint Classes. . .	334
13.22	VHDL Design Specification of Design1. . . . .	335
13.23	The Design1 Example Using the Timing1 View. . . . .	336
13.24	Refining the Timing Constraint Classes. . . . .	339
13.25	Selecting View Classes for the Timing2 View. . . . .	340
13.26	The Timing2 View Schema. . . . .	341
13.27	The Design1 Example Using the Timing2 View. . . . .	342
13.28	Further Refining the Timing Constraint Classes. . . . .	344
13.29	The Timing3 View Schema. . . . .	345
13.30	The Design1 Example Using the Timing3 View. . . . .	346
13.31	Looking at the Design1 Example Through Three Different Views. . .	348
13.32	An Example of Three Design Views for the Data Flow Graph Model. .	349
13.33	The Global Schema for the DFG Model (equal to the DFG1 View). . .	350

13.34	Integration of the Virtual Classes <b>Dfobject2</b> , <b>Dfnode2</b> , and <b>Dfnet2</b> into the Global Schema. . . . .	352
13.35	Integration of the Virtual Class <b>Dfnode3</b> into the Global Schema. . . . .	353
13.36	Selecting View Classes for the DFG2 View. . . . .	354
13.37	The DFG2 View Schema. . . . .	355
13.38	Integration of the Virtual Class <b>Dfnode4</b> into the Global Schema. . . . .	356
13.39	Integration of the Virtual Class <b>Dfnode5</b> into the Global Schema. . . . .	357
13.40	Selecting View Classes for the DFG3 View. . . . .	358
13.41	The DFG3 View Schema. . . . .	359
13.42	Growth of the Global Schema. . . . .	360
14.1	An Example of the View Construction Optimization Problem. . . . .	370
14.2	Exploiting Design Views at Higher Levels of Design Management. . . . .	373



# Acknowledgments

First and foremost, I would like to thank my advisor Professor Lubomir Bic for consistently standing behind me during all these years. I thank him for all his time, for his guidance, and for helping me to fine tune ideas. I would also like to thank Professor Daniel D. Gajski, who has taken care of me as a co-advisor for many years. He has taught me all that I ever wanted to know and more about Computer Aided Design. I thank him for his advice and for taking me into his CADLAB research group to show me what 'real' problems are all about. Lastly, I want to thank Professor Nikil Dutt for serving as a member of my doctoral committee and for providing me with insightful advice.

I would like to thank Ted Hadley for his hacking assistance with the UNIX operating system and other software. Finally, I would like to thank Joe Lis, Loganath Ramachandran, Roger Ang, Frank Vahid, Jim Kipps, Sanjiv Narayan, Allen Wu, and others in the UCI CADLab for insightful discussions during 'never-ending' Friday group meetings and for their fellowship.

Special thanks goes also to my officemates and friends, Meng-Lai Yin, Jon Gilbert, Hank Kleppinger, Lynn Stauffer, Gregory Bolcer, Ken Anderson, Wang-Chan Wong, and Mark Nagel, for their encouragement and for listening patiently to 'what-ever I had to say'. No, I didn't have a 'big' office, but I had one than one during the course of the years.

The staff at the ICS department has been an invaluable asset. I would particularly like to thank Mary Day, Carmen Mendoza, Susan Moore, Fran Paz, and Candy Mamer for their kind words and assistance. They have made ICS a more pleasant place to work and helped to get things done smoothly.

But most importantly, I would like to thank my husband Frank E. Lovering, my parents Hans and Christa Rundensteiner, and my entire family for their love and support throughout this long experience.

This work was supported in part by a number of different organizations, such as, by grants 91040 and 91041 from TRW and Rockwell International, by funding from contract NSF MIP-8922851 and from NSF grant CCR-8709817, by the Graduate Opportunity Dissertation Fellowship from the University of California, Irvine, by UC Regents' Dissertation Fellowship, by tuition fellowships from the University of California, Irvine, by a fellowship from the Germanistic Society of the Quadrille Scholarship, and by an IBM Graduate Fellowship from the IBM Data Systems Division in Poughkeepsie, NY, I am thankful for their support.

# Curriculum Vitae

Elke Angelika Rundensteiner

## RESEARCH INTERESTS

Semantic and object-oriented database management systems, object repository support technology for hardware/software systems and CAD applications, design representation and design process modeling, temporal and possibilistic databases.

## EDUCATION

### **Ph.D., Computer Science**

University of California, Irvine, California, Summer 1992. (GPA of 4.0).

### **Master of Science, Computer Science**

Florida State University, Tallahassee, Florida, Spring 1987. (GPA of 3.94).

### **Master of Science Minor (Diplomnebenfach), Business Administration**

Johann Wolfgang Goethe University, Frankfurt, West Germany, Summer 1984. (GPA of 4.0).

### **Bachelor of Science (Vordiplom), Computer Science**

Johann Wolfgang Goethe University, Frankfurt, West Germany, Summer 1983. (GPA of 4.0).

## SCHOLARSHIPS AND AWARDS

- Graduate Opportunity Dissertation Fellowship, University of California, Irvine, Winter 1991/92.
- IBM Fellowship for Computer Science Studies, 1990 - 1991.
- UC Regents' Dissertation Fellowship, University of California, Irvine, Winter 1989/90.
- Summer Research Fellowship, University of California, Irvine, Summer 1989.
- Tuition fellowship for Ph.D. studies, University of California, Irvine, 1987 - 1990.

- Travel grant from VLDB for presenting the paper "Aggregates in Possibilistic Databases" at VLDB'89 in Amsterdam, The Netherlands, 1989.
- Quadrille Scholarship for Ph.D. studies in Computer Science, 1988 - 1989.
- Tuition fellowship for graduate studies, Florida State University, Tallahassee, 1984 - 1987.
- Fulbright Scholarship for Computer Science Studies, 1984 - 1985.
- Scholarships of the German people (Studienstiftung des deutschen Volkes), Recommendation of student being among the best 1% of all German high school graduates, 1980.

## TEACHING AND RESEARCH EXPERIENCE

### **Research Associate (1987 - 1992)**

Information and Computer Science Dept., University of California, Irvine.  
Joint research with Prof. L. Bic and Prof. D. Gajski in the area of semantic and object-oriented databases, in particular, view support mechanisms for these advanced data models.

### **Teaching Assistant (1987 - 1988)**

Information and Computer Science Dept., University of California, Irvine.  
Responsible for leading discussion sections and presenting lectures for graduate Computer Science classes.

### **Research Assistant (1986 - 1987)**

Computer Science Dept., Florida State University, Tallahassee, Florida.  
Design and implementation of the student record component of the intelligent tutoring system TAPS (Training Arithmetic Problem-solving Skills), a joint project of the Psychology and Computer Science Department.

### **Teaching Assistant (1984 - 1986)**

Computer Science Dept., Florida State University, Tallahassee, Florida.  
Responsible for giving lectures for introductory Computer Science classes, for leading discussion sections for graduate level Computer Science classes, and for assisting in class design for introductory Computer Science classes.

### **Teaching Assistant (1983 - 1984)**

Computer Science Dept., Johann Wolfgang Goethe University, Frankfurt, West Germany.

Responsible for instruction and testing of undergraduate Computer Science classes while under faculty supervision.

### **Instructor (1982 - 1984)**

Institut fuer Unterrichtsvermittlung, Frankfurt, West Germany.

Responsible for instruction of small groups of students at the grammar school and high school level in the subjects of Mathematics and English.

## WORK EXPERIENCE

### **Research Programmer (1985 - 1986)**

Mechanical Engineering Dept., Florida State University, Tallahassee.

Coding and running of programs in vectorized computation of unsteady separation and combustion processes on the Cyber 205, sponsored by the Super-Computer Computations Research Institute.

### **Programmer (1983 - 1984)**

Reaktor Brennelement Union GmbH, Hanau, West Germany.

Develop and implement additional features for a RBU internal application program that facilitates the daily inventory of products manufactured by the company.

### **Programmer (1982 - 1983)**

Honeywell GmbH, Doernigheim, West Germany.

Debug and maintain a data entry spreadsheet program written in Pascal.

### **Transportation Officer (1980 - 1981)**

US Armed Forces, Hanau, West Germany.

Advise Soldiers on Transportation Regulations.

## PROFESSIONAL AFFILIATIONS

Association of Computing Machinery (ACM), SIGMOD, SIGPLAN, SIGDA.  
IEEE Computer Society, TCDE.

## REFEREE FOR PROFESSIONAL SUBMITTALS

1992 ACM Transactions on Database Systems  
1991 Journal of Intelligent Information Systems  
1990 National Science Foundation  
1990 High-Level Synthesis Workshop  
1989 Sixth International Conference on Data Engineering  
1988 Fifth International Conference on Data Engineering

## PUBLICATIONS

### Refereed Journals

- 1 Rundensteiner, E. A., Bic, L., Gilbert, J., and Yin, M., "Set-Restricted Semantic Groupings", *IEEE Transaction on Data and Knowledge Engineering*, to appear in Spring 1993.
- 2 Rundensteiner, E. A., and Bic, L., "Set Operations in Object-Based Data Models", in *IEEE Transaction on Data and Knowledge Engineering*, to appear in Volume 4, Issue 3, June 1992.
- 3 Rundensteiner, E. A., and Bic, L., "Evaluating Aggregates in Possibilistic Relational Databases", *Data and Knowledge Engineering Journal*, vol. 7, 1992, pp. 239-267.
- 4 Hawkes, L. W., Derry, S. J., and Rundensteiner, E. A., "Individualized Tutoring Using an Intelligent Fuzzy Temporal Relational Database", *International Journal of Man-Machine Studies*, vol. 33, 1990, pp. 409 - 429.
- 5 Rundensteiner, E. A., Hawkes, L. W., and Bandler, W., "On Nearness Measures in Fuzzy Relational Data Models", in *International Journal of Approximate Reasoning*, vol. 3, no. 3, July 1989, pp. 267 - 298.

### Book Chapters

- 1 Rundensteiner, E. A., "The Role of AI in Databases versus the Role of Database Theory in AI", in *Artificial Intelligence in Databases and Information Systems (DS-3)*, IFIP, (eds. Meersman, R.A., Zhongzhi, S., and Kung, C.) Elsevier Science Publishers B. V., North Holland, March 1990, pp. 233-252. (also accepted as full paper in *International Federation for Information Processing, IFIP TC2/TC8/WG 2.6/WG 8.1*, Working Conference on The Role of Artificial Intelligence in Databases and Information Systems, Guangzhou, China, July 1988).
- 2 Hawkes, L. W., Derry, S. J., Kandel, A. and TAPS Project Staff, "Fuzzy Expert Systems for an Intelligent Computer-Based Tutor", in *Fuzzy Expert Systems*, A. Kandel (Ed.), Reading, CRC Press, 1992. (also Tech. Rep. CET/86-5, in Florida State University, Sep. 1986, IST-8510894).

## Submitted Publications to Refereed Journals

- 1 Rundensteiner, E. A., and Bic, L., "Automatic View Schema Generation in Object-Oriented Databases" submitted for publication to the *IEEE Transaction on Data and Knowledge Engineering*, February 1992.
- 2 Rundensteiner, E. A., Gajski, D. D., and Bic, L., "Component Synthesis from Functional Descriptions", submitted for publication to the *IEEE Transactions on Computer-Aided Design*, August 1990. (status: accepted with minor revisions.)

## Refereed Conferences

- 1 Rundensteiner, E. A., "MultiView: A Methodology for Supporting Multiple View Schemata in Object-Oriented Databases," *18th Int. Conference on Very Large Data Bases (VLDB'92)*, Vancouver, Canada, Aug. 1992.
- 2 Rundensteiner, E. A., and Gajski, D. D., "Functional Synthesis Using Area and Delay Optimization", *29th ACM/IEEE Design Automation Conf. (DAC'92)*, Anaheim, California, June 1992.
- 3 Rundensteiner, E. A., Bic, L., Gilbert, J., and Yin, M., "A Semantic Integrity Framework: Set Restrictions for Semantic Groupings," *IEEE Int. Conf. on Data Engineering, (ICDE-7)*, Kobe, Japan, April 1991.
- 4 Rundensteiner, E. A., Bic, L., Gilbert, J., and Yin, M., "Restricting Is-A Related Groupings Using Object Equivalence", *The Second Int. Symposium on Database Systems for Advanced Applications (DASFAA '91)*, Tokyo, Japan, April 1991.
- 5 Rundensteiner, E. A., and Gajski, D. D., "A Design Database for Behavioral Synthesis", *Fifth International Workshop for High-Level Synthesis (HLSW'91)*, Buehlerhoehe, West Germany, March 1991.
- 6 Rundensteiner, E. A., Gajski, D. D., and Bic, L., Component Synthesis Algorithm: "Technology Mapping for Register Transfer Descriptions", *IEEE Int. Conf. on Computer-Aided Design*, Santa Clara, California, Nov. 1990, pp. 208 - 211.
- 7 Rundensteiner, E. A., and Bic, L., "Set Operations in a Data Model Supporting Complex Objects", *Int. Conf. on Extending Data Base Technology (EDBT)*, March 26-30, 1990, Fondazione Cini, Venice, Italy. (also published in *Lecture Notes in Computer Science*, vol. 416, 1990, pp. 286 - 300.)

- 8 Rundensteiner, E. A., and Bic, L., "Semantic data models and their potential for capturing imprecision", *Conference on Management of Data, COMAD'89*, Hyderabad, India, Nov. 1989.
- 9 Rundensteiner, E. A., and Bic, L., "Class Creation by Set Operations", *ACM SIGART Fourth Int. Symposium on Methodologies for Intelligent Systems, ISMIS'89 Poster Session*, Charlotte, NC, Oct. 1989.
- 10 Rundensteiner, E. A., and Bic, L., "Aggregates in Possibilistic Databases", *15th Int. Conference on Very Large Data Bases (VLDB'89)*, Amsterdam, The Netherlands, Aug. 1989, pp. 287 - 295.
- 11 Rundensteiner, E. A., and Bic, L., "Towards modeling imprecision in semantic data models" *Proc. of the Third International Fuzzy Systems Association World Congress (IFSA '89)*, Seattle, Washington, Aug. 1989.
- 12 Rundensteiner, E. A., and Bic, L., "Functional Enhancement of a Knowledge Base", *The Third Western Expert Systems Conference*, Anaheim, CA, June 1988.
- 13 Rundensteiner, E. A., Hawkes, L. W., and Bandler, W., "Set-Valued Temporal Knowledge Representation for Fuzzy Temporal Retrieval in ICAI", (extended abstract), *International Journal of Approximate Reasoning*, vol. 2, issue 2, April 1988.
- 14 Rundensteiner, E. A., and Bic, L., "Extending Fuzzy Relational Query Languages by Scalar Aggregates", *Proc. of NAFIPS '88*, San Francisco, CA, June 1988.
- 15 Rundensteiner, E. A., Bandler, W., Kohout, L., and Hawkes, L. W., "An investigation of fuzzy nearness measures", *Proc. of the Second International Fuzzy Systems Association Congress (IFSA '87)*, Tokyo, Japan, July 1987.
- 16 Rundensteiner, E. A., Hawkes, L. W., and Bandler, W., "Set-valued Temporal Knowledge Representation for Fuzzy Temporal Retrieval in ICAI", *Proceedings of NAFIPS '87*, West Lafayette, Indiana, May 1987, pp. 37 - 65.
- 17 Rundensteiner, E. A., and Hawkes, L. W., "Design of a Student Record", *Proceedings of the International Symposium on Methodologies for Intelligent Systems, ISMIS-86 Colloquia Program*, Knoxville, Tennessee, Oct. 1986.
- 18 Rundensteiner, E. A., and Bandler, W., "Equivalence of the Knowledge Representation Schemata 'semantic networks' and 'fuzzy relational products' ", *Proceedings of NAFIPS '86, Recent Developments in the Theory and Applications of Fuzzy Sets*, New Orleans, LA, June 1986.

- 19 Van Dommelen, L. L., and Rundensteiner, E. A., "Vortex Summation Using Bilingual Programming", *38<sup>th</sup> meeting of the American Physical Society*, Fluids dynamics section, Nov. 85, Tucson, Arizona.
- 20 Rundensteiner, E. A., "Semantic Networks compared to Fuzzy Relational Products", *Proc. on Fuzzy Expert Systems and Decision Support, NAFIPS'85*, Atlanta, Georgia, Oct. 1985.

### Non-Refereed Publications

- 1 Rundensteiner, E. A., "*MultiView*: A Methodology for Supporting Multiple View Schemata in Object-Oriented Databases" Information and Computer Science Department, Univ. of California, Irvine, Tech. Rep. 92-07, January 1992.
- 2 Rundensteiner, E. A., and Bic, L., "Automatic View Schema Generation in Object-Oriented Databases" Information and Computer Science Department, Univ. of California, Irvine, Tech. Rep. 92-15, January 1992.
- 3 Rundensteiner, E. A., and Gajski, D. D., "BDEF: The Behavioral Design Data Exchange Format" Information and Computer Science Department, Univ. of California, Irvine, Tech. Rep. 91-34, April 1991.
- 4 Rundensteiner, E. A., and Gajski, D. D., "A Design Representation Model for High-Level Synthesis", Information and Computer Science Department, Univ. of California, Irvine, Tech. Rep. 90-27, September 1990.
- 5 Rundensteiner, E. A., Gajski, D. D., and Bic, L., "Component Synthesis from Functional Descriptions", Information and Computer Science Department, Univ. of California, Irvine, Tech. Rep. 90-24, August 1990.
- 6 Rundensteiner, E. A., and Bic, L., "Set Operations in Semantic Data Models", Information and Computer Science Department, Univ. of California, Irvine, Tech. Rep. 89-22, June 1989.
- 7 Rundensteiner, E. A., and Bic, L., "Evaluating Aggregate Functions on Possibilistic Data", Information and Computer Science Department, Univ. of California, Irvine, Tech. Rep. 89-12, May 1989.
- 8 Rundensteiner, E. A., Bic, L., Gilbert, J., and Yin, M., "Set-Related Restrictions on Semantic Groupings," Information and Computer Science Department, Univ. of California, Irvine, Tech. Rep. 89-07, Jan. 1989.



- 9 Rundensteiner, E. A., The Development of a Fuzzy Temporal Relational Database (FTRDB): An Artificial Intelligence Application, Master's Thesis, Computer Science Department, Florida State University, Tallahassee, Florida, April 1987.
- 10 Rundensteiner, E. A., and Hawkes, L. W., "Different approaches towards fuzzy database systems. A Survey.", Festschrift to honor of Dr. W. Bandler, Aug. 1986. (also to be published as chapter in a book by Kluwer).

# Abstract of the Dissertation

## Object-Oriented Views: A Novel Approach for Tool Integration in Design Environments

by

Elke Angelika Rundensteiner

Doctor of Philosophy in Information and Computer Science

University of California, Irvine, 1992

Professor Lubomir Bic, Chair

Object-oriented databases have been proposed to serve as the data management component of integrated design environments. One central database represents a bottleneck, however, requiring all design tools to work on the same information model and preventing the extensibility of the system over time. In this dissertation, I propose a view-based object server that successfully addresses these problems by supporting *design views* tailored to the needs of individual design tools.

A view on an object-oriented schema corresponds to a virtual subschema graph with restructured generalization and property decomposition hierarchies. I present a methodology for supporting multiple view schemata, called *MultiView*. *MultiView* is anchored on the following four ideas: (1) the customization of individual classes using object algebra, (2) the integration of these derived classes into one global schema graph, (3) the extraction of virtual and base classes from the global schema as required by the view, and (4) the generation of a class hierarchy for these selected view classes. *MultiView's* division of view specification into these well-defined tasks, some of which have been successfully automated, makes it a powerful tool for supporting the specification of views by non-database experts while enforcing view consistency.

In this dissertation, I describe solutions for all four tasks underlying *MultiView*. For the first task, I have formulated class derivation operators modeled after the well-known relational algebra operators. For the second task, I have developed a classification algorithm that automatically integrates derived classes into one global schema. For the third task, I have designed a view definition language that can be used to declaratively specify the view classes required for a particular view. For the last task, I have developed an algorithm that generates a *complete, minimal and consistent* view schema. I present proofs of correctness, complexity analysis, and numerous illustrative examples for all algorithms.

*MultiView* is applied to address the tool integration problem in a behavioral synthesis system. For this purpose, I first develop a unified design object model for behavioral synthesis. I then formulate customized *design views* of this model tailored to the needs of particular design tools. The resulting system allows the design tools to work on their view of the information model, while *MultiView* assures the consistent integration of the diverse design data into one object model.

# Chapter 1

## Introduction

### 1.1 Engineering Design Process

An engineering design process, such as architectural design, software development, computer-aided design, and behavioral synthesis, can be described in the following manner. Given a design specification, a set of constraints, and possibly a goal, the design process synthesizes an artifact that implements the specification, meets the specified constraints and optimizes one or more goals using a particular technology.

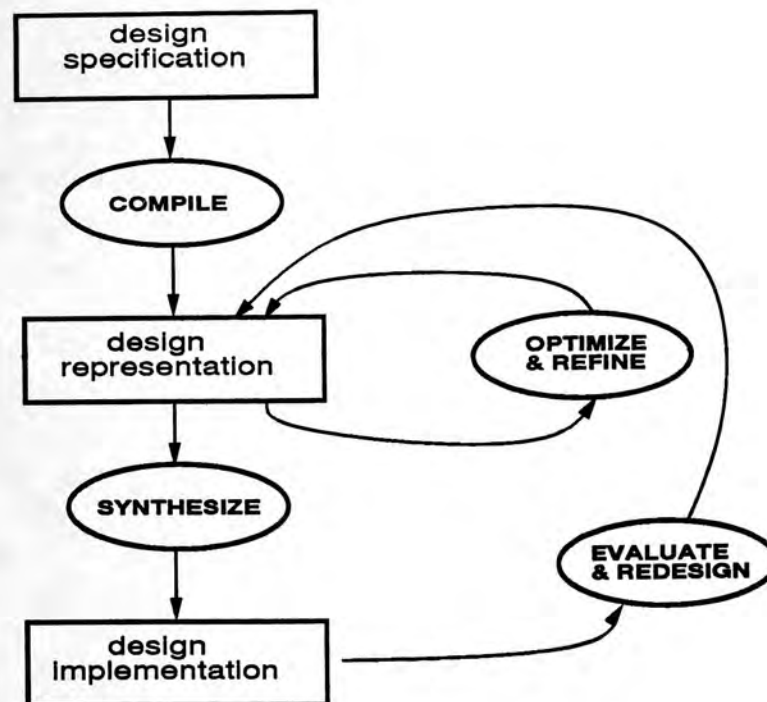


Figure 1.1: A Simple View of a Design Process.

Figure 1.1 depicts the simple model of an engineering design process. Different stages of this design process involve different types of design information. First, a *design specification* is developed which specifies the desired functionality of the design in terms of the inputs and outputs. This specification may include some constraints that should be met by the final design as well as desired goals, such as cost or performance. A *design specification* is generally specified in the English language. In some application domains special-purpose programming languages have been developed for design specification. In behavioral synthesis, for instance, VHDL [VHDL88], which recently has become the IEEE standard, is used as a hardware description language of designs.

The textual design specification is transformed into an *internal design representation* that captures the behavior of the design. At a minimum, it must include constructs for describing the flow of control as well as the data manipulations for transforming the inputs of the design to the outputs. This *internal design representation* is more amendable to synthesis than the textual specification for the following reasons. It explicitly shows data and control dependencies in the specification, it is cleaner, it is based on uniform constructs rather than the ambiguous constructs found in many languages, it is computer-accessible rather than being textual, and generally can be manipulated to reveal design styles. Furthermore, such a design representation is language-independent so that design specifications written in different description languages can be mapped into this general representation and then synthesized using the same design process.

The design representation then is optimized. First, idiosyncrasies that are due to the use of a specification language are removed. Many constructs of a language are suitable for human understanding but they often overconstrain the design. For example, the sequential nature of most programming languages may force unnecessary sequentiality in the design representation. Also, a specification language does not provide explicit constructs to describe all particulars of the application domain, and then the designer has to resort to awkward means of expressing certain constructs. For instance, VHDL does not explicitly support the description of pipelined units or detailed timing constraints. Such ambiguous information should be clarified and extraneous information should be removed from the intermediate design representation.

Thereafter, the design representation is restructured and refined. For instance, designers may want to tune the design to a particular design style, e.g., pipelined versus non-pipelined style, or to a particular technology. A reorganization of the design representation may also help in meeting requested constraints or in optimizing

for a desired goal. In short, the design representation is continuously refined by adding more detail and by making implementation choices.

Finally, the *design implementation* is synthesized from the refined behavioral design representation. The *design implementation* consists of an interconnected set of components from a particular *technology* library. In the case of architectural design, the design of a house may result in the composition of a house using windows, doors, and bricks. In behavioral synthesis, these components are arithmetic/logic units, registers and buses. The technology from which the components are chosen has a major impact on the quality of the resulting design.

It is often not clearly understood what effect a design decision at a higher level of design, e.g., the behavioral design representation, will have on the quality of the final design implementation. Consequently, the design process is exploratory and iterative. The design is evaluated and possibly some decisions are made on how to redesign by taking alternative steps at the design representation level. The final result of the design process often is a set of potential designs with different design characteristics. That is, these designs lie on a trade-off curve ranging from fast but expensive to slow but cheap design implementations. One of these designs is selected as final choice based on the relative priorities of design goals.

## 1.2 Engineering Design Environments

In most disciplines, the artifacts being designed are getting more and more complex and the technology available for implementing the design is rapidly growing. In addition, there is a vast variety of different design methodologies and algorithms for accomplishing this design process. Therefore, the use of computers has been introduced into the design process. This then is called *Computer-Aided Design*, or short, CAD. In recent years, a great number of design tools of ever increasing sophistication have become available that automate the more difficult and time consuming parts of a design process.

At first, emphasis in CAD research has been placed on the technology needed to design the desired artifact, such as the particular algorithm, the design methodology, or the underlying technology library. In this vein, tool builders develop design tools as stand-alone systems each with their internal data structures to quickly test their ideas [Katz85]. They ignore design management issues, such as the representation, maintenance, storage, and retrieval of design information, as well as tool communication and integration issues.

This then results in a *design environment* as shown in Figure 1.2. The design environment is a conglomeration of unstructured software for creating and analyzing designs and for managing the design flow. In such an environment, the design tool builder is charged with many tasks besides the actual implementation of the algorithm. He or she needs to maintain a design representation of the tool's design information, data structures to capture that data, and possibly some rudimentary data management functions to make the designs persistent over time. In other words, tool developers are reinventing over and over the data structures (and with them the semantics) of the underlying design data.

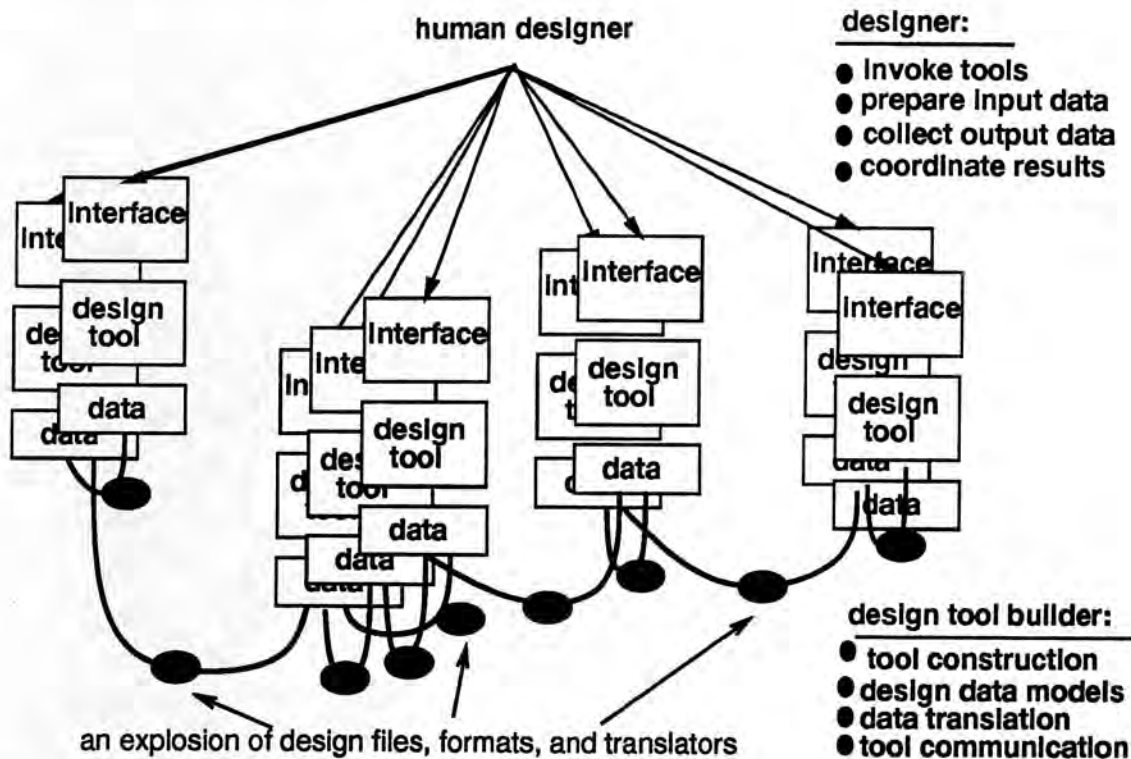


Figure 1.2: A CAD Environment.

In order to exploit the full potential of the available software, these design tools should be combined into different sequences depending on the design goal and on the different types of design data. Therefore, often as an afterthought, tool communication as well as design data exchange issues between tools is being addressed. This results in many problems, since CAD tools – each performing different design tasks – are likely to have different internal data structures and to utilize the design information in different ways. Generally, translation mechanisms as well as file formats are established in an ad-hoc manner between each pair of design tools that need to

exchange information. Hence, there could be up to  $n \times n$  such tool communication efforts for  $n$  tools (See Figure 1.2).

Such a design environment does not only place an unnecessary burden on design tool developers, but also on design tool users (designers). The latter have to know about the particulars of each design tool and about the translation and formatter routines to exchange data between two design tools. Designers furthermore have to keep track of and coordinate the intermediate and final design results. This clearly is a very complex task since design data is distributed in different representation formats among the various design tools.

### 1.3 Integrated Engineering Design Environments

While developing more and more sophisticated CAD tools for automating various design and manufacturing processes, little effort has been devoted towards the exploration of a supporting infrastructure that would allow for the integration of individual tools into one cooperative and powerful CAD system. For these reasons, this dissertation addresses the problem of how to incorporate these tools into such an *integrated design environment* [CFI92]. Such an *integrated software environment* (also called *CAD framework*) simplifies the tasks of both design tool users and design tool builders. In such a CAD framework, many of the functions done by design tool users and builders are now the responsibility of the system.

A CAD infrastructure has three major architectural components as necessary ingredients, namely, the *user interface*, the *flow manager*, and the *database system*. This elementary architecture of a CAD framework is given in Figure 1.3, while the description of more complex systems can be found in the literature [Ramm90].

The *user interface system* provides the designer with a uniform access to the CAD system. The user interface system should primarily support the following set of tasks: (1) tool invocation, (2) data preparation, (3) data browsing, (4) tool browsing, and (5) design methodology support. Important here is that the user interface presents the complex information of the design data, the design tools, and the overall design process in a comprehensive and uniform manner (usually, a graphical interface) to the designer.

The *design flow manager* is the controller of the CAD framework. It provides the designer with information on the status of the design and the availability of tools; while leaving the ultimate decision-making process to the human designer. The flow



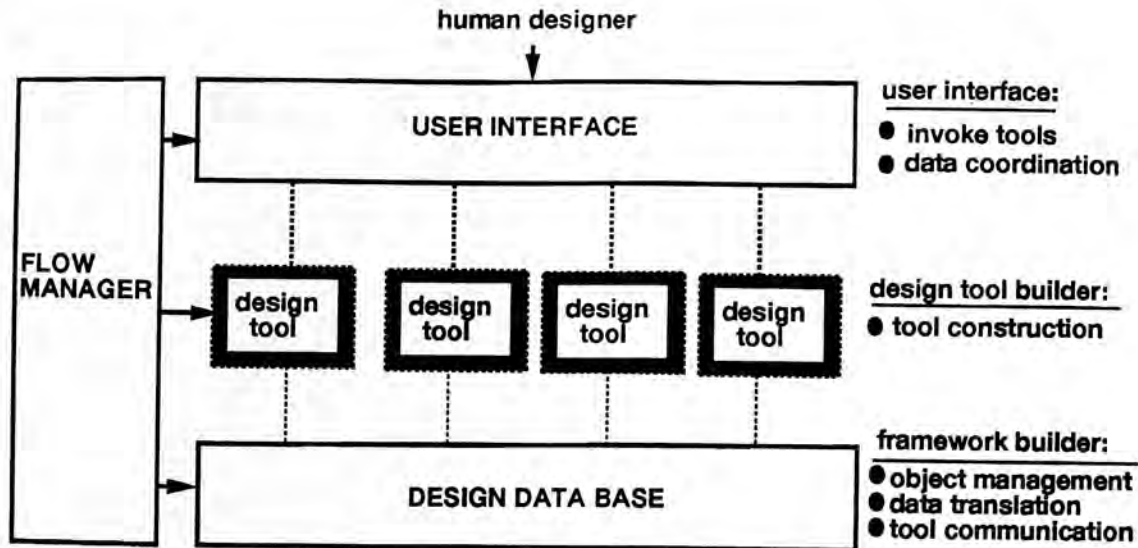


Figure 1.3: An Integrated CAD Framework.

manager, having knowledge about the design methodologies and design processes, thus helps to guide the design process. Ultimately, the design flow manager executes the requests of the designer by either invoking design tools or by making calls to the data base to retrieve or to store design data.

The *design database* provides a *repository for persistent design objects*. Such a repository would guarantee persistence of design data beyond the execution of the design tool that created the data. Support for persistence includes methods for storage, update and retrieval of persistent data. This common repository would collect all design information into one central location rather than scattering it over the various design tools, and hence encourages sharing of design data. Other usages of the *design database* are the maintenance of complex semantic relationships between partial and complete designs, such as derivation and decomposition relationships, as well as the capture of the actual design processes. See Section 1.4 for an extensive discussion of this topic.

In short, one important part of these much needed infrastructure technologies is the database for handling the complex design information and flexible tool integration schemes. It is a difficult, largely unexplored, problem that has to be solved before Computer-Aided Design (CAD) companies can achieve high-quality automated design and manufacturing. This dissertation thus concentrates on the design management task of a CAD framework. More specifically, I present a solution for the important problem of design information management and of tool integration

by providing a view-based object-oriented mechanism that allows for the sharing of complex design data among design tools.

## 1.4 A Three-Layered Model of Design Object Management

The database component of a CAD framework must store and manage all information in the design environment that is to be maintained persistently. While many different data models have been proposed in the literature [Bato85, Bret90, Dutt90, Eijn91, Katz90, Neum83], most cover only a limited subset of the design objects available in a design environment. An analysis of this design information reveals that there exist fundamentally different types of design objects; each having their own characteristics and associated tasks. In this section, I propose a comprehensive object model for design information management that classifies this design information into three layers of abstraction. This model serves numerous purposes. First, it represents a simple yet sufficient vehicle for characterizing the different types of objects that need to be supported by the database component of a CAD framework. In addition, I outline distinct functionalities that need be supported at each of these levels. On another note, this model allows us to explain how our work fits into the overall framework of design object management. More importantly, I use the model to demonstrate how our approach presented in this dissertation is distinct from, yet compatible with, existing approaches in the literature towards design object management.

Information in design environments can be characterized by a design object model that consists of three layers of abstraction:

- design process layer
- design entity layer
- design data layer

Figure 1.4 gives an example of the types of objects and their relationships that belong to each layer for the behavioral synthesis domain. In the remainder of this section, I characterize the information captured by each of the three levels and describe the respective database functionalities that should be supported at each of these levels.

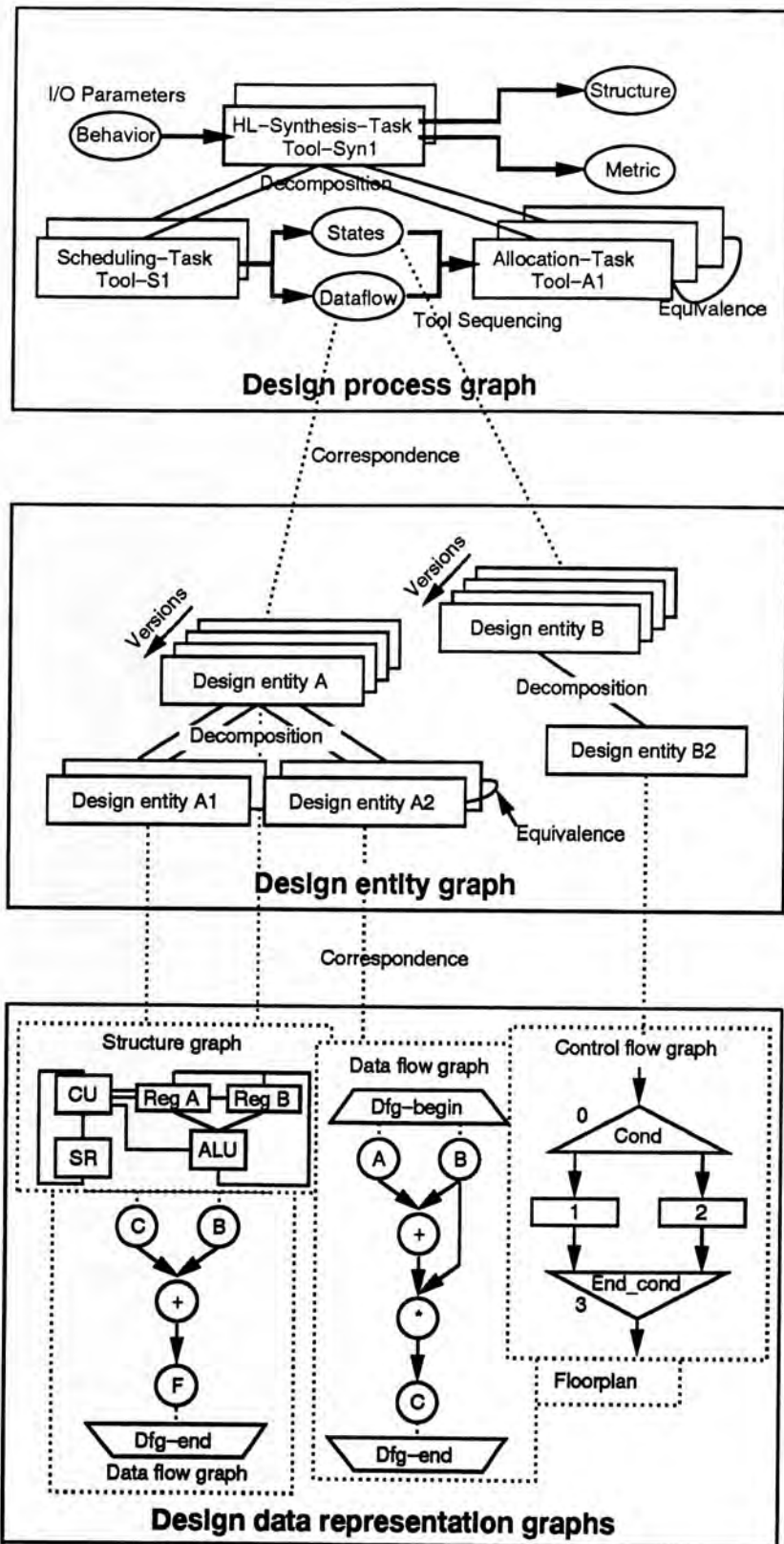


Figure 1.4: A Three-Layered Object Model for Design Environments.

### 1.4.1 Design Process Management

The top-most layer of the design object model is concerned with the management of *design processes*. The model includes the capture of static design information, such as the *design tools* available in the system. Since CAD tools perform particular *design tasks*, such as synthesis, verification, scheduling, and allocation. The model must also capture the semantics of these *design functions*, their required and/or optional data inputs and outputs, control parameters, and the (potentially dynamic) mapping to actual design tools. Most importantly, the model captures the actual design process, i.e., the concatenation of design tasks for correctly performing the desired design function. Concatenation means both data flow, i.e., the consistency of using the output of one design task as input for the next design task, as well as control flow, i.e., the consistency of design decisions made during the course of the design process and the conditional dependence between tasks.

Most work in the literature approaches the design process management problem based on the following three ideas: first, the encapsulation of design tools by a stable and uniform front-end, second, the mapping of encapsulated design tools into CAD tasks, and third, the construction of a flow map of CAD tasks capturing a plan of execution. As shown in Figure 1.4, the latter may include sequencing relationships (e.g., simulation before synthesis), the decomposition of one CAD task into an execution plan of several lower-level ones (decomposing the high-level synthesis task into scheduling, allocation and binding CAD tasks), the modeling of alternatives (performing either simulation or verification of a design before releasing it), and so on.

There clearly is a trade-off between maximum support in automating the CAD tool sequencing versus allowing for freedom in performing arbitrary (possibly inconsistent) design processes. The former would be appropriate for fixed design methodologies, i.e, bottom-up or top-down, with predetermined CAD task sequencing. This allows for the configuration of the methodology into a fixed execution plan, called the design process. In this case, the system could be fine-tuned to execute these static design flows. Furthermore, the system could support the design process in a number of different ways, for instance, by pre-processing necessary data formats before presenting them to CAD tools, by maintaining a detailed and efficient capture of design flows, by performing history tracing on these captured design flows, by performing flow sequence optimization, by automating tool invocation, and by avoiding unnecessary tool invocations or data translations. On the other hand, such automated support may be too rigid, thus limiting the generation of

solutions for non-routine designs problems, which requires a non-standard sequencing of CAD tasks. While initial proposals of design process models have emerged [Bing90, Hame90, Bush89, Dani99, Zane92, Kash92], many of the issues discussed above are not well-understood and thus need to be further investigated.

### 1.4.2 Design Entity Management

A complex design is broken into smaller pieces of design and each piece is solved by a possibly different tool one at a time. Each piece of design that is prepared as input to a CAD task, that is being generated as output by a CAD task, that is being decomposed into smaller pieces of design objects for consumption by CAD tasks, or that is being re-assembled into a more higher level design, corresponds to a basic unit of information at the *design entity* layer, called design entity object. The management of the design entity objects generated and/or consumed during this process and their coordination into a consistent design is the focus of the second layer. As shown in Figure 1.4, the design entity layer thus captures the overall administrative organization of the design data by maintaining various semantic relationships between the different pieces of design data. It supports, for instance, design data hierarchy, version derivation, and configuration relationships.

The design database becomes a valuable tool for the *coordination* of evolving designs when supporting these semantic relationships explicitly rather than leaving this task to the designer. This conceptual data model breaks the complex design information into several independent workable units at the request of the designer and maintains linkage between the parts. This *design modularization* can be based on design level boundaries, on abstraction levels of the design, on information domains, or on any other desired criteria. The designer or design tool will be able to focus in on the chosen portion of the design without having to keep track of how this current piece of design relates to the design as a whole. The database therefore supports *traceability* of information across levels of design and across different information domains.

*Version control* and *configuration management* are two important issues for design management that should be addressed at this level of abstraction. The engineering design process is exploratory, that is, many different alternatives may be generated for each piece of design, and iterative, that is, evolutionary versions of a design may exist that get continuously refined. Therefore the design entity model supports the capture and manipulation of many different versions of a design as well as the version derivation relationship among them. It also allows for the selection of different synthesized pieces of the design to compose a particular configuration.

It is at this meta-data level that the designer may want to perform object selection for design exploration. The designer may for instance be interested in locating the most recent design implementation that has been generated for a given design specification. Or, the designer may want to determine the most favorable design among a set of alternatives according to some quality measure (i.e., smallest in area or fastest in delay). The latter is a potentially difficult problem since the design may be incomplete or it may be too time-consuming to obtain the design information at the abstraction level required for extracting exact measures (i.e., it may be unreasonable to wait for detailed layout calculations before giving an area estimate). Note that the database needs to support both temporal as well as fuzzy queries on possibly incomplete information to handle the above mentioned requests [Rund89]. While a number of models for version and configuration management have been proposed in the literature, some of the more difficult problems, such as incomplete design information, estimation, temporal or approximate queries, remain unaddressed.

### 1.4.3 Design Data Management

The third level of the model describes the actual design data that is being generated and consumed by design tools. The design data is arbitrarily complex with component hierarchies, different abstraction levels (such as algorithmic, functional, register-transfer and transistor), and multiple information domains (such as specification, behavioral, structural, or physical domains). One is thus concerned with developing an appropriate design representation for the application at hand. The design representation model needs to capture all design information related to the stages of the engineering design process depicted in Figure 1.1. The design information ranges from the initial design specification, over various internal behavioral representations, down to the structural implementation of the design. The development of such an explicit design representation model would then replace the often ad-hoc efforts of design tool builders to establish their own local data structures. An added benefit of this effort will of course be a better understanding of the engineering design information.

In this dissertation, I present a solution to this problem for the behavioral synthesis domain by developing a unified design data model. I model the design data in behavioral synthesis by the behavioral and the physical graphs (in Figure 1.4). The behavioral graph model captures the hierarchical description of the design behavior on the operation level, which is augmented with control and data dependencies, timing and physical constraints, and state and component bindings. The physical graph

model captures the hierarchical structure of interconnected components augmented by timing and physical design information [Rund90b].

The database needs to gracefully handle the description and manipulation of *incomplete design information*. The purpose of a design database is to support the incremental building of a design from the original textual specification, over its gradual refinements, down to the eventual complete design implementation. Hence, the database needs to be able to (1) gracefully handle the specification and manipulation of partial designs (*incomplete data*) and (2) to respond in a sensible manner to retrieval requests or queries on such data. This also touches on the issue of *design estimation*, since the database needs to supply approximate answers about characteristics of the possibly incomplete design by using estimation routines rather than complete synthesis. For instance, given a request, such as "Provide area information on a design that has not been laid out yet," the database may provide an estimate of the area of a data path based on the given netlist in a few minutes, whereas the layout tool that determines the exact area of a data path would have a running time of several days.

As shown in Figure 1.4, the design entity level partitions the design data into collections of related design objects. Each design entity abstracts a collection of related design data objects, e.g., all components describing a register-transfer design would be grouped together. A design entity object may contain domain-specific attributes that characterize the abstracted design data. In the behavioral synthesis model, for instance, a design entity will have attributes such as estimated-area, maximum-delay, and number-of-components. These summary attributes at the design entity level provide quality measures of the abstracted design information that can be used to retrieve different design versions by one or a combination of quality measures without having to inspect the actual content of the complex design [Katz90] and [Chiu92]. To ensure the consistency between the attributes kept in the design-entity graph and the design data, the database associates with each attribute an application-specific procedure that automatically calculates the value of the attribute from the underlying design data.

Since the design data level is concerned with actual design data that is generated and consumed by diverse design tools, we need to provide support for design tool access at this fine-grained level. There are generally two approaches taken in the literature to address this problem: one, the development of data translators between the global object model and the local tool data structures and, two, the development of one abstract-data-type interface to the data kept in the database that all tools need to use. In this dissertation, I present a new solution to this problem by utilizing

object-oriented views as a flexible mechanism for tool access to the complex design data model.

#### 1.4.4 Status of Design Object Management

In the past, most work on CAD databases has focused on the second level of the design object spectrum shown in Figure 1.4 [Katz82], [Afsa89], [Gupt89] [Siep89], [Caso90], [Katz90], [Wolf90], [Bing90], [Chiu92], and [CFI92]. In particular, they have developed generic, i.e., application-independent, models of maintaining this administrative information. In the last few years, however, there has been more interest in the CAD community in supporting the design process itself, i.e., in the first level of our model [Reiss90], [Bing90], [Caso90]. Little effort on the other hand has been devoted to the third level of design object management. One reason for this may be that this level deals with application-specific design information. Work at this level thus requires expertise of both the database field and the application domain. Secondly, it is more difficult to provide solutions that are of general applicability to other (or better yet all) application domains.

This dissertation focuses on the third level of object management. Most data models at the higher two levels assume a trivial solution for the design data level. Namely, they assume that a design file is associated with each design entity and that individual design tools manage the content and format of these design files. [Wolf90, Bing90]. I, on the other hand, propose fine-grained design management support to improve and support this interaction of tools at the design data level. Our approach offers many advantages as already described in this section. As can be seen by the three-layered object model, the solution that I proposed for the third level is independent from existing proposals for the higher levels. In fact, design management support at the design data level can easily be combined with existing approaches at the higher levels to create an object server supporting the complete spectrum of design information.

### 1.5 Approaches towards Tool Integration

Most work on tool integration addresses the control or process integration aspect, i.e., they develop control mechanisms for tool sequencing, remote procedure calls, and inter-process communication mechanisms [Reiss90]. In terms of the model presented in Section 1.4, this work is at the process or at the design entity level of



the design object space. In this dissertation, on the other hand, I address the tool integration problem at the design data level. Namely, I present a mechanism for effectively sharing complex design data structures between tools, a problem generally ignored by the above mentioned approaches. Our mechanism allows tools to work on their own model of the design information while automatically assuring the consistency of the data when operated on by other tools.

In this section, I review existing approaches towards tool integration at the design data level, while in the next section I will outline our own approach. The discussion in this section is cast in terms of the following three categories of tool integration:

- The File-Based Approach
- The Traditional Database Approach
- The Object Server Approach

### 1.5.1 The File-Based Approach

As already discussed earlier, most CAD systems support a simple file-based approach towards tool integration at the design data level [Thom92, Siep91, Caso90, EDIF]. They assume that the actual design data is kept persistently on design files. There is no uniform design representation model established to hold the combined information. Instead the design information spread over a number of different files is encoded into a number of different file formats. In the file-based approach, the system generally imposes no interpretation on the contents of design files, leaving this to the tools.

In such a CAD system, tool communication is taken care of by exchanging design data between tools using design files. In most cases, translation mechanisms as well as file formats are established in an ad-hoc manner between each pair of design tools that need to exchange information. Hence, there often is an explosion of file formats and translators. Other disadvantages of this arrangement are the vast number of different exchange formats, the mismatch in diverse representation models, and the cost in developing format translators every time a new tool is added or the present representation is changed. Since complete design files are being interchanged, incremental update is not possible. Also since no one is in charge of coordinating the partial design information, there will possibly be many different alternative versions of the same, slightly modified, pieces of data. This file-based approach is depicted in Figure 1.5.a.

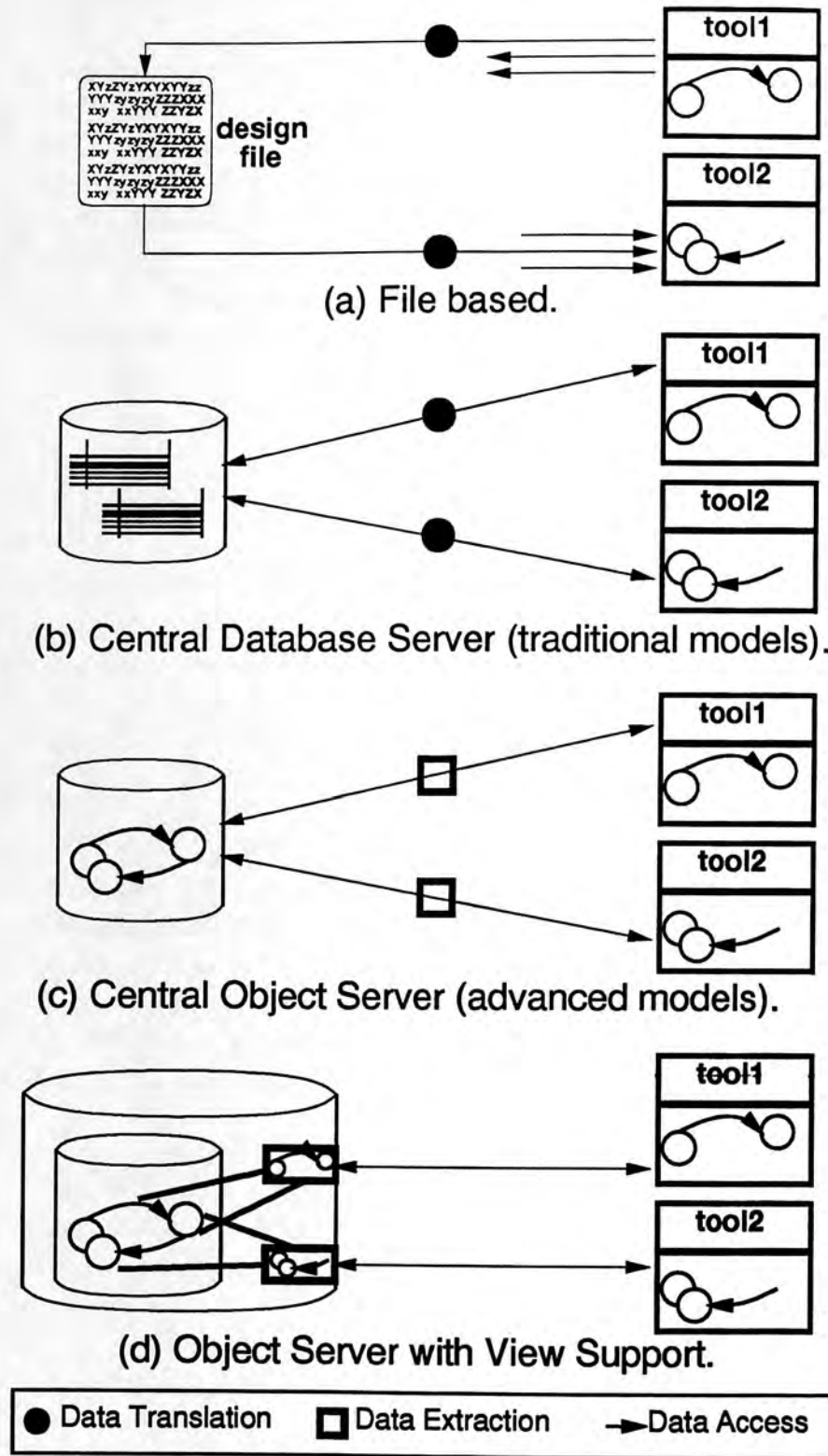


Figure 1.5: Different Tool Integration Schemes.

There has been some interesting work related to the exchange of structured design data using a file-based approach based on the interface description language (IDL) [Snod86]. IDL is an abstraction description language designed to describe the properties of structured data using typed, attributed directed graphs as data model. Snodgrass and Shannon [Snod86] have developed an IDL tool kit that given an IDL description generates declarations of semantically equivalent data structures in a target programming language, utilities for manipulation of main-memory data, and input and output routines between instances of that data structure and their IDL textual description. This approach does of course still suffer from all of the disadvantages of the file-based approach outlined above. It does however offer the advantage that design tools are not constrained to work on one common data model.

### 1.5.2 The (Traditional) Database Approach

Due to the above mentioned disadvantages, there have been proposals in the literature to utilize a (traditional) database system for maintaining all design data [Afsa89, Neum83]. The major point here is not just the use of a database system in place of a file system for the storage of information, but rather the concept of a centralized and unified design representation that captures and organizes the information from all tools into a consistent global model.

Different models have been proposed in the literature to this effect, but generally - and not surprisingly - they are limited to a particular subset of an application domain. The development of such a uniform design representation model for a given application domain is of course a very challenging problem. A final solution can practically only be reached for a well-understood application domain in which a generally accepted standard of the information model can be agreed upon. Otherwise, one can at best expect to develop such a uniform model for a given application site.

Note here that a central design database would make point-to-point communication between design tools unnecessary. Using one common data model for the design information will thus tremendously simplify the task of tool communication, since now each design tool has to interface only with one other system, namely, the design database. This then provides an indirect connection with all other design tools as long as they are also communicating with the database. It not only simplifies tool integration but it also eliminates the loss of design information caused by numerous translations of data from one tool's local data structure to another.

To summarize, advantages of the central database server approach are that (1) each tool has to develop but one interface (namely, to the design database)

rather than to numerous different tools and file formats, (2) design data generated by various tools is being integrated into one consistent database using the uniform design representation model, (3) incremental update of the design data, and (4) the database provides traditional database support functionalities, such as controlled access to shared data, concurrent data access, and integrity support.

Initial proposals of CAD databases suggested the use of traditional database models [Date90, Ullm88] for this purpose [Neum83], since those were the only database technology available at that time. These traditional database models [Date90, Ullm88], the best known of which is the relational model [Date90, Codd79], though being advanced for their time have been shown to be inadequate for supporting complex applications like CAD. I list some of these reasons below, while more extensive discussions of this topic can be found elsewhere [Kent79].

Traditional database models were the first effort of organizing large amounts of data in some well-defined, rigorous manner. Consequently, their primary focus has been on designing an internal organization of data which complies with physical implementation considerations, such as compact storage, efficient retrieval facilities, etc. [Kent79]. The result is a rigid, inflexible structure into which all the data has to be forced - disregarding its meaning. These record structures arrange data in fixed linear sequences of values [Kent79]. This record construct is a machine-oriented structure which is not powerful enough to model complex design data unambiguously and accurately. In fact, a given construct is being used to represent several distinct semantic constructs. Hence supplementary information not found in the database needs to be known to determine its particular usage at any given point of time. For instance, multiple fields in a record could be one of three things, the compound name of an entity, a relationship among entities, or multiple independent facts about an entity. Thus, much of the information to be modeled is implicitly hidden in special-purpose application programs, instead of being directly incorporated in the database structure.

Being of limiting modeling power, there is often no direct correspondence between entities in the application domain and record structures in the database. In fact, records often do not correspond to any real world information but are only artificial modeling constructs (implementation details) required to represent the complex real world information using the tabular format. The user thus is often required to be conscious about these low-level modeling choices. Note also that, for example, a person can be viewed as a single entity or as an instance of several entity types, such as person, employee, student, etc. The latter cannot be modeled by record structures since a record cannot be member of several record types. Hence, these models, which

are easily adapted to the computer, are often awkward to the inexperienced user and semantically inadequate for modeling most advanced application environments.

In addition, the traditional models allow operations to be performed which do not have any real world equivalent, e.g., a join on two possibly unrelated fields is a legal database operation though being meaningless in the real world. This lack of semantic expressiveness is conducive to errors, since a user can easily misinterpret the intended semantics of the represented information, which then can hardly lead to sensible interpretation, successful usage or effective manipulation. In addition, the associated query languages are in general very low-level and complex and require knowledge about the actual physical structure underlying the database. For example, to retrieve information about one particular entity, a user may have to combine (join) various records to reconstruct this information.

In short, being limited in their modeling capabilities, traditional models cannot easily cope with the complex information modeling needs of application domains such as Computer-Aided Design. The gap between the operations to be performed by design tools and the machine-oriented operations supported by the data model is too large, making an effective use of a design database extremely slow and cumbersome. Therefore, initial efforts of CAD databases based on traditional database technology were rejected by the CAD community and thus generally failed.

### 1.5.3 The Object Server Approach

Conventional databases that handle simple and regularly structured entities had been developed for business and banking application domains. New application domains, such as Computer-Aided Design, handle entities with very complex internal structures and a potentially large number of unstructured properties, also called *complex objects*, and thus have other modeling requirements. For these reasons, the database community has developed new conceptual data models to deal with information which is less regularly structured (and more complex) than would be found in traditional database applications. The most prevailing one is the object-oriented data model, which derived from work in data abstraction and object-oriented programming, has become increasingly popular in the last few years. Detailed descriptions of different object-oriented models can be found in the literature [Maie87, Maie86, Bane87b, Kim89, Cope84], while below I discuss the basics only.

In object-oriented programming, the concept of abstract data types has been combined with the idea of message passing. In object-oriented systems, all conceptual entities are *objects* with an object encapsulating a private state and its behavior (methods), similar to an abstract data type. Objects communicate with each other by passing messages. An object responds to messages sent by other objects by executing the appropriate method, i.e., it generally changes its state or returns a result. The set of messages an object responds to is called its protocol and objects may be inspected or changed only through its protocol. Similar objects are grouped together into *classes* - each object is said to be an *instance* of one or more classes. Important constructs inherent to object-oriented data models are data abstraction (the class concept encapsulates both the representations and operations associated with a data type in one declaration), object identity (a time-invariant, unique, system-assigned identity for each entity in the database independent from the status of the object that can be used to uniquely reference objects), generalization hierarchy (classification of classes according to their generalization and specialization of one another), decomposition hierarchy (capability to support complex objects), and property inheritance (classes are organized into an inheritance structure so that objects that belong to different classes can share common properties). Examples of object-oriented database prototypes and systems are ORION [Kim87, Bane87b], Iris [Fish87], Cactis [Huds86], POSTGRES [Ston86], EXODUS [Care88], GemStone system [Cope84].

The object-oriented model is therefore a powerful model, that is more suitable to capturing complex information found in design applications than the relational model. In addition, operations of the application domain can be directly supported by the model rather than requiring complex translations between database and application operations. Using the object-oriented model as a model for the centralized object server of a CAD framework therefore is a rather natural suggestion, offering all advantages of the traditional database system with the added benefit of increased modeling power. Indeed, initial proposals can be found in the literature that propose the third approach, namely, the use of an object-oriented database for design object management [Gupt89, Wolf90].

While being promising, the object server approach (and thus the systems mentioned above) assume that all design tools interface with this *one* complex object schema containing the complete design information. It does not address the fact that different design tools are interested in different pieces of the information content as well as in different organizations of the information. Therefore, I propose to extend the object server approach by design view mechanisms as further discussed in the next section.

## 1.6 Tool Integration Using the View-Based Object Server Approach

The study of existing object models and tool integration approaches presented in the previous section has revealed that there is no adequate solution for tool integration at the lowest level of the design information spectrum (Figure 1.4). In this section, we therefore propose a view-based approach that represents a simple yet powerful solution for tool integration.

The object server approach discussed in the previous section assumes that *all* design data produced and consumed by different design tools has been unified into one design representation model. Ideally, all design tools should be able to work directly on this uniform design representation. In reality, however, design tools work on different levels of abstraction as well as in different information domains. Therefore every tool requires a different subset of this global design information. Furthermore, design tools require a different format or data organization of the same data to best suit their individual design tasks. The global data model, capturing the complete spectrum of design objects and serving all design tasks, is generic and very complex, and therefore cannot fulfill these diverse requirements of different design tools.

In addition, a design database must support designer intervention in the design process. Similar to a tool, a human designer does generally not want to see all aspects of the design at one time since the complexity would be too overwhelming. By generalizing, one can say that different aspects or characteristics of a design are required by different users depending on the type of user (e.g., a human or a program), the task at hand (e.g., optimization or verification), and the desired goal (e.g., state-by-state usage of available resources). This idea is based on the philosophy that all truth is relative to the individual and the time and place in which he acts [Gilb90b].

In this dissertation, I suggest that the design database should account for this problem rather than burdening every user with irrelevant information. I want to be supporting multiple, perhaps conflicting, perceptions of the organization and information content of the data in a database system. In order to support this variety of user expectations and needs, the database system must provide diverse customized interfaces to the global design representation model suitable for different types of user groups and application goals. These customized interfaces, also called *design views*, generally correspond to a subset of the global design information and have a possibly reorganized format specified by the tools. Design views thus are

*tool-specific local schemata* that have been defined based on the global schema. A design view is as close as possible to the perception that the users of this view have about the design process and its design data. The global schema on the other hand carries all available information related to the design, and therefore cannot directly address the particular representation needs of all users.

The view problem of supplying specially-tuned representations to specific database users – though a recognized problem in database research – has been ignored in the CAD literature. Existing CAD databases assume a central data model and require all design tools to directly interface and live with whatever the data model provides [Knap85, Katz85, Katz90, Wolf90, Caso90] (See the traditional database approach and the object server approach described in Sections 1.5.2 and 1.5.3, respectively). The burden of interfacing and of mapping from the tool's local view to the database's global data structures and back is left with the individual design tools.

In this dissertation, I propose to develop a view model that incorporates the design view descriptions as well as their access methods directly with the unified design object model. In other words, the database takes care of implementing and maintaining these view descriptions rather than the individual design tools. The database therefore needs to provide a *view description language* for the creation of these design views. A tool or a set of tools can then use this language to define its own local schema, through which it wishes to view the current design, as well as its own access operations on this schema. This organization has many advantages as will be discussed below.

The view-based object server approach offers all the advantages of the object-server approach, such as the integration of all design data into one explicit design information model, controlled access to shared data, integrity control, the development of one interface per tool only, the possibility for incremental update, and a flexible object model that is powerful enough to capture the complex design information. Furthermore, the approach offers the following additional advantages:

- narrowing the *gap* between global and tool-specific information models by providing tool interface construction support,
- *simplifying* the complex global information model through view customization,
- tuning of the design view to design task by adding *application-specific functions*,
- increasing the *robustness* of the CAD system,
- providing *flexibility* for rapidly adding new customized tool interfaces,



- supporting *extensibility* of the global data model since tools access only through views,
- and increasing data *consistency* since design views provide controlled access privilege to the global information

Views *customize* the global object model so that it suits the needs of particular design tools, i.e., they may hide irrelevant information or restructure the information according to the user's perception. Narrowing the gap between the database and the tool's local model of the design information simplifies the task of tool interfacing. It is thus likely to reduce errors and misunderstandings caused by differences in the global database schema and the local tool models of the design information.

*Robustness* of the CAD system is achieved by shielding tools from changes in the data model. As long as the view on which a particular tool operates is maintained the tool need not worry about any changes to the global data model, such as extensions of the conceptual model or the reimplementations of the model using different data structures. The database will take care of these mappings between the view and the global schema automatically.

The view-based organization provides *flexibility* since new customized views, i.e., tool interfaces, can be created rapidly. This supports the extensibility of the system since new tools can be easily added to the system by simply defining a new view (or possibly using an existing one). Existing tools can work with new ones without having to develop a new interface to these tools. This is likely to decrease the development time of new tools, since they can exploit existing access functions of the global object model. It will thus increase the productivity of design tool developers, who generally spend a major portion of their time designing and implementing tool interfaces.

The approach also guarantees the *extensibility* of the data model since new information can be added to the design data without disturbing existing views. One unified object model as suggested in the literature represents a bottleneck to the overall system, since if a change of the object model were required due to the introduction of a new tool, then all current tools using the object model may have to be revised.

An added advantage of this approach is the increased level of *consistency* since each design tool would only have a restricted access privilege to the global design data. In fact, views can be used to control the access privileges of design tools at any stage during the design process. For instance, the floorplan view of the register-transfer design may allow only for the modification of the floorplan attributes, such

as the set-position() and modify-position() operators, but not the modification of the actual components themselves. The database system can thus guarantee that - as long as a tool operates during this view - the semantics of the design have not been modified or corrupted, meaning, the positioning of components may have changed but that the type of components and their interconnectivity have not been changed. Note that this could be guaranteed without actually having to verify it. This is a very important contribution since the verification of the equivalence of designs would be a costly and in some circumstances an impossible task.

## 1.7 Problem Definition

The concept of *views* is well-known in the context of traditional database models, in particular, the relational model [Date90, Codd79]. In the relational model, a view is defined to be a named, derived, virtual relation specified via a query [Date90]. Rather than executing the query and returning the result to the user, the query is stored as view specification for the virtual relation in the database catalog. This defines the virtual relation in terms of other relations in the database. In short, relational views cover simple variations of base relations [Cham75, Banc81, Daya82], such as renaming or permuting columns, converting units of a column, selecting a subset which satisfies some predicate, and linking relations together into joins.

The view update problem, i.e., the problem of properly translating the update request on a view onto an update request on the global schema, arises because an update on the view may not have a corresponding unique and unambiguous update on the global schema, or it may have undesirable side effects on the original or even other views. Therefore most existing database systems severely restrict the types of updates on views or they even support only non-updatable views [Date90]. Due to the view update problem, views in the relational model are considered to be of limited use for application development. Since I am interested in object management for design environments, with design defined to be the process of continuous and incremental refinement of design data, the updatability of the view mechanism is an important requirement for this work. Fortunately, view update ambiguity is less of a problem for object-oriented database systems as will be discussed below.

As discussed in Section 1.5, the object-oriented model is more suitable for representing and manipulating the complex design information found in design environments than traditional data models. Unfortunately, since object-oriented models are relative newcomers to the database world, the development of view mechanisms for these models is largely unexplored. Therefore, the goal of this dissertation is (1)

Comparison of Concepts	Concepts	Relational Data Model	Object Data Model
<b>data model</b>	<b>entity</b>	tuple = set of simple values	object = set of property functions and relationships
	<b>relationship</b>	no explicit relationships between tuples	explicit relationships between objects (complex objects)
	<b>container</b>	relation = set of tuples	class is composed of -type=set of property functions -content=set of object instances
	<b>container relationships</b>	no explicit relationships between relations	support for explicit relationships between classes, in particular, - generalization hierarchy - type decomposition hierarchy
	<b>schema</b>	schema = set of relations	schema = graph of classes connected by relationships
<b>view constructs</b>	<b>view entity</b>	virtual relation = - defined by a stored query - independent from all other relations in schema	virtual class = - defined by a stored query - interrelated with other classes via diverse relationships
	<b>view schema</b>	view schema = - collection of view definitions - set of 'unrelated' relations	view schema = - schema graph of virtual classes - complex interrelated graph structure

Figure 1.6: Comparison of the Relational and the Object-Oriented Data Models.

to develop a methodology for specifying and maintaining views in object-oriented databases, and (2) to test this methodology by utilizing it for the specification of *design views* for diverse design tools in a Computer-Aided Design (CAD) environment. As stated earlier, I have chosen behavioral synthesis as an example application domain.

In the following, I compare basic constructs of the relational and the object-oriented model to determine how the view mechanism defined for relational models can be applied to object-oriented models. This comparison shown in Figure 1.6 reveals the following simple, yet important, difference between views in relational and in object-oriented systems. A relational data schema is simply a set of 'unrelated' relations [Date90], whereas an object-oriented data schema corresponds to a complex structure of classes interrelated via generalization and decomposition relationships [Kim89, Maie86]. I therefore define An object-oriented view therefore corresponds to a *virtual, possibly restructured, subschema graph* of the global schema, also called a *view schema* [Rund92d]. The construction of these view schemata raises challenging research issues in terms of how to restructure view schema graphs, how to relate them with the global schema structure, how to assure the consistency of the property decomposition hierarchy in a view schema, and how to effectively share information between views. These issues did not arise in the context of the relational model due to its simplicity and set-orientation and thus are important open problems.

In this dissertation, I will present a solution to these problems by proposing an approach for view support in object-oriented models that accomplishes all of the above plus more. A summary of the functionalities that should be supported by object-oriented views, and that are accomplished by our approach, is given below:

- *modification* of object instances, type descriptions of classes, and object memberships of classes as needed in the view,
- *hiding* of object instances, classes, object relationships, and class relationships from the view schema,
- *restructuring* of the generalization and the property decomposition hierarchy of the global schema for a view schema,
- *sharing* of property functions and object instances among classes without unnecessary duplication, and sharing of objects, classes and class relationships across views.

These functionalities must be supported while assuring the consistency of the views with the global schema and the consistency of updates through views.

global object schema (for behavioral synthesis)

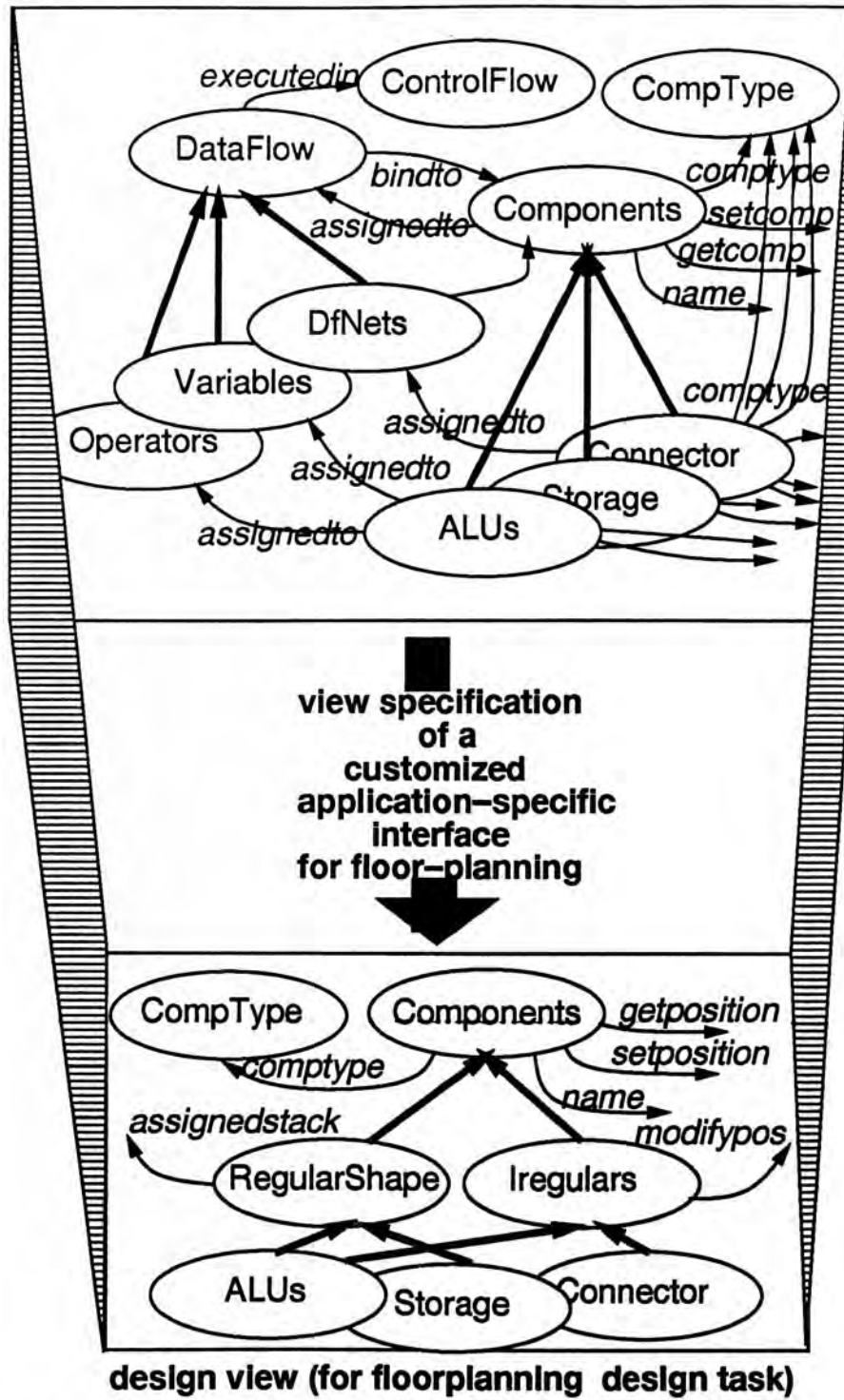


Figure 1.7: An Object-Oriented View for the Floorplanning Design Task.

In Figure 1.7, I give an example of an object-oriented view that demonstrates some of these desired features. This view is designed to support the floorplanning step of behavioral synthesis; with floorplanning defined to be the task of determining the correct positions of components of a chip on a given grid. The global object schema graph for behavioral synthesis is shown on top of Figure 1.7. The view schema is the virtual schema graph defined on top of a global object schema graph shown at the bottom of the figure. First, data-flow and control-flow graph information is not relevant to the task of floorplanning. Therefore, the classes and class relationships from the global object schema that model the data-flow and the control-flow graph are hidden from the view schema. This of course must include a modification of the classes still visible in the view that have references to the now hidden classes. For instance, the `assignedto()` function of the `Components` class has to be removed, since the domain class of this function is no longer part of the view. The figure also gives an example of how the class generalization hierarchy of the global schema is restructured according to the needs of the view. For instance, the floorplanning view may require the classification of components into the two groups of regularly and irregularly structured components, since different floorplanning algorithms are likely to be applicable to components in these two groups. Since floorplanning is concerned with the modification of the positioning of components, new access functions that support such operations have to be introduced. For instance, the functions `getposition()` and `setposition()` have been added to the `Components` class in the view and thus are also inherited by all its subclasses in the view. On the other hand, modification of the design in terms of adding, deleting or modifying existing components is not allowed during floorplanning. Hence, operators of that nature are hidden from the floorplanning view. For instance, the functions `setcomp()` and `getcomp()` are removed from the `Components` class and all its subclasses in the view. Lastly, the `CompType` class and its relationships with other classes are shared between the global and the view schema.

In an object-oriented model, a class encapsulates data types by associating type-specific operators with the object type. An object-oriented view thus consists of type descriptions as well as access and update methods associated with the view. Consequently, at view definition time the meaning of each view access and update is determined. Therefore, no ambiguities will arise when processing through the view. Also, an object-oriented model maintains a unique time-invariant identity for each object in the database, the object identity, which is independent from the external characteristics of the object. Being object- rather than value-based provides the database a mechanism for correctly recognizing and processing an object even if accessed through a view that may hide or modify some of the external characteristics

of the object. For these two reasons, the view update ambiguity is less of a problem for object-oriented database systems compared to traditional - value-based - systems.

It is important to note that I cannot simply modify the existing global object schema so that it suits the requirements of one particular user. Instead, I need to support a number of *different, potentially conflicting, view schemata* of the same data model, each of which supports a particular user's point of view. Consequently, I am concerned here with the *virtual restructuring* for each given view while maintaining all other view schemata; rather than with permanently changing the global database as is done in schema evolution [Bane87a].

## 1.8 Road Map of the Dissertation

In this chapter, I have provided an introduction to the problem of object management for design environments. In particular, I have proposed a three-layered model of design object management and I have shown how my approach is targeted to provide a solution to the third layer. Thereafter, I define the view-based approach towards tool integration. The main part of the dissertation, namely, Chapters 4 to 14, then is devoted towards presenting a methodology, called *MultiView*, for supporting views on object-oriented databases. In Chapter 2, I compare the *MultiView* approach to related work on views for relational and object-oriented database models. Chapter 3 introduces object-oriented concepts related to views. In Chapter 4, I outline the *MultiView* methodology. Chapters 5 to 9 describe the solutions to the different tasks into which *MultiView* decomposes view specification. Chapter 5, for instance, describes the object algebra used for class derivation. Chapter 6 analyzes in more detail the set-theoretic operators of the object algebra. Chapter 7 presents a solution to the class integration problem, i.e., it addresses task two of *MultiView*. Chapter 8 discusses the proposed view definition language used for the third task of *MultiView*. Chapter 9 presents algorithms for view generalization hierarchy generation, which represents a solution for automating the fourth and last task of *MultiView*. Chapter 10 presents two algorithms for verifying and assuring the consistency of a view schema, while Chapter 11 introduces the concept of view independence and shows *MultiView* to be view independent. In Chapter 12, I introduce the application domain chosen to demonstrate the concepts developed in this dissertation, namely, behavioral synthesis. I'll describe the design tasks in behavioral synthesis and present basic elements of a design database for behavioral synthesis. I have developed a unified behavioral design object model to capture the design information of the behavioral domain. This model is also shortly described in Chapter 12. Chapter 13 then demonstrates the feasibility of the proposed view

mechanism for tool integration by applying it to the behavioral design object model. In particular, I specify diverse *design views* customized to the requirements of various design tools in the behavioral synthesis system. I conclude the dissertation with a discussion about contributions and suggestions for future research in Chapter 14.



## Chapter 2

# Related Work on Database Views

In this section, I discuss work presented in the literature on view mechanisms for databases. Since the concept of views has been invented and extensively explored in the context of the relational database model, I review this work on relational views first. I then discuss view support using abstract data types in Section 2.2 and using multiple representations in Section 2.3. The last four sections are devoted to related work in object-oriented views. I roughly divide this discussion into the following four categories. First, I discuss methods for the derivation of virtual classes in object-oriented databases based on query languages. Second, I discuss alternatives to this language-based approach that are based on the idea of multiple protocols. Then, I present work in schema virtualization, which conceptually comes the closest to our work presented in this dissertation. Lastly, I discuss the relationship of particular algorithms that I have used to solve subproblems of our view specification approach with work in the literature in the miscellaneous section.

### 2.1 Relational Views

The concept of *views* is well-known in the context of traditional database models. Since most research concerned with user views in traditional models focused on the relational model [Codd79, Ston86, Ullm88, Date90]. I will review in the following the basic concepts of the relational model and of relational views. This discussion represents a foundation for our work on object-oriented views presented in this dissertation.

In a relational database, information is organized in (relational) tables. Each table, a special case of the set-theoretic *relation*, has a unique name. The rows of a relation table are called *tuples*; and the columns are called *attributes*. Each column within a relation has a unique name, called the *attribute name*  $A_i$ . A relation name  $R$  and its set of attribute names  $\{A_1, \dots, A_n\}$  is called a *relational schema*,  $R(A_1, \dots, A_n)$ , or short  $R$ . A collection of relational schemata is called a *data schema*. The set of

values from which actual values in a column are selected is called the *domain* of the attribute. Let  $U_i$  be the domain of the attribute  $A_i$ , with  $1 \leq i \leq n$ . A *relation*  $r_i$  on  $R$  is defined as a subset of the Cartesian product of the domains  $U_i$ , i.e.,  $r \subseteq U_1 \times \dots \times U_n$ . Each tuple  $t_i \in r$  has the form  $\langle a_{i1}, \dots, a_{in} \rangle$  with  $a_{ij} \in U_j$ .  $t_i[A_j]$  denotes the value of tuple  $t_i$  on attribute  $A_j$ , i.e.,  $t_i[A_j] = a_{ij}$ . The collection of relations  $r_i$  is called the *relational database*.

## Components

Name (String)	AreaCost (Integer)	PartOf (ChipNames)
C1	1000	Chip1
C2	2000	Chip1
C3	1500	Chip2
C4	3000	Chip2
C5	2000	Chip1
C6	1000	Chip2
C7	2500	Chip2

(a) Base table.

```
VIEW Comps-of-Chip1
defined by
SELECT Name, AreaCost
FROM Components
WHERE PartOf="Chip1".
```

(b) View specification.

## Comps-of-Chip1

Name (String)	AreaCost (Integer)
C1	1000
C2	2000
C5	2000

(c) View table.

Figure 2.1: A CAD Example Using Relational Views.

I now explain these concepts based on the example shown in Figure 2.1. The relational schema `Components(Name,AreaCost,PartOf)` is for instance depicted in Figure 2.1.a. `Name`, `AreaCost`, and `PartOf` are attributes with the domains of the attributes being `String`, `Integer`, and `ChipNames`, respectively. The `Components` relation has seven tuples, each modeling an individual component from the application domain. The first tuple  $t1 = \langle C1, 1000, Chip1 \rangle$  represents for instance a component with  $Name[t1] = C1$ ,  $AreaCost[t1] = 1000$ , and  $PartOf[t1] = Chip1$ .

The relational model is *value-oriented*, that is, each of the data items  $t_i[A_j]$  is a single simple value from a primitive data or enumeration type, such as `Integer` or `ChipNames`. This important feature of the relational approach is also its biggest drawback, since relationships between tuples are not directly represented except those modeled implicitly by matching attribute domains and values. For instance, a component tuple in the the `Components` relation in Figure 2.1.a may correspond to a chip modeled by the *PartOf* attribute of the relation. In this example, this can only be determined by checking whether the name of a component `comp`,  $Name[comp]$ , also appears in the third column for another tuple in that relation. Similarly, the relational model cannot explicitly model relationships between relational schemata. Again, possible relationships between relational schemata would have to be inferred by comparing the domains of their respective attributes. Hence, a *data schema* corresponds to a set of apparently unrelated relation tables.

In the relational model, a *view* corresponds to a *named, derived, virtual relation* defined via a query [Date90]. Note that rather than executing the query and returning the result to the user, the query is stored as view specification for the virtual relation in the database catalog. This definition defines the virtual relation in terms of other relations in the database. For a relation to be *virtual* means that the underlying table does not exist in its own right but only looks to the user as if it did. There is no physically separate, distinguishable data stored for these relations. In short, relational views cover simple variations of base relations [Cham75, Banc81, Daya82], such as renaming or permuting columns, converting units of a column, selecting a subset which satisfies some predicate, and linking relations together into joins.

The view relation `Comps-of-Chip1` depicted in Figure 2.1.c, for instance, is defined by the view specification shown in Figure 2.1.b: “VIEW `Comps-of-Chip1` defined-by SELECT `Name, AreaCost` FROM `Components` WHERE `PartOf='Chip1'`”. The `SELECT` clause of the query determines the two attributes of the new relation; in this example it is `Name` and `AreaCost`. The `FROM` clause indicates from what source relations to extract the information for the new relation; in this example from the `Components` relation. The `WHERE` clause specifies the condition that all tuples must fulfill in order to be included in the result relation; in this example the

components must belong to Chip1. Since the condition  $\text{PartOf}[t_i]=\text{Chip1}$  evaluates to true for the tuples  $t_1$  with  $\text{Name}[t_1]=C_1$ ,  $t_2$  with  $\text{Name}[t_2]=C_2$ , and  $t_5$  with  $\text{Name}[t_5]=C_5$ , these three tuples are being selected for the result relation. As specified in the SELECT clause of the query, the attributes Name and AreaCost are being projected out for these three tuples. It can easily be seen that the Comps-of-Chip1 relation represents the set of all components that are part of Chip1.

A *view schema* corresponds to a collection of relational tables, some of them base and some of them view relations. For instance, the schema consisting of the base relation Components and the virtual relation Comps-of-Chip1 in Figure 2.1 is an example of a view schema.

Note that the view relation Comps-of-Chip1 is a 'window' into the Components relation that may be desirable to a designer during area estimation of Chip1. This window is dynamic, that is, changes of the base relation will be automatically visible through the view relation, and vice versa. The view relation can thus be treated like a real table (with certain exceptions discussed below). The *generic* set of query and update operators of the relational data model (the relational query language) that can be applied to both base relations and view relations. Querying and updating of views is done by converting operations on the view relation into equivalent operations on the underlying base relation(s). Whenever a view is accessed for further querying, data retrieval and update, the stored query is executed by the query processor to retrieve or update the requested design data from the global schema. This conversion process is relatively straightforward for retrieval but not necessarily for update operations. The view update problem, the problem of properly translating the update request on a view onto an update request on the base schema, arises because an update on the view may not have a corresponding unique and unambiguous update on the global schema, or it may be ill-defined and have undesirable side effects on the original or even other views. Since there is no mechanism to restrict this application of these generic update operator or to support the interpretation of their application for a given view, the view update problem arises. In fact, it has been found that there is an extremely limited set of correctly translatable view updates. Therefore most existing database systems severely restrict the types of updates on views or they even support only non-updatable views [Date90].

The fact that not all views are updatable is demonstrated with the example given in Figure 2.2. The view table Chip1-Areas defined by the query "VIEW Chip1-Areas defined-by SELECT AreaCost, PartOf FROM Components WHERE PartOf='Chip1'" is theoretically not updatable. This is so since it does not preserve the primary key of the underlying base relation. The view relation in Figure 2.1 preserves the primary key, and thus is more likely to be updatable. I can for

## Components

Name (String)	AreaCost (Integer)	PartOf (ChipNames)
C1	1000	Chip1
C2	2000	Chip1
C3	1500	Chip2
C4	3000	Chip2
C5	2000	Chip1
C6	1000	Chip2
C7	2500	Chip2

(a) Base table.

VIEW Chip1-Areas  
defined by  
SELECT AreaCost, PartOf  
FROM Components  
WHERE PartOf="Chip1".

(b) View specification.

## Chip1-Areas

AreaCost (Integer)	PartOf (ChipNames)
1000	Chip1
2000	Chip1
2000	Chip1

## Chip1-Areas

AreaCost (Integer)	PartOf (ChipNames)
1000	Chip1
2000	Chip1

(c) Non-updatable view table.

Figure 2.2: An Example of the Update Ambiguity of Relational Views.

instance delete an existing tuple from the Comps-of-Chip1 relation, say the tuple  $\langle C1, 1000 \rangle$ , by deleting the corresponding tuple  $t1 = \langle C1, 1000, Chip1 \rangle$  from the underlying base relation. I can insert a new tuple into the view, say  $\langle C9, 5000 \rangle$ , by inserting the tuple  $\langle C9, 5000, Chip \rangle$ . Note that in this case I assume that inserting a tuple into a view means that it should be visible in the view, i.e., that it fulfills the condition of the view specification. On the other hand, if I attempt to delete the tuple  $\langle 2000, Chip1 \rangle$  from the Chip1-Areas view in Figure 2.2, then the system will not know which tuple to delete from the underlying base relation: it could be  $\langle C2, 2000, Chip1 \rangle$ ,  $\langle C5, 2000, Chip1 \rangle$ , or both. Similarly, if I try to change the tuple  $\langle 2000, Chip1 \rangle$  to  $\langle 5000, Chip1 \rangle$  in the Chip1-Areas relation, then the intended action on the base table is again ambiguous.

This view update problem for relational views has been studied extensively in the literature [Cham75, Banc81, Daya82]. Due to this problem, views in the relational model are considered to be of limited use for application development. Since I am interested in object management for design environments, with design defined to be the process of continuous and incremental refinement of design data, the updatability of the view mechanism is an important requirement for tool integration. Fortunately, the view update ambiguity is less of a problem for object-oriented database systems. In an object-oriented methodology, each view description would consist of type descriptions as well as all access and update methods associated with the view. Consequently, at view definition type the meaning of each view access and update has to be determined and therefore no ambiguities will arise during processing. A more detailed discussion of this can be found in a later chapter.

There is yet another problem with the view designed for supporting area determination of Chip1 (Figure 2.2). The relational model being value-based does not allow for duplicate tuples in a relation. As can be seen in the example in Figure 2.2.c, the two tuples  $\langle C1, 2000, Chip1 \rangle$  and  $\langle C5, 2000, Chip1 \rangle$  look the same (and, in fact, are identical) in the view relation, namely, they are equal to  $\langle 2000, Chip1 \rangle$ . Therefore, in the Chip1-Areas relation it will appear as if the Chip1 design has two (rather than three) components with the areas of 1000 and 2000, respectively. This problem is overcome by the object-oriented model due to the concept of object identities [Khos86]. An object identity is a unique, time-invariant, system-supported identifier of each object in the system that is independent from the values that the object takes on during the course of its existence. For instance, a component that changes its position in the floorplan or its area will still be modeled as the same component in the database. Therefore, in a situation similar to the one in Figure 2.2, the object-oriented model would maintain two, externally indistinguishable, object instances in the view [Khos86, Rund93].

## 2.2 Extending Relational Views Using Abstract Data Types

There have been some efforts to overcome these problems of relational views using the abstract data type (ADT) approach [Rowe79, Brod84, Brod87]. An ADT approach characterizes the objects of a type by the operations performed on them and hides from the programmers details of how such objects are represented. The basic ideas of these approaches is that the view definition contains not only a description of how to derive the view from the underlying database, but it also contains a description of the set of allowed view updates together with their translations into updates over the base schema. This approach requires extended view definition capabilities for specifying the operators and their translations, methods of verifying that update translations are indeed correct have to be devised. The disadvantages of this approach are that (1) the view definition language is no longer within the relational model, and (2) the view definer is required to give an exact translation of each operation on the view onto the underlying stored relations.

## 2.3 Maintaining Multiple Representations

Garlan's work [Garl87] is based on constructive rather than derivative views. There is no underlying common database, instead a number of different basic views are first defined and then 'merged' with one another. In conventional database terminology, basic views are really type descriptions. Merging of views is accomplished using a purely syntactic approach, i.e., if two types have the same name and the same domain then they are considered to be different descriptions of the same data. This is equivalent to maintaining multiple representations of the same data, and propagating the update on one instance through a view to all other representations of that instance. Garlan [Garl87] also proposes the concept of dynamic views, which corresponds to a materialized derived attribute defined via a function decomposition that returns a collection of homogeneous objects. Update is not allowed on these dynamic views. Garlan [Garl87] claims that his underlying model is object-oriented. However, since the model does not support concepts, such as object identifiers, encapsulation, and generalization hierarchy, I have placed the discussion of his work in this section.

## 2.4 The Derivation of Virtual Classes

An immediate extension of the view mechanism from relational databases to OODB systems is to define a view to be equal to an object-oriented query. In fact, many efforts of defining views for OODBs follow this approach; that is, they suggest the use of the query language defined for their respective object model to derive a virtual class. Some examples are view mechanisms for the Fugue Model in [Heil90], for the Orion model in [Kim89], and for integrating databases in [Kaul90]. *MultiView* can use any of these proposed class derivation mechanisms to implement the first phase of view schema generation, i.e., the customization of individual classes. In this sense, *MultiView* is a superset of these approaches.

Most of these approaches do not discuss the integration of derived classes into the global schema. Instead, the derived classes are treated as “stand-alone” objects [Heil90], or they are attached directly as subclasses of the schema root class [Kim89]. Scholl et al.’s recent work [Scho91] is one of the exceptions. They sketch the class integration process for a selected subset of the operators of the query language COOL. This work is similar in flavor to what I present in Section 5.2 on object algebra. Namely, they determine whether a derived class should be placed lower or higher than their source classes. This localized class integration approach is directly guided by the derivation of a virtual class, and it is not, as I have shown in this dissertation, a solution to finding the globally most appropriate location in the schema graph. Scholl et al. [Scho91] do not consider the problem of generating multiple view schemata, which is an integral part of *MultiView*. *MultiView* can thus be considered to be a compatible extension of their work.

Abiteboul and Bonner [Abit91] present a view mechanism for the  $O_2$  database system. In this context, they also discuss class integration as an important problem. However, their suggested solution is again a simplistic approach that results in partial rather than in complete classification (See Section 7.1). No precise algorithm for class integration is presented.

In [Rund92b], we discuss the integration of virtual classes derived using set operators into a schema graph. This work focuses on the semantics of set operators and the inheritance of property characteristics, such as single- versus multi-valued or required versus optional. Again, I discuss the relative positioning of the virtual class with respect to its source classes without presenting a general solution for classification.



## 2.5 Multiple Protocol Approaches

Shilling and Sweeney's approach [Shil89] for supporting object-oriented views is based on extending the concept of a class from having one type (one ADT interface) to having multiple type interfaces. The purpose is to limit the access rights to property functions and to control the visibility of instance variables. I accomplish the same goal of specializing types by using the type refinement capability of the generalization hierarchy. Our work is simpler, however, since it does not require the extension of the traditional class concept. *MultiView* can hence be implemented directly on top of existing object-oriented database technology, while theirs cannot. Furthermore, Shilling and Sweeney approach the problem from the programming language point of view and thus are not concerned with the sets of objects attached to a class. Consequently, they do not address the derivation of new classes by restricting the membership of a class via a select-like query. Shilling and Sweeney's work focuses on one class only, and the effects of multiple interfaces on the class generalization hierarchy are not addressed. Powerful view formation by restructuring of the is-a hierarchy as done by *MultiView* is therefore not handled. Consequently, the question of whether a new type interface associated with an existing class is properly integrated with the complete schema remains open. Of course, class integration, which does not become an issue when dealing with an individual class only, is not addressed.

Gilbert's proposal [Gilb90], similar to [Shil89], is also based on the idea of defining multiple interfaces for a class object. *MultiView* does not require the extension of the traditional class concept, and thus can be implemented directly with the existing object-oriented database technology, while Gilbert's approach could not. Nonetheless, *MultiView* is as powerful as the multi-interface approach; any view schema that can be defined using the multi-interface approach can also be defined using our strategy. In addition, our work allows for the direct application of the class derivation mechanisms proposed in the literature. The use of general query operators is currently not handled by [Gilb90]. Emphasis of this approach is instead on issues of parallel processing of queries.

## 2.6 Schema Virtualization

Tanaka et al.'s work on schema virtualization [Tana88] comes the closest to our approach in as much as they are concerned with complete view schemata rather

than just individual virtual classes. However, their work does not distinguish between the task of integrating derived classes into a common schema and the task of generating view schemata. In fact, the interplay between these tasks is not well-defined in their approach. Also, they allow for the arbitrary addition of *is-a* edges in a virtual schema, which in many cases will lead to an inconsistent schema, rather than supporting the automatic generation of the class hierarchy of a view schema as done in *MultiView*. Their approach thus does not assure the validity of a view schema. While recognizing the need for class integration, they do not present a general classification algorithm. They point out that work is needed for developing a systematic approach towards view specification in OODBs, in particular, a view definition language. In this dissertation, I have provided a solution for this. In fact, by breaking the view schemata definition process into a number of distinct phases, I was able to reduce the view definition to a simple yet powerful mechanism. In summary, *MultiView* is a more systematic solution approach compared to their rather ad-hoc (and completely manual) proposal.

## 2.7 Miscellaneous Topics

Algorithms for special forms of the classification problem have been proposed in the Artificial Intelligence literature. Schmolze and Lipkis [Schm83], for instance, describe a classifier for 'concepts' in the KL-ONE Knowledge Representation System. The KL-ONE Knowledge Representation scheme does not include behavioral abstractions and abstract data types as done in an object-oriented model. Hence, the type inheritance mismatch problem is not addressed by their solution. Furthermore, the KL-ONE classifier deals with single-inheritance only, while our class placement algorithm can handle both multiple-inheritance schema graphs. More importantly, they do not assume that a 'concept' has an associated set of object members. Consequently, they do not have to tackle the problem of *is-a* incompatibility between the subset and the subtype hierarchies underlying the general representation scheme. Lastly, since the derivation of new classes is accomplished in *MultiView* using well-defined object algebra operators, our approach is able to reduce the complexity of classification depending on the operator used for derivation. This issue is not addressed in [Schm83].

Our work on classification probably comes the closest to the research by Missikoff et al. [Missi89] on inserting types into a lattice structure. Rather than dealing with an object-oriented model, they assume a simple record-oriented type system. Our classification algorithm, on the other hand, is extended to be applicable to a class

generalization hierarchy. Since a class represents both a type and a set, our classification algorithm solves the *is-a* incompatibility problem. This problem does not arise, and thus is not addressed in [Missi89], when dealing with classification in a type structure rather than in a schema graph.

# Chapter 3

## Object-Oriented Concepts

### 3.1 The Object Model

Below, I introduce the basic concepts of object-oriented database (OODB) models needed for the remainder of the dissertation. Let  $P$  be an infinite set of property functions. property abstractions that includes attributes Each property function  $p \in P$  can be a value from a simple predefined enumeration type, an object instance from some class, or an arbitrarily complex function. Each property function  $p \in P$  has a name and signature (i.e., domain types). For simplicity, I assume for the following that all properties in the schema have unique property names<sup>1</sup>. Let  $T$  be the set of all types. Let  $t \in T$  be a type in  $T$ . Let **properties<sub>t</sub>** be the set of attributes property functions (attributes) of type  $t$  and **domain<sub>p</sub>(t)** denote the range (domain) of the property function  $p$  in type  $t$ . Let  $O$  be an infinite set of object instances. Each element  $o \in O$  is an instance of an abstract data type (ADT), i.e., it can be manipulated by means of the interface of the respective ADT.

Let  $C$  be the set of all classes. A class  $C_i \in C$  has a unique class name, a type description and a set membership. The type associated with a class corresponds to a common interface for all instances of the class, that is, the collection of applicable property functions. I refer to the name of the type associated with a class  $C$  by **type(C)** and to the set of property functions defined for  $C$  by **properties(type(C))**, or short, **properties(C)**. If  $p \in P$  is a property function defined for  $C$ , i.e.,  $p \in \mathbf{properties}(C)$ , then I refer to the domain of the property function  $p$  for  $C$  by **domain<sub>p</sub>(C)**. A class is also a container for a set of objects. The collection of objects that belong to a class  $C$  is denoted by **extent(C) := {o | o ∈ C}** with

---

<sup>1</sup>Note that to determine whether two property functions are identical is equally hard to proving that two programs are equivalent. The uniqueness of property names assumption thus provides a simple yet elegant solution for this otherwise np-complete problem. I ensure uniqueness of properties by associating a unique property identifier with each newly defined property. Two properties that have the same property name can thus be distinguished internally based on their identifier. For other schemes of disambiguation of property names see [Rund92b].

the member-of predicate “ $\in$ ” defined based on the object identities of the object instances [Rund93].

### 3.2 Type Hierarchy and Type Relationships

**Definition 1.** I define a partial order on types  $T$  as follows. For two types  $t1$  and  $t2 \in T$ ,  $t2$  is called a **subtype** of  $t1$ , denoted by  $t2 \preceq t1$ , if and only if

- $\text{properties}_{t1} \subseteq \text{properties}_{t2}$ , and
- $(\forall p \in \text{properties}_{t2})(\text{domain}_p(t2) \subseteq \text{domain}_p(t1))$ .

The first condition of Definition 1 states that a subtype must have the same property functions as its supertype and possibly additional ones. The second condition states that the domains of the property functions of a subtype must be contained within the domains of the respective property functions of the supertype, but that they could possibly be restricted.

Given a finite set of types  $T$ , I call  $t_1$  a *direct subtype* of  $t_n$  and  $t_n$  a *direct supertype* of  $t_1$ , denoted by  $t_1 \prec t_n$ , if  $(t_1 \preceq t_n)$  and  $(t_1 \neq t_n)$  and there are no other types  $t_{k_j} \in T$  (with  $j=1, \dots, m$ ) for which the following subtype relationships hold:  $(t_1 \preceq t_{k_1})$  and  $(t_{k_1} \preceq t_{k_2})$  and ... and  $(t_{k_m} \preceq t_n)$ . Based on this partial ordering function “ $\preceq$ ”, I can define a set of types  $T$  to correspond to a *type hierarchy* that explicitly represents all *direct subtype* relationships in terms of edges of a graph.

**Definition 2.** A **type hierarchy** is a directed acyclic graph  $TH=(TV,TE)$ , where  $TV$  is a finite set of types and  $TE$  is a finite set of directed edges. Each element in  $TV$  corresponds to a type  $t_i$ , while  $TE$  corresponds to a binary relation on  $TV \times TV$  that represents all direct subtype relationships between pairs of types in  $TV$ . In particular, each directed edge  $e$  from  $t_1$  to  $t_2$ , denoted by  $e = \langle t_1, t_2 \rangle$ , represents the direct subtype relationship  $(t_1 \prec t_2)$  between  $t_1$  and  $t_2$  in  $TV$ .

Next, operations on type descriptions are introduced which form a new type based on the sets of properties of two existing types [Rund92b].

**Definition 3.** Let  $t1, t2 \in T$ . Then  $\sqcup$  is a function from  $T^2 \rightarrow T$  that defines a new type  $t3$  by

$$t3 = t1 \sqcup t2.$$

The property functions  $\text{properties}_{t3}$  of  $t3$  are defined by:

$$\mathbf{properties}_{t_3} = \mathbf{properties}_{t_1} \cup \mathbf{properties}_{t_2}.$$

And, for each property function  $p \in \mathbf{properties}_{t_3}$  of  $t_3$ , the following domain is defined:

$$\mathbf{domain}_{t_3}(p) = \mathbf{domain}_{t_1}(p) \cap \mathbf{domain}_{t_2}(p).$$

Intuitively, the new type  $t_1 \sqcup t_2$  denotes the collection of all properties defined for either  $t_1$  or  $t_2$ . In other words, the  $\sqcup$  function creates a new type from two existing ones (1) by building the union of the properties of both types and (2) by forming the intersection of the domains of properties, for all properties common to both source types. The following type relationships hold between the new type ( $t_1 \sqcup t_2$ ) and the two source types  $t_1$  and  $t_2$ :

$$(t_1 \sqcup t_2) \preceq t_1, \text{ and}$$

$$(t_1 \sqcup t_2) \preceq t_2.$$

In fact,  $(t_1 \sqcup t_2)$  is the *greatest common subtype* of  $t_1$  and  $t_2$ .

**Definition 4.** Let  $t_1, t_2 \in T$ . Then  $\sqcap$  is a function from  $T^2 \rightarrow T$  that defines a new type  $t_3$  by:

$$t_3 = t_1 \sqcap t_2.$$

The property functions  $\mathbf{properties}_{t_3}$  of  $t_3$  are defined by:

$$\mathbf{properties}_{t_3} = \mathbf{properties}_{t_1} \cap \mathbf{properties}_{t_2}.$$

And, for each property function  $p \in \mathbf{properties}_{t_3}$  of  $t_3$ , the following domain is defined:

$$\mathbf{domain}_{t_3}(p) = \mathbf{domain}_{t_1}(p) \cup \mathbf{domain}_{t_2}(p).$$

Intuitively, the new type  $t_1 \sqcap t_2$  denotes the collection of properties common to both types  $t_1$  and  $t_2$ . In other words, the  $\sqcap$  function creates a new type from two existing ones by building the intersection of the properties of both types and by forming the union of the domains of properties, for all properties common to both source types. The following type relationships hold between the new type ( $t_1 \sqcap t_2$ ) and the source types  $t_1$  and  $t_2$ :

$$t_1 \preceq (t_1 \sqcap t_2), \text{ and}$$

$$t_2 \preceq (t_1 \sqcap t_2).$$

The type  $t1 \sqcap t2$  is the *lowest common supertype* of  $t1$  and  $t2$ . I distinguish between four cases for applying the  $\sqcap$  function to two types  $t1$  and  $t2$ , which result in different placements of the resulting type  $t1 \sqcap t2$  in the type hierarchy:

1.  $t1 \sqcap t2 = \emptyset$ .
2.  $(t1 \sqcap t2 = t1)$ .
3.  $(t1 \sqcap t2 = t2)$ .
4.  $(t1 \sqcap t2) \notin \{t1, t2, \emptyset\}$ .

In the first case,  $t1$  and  $t2$  have no common properties. By Definition 2, the lowest common supertype of  $t1$  and  $t2$  is then equal to the root of the type hierarchy. This implies that  $t1$  and  $t2$  will be placed into different subgraphs of the type hierarchy.

In the second case,  $t2$  has the same properties that are defined for  $t1$ , and,  $t2$  may also have additional properties. Also the domains for some of  $t1$ 's properties may be further restricted for  $t2$ . By Definition 1,  $t2$  is a *subtype* of  $t1$ .  $t2$  must therefore be placed below  $t1$  in the type hierarchy. Case 3 is identical to case 2 with the roles of  $t1$  and  $t2$  reversed. If, by default,  $(t1 \sqcap t2 = t1 = t2)$  then the two types  $t1$  and  $t2$  are identical and their relative positioning in a type hierarchy would be the same location.

In the fourth case,  $t1$  and  $t2$  share some common property functions, but, in addition, each of them has their own property functions. By Definition 1,  $t1$  and  $t2$  are *type incompatible*. Since no type relationship can be established among them, they have to be placed into different subgraphs of the type hierarchy. As I will show below, this fourth case plays an important role in type classification.

### 3.3 Class Hierarchy and Class Relationships

**Definition 5.** For two classes  $C1$  and  $C2 \in C$ ,  $C1$  is called a **subset** of  $C2$ , denoted by  $C1 \subseteq C2$ , if and only if  $(\forall o \in O) ((o \in C1) \implies (o \in C2))$ .

**Definition 6.** For two classes  $C1$  and  $C2 \in C$ ,  $C1$  is called a **subtype** of  $C2$ , denoted by  $C1 \preceq C2$ , if and only if  $(\mathbf{properties}(C1) \supseteq \mathbf{properties}(C2))$  and  $(\forall p \in \mathbf{properties}(C2)) (\mathbf{domain}_p(C1) \subseteq \mathbf{domain}_p(C2))$ .

Definition 6 is directly based on Definition 1, i.e., a class  $C1$  is defined to be a subtype of  $C2$  if and only if  $\mathbf{type}(C1)$  is a subtype of  $\mathbf{type}(C2)$ .

**Definition 7.** For two classes  $C1$  and  $C2 \in C$ ,  $C1$  is called a **subclass** of  $C2$ , denoted by  $C1$  is-a  $C2$ , if and only if  $(C1 \preceq C2)$  and  $(C1 \subseteq C2)$ .

Informally, I say that  $C1$  is *is-a* related to  $C2$  if (1) every member of  $C1$  is a member of  $C2$  (the subset relationship) and (2) every property defined for  $C2$  is also defined for  $C1$  (the subtype relationship).

Given a collection of classes for a particular database application, I want to organize them in a fashion such that these class relationships are explicitly represented rather than having to recompute them continuously. The maintenance of the subset class relationships allows us to determine the containment of the object instances associated with one class within the extent of another class. This may for instance be useful for query processing. The maintenance of the subtype relationship is useful for the reuse of property function code; this feature is commonly known as property inheritance.

Let  $S = \{ C_i \mid i = 1, \dots, n \}$  be a set of classes. I call  $C_1$  a *direct subclass* of  $C_n$  and  $C_n$  a *direct superclass* of  $C_1$  if  $(C_1 \text{ is-a } C_n)$  and  $(C_1 \neq C_n)$  and there are no other classes  $C_{k_j} \in S$  (with  $j=1, \dots, m$ ) for which the following *is-a* relationships hold:  $(C_1 \text{ is-a } C_{k_1})$  and  $(C_{k_1} \text{ is-a } C_{k_2})$  and ... and  $(C_{k_m} \text{ is-a } C_n)$ .  $C_1$  is called an (*indirect*) *subclass* of  $C_n$  and  $C_n$  an (*indirect*) *superclass* of  $C_1$  if there are one or more classes  $C_{k_j} \in S$  (with  $j=1, 2, \dots, m$ ) for which the above *is-a* relationships hold. The *direct subclass* relationship between  $C_1$  and  $C_n$  is denoted by  $(C_1 \text{ is-a}^d C_n)$ ; the *indirect subclass* relationship with  $(j \geq 1)$  by  $(C_1 \text{ is-a}^* C_n)$ . A graph-theoretic representation of a set of classes  $S$  that explicitly represents all *direct subclass* relationships among the classes in terms of edges is defined below.

**Definition 8.** An **object schema** is a directed acyclic graph<sup>2</sup>  $S=(V,E)$ , where  $V$  is a finite set of vertices and  $E$  is a finite set of directed edges. Each element in  $V$  corresponds to a class  $C_i$ , while  $E$  corresponds to a binary relation on  $V \times V$  that represents all *direct is-a* relationships between all pairs of classes in  $V$ . In particular, each directed edge  $e$  from  $C_1$  to  $C_2$ , denoted by  $e = \langle C_1, C_2 \rangle$ , represents the *direct is-a* relationship between the two classes ( $C_1 \text{ is-a } C_2$ ). There is one designated root node, called **Object**, which contains all object instances of the database and its type description is empty<sup>3</sup>.

I refer to the collection of *is-a* relationships of a set of classes as the **generalization hierarchy** of the object schema. Since the *is-a* relationship is *reflexive*,

<sup>2</sup>A schema without multiple inheritance corresponds to a tree rather than a DAG.

<sup>3</sup>The schema root class provides a unique entry point into the database. Note that this definition is not limiting, since in reality the database schema may correspond to a set of DAGs (some of which may of course be isolated classes) – with their interconnection to the **Object** root possibly hidden to the user.



*antisymmetric* and *transitive*, the schema graph is a directed acyclic graph without any loops. An edge  $e = \langle C_i, C_j \rangle$  is called a self-loop if its source node  $C_i$  and its sink node  $C_j$  are identical, i.e.,  $i=j$ . Furthermore, since I only store the direct subclass relationships, there will be no self-loops in a schema graph. Two or more edges are called multi-edges if they have the same source and the same sink node, respectively. For instance, the edges  $e_1 = \langle C_i, C_j \rangle$  and  $e_2 = \langle C_k, C_l \rangle$  with  $(i=k)$  and  $(j=l)$  are multi-edges. The schema graph also has no multi-edges, since each direct subclass relationship is stored but once.

**Definition 9.** Given a set of classes  $S = \{ C_i \mid i = 1, \dots, n \}$ .

- a. For all classes  $C_1, C_2$  in  $S$ , an arc from source  $C_1$  to sink  $C_2$  is defined to be **required** in  $S$ , if  $(C_1 \text{ is-a } C_2)$  in  $S$  and there is no  $C_x$  in  $S$  such that  $(C_1 \text{ is-a } C_x)$  and  $(C_x \text{ is-a } C_2)$ , i.e.,  $C_1$  is a direct subclass of  $C_2$  in  $S$  denoted by  $(C_1 \text{ is-a}^d C_2)$ .
- b. For all classes  $C_1, C_2$  in  $S$ , an arc from source  $C_1$  to sink  $C_2$  is defined to be **redundant** in  $S$ , if there is a class  $C_x$  in  $S$  such that  $(C_1 \text{ is-a } C_x)$  and  $(C_x \text{ is-a } C_2)$ , i.e.,  $C_1$  is an indirect subclass of  $C_2$  in  $S$  denoted by  $(C_1 \text{ is-a}^* C_2)$ .
- c. For all classes  $C_1, C_2$  in  $VV$ , an arc from source  $C_1$  to sink  $C_2$  is defined to be **inconsistent** in  $S$  if the subclass relationship  $(C_1 \text{ is-a } C_2)$  does not hold.
- d. The object schema graph  $G=(V,E)$  is defined to be **valid** if and only if  $(V=S)$  and the set  $E$  of is-a arcs of  $G$  contains all required and no redundant and no inconsistent arcs in  $S$ .

A valid schema graph  $G$  is complete since, by Definition 9.a and by the transitivity property of the *is-a* relationship, two classes  $C_1$  and  $C_2$  in  $VS$  are – directly or indirectly – connected via an arc in  $G$  if and only if they are also *is-a* related in  $S$ . A valid schema graph  $G$  is minimal since, by Definition 9.b, there is no direct *is-a* arc between two classes if there is already an indirect *is-a* path between them. Lastly, a valid schema graph is consistent since, by Definition 9.c, an *is-a* arc from source  $C_1$  to sink  $C_2$  exists in  $G$  if and only if the two classes are *is-a* related in  $S$ .

Once these class relationships are compiled and stored in this graph format, I can read them directly from the structure of the graph without having to repeatedly compute the *subclass* relationships. For instance,  $C_1$  is a *direct subclass* of  $C_n$  if the edge  $e = \langle C_1, C_n \rangle$  exists in  $E$ .  $C_1$  is an *indirect subclass* of  $C_n$ , denoted by  $(C_1 \text{ is-a}^* C_n)$ , if there is a path through the class hierarchy of length one or longer connecting  $C_1$  and  $C_n$ .

A class is related to other classes via property relationships. For example, if  $C_1$  has defined a property function  $p$  with  $\text{domain}_p(C_1) = C_2$ , then I say that there

is a property decomposition arc between  $C1$  and  $C2$  labeled 'p'. I refer to the set of all property relationships of a schema as its **property decomposition hierarchy**. The concept of a property decomposition hierarchy is more formally defined below.

**Definition 10.** Let  $S=(V,E)$  be an object schema. Let  $L$  be a set of labels that correspond to the names of the property functions in  $P$ . Then the **property decomposition hierarchy** of  $S$  is defined to be a directed graph  $PD=(V,A,L)$  with  $V$  the set of vertices and  $A$  the set of arcs.  $A$  is a ternary relation on  $V \times V \times L$ , called the property decomposition edges. An edge  $a = (C1,C2,l) \in A$  if and only if there is a property function defined for class  $C1$  with the property label  $l$  and the domain class  $C2$ .

A property decomposition hierarchy consists of one or more disconnected subgraphs with possibly loops, self-loops, and multi-edges. The latter are distinguished based on their labels. The property decomposition and the generalization hierarchy graphs are orthogonal concepts, each having their own graph-theoretic restrictions.

Definitions 3 and 4 that define operators on types are now extended to operators on classes.

**Definition 11.** Let  $C1$  and  $C2 \in C$  be two classes. Then  $\sqcup$  is a function from  $C^2 \rightarrow C$  that defines a new class  $C3$  by

$$C3 := C1 \sqcup C2$$

with

$$type(C3) := type(C1) \sqcup type(C2)$$

with the later  $\sqcup$  function equal to the function on types given in Definition 3. And  $content(C3)$  is not specified.

$C3$  is a common subclass of both  $C1$  and  $C2$  if and only if  $C3 \preceq C1 \sqcup C2$  and  $content(C3) \subseteq C1 \cap C2$ .  $C3$  is called the *greatest common subclass* of  $C1$  and  $C2$  if and only if  $C3 = C1 \sqcup C2$  and  $content(C3) = C1 \cap C2$ .

**Definition 12.** Let  $C1$  and  $C2 \in C$  be two classes. Then  $\sqcap$  is a function from  $C^2 \rightarrow C$  that defines a new class  $C3$  by

$$C3 := C1 \sqcap C2$$

with

$$\text{type}(C3) := \text{type}(C1) \sqcap \text{type}(C2)$$

with the later  $\sqcap$  function as defined in Definition 4.  $\text{Content}(C3)$  is not specified.

$C3$  is a common superclass of both  $C1$  and  $C2$  if and only if  $C3 \succeq C1 \sqcap C2$  and  $\text{content}(C3) \supseteq C1 \cup C2$ . I call  $C3$  the *lowest common superclass* of  $C1$  and  $C2$  if and only if  $C3 = C1 \sqcap C2$  and  $\text{content}(C3) = C1 \cup C2$ .

In both definitions, the overloading of the functions  $\sqcup$  and  $\sqcap$  for the class and for the type parameter is a matter of convenience, since I generally deal with classes rather than with types. I use the notation " $C_i \in G$ " to denote that the class  $C_i$  is a member of the set  $V$  of the schema graph  $G=(V,E)$ . For notational convenience, I overload the " $\in$ " operator to also apply to types. For  $t_i$  an element of the set of types  $T$  underlying the schema graph  $G=(V,E)$ , I say that  $t_i$  is an element of  $G$ , denoted by  $t_i \in G$ , if and only if  $(\exists C_i \in G)(\text{type}(C_i)=t_i)$ .

### 3.4 View Schema Definition

I distinguish between **base** and **virtual** classes. **Base classes** are defined during the initial schema definition. Object instances that are members of base classes are explicitly stored as base objects. **Virtual classes** are defined during the lifetime of the database using object-oriented queries, i.e., their definitions are dynamically added to the existing schema. A virtual class has an associated membership derivation function that will determine its exact membership based on the state of the database. The extent of a virtual class is generally not explicitly stored, but rather computed upon demand.

**Definition 13.** *The base schema (BS) is an object schema  $S=(V,E)$ , where all nodes in  $V$  correspond to base classes with stored rather than derived object instances.*

**Definition 14.** *Let BS be a base schema. The global schema (GS) is an extension of the base schema that is augmented by the collection of all virtual classes defined during the lifetime of the database as well as is-a relationships among this extended set of classes.*

A subgraph of the global schema which contains only virtual classes and their is-a relationships is commonly called a *virtual schema* [Tana88, Abit91].

**Definition 15.** *Given a global schema  $GS=(V,E)$ , then a view schema (VS), or short, a **view**, is defined to be a schema  $VS=(VV,VE)$  with the following properties:*

1.  $VS$  has a unique view identifier denoted by  $\langle VS \rangle$ ,
2.  $VV \subseteq V$ , and
3.  $VE \subseteq \text{transitive-closure}(E)$ .

The first condition states that each view schema is uniquely identifiable. The second property states that all classes of  $VS$  also have to be classes in  $GS$ , i.e., they have been properly integrated with the global information. The third property states that the view schema maintains only *is-a* relationships among its view classes that are directly derivable from  $GS$ . In other words, an edge  $\langle C_i, C_j \rangle$  can only exist in  $VE$  if either  $\langle C_i, C_j \rangle$  exists directly in  $E$  or if it is indirectly derivable via the transitivity of the *is-a* relationship, i.e., only if  $(C_i \text{ is-a}^* C_j)$  in  $GS$ . A view schema is a special case of an object schema. Therefore all properties of an object schema defined in Section 3.1 must also hold. I call the classes in a view schema (both the base and the virtual ones) *view classes* and the *is-a* relationships among these view classes *view is-a relationships*.

**Example 1.** *Figure 3.1 shows the relationship between (a) the base schema, (b) the global schema, and (c) and (d) two different view schemata. I depict base and virtual classes by circles and dotted circles, respectively. The global schema in Figure 3.1.b is derived from the base schema in Figure 3.1.a by adding the virtual classes **Minor** and **TeenageBoy** and by interconnecting them with the remaining classes to create a valid schema. The view schematas in Figure 3.1.c and 3.1.d are derived from the global schema by selecting a subset of its classes and interconnecting them into a valid schema using view *is-a* arcs. The view  $VS1$  in Figure 3.1.c contains the classes **Person**, **Minor** and **TeenageGirl**; and the view  $VS2$  in Figure 3.1.d the classes **Person**, **Adult**, **TeenageGirl** and **TeenageBoy**.*

Note that the base schema is a special case of a view schema that consists exclusively of all base classes and no virtual classes. A base, a global, and a view schema are all special forms of an object schema, and, for that reason, each of them must obey the conditions of schema graph validity outlined in Definition 9. In addition, I have developed criteria for evaluating the consistency of the class generalization hierarchy of a view schema with the underlying global schema as well as the closure of the property decomposition hierarchy of a view schema [Rund92d]. These issues are not particular to the work presented in this dissertation, and therefore are omitted.

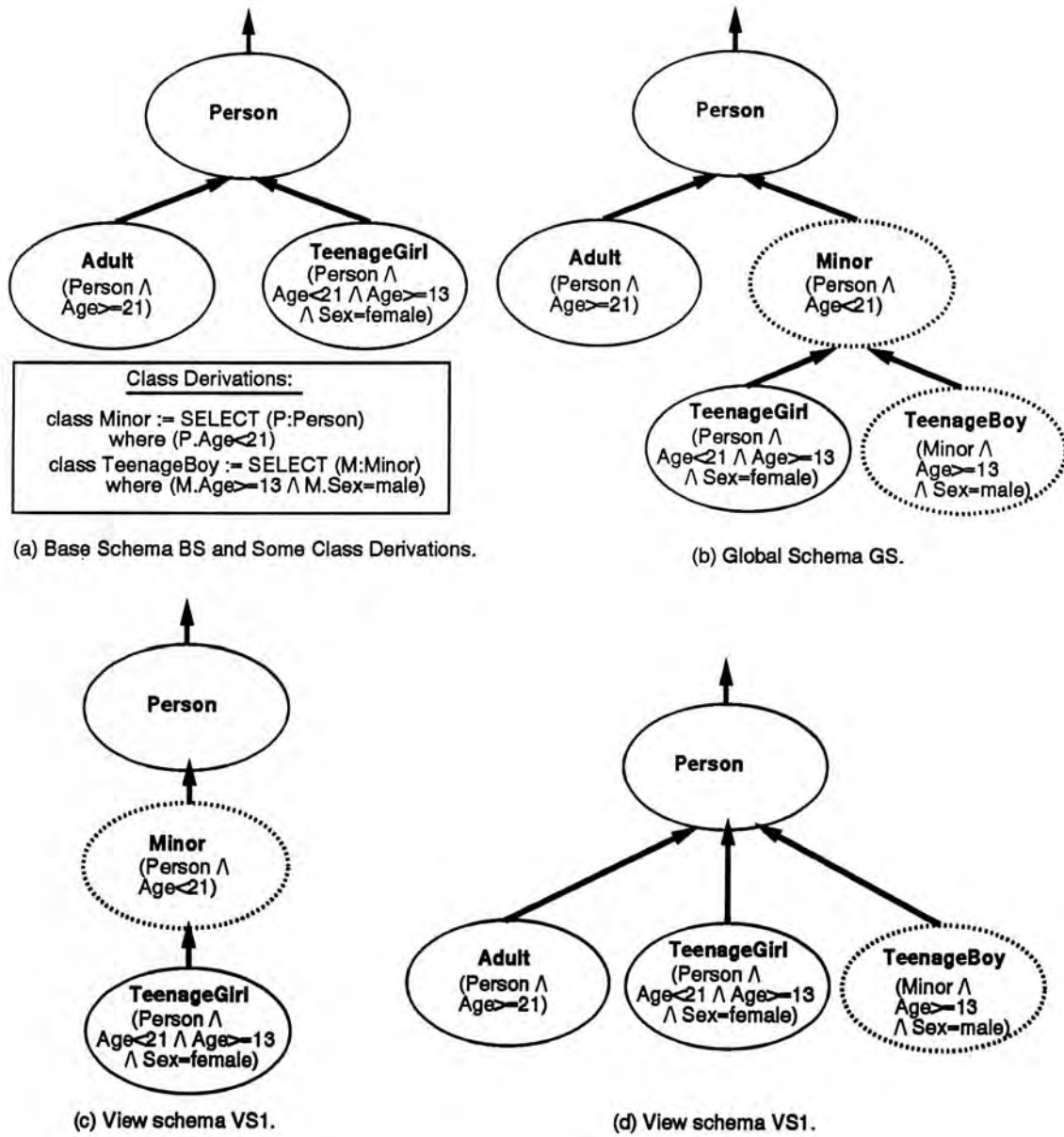


Figure 3.1: Examples of Base, Global and View Schemata.

### 3.5 The Validity of the View Generalization Hierarchy

Next, I introduce criteria that indicate whether the class generalization hierarchy of a view schema is consistent with the one of the underlying global schema.

**Definition 16.** Given a view schema  $VS=(VV,VE)$  defined on the global schema  $GS=(V,E)$ . For all classes  $C_1, C_2$  in  $VV$ , an *is-a* arc from source  $C_1$  to sink  $C_2$  is **required** in  $VS$ , if  $(C_1 \text{ is-a}^* C_2)$  in  $GS$  and there is no  $C_x$  in  $VV$  such that  $(C_1 \text{ is-a}^* C_x)$  in  $GS$  and  $(C_x \text{ is-a}^* C_2)$  in  $GS$ . The view  $VS$  is **complete**, if the set  $VE$  of all its view *is-a* relationships contains all required arcs in  $VS$ .

This defines the completeness criterion of *is-a* arcs as follows: if two classes  $C_1$  and  $C_2$  in  $VS$  are *is-a* related in  $GS$  then they also have to be *is-a* related in  $VS$ . If there is no indirect path of length greater than one between  $C_1$  and  $C_2$  in  $VS$  (such that,  $C_1 \text{ is-a}+ C_2$  in  $VS$ ), then the edge  $(C_1 \text{ is-a} C_2)$  is *required* in  $VS$ .

**Definition 17.** Given a view schema  $VS=(VV,VE)$  defined on the global schema  $GS=(V,E)$ . For all classes  $C_1, C_2$  in  $VV$ , an *is-a* arc from source  $C_1$  to sink  $C_2$  is **redundant** in  $VS$ , if there is a class  $C_x$  in  $VV$  such that  $(C_1 \text{ is-a}^* C_x)$  in  $GS$  and  $(C_x \text{ is-a}^* C_2)$  in  $GS$ . The view  $VS$  is **minimal**, if none of the view *is-a* arcs in the set  $VE$  is *redundant* in  $VS$ .

This defines the minimality criterion of *is-a* arcs as follows: if there is an indirect *is-a* path (i.e., a path of length greater or equal to two) between two classes then there should not also be a direct *is-a* arc between them. Stated differently, an *is-a* arc from source  $C_1$  to sink  $C_2$  is *redundant* in  $VS$  if there also is an *is-a* arc path of length greater or equal to two from  $C_1$  to  $C_2$  in  $VS$  (i.e.,  $C_1 \text{ is-a}+ C_2$ ).

**Definition 18.** Given a view schema  $VS=(VV,VE)$  defined on the global schema  $GS=(V,E)$ . For all classes  $C_1, C_2$  in  $VV$ , an *is-a* arc from source  $C_1$  to sink  $C_2$  is **incompatible** in  $VS$  if the edge  $\langle C_1, C_2 \rangle$  is in  $VE$  and  $\text{not}(C_1 \text{ is-a}^* C_2)$  in  $GS$ . The view  $VS$  is **consistent**, if none of its view *is-a* arcs in the set  $VE$  is *incompatible*.

This defines the consistency criterion of *is-a* arcs as follows: an *is-a* arc from source  $C_1$  to sink  $C_2$  can exist in  $VS$  if and only if the two classes are *is-a* related in  $GS$ . In other words, it indicates the following equivalence relationship for all  $C_1, C_2$  in  $VV$ :  $(C_1 \text{ is-a}^* C_2)$  in  $VS \iff (C_1 \text{ is-a}^* C_2)$  in  $GS$ .

Note that while the consistency criterion defined in Definition 18 is important to assure the correctness of the view schema, the completeness and minimality criteria are useful but not necessarily required.

**Definition 19.** A view schema  $VS=(VV, VE)$  for a given global schema  $GS=(V, E)$  is *is-a valid* (or *valid*) if the set of all view is-a relationships  $VE$  among its view classes  $VV$  is complete and minimal and consistent.

This definition states that a valid view schema  $VS=(VV, VE)$  contains *all* required is-a relationships and *no* redundant or incompatible is-a relationships.

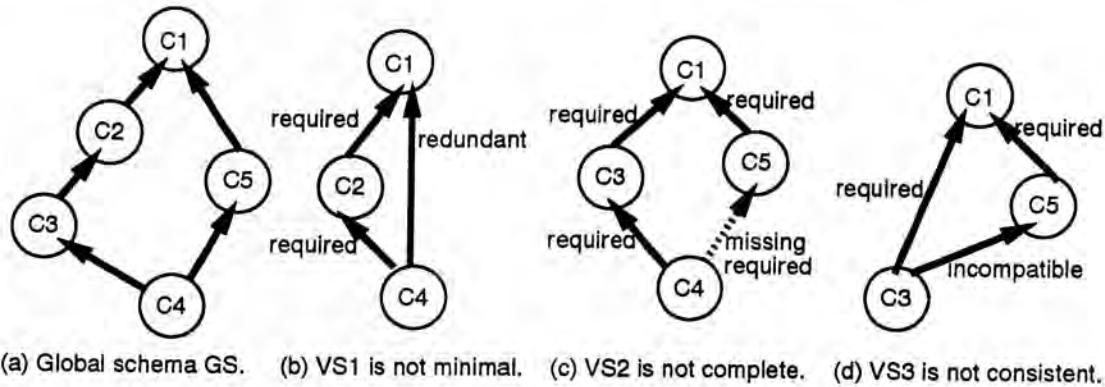


Figure 3.2: Examples of the View Schema Validity Criteria.

**Example 2.** Figures 3.2.b, 3.2.c, and 3.2.d depict three different view schemata defined on the global schema  $GS$  depicted in Figure 3.2.a. The view  $VS1$  in Figure 3.2.b is not a valid view schema, since it violates the minimality criterion. The redundant edge  $e_{4,1} = \langle C_4, C_1 \rangle$  can be removed from  $VS1$  without losing the information that ( $C_4$  is-a  $C_1$ ). The view  $VS2$  in Figure 3.2.c is also not is-a valid.  $VS2$  violates the completeness criterion, since the required edge  $e_{4,5} = \langle C_4, C_5 \rangle$  is missing. The view  $VS3$  in Figure 3.2.d is not is-a valid, since it violates the consistency criterion. The edge  $e_{3,5} = \langle C_3, C_5 \rangle$  is incompatible in  $VS$ , since the relationship ( $C_3$  is-a\*  $C_5$ ) does not hold in  $GS$ .

### 3.6 The Closure of the View Property Decomposition Hierarchy

This section addresses the consistency of the property decomposition hierarchy [Tana88, Heil90], while the consistency of the generalization hierarchy is handled in [Rund92a]. Let the function  $Uses(C)$  represent the set of classes that are used by  $C$ 's type interface. For example, if  $p$  corresponds to an object pointer defined by  $domain_p(C)=C_2$ , then  $Uses(C)$  contains  $C_2$ .

**Definition 20.** Let  $C$  be a finite set of classes,  $L$  a finite set of property labels,  $PD=(C,A,L)$  a property decomposition hierarchy. Then  $Uses:C \rightarrow 2^C$  is a function defined by: For  $C_i, C_j \in C$ , for  $pk \in L$ ,  $Uses(C_i)=\{C_j \in C | a_{ij} = \langle C_i, C_j, pk \rangle \in A\}$ . For  $S \subseteq C$ ,  $Uses(S) = \cup_{C_i \in S} Uses(C_i)$ . I define the closure operator  $*$  by  $Uses^*(C_i) = \cup_{j=1}^{|V|} Uses^j(C_i)$  with  $Uses^1(C_i) = Uses(C_i)$  and  $Uses^j(C_i) = Uses(Uses^{j-1}(C_i))$  for  $j > 1$ .

$Uses(C_i)$  ( $Uses^*(C_i)$ ) corresponds to the classes that are directly (directly or indirectly via transitive closure) used by the class  $C_i$ .

**Definition 21.** A view schema  $VS=(VV,VE)$  is defined to be a closed view if the following holds:  $VV = (\cup_{C_i \in VV} (Uses^*(C_i))) \cup VV$ .

The closure criterion assures that all classes that are being used in a view (i.e., whose class names are visible in the  $Uses^*$  set of a view class) are also defined within the view (i.e., they themselves are view classes).

**Example 3.** Figure 3.3 depicts is-a and property decomposition relationships by bold arrows without and by regular arrows with labels, respectively. A (view) schema is denoted by encircling its (view) classes by a dotted line. The views  $VS1$  and  $VS2'$  are closed. The view  $VS2 = \{\text{Statenode2}, \text{Statetrans2}\}$  is not closed, since the 'actions-in-state' property defined for the view class **Statenode2** has the domain class **Dataflow**, which is not contained in  $VS2$ .



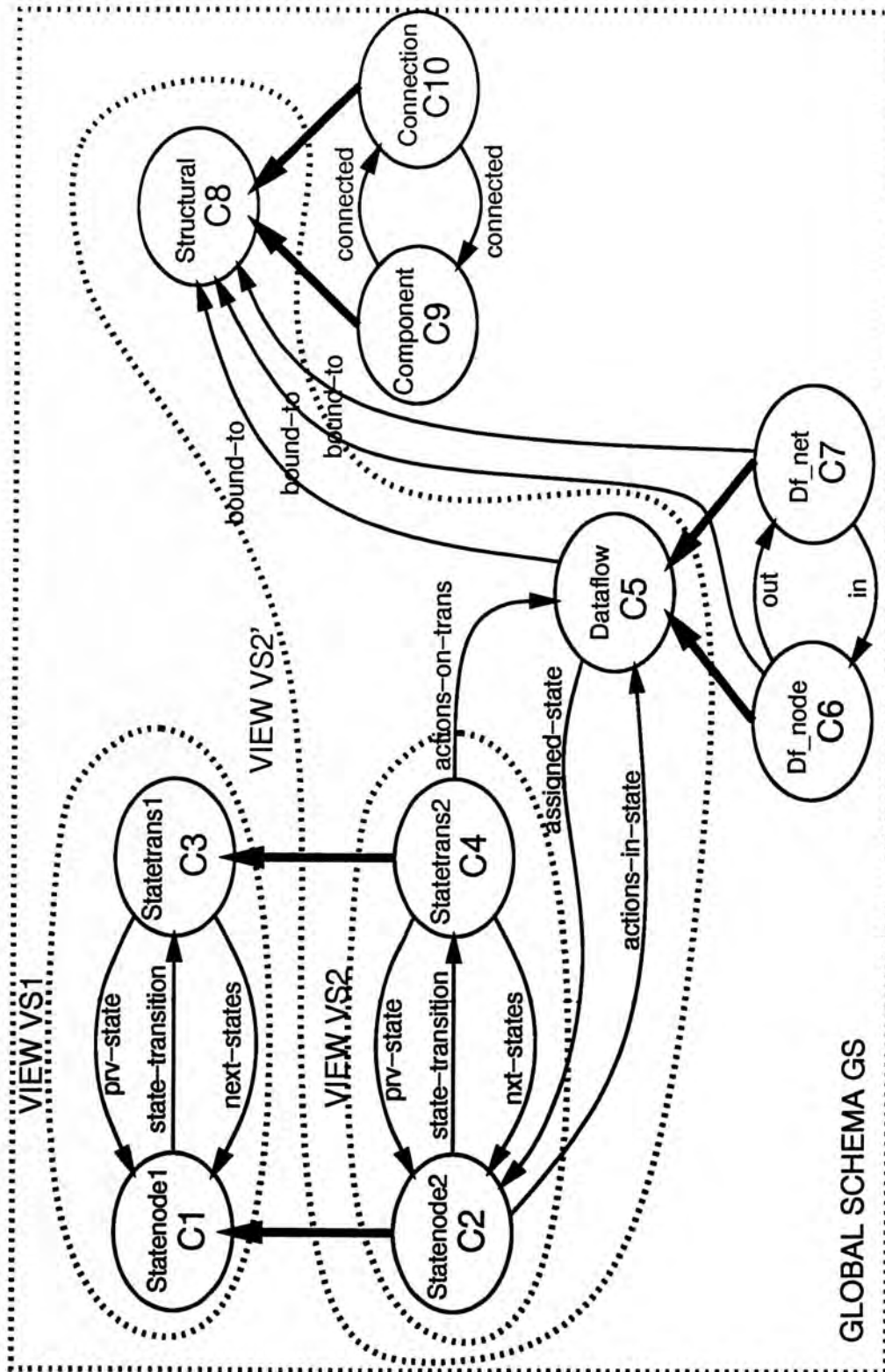


Figure 3.3: Examples of Closed and Non-Closed Views.

# Chapter 4

## The MultiView Methodology

### 4.1 The Basic Features of *MultiView*

In this chapter, I outline our approach for supporting *multiple view schemata* in OODBs, called the *MultiView* methodology. First, some of the key ideas underlying the design of *MultiView* can be summarized as follows:

1. An object-oriented view should look like a (regular) object schema, so that it can for instance be used to define other views.
2. The view specification mechanism provided to the user should be as simple as possible, so that non-database experts may be able to use the view system.
3. The user should be relieved of tedious tasks, whenever they can be automated.
4. The view system should help the user in enforcing the consistency of the view schema structure while defining the view.

*MultiView*, which as I will show later fulfills these requirements.

One important aspect of *MultiView* is that it breaks view specification into three independent tasks:

1. customization of existing type structures and object sets by deriving virtual classes via object-oriented queries,
2. integration of derived classes into *one* consistent global schema graph, and
3. the specification of arbitrarily complex, but consistent, view schemata composed of both base and virtual classes on top of this comprehensive global schema.

*MultiView's* division of view specification into a number of well-defined tasks, some of which have been successfully automated, makes it a powerful tool for supporting the specification of views by non-database experts while enforcing view

consistency. The separation of the view schema design process into a number of well-defined subtasks has several additional advantages. First, it simplifies the view specification and maintenance process, since each of the subtasks can be solved independently from the others. Second, it increases the level of schema design support that can be provided to the view definer by allowing for the automation of some of the subtasks. In Chapter 10.2, I present, for instance, algorithms that automate the second subtask of integrating derived classes into *one* consistent global schema graph. Similarly, I have proposed algorithms for the automatic generation of the view schema hierarchy. The later reduces the third subtask of view schema specification to the simple task of selecting classes to be included in the view schema. Furthermore, the integration of virtual into one global schema assures the consistency of all views with the global schema and with one another. Lastly, the definition of an arbitrary view schema on top of the augmented global schema provides the flexibility to define practically any desired view schema.

The first subtask of *MultiView* supports the virtual customization of existing classes by deriving new virtual classes with a possibly modified type description and membership extent. *MultiView* uses these class derivation mechanisms for a number of different purposes, e.g., to customize type descriptions, to limit the access to property functions, to collect object instances into groups meaningful for the task at hand, and so on. For this I assume that virtual classes are derived from the global schema using object-oriented queries. This fulfills the first feature required for a view support system listed earlier in this chapter.

While there is no generally agreed-upon object algebra, there are a number of proposals for object algebras in the literature (e.g., see [Kim89, Heil90, Scho91]). For the purpose of this work (and for the first prototype implementation of *MultiView*), I define an object algebra, which is similar in flavor to the ones proposed in the literature. (see Chapter 5). Our treatment of the object algebra focuses on the *subset*, *subtype* and *subclass* relationships among the source and result classes, since this is the foundation for successfully addressing the class integration problem. This issue is generally ignored in the literature on object algebras. I want to stress that *MultiView* is independent from the particular choice of operators.

*MultiView* supports the integration of virtual classes into one underlying global schema. This integration takes care of the maintenance of explicit relationships between stored and derived classes in terms of type inheritance and subset relationships. This is useful for sharing property functions and object instances consistently among classes without unnecessary duplication. Furthermore, this organization may result in performance increases, since the query optimizer could exploit these known relationships among classes for query processing. For instance, the union of two

classes  $C1$  and  $C2$  could be reduced to be equal to  $C1$  without actual query processing if one knew that  $C2$  is a subclass of  $C1$ . Last but not least, the integration of all virtual classes into one schema graph is a necessary basis for the third task of *MultiView*, namely, for the formation of arbitrarily complex view schema graphs composed of both base and virtual classes. If the virtual classes are not integrated with the classes in the global schema, then a view schema would correspond to a collection of possibly 'unrelated' classes rather than a generalization schema graph as defined in Definition 15. In short, the integrated global schema graph represents the backbone of our view support system based on which view schemata are being designed. Details of the class integration process are given in Chapter 7.

Class integration tackles the problem of how a virtual class (as well as a complete view schema) derived during the first step of *MultiView* relates to, and can be integrated with, the remaining classes in the global schema. Note that in the relational model, where each relation is physically independent from all other relations, the integration of a virtual relation with the global schema corresponds to simply adding it to the list of existing relations (the data dictionary). In the context of OODBs, however, this is less straightforward. A class in an object schema is interrelated with other classes via an is-a hierarchy (for property inheritance and subsetting) and via a property decomposition hierarchy (for forming complex objects). Class integration in OODBs needs to guarantee the consistency of these class relationships when adding new classes (and thus new relationships) into the schema graph.

Not just individual virtual classes but complete (possibly conflicting) view schemata have to be integrated with another and with the underlying global schema into one consistent whole. This integration has to maintain the difference in the generalization and decomposition hierarchies of the view schemata. The proposed *MultiView* methodology solves this problem by separating the definition of view schemata into two independent steps, namely, one, the integration of virtual classes into *one* consistent global schema graph and, two, the definition of view schemata composed of both base and virtual classes on top of this augmented global schema. View schemata are consistently integrated with one another simply by being consistently integrated with the same underlying global schema.

I have identified two class integration problems that existing work does not appropriately handle, which are (1) the type inheritance mismatch for virtual classes and (2) the composition of *is-a* incompatible subset and subtype hierarchies into one consistent class hierarchy. The characterization of both problems is made possible by our approach of distinguishing between the type and the set content of a class as two independent concepts [Rund92b]. The first problem is concerned with constructing

a type (and also class) hierarchy that includes the type of the new virtual class while assuring the correct type inheritance for all classes. To demonstrate this problem, I present examples for which a correct placement for a class *C* in a given schema graph *G* cannot be found. The second problem is caused by the fact that the class hierarchy combines the subset and the subtype relationships among classes into one *is-a* relationship. Differences between the subtype and the subset relationships of a class may cause problems in determining one consistent class hierarchy. For example, if a class's set content is lower in the corresponding set hierarchy and the class's type is higher in the corresponding type hierarchy, then there is a conflict in where to place the class in the combined class generalization hierarchy. In this dissertation, I present a class integration algorithm that successfully solves both problems.

Inserting arbitrary subclass relationships between classes may result in an inconsistent schema graph in terms of property inheritance and subset relationships. Therefore, rather than requiring manual placement of classes in the schema graph and then checking the entered information for consistency, *MultiView* supports the automatic integration of classes. Automatic class integration does not only prevent the introduction of inconsistencies into the schema, but it also simplifies the task of the view definer. It decreases the time needed for view specification, and, more importantly, it makes it possible for a non-database expert to specify an application-specific view on his or her own.

For these reasons, I have developed an algorithm for the automatic classification of virtual classes into a global schema graph. This algorithm solves the two problems of type inheritance and of *is-a* incompatibility. The solution is based on type lattice theory [Missi89], the essence of which is the creation of additional intermediate classes that restructure the schema graph. I present proofs of correctness and a complexity analysis for the classification algorithm. Furthermore, I characterize classification requirements of virtual classes derived by different object algebra operators. This characterization helps us to reduce the complexity of the classification task for most cases. For instance, I reduce classification from quadratic to linear complexity for classes derived by the Select, the Union, and the Difference operators and to constant complexity for those derived by the Refine operator.

The third subtask of *MultiView* utilizes this augmented global schema graph for the selection of both base and virtual classes and for arranging these view classes in a consistent class hierarchy, called a *view schema*. This phase handles all remaining requirements for a view support system listed earlier in this chapter. It supports for instance the virtual restructuring of the *is-a* hierarchy by allowing to hide from and to expose classes within a view schema. For the explicit selection of view classes from the global schema, I have developed a view schema definition language that

can be used by the view definer to specify the classes required for a particular view schema (see Chapter 8).

In addition, I present in this dissertation an algorithm for checking the closure property of a view schema graph. Given a non-closed view schema, the algorithm will automatically generate a closed view schema that contains the minimal number of view classes required to make the view closed (Chapter 10.2). Lastly note that the *is-a* relationships among the set of selected view classes of a view schema are dictated by their subset and subtype relationships as defined in Chapter 3. Inserting arbitrary *is-a* relationships between classes in a view schema may result in an incorrect schema in terms of property inheritance and subset relationships. Therefore, rather than requiring the manual insertion of *view is-a* arcs by the view definer, I have developed algorithms that automatically augment the set of selected view classes to generate a *valid* view schema class hierarchy [Rund92a].

To make the presented ideas more concrete I now give an example of the steps involved in constructing a *view schema* in *MultiView*.

**Example 4.** *This example of the view schema construction process is based on Figure 4.1. In this figure I depict base and virtual classes by circles and dotted circles, respectively. Given the global schema GS in Figure 4.1.a, the view definer first specifies the two virtual classes VC4 and VC5 using object-oriented queries (Figure 4.1.b). Class VC4, for instance, is derived based on the two source classes C1 and C3 as depicted by the dotted arrows pointing from Figure 4.1.a to Figure 4.1.b. The integration of the virtual classes VC4 and VC5 into GS is given in Figure 4.1.c. View schema definition now proceeds by selecting a subset of classes from the augmented schema GS. As depicted in Figure 4.1.d, the selected view classes can be both base and virtual classes. Lastly, the chosen view classes are interconnected into one schema graph. The resulting virtual schema graph, called a view schema, is given in Figure 4.1.e.*

## 4.2 Realization of *MultiView*

While steps one and two of *MultiView* (Figures 4.1.b and 4.1.c) are real in as much as they actually modify the underlying global schema, steps three and four (Figures 4.1.d and 4.1.e) are virtual since they leave the underlying global schema intact (Section 4). This is supported by maintaining information on view schemata, such as their view classes and their *view is-a* relationships, in separate view object tables as described in this section. There are two equivalent ways in

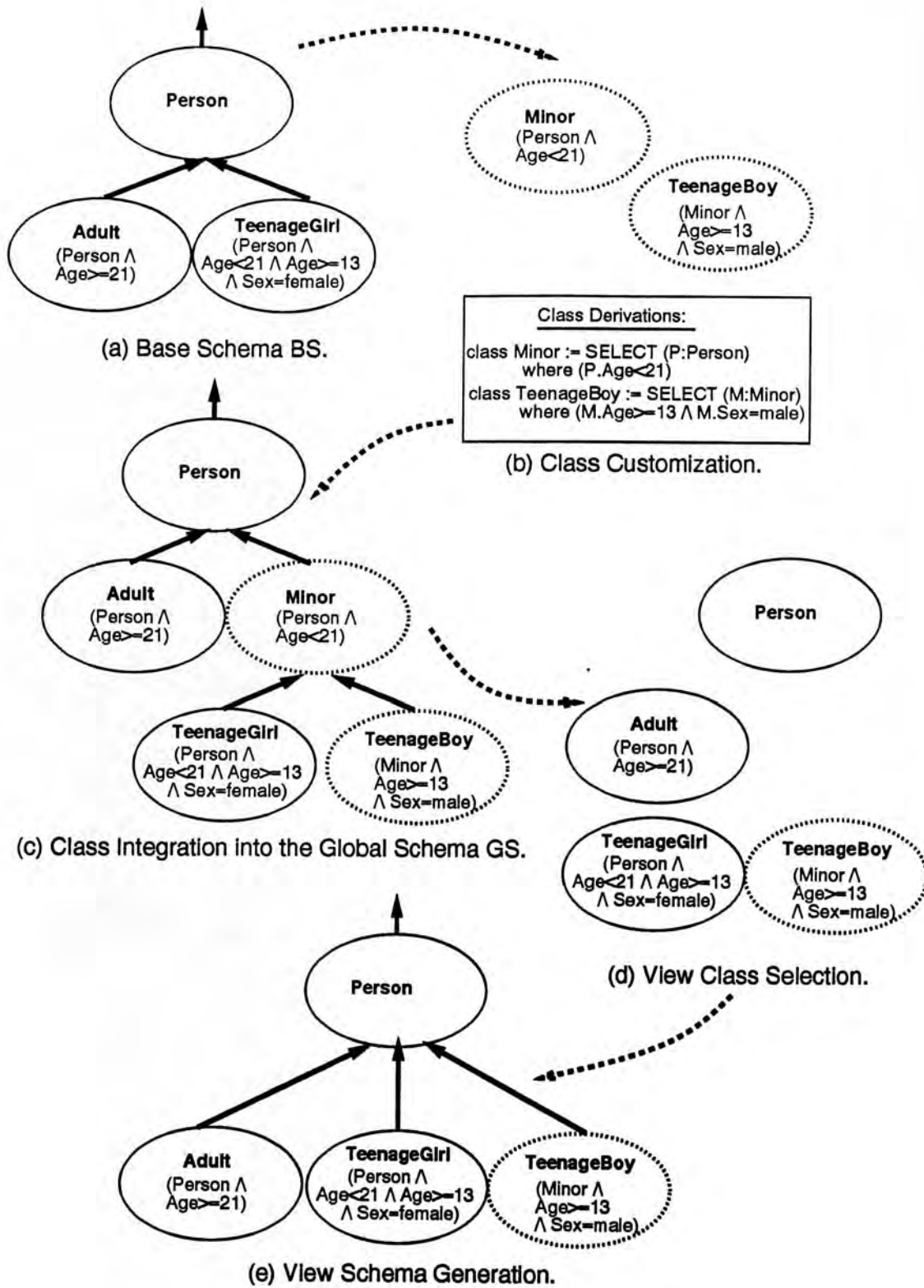


Figure 4.1: An Example of the *MultiView* Approach.

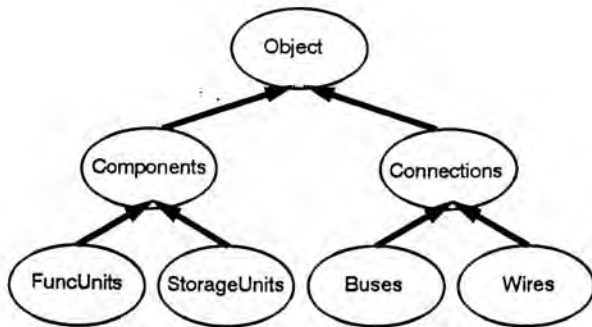
which to maintain the information about multiple view schemata: a centralized or a distributed approach. An example of these two approaches is given in Figure 4.2.b and 4.2.c, respectively, while Figure 4.2.a depicts the schemata in the graphical form used throughout the dissertation.

The centralized approach (Figure 4.2.b) maintains one *view schema table* for each view schema (similar to a data dictionary in the relational model). The *base schema table* and the *global schema table* are two special schema tables kept for the base and the global schema, respectively. Such a view schema table contains the following information: a list of class names of classes that belong to the schema, their internal class identifiers, and their direct sub- and superclasses. Figure 4.2.b, for instance, shows how the schemata of Figure 4.2.a are captured by the centralized approach. There are three view (schema) tables, one for the global schema and one for each view schema. The view table for View1, for instance, has been assigned the view identifier  $\langle VS1 \rangle$ . The table enumerates all classes that belong to View1, which are Objects, Components, FuncUnits, and StorageUnits, as well as their direct sub- and superclasses in the view.

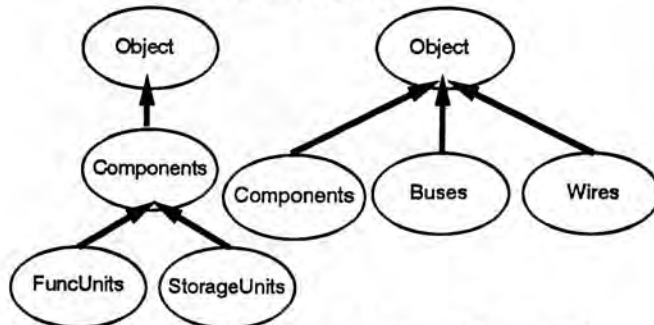
The second approach distributes the above described schema information across the classes of the global schema. More precisely, each class of the global schema would be extended with a list of view identifiers of the view schemata to which that class belongs. For each such view identifier, it would enumerate the direct sub- and superclasses visible within the respective view. Figure 4.2.c, for instance, shows how the schemata of Figure 4.2.a are captured by the distributed approach. In Figure 4.2.c, for instance, the Components class belongs to both views View1 and View2. Hence, the class definition of the Components class lists their view identifiers  $\langle VS1 \rangle$  and  $\langle VS2 \rangle$ . It also lists sub- and superclasses of the Component class for each view. For instance, the Components class has two *is-a* related subclasses for View1 and none for View2.

I have chosen the centralized approach over the distributed one for the following reasons. First, this does not require any extension of the class concept. Hence, I can directly use existing object-oriented database technology for an implementation of *MultiView*. Also, no modification of a class description is required due to the insertion of the class into and deletion of the class from of a view schema. In addition, an operation to remove an obsolete view schemata or to copy one view schemata into another view schemata could easily be accomplished by manipulating the class dictionaries used in the centralized approach (without requiring any search through all the classes distributed throughout the global schema).





a.1. Global Schema GS.



a.2. View Schema <VS1>. a.3. View Schema <VS2>. a. Graphical Representation of Global and View Schemata.

Global	<GS>	
Classes	Superclasses	Subclasses
Object	—	Components Connections
Components	Object	FuncUnits StorageUnits
FuncUnits	Components	—
StorageUnits	Components	—
Connections	Object	Buses Wires
Buses	Connections	—
Wires	Connections	—

View1	<VS1>	
Classes	Superclasses	Subclasses
Object	—	Components
Components	Object	FuncUnits StorageUnits
FuncUnits	Components	—
StorageUnits	Components	—

View2	<VS2>	
Classes	Superclasses	Subclasses
Object	—	Components Buses Wires
Components	Object	—
Buses	Object	—
Wires	Object	—

b. Centralized Approach: Schema Representation using a separate View Table for each Schema.

**Class: Object**  
**GlobalSchema (<GS>):**  
 Superclasses: —  
 Subclasses: Components, Connections  
**View1 (<VS1>):**  
 Superclasses: —  
 Subclasses: Components  
**View2 (<VS2>):**  
 Superclasses: —  
 Subclasses: Components, Buses, Wires

**Class: Components**  
**GlobalSchema (<GS>):**  
 Superclasses: Object  
 Subclasses: FuncUnits StorageUnits.  
**View1 (<VS1>):**  
 Superclasses: Object  
 Subclasses: FuncUnits StorageUnits  
**View2 (<VS2>):**  
 Superclasses: Object  
 Subclasses: none

**Class: FuncUnits**  
**GlobalSchema (<GS>):**  
 Superclasses: Components  
 Subclasses: none  
**View1 (<VS1>):**  
 Superclasses: Components  
 Subclasses: none

**Class: StorageUnits**  
**GlobalSchema (<GS>):**  
 Superclasses: Components  
 Subclasses: none  
**View1 (<VS1>):**  
 Superclasses: Components  
 Subclasses: none

**Class: Connections**  
**GlobalSchema (<GS>):**  
 Superclasses: Object  
 Subclasses: Buses, Wires

**Class: Buses**  
**GlobalSchema (<GS>):**  
 Superclasses: Connections  
 Subclasses: none  
**View2 (<VS2>):**  
 Superclasses: Object  
 Subclasses: none

**Class: Wires**  
**GlobalSchema (<GS>):**  
 Superclasses: Connections  
 Subclasses: none  
**View2 (<VS2>):**  
 Superclasses: Object  
 Subclasses: none

c. Distributed Approach: Distributing Schema Information across Classes.

Figure 4.2: Centralized versus Distributed Realization of *MultiView*.

# Chapter 5

## Class Customization Using Object Algebra

### 5.1 Dual Aspects of a Class: Types and Sets

The *MultiView* methodology is independent from the particular object algebra operators chosen for the class derivation subtask. However, since there is no agreed-upon standard for object algebra, I present below a representative set of algebra operators. The result of a class derivation using an algebra operator is a virtual class VC that has a possibly derived type description and a derived membership extent. I have shown the distinction between the type and the set aspect of classes to be a valuable tool for characterizing the semantics of query operators on object-based data models [Rund92b]. As defined in Section 3, a *type* description of a class determines which property functions can be used to access the instances associated with the class. A *set* aspect of a class refers to the set of objects that are members of this class. In the section, I describe the semantics of the object algebra operators by defining their effect on the type and the set aspect of the resulting virtual class.

I distinguish between *type* and *set manipulating* query operators. *Type manipulating* operators restrict or elaborate on the type description of an existing class and determine which property functions can be applied to the set of objects associated with the class. They thus limit the visibility of property functions and the access rights to the underlying object instances. A typical example is the **hide** operator, which is similar to the **project** operator used in relational algebra. *Set manipulating* operators group sets of objects into smaller constrained sets or combine several sets into larger sets of objects. A typical example is the **select** operator, also called a predicate-based query [Kim89], which is similar to the **selection** operator used in relational algebra. Some operators manipulate both the type and the set properties of classes. The set operators, such as **union**, **intersection**, and **difference**, fall into this category.

Note that the derivation relationship between the source classes (arguments to a query operator) and the derived class (the result class of the query operation) do not necessarily correspond to *is-a* relationships between these classes. As I will demonstrate in this section, the resulting class relationships vary with the type of the query operator. The determination of these *subclass* relationships is a necessary basis for the integration of virtual classes into the global schema and thus is generally not covered in the literature.

The table in Figure 5.1 summarizes the object algebra operators, in particular, it gives their syntax, semantics and the resulting class relationships. The proposed object algebra consists of the following six operators, **hide**, **refine**, **select**, **union**, **intersect**, and **diff**. A more detailed description of the operators and examples follow (See also [Rund92d]).

## 5.2 The Object Algebra Operators

### 5.2.1 The Hide Operator

The **hide** operator modifies the type description of a class by hiding some of its property functions. It is similar to the **project** operator in the classical relational algebra, which projects some columns from a relation. It has the following syntax:

$$\langle \text{virtual-class} \rangle := \text{hide} [\langle \text{prop-functions} \rangle] \text{ from } (\langle \text{source-class} \rangle);$$

with  $\langle \text{prop-functions} \rangle$  being one or more property functions defined for the class  $\langle \text{source-class} \rangle$ . The semantics of the **hide** operator are to remove the property functions listed in the set  $\langle \text{prop-functions} \rangle$  from the source class while preserving all other property functions visible in the class. More formally stated,

$$\text{type}(\langle \text{virtual-class} \rangle) := \{p \in P \mid p \in \text{properties}(\langle \text{source-class} \rangle) \wedge p \notin \langle \text{prop-functions} \rangle\},$$

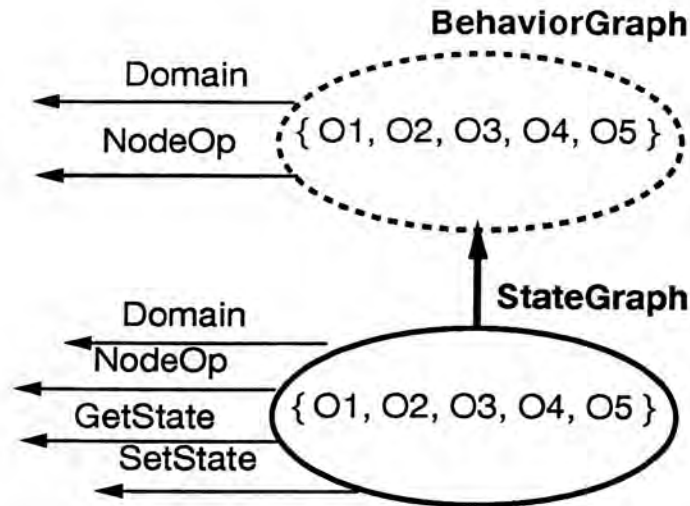
By Definition 6, the type of the  $\langle \text{virtual-class} \rangle$  is a supertype of the type of the  $\langle \text{source-class} \rangle$ , since some of the property functions defined for the  $\langle \text{source-class} \rangle$  are not defined for the  $\langle \text{virtual-class} \rangle$ . This is denoted by  $\langle \text{source-class} \rangle \preceq \langle \text{virtual-class} \rangle$ . The extent of the result class is equal to the extent of the source class, denoted by

$$\text{extent}(\langle \text{virtual-class} \rangle) := \text{extent}(\langle \text{source-class} \rangle).$$

<b>hide</b>	syntax	$\langle \text{virt-class} \rangle := \text{hide } \langle \text{prop-fcts} \rangle \text{ from } \langle \text{src-class} \rangle$
	semantics	$\text{type}(\langle \text{virt-class} \rangle) := \{p \in P \mid p \in \text{prop}(\langle \text{src-class} \rangle) \wedge p \notin \langle \text{prop-fcts} \rangle\}$ $\text{extent}(\langle \text{virt-class} \rangle) := \text{extent}(\langle \text{src-class} \rangle)$
	class rels	$\langle \text{src-class} \rangle \preceq \langle \text{virt-class} \rangle$ $\langle \text{src-class} \rangle \subseteq \langle \text{virt-class} \rangle$ $\langle \text{src-class} \rangle \text{ is-a } \langle \text{virt-class} \rangle$
<b>refine</b>	syntax	$\langle \text{virt-class} \rangle := \text{refine } \langle \text{fct-defs} \rangle \text{ for } \langle \text{src-class} \rangle$
	semantics	$\text{type}(\langle \text{virt-class} \rangle) := \{p \in P \mid p \in \text{prop}(\langle \text{src-class} \rangle) \vee p \in \langle \text{fct-defs} \rangle\}$ $\text{extent}(\langle \text{virt-class} \rangle) := \text{extent}(\langle \text{src-class} \rangle)$
	class rels	$\langle \text{virt-class} \rangle \preceq \langle \text{src-class} \rangle$ $\langle \text{virt-class} \rangle \subseteq \langle \text{src-class} \rangle$ $\langle \text{virt-class} \rangle \text{ is-a } \langle \text{src-class} \rangle$
<b>select</b>	syntax	$\langle \text{virt-class} \rangle := \text{select from } \langle \text{src-class} \rangle \text{ where } \langle \text{predicate} \rangle$
	semantics	$\text{type}(\langle \text{virt-class} \rangle) := \text{type}(\langle \text{src-class} \rangle)$ $\text{extent}(\langle \text{virt-class} \rangle) := \{o \in O \mid o \in \langle \text{src-class} \rangle \wedge \langle \text{predicate} \rangle(o)\}$
	class rels	$\langle \text{virt-class} \rangle \preceq \langle \text{src-class} \rangle$ $\langle \text{virt-class} \rangle \subseteq \langle \text{src-class} \rangle$ $\langle \text{virt-class} \rangle \text{ is-a } \langle \text{src-class} \rangle$
<b>union</b>	syntax	$\langle \text{virt-class} \rangle := \text{union}(\langle \text{src-class1} \rangle, \langle \text{src-class2} \rangle)$
	semantics	$\text{type}(\langle \text{virt-class} \rangle) := \text{type}(\langle \text{src-class1} \rangle) \sqcup \text{type}(\langle \text{src-class2} \rangle)$ $\text{extent}(\langle \text{virt-class} \rangle) := \{o \in O \mid o \in \langle \text{src-class1} \rangle \vee o \in \langle \text{src-class2} \rangle\}$
	class rels	$\langle \text{src-class1} \rangle \preceq \langle \text{virt-class} \rangle \wedge \langle \text{src-class2} \rangle \preceq \langle \text{virt-class} \rangle$ $\langle \text{src-class1} \rangle \subseteq \langle \text{virt-class} \rangle \wedge \langle \text{src-class2} \rangle \subseteq \langle \text{virt-class} \rangle$ $\langle \text{src-class1} \rangle \text{ is-a } \langle \text{virt-class} \rangle \wedge \langle \text{src-class2} \rangle \text{ is-a } \langle \text{virt-class} \rangle$
<b>intersect</b>	syntax	$\langle \text{virt-class} \rangle := \text{intersect}(\langle \text{src-class1} \rangle, \langle \text{src-class2} \rangle)$
	semantics	$\text{type}(\langle \text{virt-class} \rangle) := \text{type}(\langle \text{src-class1} \rangle) \sqcap \text{type}(\langle \text{src-class2} \rangle)$ $\text{extent}(\langle \text{virt-class} \rangle) := \{o \in O \mid o \in \langle \text{src-class1} \rangle \wedge o \in \langle \text{src-class2} \rangle\}$
	class rels	$\langle \text{virt-class} \rangle \preceq \langle \text{src-class1} \rangle \wedge \langle \text{virt-class} \rangle \preceq \langle \text{src-class2} \rangle$ $\langle \text{virt-class} \rangle \subseteq \langle \text{src-class1} \rangle \wedge \langle \text{virt-class} \rangle \subseteq \langle \text{src-class2} \rangle$ $\langle \text{virt-class} \rangle \text{ is-a } \langle \text{src-class1} \rangle \wedge \langle \text{virt-class} \rangle \text{ is-a } \langle \text{src-class2} \rangle$
<b>diff</b>	syntax	$\langle \text{virt-class} \rangle := \text{diff}(\langle \text{src-class1} \rangle, \langle \text{src-class2} \rangle)$
	semantics	$\text{type}(\langle \text{virt-class} \rangle) := \text{type}(\langle \text{src-class1} \rangle)$ $\text{extent}(\langle \text{virt-class} \rangle) := \{o \in O \mid o \in \langle \text{src-class1} \rangle \wedge o \notin \langle \text{src-class2} \rangle\}$
	class rels	$\langle \text{virt-class} \rangle \preceq \langle \text{src-class1} \rangle$ $\langle \text{virt-class} \rangle \subseteq \langle \text{src-class1} \rangle$ $\langle \text{virt-class} \rangle \text{ is-a } \langle \text{src-class1} \rangle$

Figure 5.1: Syntax, Semantics and Class Relationships for the Object Algebra Operators.

By default, this means that  $\langle \text{source-class} \rangle \subseteq \langle \text{virtual-class} \rangle$ . By Definition 7, this implies that the  $\langle \text{source-class} \rangle$  is a subclass of the  $\langle \text{virtual-class} \rangle$ , denoted by  $\langle \text{source-class} \rangle$  is-a  $\langle \text{virtual-class} \rangle$ .



**BehaviorGraph = HIDE [ SetState, GetState ] from StateGraph;**

Figure 5.2: An Example of the hide Operator.

**Example 5.** In Figure 5.2, the query “**BehaviorGraph = hide [SetState, GetState] from (StateGraph)**” is used to derive the virtual class **BehaviorGraph** from the source class **StateGraph**. Then  $\text{extent}(\text{BehaviorGraph}) = \text{extent}(\text{StateGraph})$ , i.e.,  $\text{StateGraph} \subseteq \text{BehaviorGraph}$ . Also  $\text{type}(\text{StateGraph}) = [ \text{Domain}, \text{NodeOp}, \text{SetState}, \text{GetState} ]$  and  $\text{type}(\text{BehaviorGraph}) = [ \text{Domain}, \text{NodeOp} ]$  imply  $\text{StateGraph} \preceq \text{BehaviorGraph}$ . The is-a relationship (**StateGraph** is-a **BehaviorGraph**) is indicated by the edge from **StateGraph** to **BehaviorGraph**.

Note that if the property function  $a_1$  is not directly defined for the class  $C_1$  but inherited from another class, then this integration of the virtual class  $C_1'$  into the global schema may create other intermediate classes up to the class where the attribute  $a_1$  has been originally defined. A discussion of this can be found in a later section of this dissertation.

### 5.2.2 The Refine Operator

The **refine** operator is a type manipulating operator that adds additional property functions to a type rather than removing existing ones. It is similar in flavor

to calculating a derived value for each tuple of a relation and then adding (joining) this derived value to the relation in the form of an additional column. It has the following syntax:

$$\langle \text{virtual-class} \rangle := \text{refine} [\langle \text{prop-function-defs} \rangle] \text{ for } (\langle \text{source-class} \rangle);$$

with  $\langle \text{prop-function-def} \rangle$  being the definition of a new property function in form of a new property name and a function body with the latter a legal arithmetic, boolean or set expression. For instance, the expression “age := today-date - birth-date” is an example of a legal  $\langle \text{prop-function-defs} \rangle$ . The property functions in  $\langle \text{prop-function-defs} \rangle$  are assumed to be distinct from all other property functions in the global schema; and I therefore associate a unique property identifier with them. The semantics of the **refine** operator are to refine the type description of the source class by adding one or more new derived attributes to the source class, namely, those listed in  $\langle \text{prop-function-defs} \rangle$ . All other attributes visible in the source class are preserved. More formally stated,

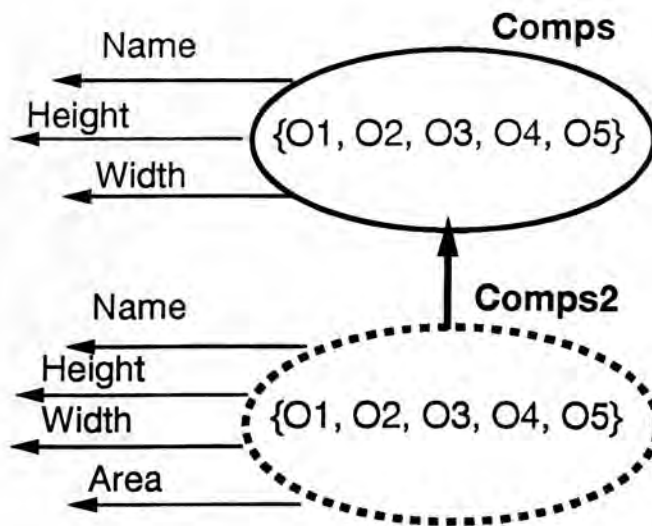
$$\text{type}(\langle \text{virtual-class} \rangle) := \{p \in P \mid p \in \text{properties}(\langle \text{source-class} \rangle) \vee p \in \langle \text{prop-function-def} \rangle\}.$$

The type of the  $\langle \text{virtual-class} \rangle$  is a subtype of the type of the  $\langle \text{source-class} \rangle$ , since all of the property functions defined for the  $\langle \text{source-class} \rangle$  are also defined for the  $\langle \text{virtual-class} \rangle$  (Definition 6), i.e.,  $\langle \text{virtual-class} \rangle \preceq \langle \text{source-class} \rangle$ . The content of the result class is again equal to the content of the source class,

$$\text{extent}(\langle \text{virtual-class} \rangle) := \text{extent}(\langle \text{source-class} \rangle).$$

By default,  $\langle \text{virtual-class} \rangle \subseteq \langle \text{source-class} \rangle$ . By Definition 7, this implies  $\langle \text{virtual-class} \rangle$  *is-a*  $\langle \text{source-class} \rangle$ .

**Example 6.** In Figure 5.3, the **refine** operator is used in the following query to derive **Comps2** from **Comps**: **Comps2** = **refine** [*Area* = *Height* \* *Width*] **for** (**Comps**). I have  $\text{extent}(\text{Comps2}) = \text{extent}(\text{Comps})$ , i.e.,  $\text{Comps2} \subseteq \text{Comps}$ . The type of **Comps2** has been extended by the new method *Area*. Hence  $\text{type}(\text{Comps}) = [\text{Name}, \text{Height}, \text{Width}]$  and  $\text{type}(\text{Comps2}) = [\text{Name}, \text{Height}, \text{Width}, \text{Area}]$ , which together imply  $\text{Comps2} \preceq \text{Comps}$ . **Comps2** is integrated into the global schema by placing **Comps2** below **Comps** as direct subclass.



**Comps2 = REFINE (Comps) by [Area := Height \* Width].**

Figure 5.3: An Example of the refine Operator.

### 5.2.3 The Select Operator

The **Select** operator is a set manipulating operator that selects a subset of object instances from a given set of objects – similar to the selection operator defined for relational algebra [Date90]. The select operator has the following syntax:

**<virtual-class> := select from (<source-class>) where (<predicate>);**

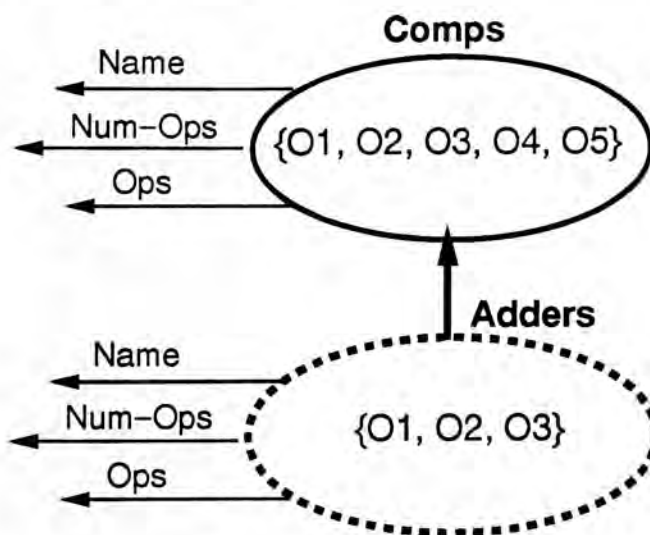
with the <predicate> being some possibly complex function on the source class and its type description. Its semantics are to return a subset of object instances of the source class based on the evaluation of the associated predicate, namely, all object instances that satisfy the predicate are collected into the virtual class. More formally stated,

**extent(<virtual-class>) := {o ∈ O | o ∈ <source-class> ∧ <predicate>(o) = true}.**

The extent of the virtual class derived by selection thus is a subset of the extent of the source class, denoted by  $\langle \text{virtual-class} \rangle \subseteq \langle \text{source-class} \rangle$ . Furthermore, the type description defined for the derived class is unchanged, i.e., I have

**type(<virtual-class>) := type(<source-class>).**

By default,  $\langle virtual-class \rangle \preceq \langle source-class \rangle$ . By Definition 7, this implies  $\langle virtual-class \rangle$  is-a  $\langle source-class \rangle$ .



**Adders = SELECT (Comps) where (Plus in Comps.Ops);**

Figure 5.4: An Example of the select Operator.

**Example 7.** In Figure 5.4, the select operator is used in the query “Adders = select from (Comps) where (Plus in Comps.Ops)” to derive Adders from Comps. The Adders class consists of a selected subset of object members from the Comps class, namely, all components that implement the Plus operator. Thus  $\text{Adders} \subseteq \text{Comps}$ . Also  $\text{type}(\text{Adders}) = \text{type}(\text{Comps})$ . The is-a relationship (Adders is-a Comps) has been added as indicated by the edge from Adders to Comps.

## 5.2.4 The Union Operator

Set operators, like the union operator, manipulate both the type description and the set membership of their two source classes. A detailed analysis of these set operators for OODBs can be found in [Rund92b]. For the purpose of this work, I use simple semantics (rather than utilizing a more flexible scheme for property inheritance proposed in [Rund92b]). The union operator has the following syntax:

$\langle virtual-class \rangle := \text{union}(\langle source-class1 \rangle, \langle source-class2 \rangle)$ .



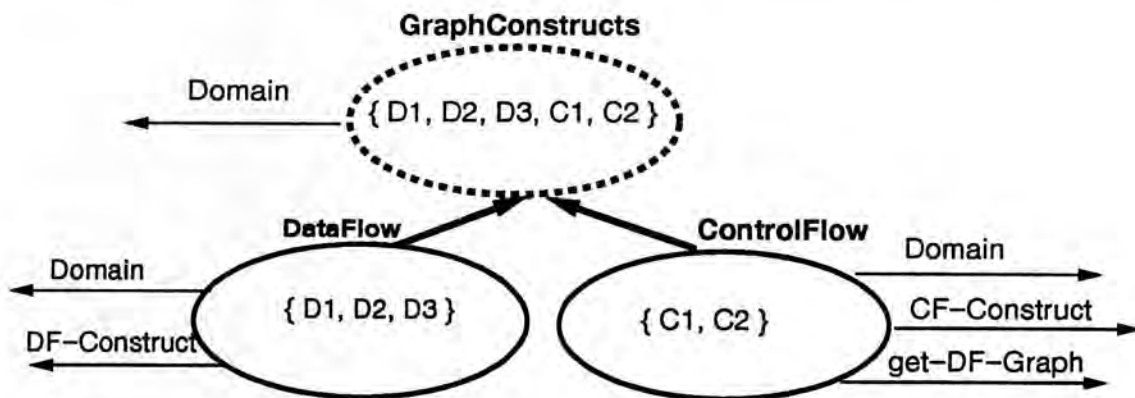
Its semantics are to return a set of object instances composed of the members of both source classes. More formally stated,

$$\text{extent}(\langle \text{virtual-class} \rangle) := \{o \in O \mid o \in \langle \text{source-class1} \rangle \vee o \in \langle \text{source-class2} \rangle\}.$$

The semantics of the union operator imply the following subset relationships  $\langle \text{source-class1} \rangle \subseteq \langle \text{virtual-class} \rangle$  and  $\langle \text{source-class2} \rangle \subseteq \langle \text{virtual-class} \rangle$ . Furthermore, the type description of the virtual class is equal to the lowest common super-type of the two source classes as defined in Definition 4. This is denoted by

$$\text{type}(\langle \text{virtual-class} \rangle) := \text{type}(\langle \text{source-class1} \rangle) \sqcap \text{type}(\langle \text{source-class2} \rangle).$$

This implies the following two subtype relationships  $\langle \text{source-class1} \rangle \preceq \langle \text{virtual-class} \rangle$  and  $\langle \text{source-class2} \rangle \preceq \langle \text{virtual-class} \rangle$ . This then implies the subclass relationships  $\langle \text{source-class1} \rangle \text{ is-a } \langle \text{virtual-class} \rangle$  and  $\langle \text{source-class2} \rangle \text{ is-a } \langle \text{virtual-class} \rangle$ .



$\text{GraphConstructs} = \text{UNION}(\text{DataFlow}, \text{ControlFlow})$

Figure 5.5: An Example of the union Operator.

**Example 8.** In Figure 5.5, the union operator derives the virtual class **GraphConstructs** from the source classes **DataFlow** and **ControlFlow**. This is specified using the query “**GraphConstructs = union(DataFlow,ControlFlow)**”. Then  $\text{extent}(\text{GraphConstructs}) = \text{extent}(\text{DataFlow}) \cup \text{extent}(\text{ControlFlow}) = \{D1, D2, D3\} \cup \{C1, C2\} = \{D1, D2, D3, C1, C2\}$ . Therefore we have  $\text{DataFlow} \subseteq \text{GraphConstructs}$  and  $\text{ControlFlow} \subseteq \text{GraphConstructs}$ . Also we have  $\text{type}(\text{GraphConstructs}) = \text{type}(\text{DataFlow}) \sqcap \text{type}(\text{ControlFlow}) = [ \text{Domain}, \text{DF-Construct} ] \sqcap [ \text{Domain}, \text{CF-Construct}, \text{get-DF-Graph} ] = [ \text{Domain} ]$ . Hence  $\text{DataFlow} \preceq \text{GraphConstructs}$  and  $\text{ControlFlow} \preceq \text{GraphConstructs}$ . The

*is-a relationships (DataFlow is-a GraphConstructs) and (ControlFlow is-a GraphConstructs) are indicated by edges from DataFlow to GraphConstructs and from ControlFlow to GraphConstructs, respectively.*

### 5.2.5 The Intersection Operator

The **intersect** operator manipulates both the type description and the set membership of the two source classes. It has the following syntax:

$$\langle \text{virtual-class} \rangle := \text{intersect}(\langle \text{source-class1} \rangle, \langle \text{source-class2} \rangle).$$

Its semantics are to return a set of object instances that are members of both source classes. More formally stated,

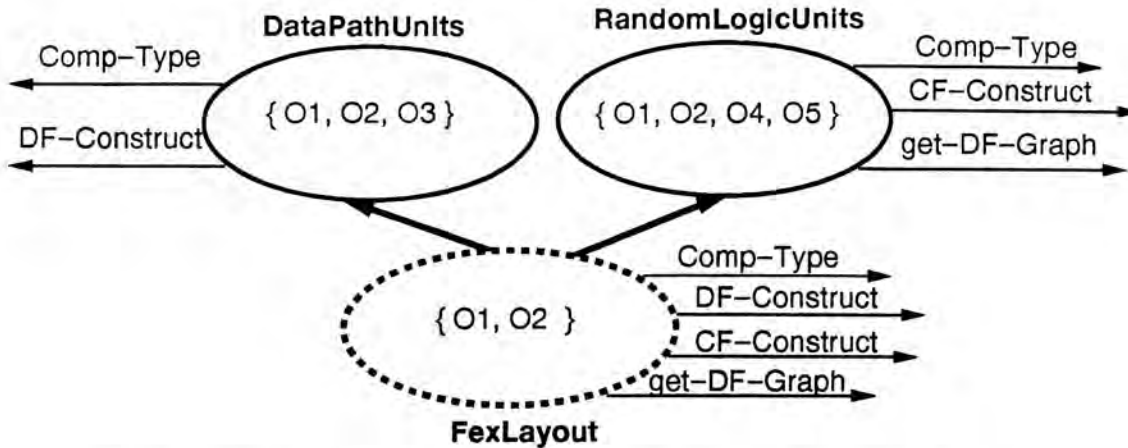
$$\text{extent}(\langle \text{virtual-class} \rangle) := \{o \in O \mid o \in \langle \text{source-class1} \rangle \wedge o \in \langle \text{source-class2} \rangle\}.$$

The semantics of the intersection operator imply the following subset relationships,  $\langle \text{virtual-class} \rangle \subseteq \langle \text{source-class1} \rangle$  and  $\langle \text{virtual-class} \rangle \subseteq \langle \text{source-class2} \rangle$ . Furthermore, the type description of the virtual class is equal to the greatest common subtype of the two sources classes as defined in Definition 3. This is denoted by

$$\text{type}(\langle \text{virtual-class} \rangle) := \text{type}(\langle \text{source-class1} \rangle) \sqcup \text{type}(\langle \text{source-class2} \rangle).$$

This implies the subtype relationships  $\langle \text{virtual-class} \rangle \preceq \langle \text{source-class1} \rangle$  and  $\langle \text{virtual-class} \rangle \preceq \langle \text{source-class2} \rangle$ , and finally also the subclass relationships  $\langle \text{virtual-class} \rangle \text{ is-a } \langle \text{source-class1} \rangle$  and  $\langle \text{virtual-class} \rangle \text{ is-a } \langle \text{source-class2} \rangle$ .

**Example 9.** In Figure 5.6, the **intersect** operator is used to derive the virtual class **FexLayout** from **DataPathUnits** and **RandomLogicUnits** using the query  $\text{FexLayout} = \text{intersect}(\text{DataPathUnits}, \text{RandomLogicUnits})$ . Then  $\text{extent}(\text{FexLayout}) = \text{extent}(\text{DataPathUnits}) \cap \text{extent}(\text{RandomLogicUnits}) = \{O1, O2, O3\} \cap \{O1, O2, O4, O5\} = \{O1, O2\}$ . Therefore the following relationships  $\text{FexLayout} \subseteq \text{DataPathUnits}$  and  $\text{FexLayout} \subseteq \text{RandomLogicUnits}$  hold.  $\text{type}(\text{FexLayout}) = \text{type}(\text{DataPathUnits}) \sqcup \text{type}(\text{RandomLogicUnits}) = [ \text{Comp-Type}, \text{DF-Construct} ] \sqcup [ \text{Comp-Type}, \text{CF-Construct}, \text{get-DF-Graph} ] = [ \text{Comp-Type}, \text{DF-Construct}, \text{CF-Construct}, \text{get-DF-Graph} ]$ . Thus **FexLayout**



$$\text{FexLayout} = \text{INTERSECT}(\text{DataPathUnits}, \text{RandomLogicUnits})$$

Figure 5.6: An Example of the intersect Operator.

$\preceq$  **DataPathUnits** and **FexLayout**  $\preceq$  **RandomLogicUnits**. The is-a relationships (**FexLayout** is-a **DataPathUnits**) and (**FexLayout** is-a **RandomLogicUnits**) are indicated by the edges from **FexLayout** to **DataPathUnits** and from **FexLayout** to **RandomLogicUnits**, respectively.

### 5.2.6 The Difference Operator

The **difference** operator has the following syntax:

$\langle \text{virtual-class} \rangle := \text{diff}(\langle \text{source-class1} \rangle, \langle \text{source-class2} \rangle)$ .

Its semantics are to return a set of object instances that are members of the first but not of the second source class. More formally stated,

$\text{extent}(\langle \text{virtual-class} \rangle) := \{o \in O \mid o \in \langle \text{source-class1} \rangle \wedge o \notin \langle \text{source-class2} \rangle\}$ .

The following subset relationship holds  $\langle \text{virtual-class} \rangle \subseteq \langle \text{source-class1} \rangle$ . Furthermore, the type description of the virtual class is equal to the type description of the first source class, i.e.,

$\text{type}(\langle \text{virtual-class} \rangle) := \text{type}(\langle \text{source-class1} \rangle)$ .

By default,  $\langle \text{virtual-class} \rangle \preceq \langle \text{source-class1} \rangle$ . This then implies the subclass relationship  $\langle \text{virtual-class} \rangle \text{ is-a } \langle \text{source-class1} \rangle$ . No subset, subtype or subclass relationships hold between the second  $\langle \text{source-class2} \rangle$  and the  $\langle \text{virtual-class} \rangle$ .

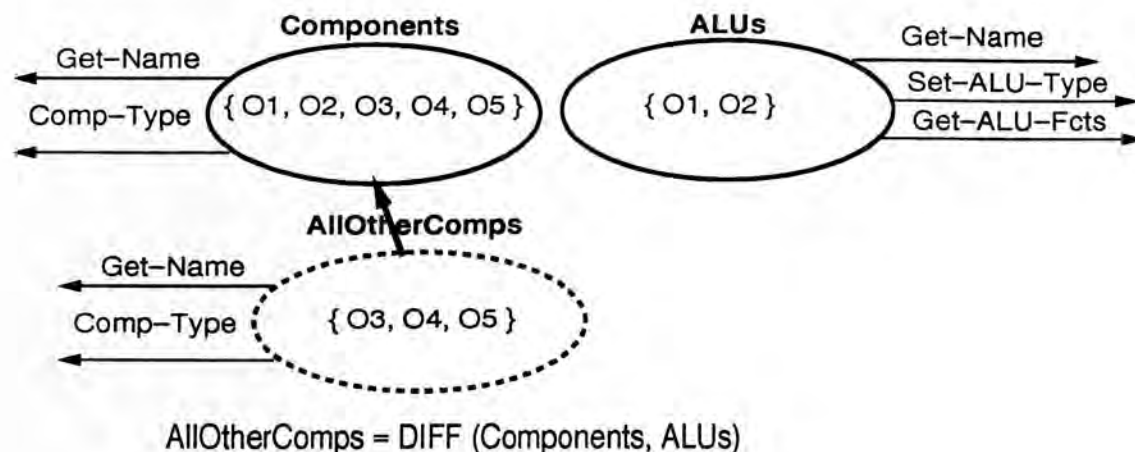


Figure 5.7: An Example of the **diff** Operator.

**Example 10.** In Figure 5.7, the **diff** operator is used in the query “**AllOtherComps** = **diff(Components,ALUs)**” to derive **AllOtherComps** from **Components** that are not in **ALUs**. I have  $\text{extent}(\text{AllOtherComps}) = \text{extent}(\text{Components}) - \text{extent}(\text{ALUs}) = \{O1, O2, O3, O4, O5\} - \{O1, O2\} = \{O3, O4, O5\}$ . Therefore  $\text{AllOtherComps} \subseteq \text{Components}$ . We also have  $\text{type}(\text{AllOtherComps}) = \text{type}(\text{Components}) = [\text{Get-Name}, \text{Comp-Type}]$  and  $\text{AllOtherComps} \preceq \text{Components}$ . The relationship (**AllOtherComps is-a Components**) has been added to Figure 5.7.

In this section, I have shown how virtual classes are created by the object algebra operators. In the example given below, I now show how these virtual classes are integrated with their source schema. A more thorough treatment of how to integrate them with the complete global schema rather than only the source subschema is presented in a later chapter.

**Example 11.** An example of each of the object algebra operators is given in Figure 5.8, where I depict the properties that are explicitly defined for a class and not those that are inherited from other classes. In Figure 5.8.a, the query “**C1' := hide [a1] from (C1)**” is used to derive the virtual class **C1'** from the source class **C1**. I have  $\text{extent}(\text{C1}') := \text{extent}(\text{C1})$ , i.e.,  $\text{C1} \subseteq \text{C1}'$ . Also  $\text{type}(\text{C1}) := \{a0, a1, \dots\}$ ,  $\text{type}(\text{C1}') := \{a0, \dots\}$ , and  $\text{C1} \preceq \text{C1}'$ . The *is-a* relationship (**C1 is-a C1'**) is indicated by the edge from **C1** to **C1'**.

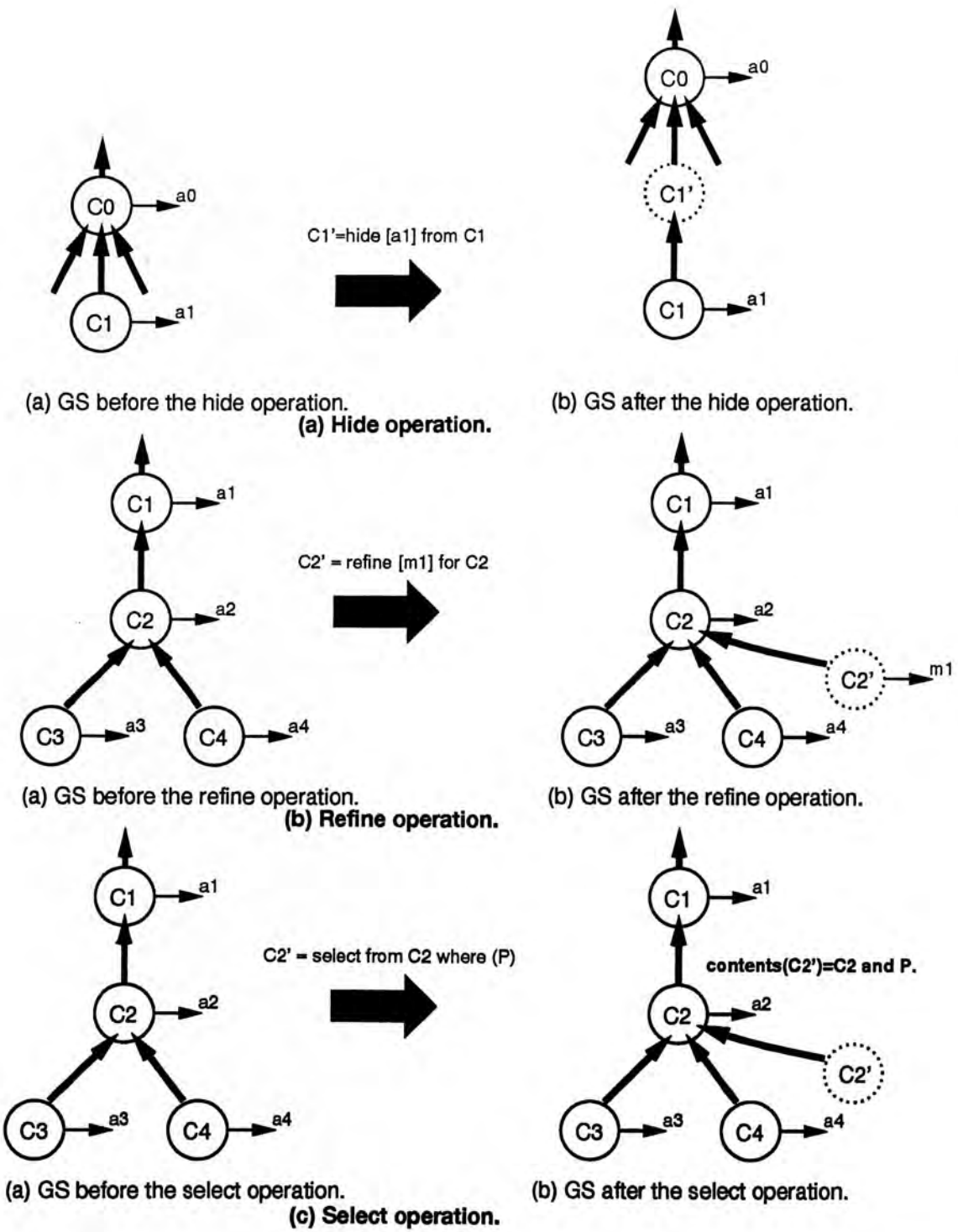


Figure 5.8: Examples of Class Derivation and Integration Using Object Algebra.

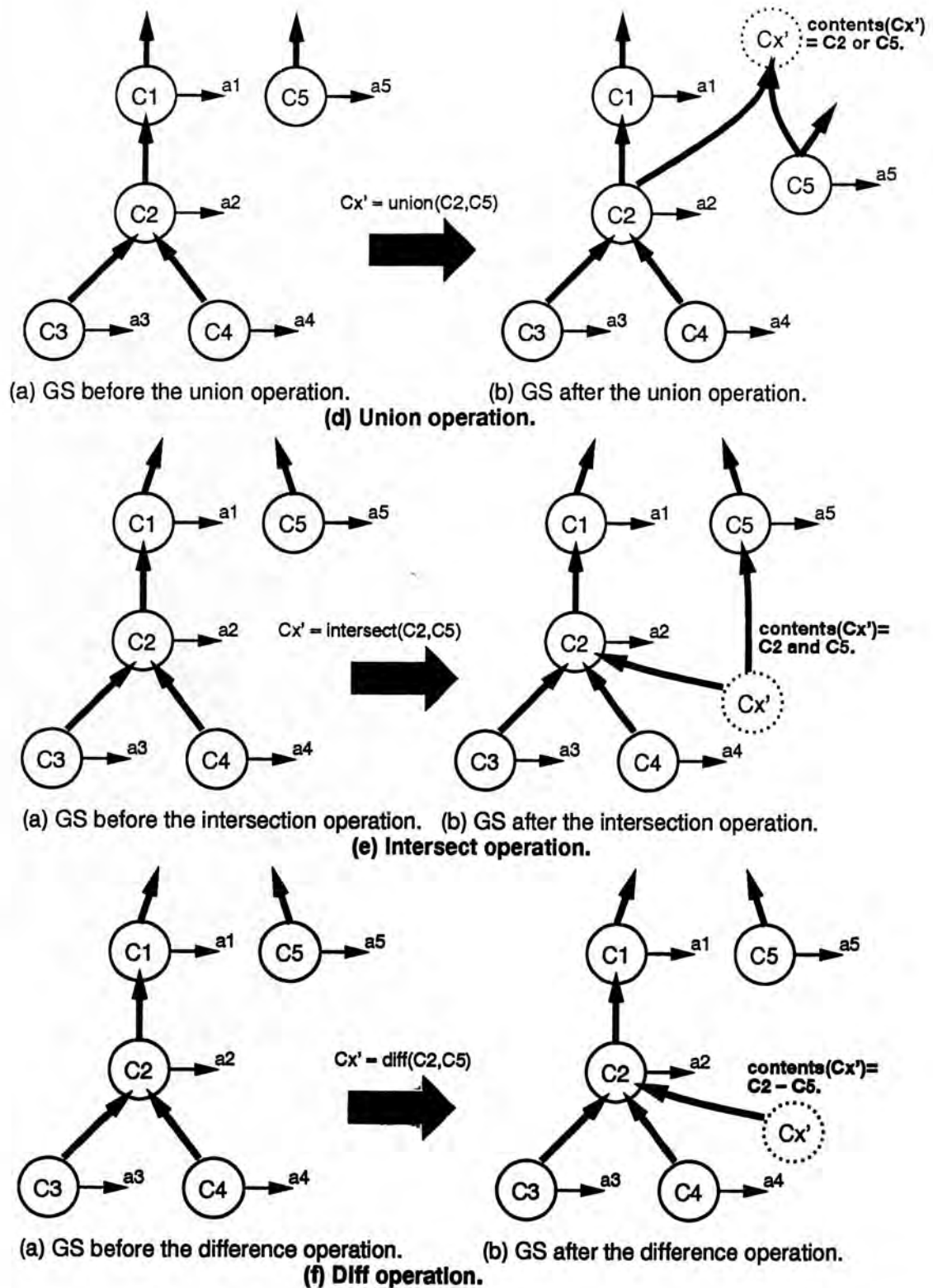


Figure 5.9: Examples of Class Derivation and Integration (cont.).

In Figure 5.8.b, the **refine** operator is used in the following query to derive  $C2'$  from  $C2$ :  $C2' := \text{refine } [m1 := \text{fct}] \text{ for } (C2)$ . I have  $\text{extent}(C2') := \text{extent}(C2)$ , i.e.,  $C2' \subseteq C2$ . The type of  $C2'$  has been extended by the new method  $m1$ ,  $\text{type}(C2) := \{a0, a1, \dots\}$ ,  $\text{type}(C2') := \{a0, a1, m1, \dots\}$ , and  $C2' \preceq C2$ .  $C2'$  is integrated into the global schema by placing  $C2'$  below  $C2$  as direct subclass.

In Figure 5.8.c, the **select** operator is used in the query " $C2' := \text{select from } (C2) \text{ where } (\langle \text{pred} \rangle)$ " to derive  $C2'$  from  $C2$ . Then  $C2' \subseteq C2$  and  $\text{type}(C2') := \text{type}(C2)$ . The is-a relationship ( $C2'$  is-a  $C2$ ) has been added as indicated by the edge from  $C2'$  to  $C2$ .

In Figure 5.9.d, the **union** operator is used in the query " $Cx' := \text{union}(C2, C5)$ " to derive the virtual class  $Cx'$  from the source classes  $C2$  and  $C5$ . Then  $\text{extent}(Cx') := \text{extent}(C2) \cup \text{extent}(C5)$ . Hence  $C2 \subseteq Cx'$  and  $C5 \subseteq Cx'$ . Also  $\text{type}(Cx') := \text{type}(C2) \sqcup \text{type}(C5)$ . Hence  $C2 \preceq Cx'$  and  $C5 \preceq Cx'$ . The is-a relationships ( $C2$  is-a  $Cx'$ ) and ( $C5$  is-a  $Cx'$ ) are indicated by the edges from  $C2$  to  $Cx'$  and from  $C5$  to  $Cx'$ , respectively.

In Figure 5.9.e, the **intersect** operator is used to derive the virtual class  $Cx'$  from  $C2$  and  $C5$ , namely, the query  $Cx' := \text{intersect}(C2, C5)$ . Then  $\text{extent}(Cx') := \text{extent}(C2) \cap \text{extent}(C5)$ . Hence  $Cx' \subseteq C2$  and  $Cx' \subseteq C5$ . Also  $\text{type}(Cx') := \text{type}(C2) \sqcap \text{type}(C5)$ . Hence  $Cx' \preceq C2$  and  $Cx' \preceq C5$ . The is-a relationships ( $Cx'$  is-a  $C2$ ) and ( $Cx'$  is-a  $C5$ ) are indicated by the edges from  $Cx'$  to  $C2$  and from  $Cx'$  to  $C5$ , respectively.

In Figure 5.9.f, the **diff** operator is used in the query " $Cx' := \text{diff}(C2, C5)$ " to derive  $Cx'$  from  $C2$  and  $C5$ . I have  $\text{extent}(Cx') := \text{extent}(C2) - \text{extent}(C5)$  and  $Cx' \subseteq C2$ . Also  $\text{type}(Cx') := \text{type}(C2)$  and thus  $Cx' \preceq C2$ . ( $Cx'$  is-a  $C2$ ) has been added to Figure 5.9.f.b.

### 5.3 The Macro Object-Algebra Operators

While the operators presented in the previous section work on individual classes, below I present object algebra operators that can be used to derive new virtual schema from existing schemata. These operators can be built by composing primitive operators on individual classes into meaningful operators on schemata. Hence I refer to these macro operators also as macro-operators. These operators serve two purposes. First, they simplify the creation of a virtual schema by grouping the primitive class operators into more complex well-defined macro operators. Second, they

simplify the integration of the set of virtual classes into the global schema by explicitly stating (and thus preserving) the is-a relationship between the set of virtual classes that make up the resulting virtual subschema.

For this section I assume the following notations. For  $\langle \text{class} \rangle$  the name of a class of a schema  $S$ , the term  $\langle \text{class} \rangle^*$  refers to the complete subschema of  $S$  that is rooted at the class  $\langle \text{class} \rangle$ . Hence, the subschema  $\langle \text{class} \rangle^*$  includes the class  $\langle \text{class} \rangle$  and all its subclasses in  $S$ . The macro operators introduced in this section take as input argument a subschema of the global schema called  $\langle \text{source-class} \rangle^*$  and they return a virtual (sub)-schema indicated by  $\langle \text{virtual-class} \rangle^*$  with  $\langle \text{virtual-class} \rangle$ . This virtual schema  $\langle \text{virtual-class} \rangle^*$  then has to be integrated with the source schema  $\langle \text{source-class} \rangle^*$  as well as all other classes in the global schema. I assume that the classes in the input schema are named  $C_1, C_2, \dots, C_n$ , and I thus refer to the classes of the output schema by the class names  $C_1', C_2', \dots, C_n'$ , respectively. Of course, explicit renaming of these classes by the view definer can and should take place. Each newly generated class is given some unique internal class identifier.

### 5.3.1 The Hide\* Macro Operator

The **hide\*** operator, an extension of the **hide** operator defined in Section 5.2.1, works on a complete schema graph rather than on one individual class. Consequently, the result of the **hide\*** operator is a virtual schema graph rather than one virtual class. The **hide\*** operator removes one or more attributes from all classes in the source schema, while preserving all other attributes visible in the source schema. It now becomes apparent why I prefer the **hide** operator over the **project** operator. Both accomplish the same effect when applied to an individual class, namely, for the **hide** operator I specify which property functions to remove (hide) while for the **project** operator I specify which property functions to keep (project). Once I deal with a complete schema, I achieve different results. The **project\*** operator projects out a *fixed* set of attributes for all its result classes; and thus all classes in the virtual subschema will have the same projected result type. The **hide\*** operator on the other hand hides the same *fixed* set of attributes from all classes; and thus all classes in the virtual subschema will potentially have different types - namely their original property functions minus the hidden property functions.

The **hide\*** operator may for instance be useful for the floorplan view schema. Assume that in a floorplan view, the structure of the design is not allowed to be modified (via the *create-instance* property function or the *delete-instance* property function). However, the geometric placement information of existing component



object instances can be modified, e.g., by changing the placement of a component or by resizing a component. Therefore, I would use the **hide\*** operator to remove the *create-instance* and *delete-instance* property functions from the complete schema while leaving all other operators intact.

The **hide\*** operator has the following syntax:

$$\langle \text{virtual-class} \rangle^* := \mathbf{hide}^* [\langle \text{prop-functions} \rangle] \text{ from } (\langle \text{source-class} \rangle^*).$$

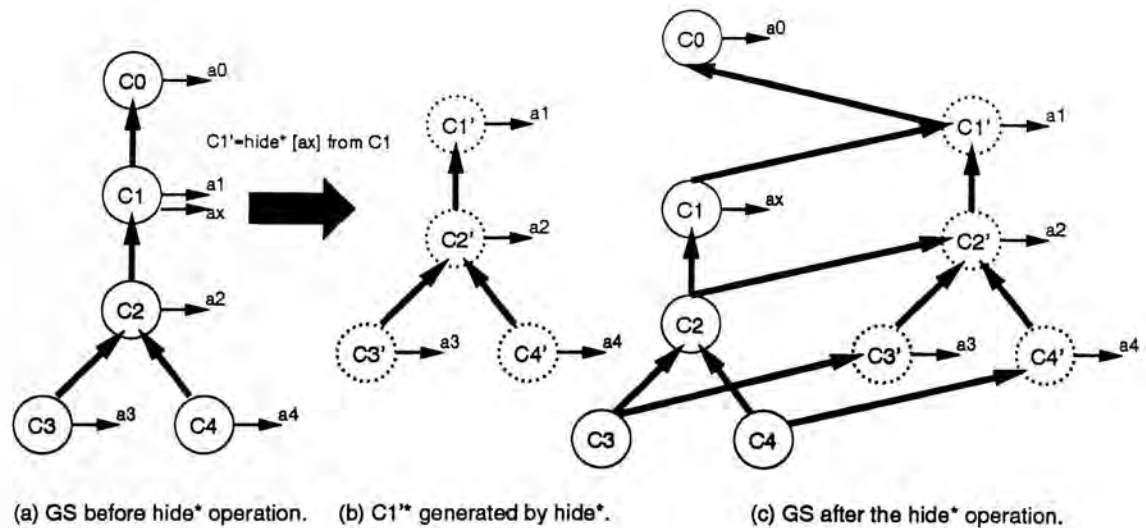
The semantics of the **hide\*** operator are to remove the property functions listed in the set  $\langle \text{prop-functions} \rangle$  from all classes in the source schema while preserving all other attributes that are visible in the different classes of the schema. More precisely, for all classes  $C_i$  in the source schema  $\langle \text{source-class} \rangle^*$ , derive a virtual class  $C_i'$  in the result schema  $\langle \text{virtual-class} \rangle^*$  by the query  $C_i' := \mathbf{hide} [\langle \text{prop-functions} \rangle] \text{ from } (C_i)$ . Due to the semantics of the **hide\*** operator, which creates the virtual classes by hiding the same set of property functions from all source classes, I can deduce the following:

$$(\forall C_i, C_j \text{ in } \langle \text{source-class} \rangle^*) (\forall C_i', C_j' \text{ in } \langle \text{virtual-class} \rangle^*) (C_i \text{ is-a } C_j \Leftrightarrow C_i' \text{ is-a } C_j').$$

In addition, the semantics of the **hide** operator imply an *is-a* relationship between each source and virtual class pair. Therefore, the following *is-a* relationships hold between the classes in the source and the result schema:

$$(\forall C_i \text{ in } \langle \text{source-class} \rangle^*) (\forall C_i' \text{ in } \langle \text{virtual-class} \rangle^*) (C_i \text{ is-a } C_i').$$

**Example 12.** *An example of the **hide\*** macro operator is given in Figure 5.10. The query " $C1'^* := \mathbf{hide}^* [ax] \text{ from } (C1^*)$ " is used to derive the virtual schema  $C1'^*$  from the source schema  $C1^*$ . Since the source schema  $C1^*$  rooted at the class  $C1$  has four classes, the virtual schema  $C1'^*$  rooted at the class  $C1'$  is also composed of four classes. All four virtual classes are identical to their source classes except for the removal of the property function *ax* from their respective interfaces. In other words, the **hide\*** operator effectively removed the property function *ax* from the subschema  $C1^*$ . The *is-a* relationships among the classes of the virtual schema are directly derived from the *is-a* relationships among the classes of the source schema, e.g., ( $C2$  is-a  $C1$ ) implies ( $C2'$  is-a  $C1'$ ), etc. Also, the source classes are *is-a* related with the virtual classes from which they are derived, e.g., ( $C2$  is-a  $C2'$ ), etc.*

Figure 5.10: The **hide\*** Macro Operator.

### 5.3.2 The **Refine\*** Macro Operator

The **refine\*** operator, an extension of the **refine** operator defined in Section 5.2.1, also works on a complete schema. The **refine\*** operator adds one or more new derived property functions to all classes in a subschema, while preserving all other attributes visible in the schema. The **refine\*** operator has the following syntax:

$$\langle \text{virtual-class} \rangle^* := \text{refine}^* [\langle \text{prop-function-defs} \rangle] \text{ for } (\langle \text{source-class} \rangle^*).$$

with  $\langle \text{prop-function-def} \rangle$  the definition of one or more new property functions. The semantics of the **refine\*** operator are to refine the type description of all classes in the source schema by adding the property functions listed in the set  $\langle \text{prop-function-defs} \rangle$  to their type definitions. More precisely, for all classes  $C_i$  in the source schema  $\langle \text{source-class} \rangle^*$ , derive a virtual class  $C_i'$  in the result schema  $\langle \text{virtual-class} \rangle^*$  by the query  $C_i' := \text{refine} [\langle \text{prop-function-defs} \rangle] \text{ for } (C_i)$ . Since all virtual classes are changed in the same manner, namely, by adding the properties  $\langle \text{prop-function-defs} \rangle$  to them, I can deduce the following:

$$(\forall C_i, C_j \text{ in } \langle \text{source-class} \rangle^*) (\forall C_i', C_j' \text{ in } \langle \text{virtual-class} \rangle^*) (C_i \text{ is-a } C_j \Leftrightarrow C_i' \text{ is-a } C_j').$$

In addition, the semantics of the **refine** operator imply an *is-a* relationship between each pair of source and virtual class:

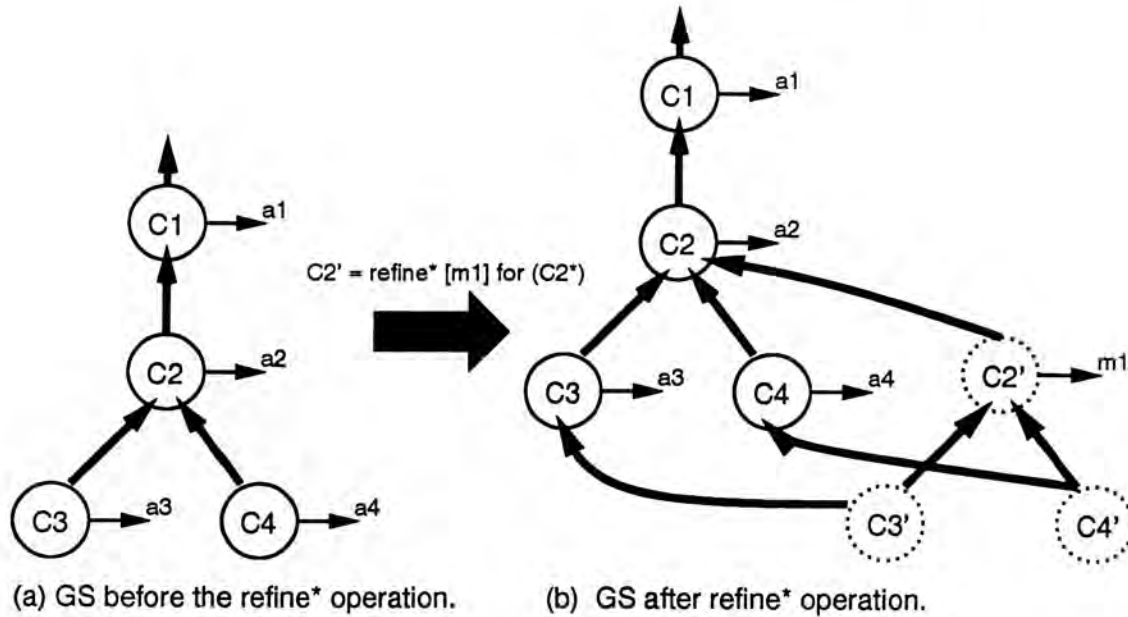
$$(\forall C_i \text{ in } \langle \text{source-class} \rangle^*) (\forall C_i' \text{ in } \langle \text{virtual-class} \rangle^*) (C_i' \text{ is-a } C_i).$$


Figure 5.11: The **refine\*** Macro Operator.

**Example 13.** An example of the **refine\*** macro operator is given in Figure 5.11. The query " $C2'^* := \text{refine}^* [m1] \text{ for } (C2^*)$ " is used to derive the virtual schema  $C2'^*$  from the source schema  $C2^*$ . The virtual schema  $C1'^*$  is composed of the three classes  $C2'$ ,  $C3'$  and  $C4'$ , which are identical to their source classes  $C2$ ,  $C3$  and  $C4$  except for the addition of the new property function  $m1$  to their respective interfaces. The **refine\*** operator effectively refined the complete subschema  $C2^*$  by the new property function  $m1$ . In particular,  $m1$  is defined at the root of the subschema  $C2'^*$  and then inherited by all classes in  $C2'^*$ . The is-a relationships among the classes of the virtual schema are directly derived from the is-a relationships among the classes of the source schema, e.g.,  $(C3 \text{ is-a } C2)$  implies  $(C3' \text{ is-a } C2')$ , etc. Also, the source classes are is-a related with the virtual classes from which they are derived, e.g.,  $(C2' \text{ is-a } C2)$ , etc.

### 5.3.3 The **Select\*** Macro Operator

The **select\*** operator hides a subset of objects from all classes in the given source subschema. Note that this is different from completely removing a subschema from a view. The **select\*** operator has the following syntax:

$\langle virtual-class \rangle^* := select^* from (\langle source-class \rangle^*) where (\langle predicate \rangle),$

with  $\langle predicate \rangle$  some possibly complex predicate function on the source class and its type description. The semantics of the  $select^*$  operator are to select a subset of objects from all classes in the given source subschema based on the given predicate. More precisely, for all classes  $C_i$  in the source schema  $\langle source-class \rangle^*$ , derive a virtual class  $C_i'$  in the result schema  $\langle virtual-class \rangle^*$  by the query  $C_i' := select^* from (C_i) where (\langle predicate \rangle)$ . I can deduce the following:

$(\forall C_i, C_j \text{ in } \langle source-class \rangle^*) (\forall C_i', C_j' \text{ in } \langle virtual-class \rangle^*) (C_i \text{ is-a } C_j \Leftrightarrow C_i' \text{ is-a } C_j')$ .

Secondly, the following *is-a* relationships hold between the classes in the source and result schema:

$(\forall C_i \text{ in } \langle source-class \rangle^*) (\forall C_i' \text{ in } \langle virtual-class \rangle^*) (C_i' \text{ is-a } C_i)$ .

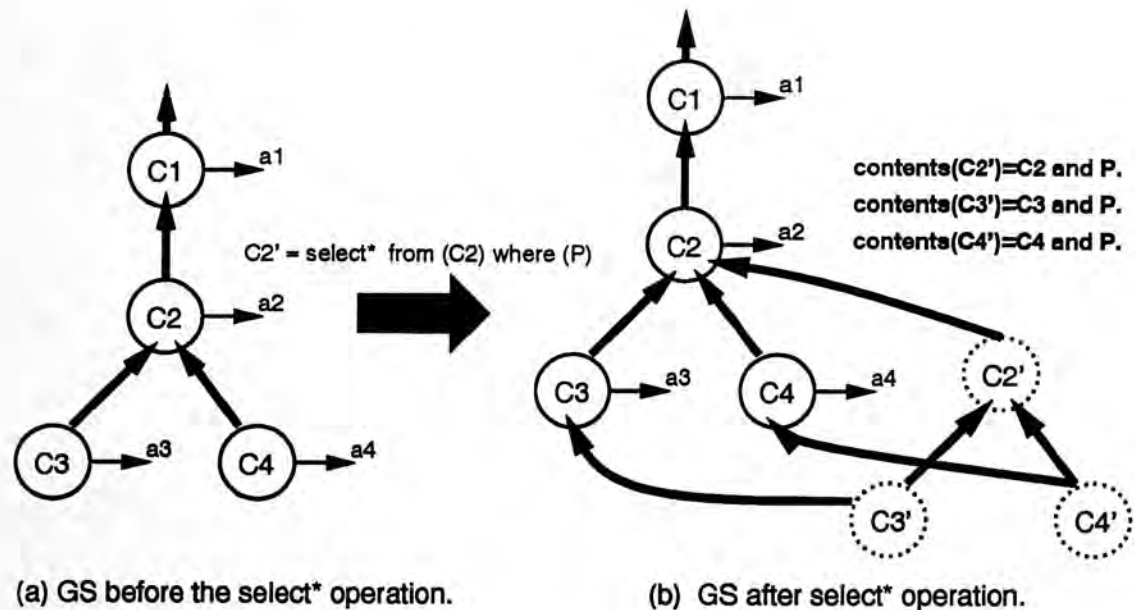


Figure 5.12: The  $selection^*$  Macro Operator.

**Example 14.** An example of the  $select^*$  macro operator is given in Figure 5.12. The query  $C2'^* := select^* from (C2^*) where (\langle predicate \rangle)$  is used to derive the virtual schema  $C2'^*$ . The  $select^*$  operator modified the subschema  $C2^*$  by effectively removing a set of object instances from all classes in  $C2^*$  with the subtracted set defined by  $\{ o \in O \mid o \in C2 \wedge \langle predicate \rangle(o) = true \}$ .

### 5.3.4 The Union\* Macro Operator

The schema extensions of the set operators would work on one input schema and on one input class. The **union\*** operator adds additional object instances to all classes in a subschema, where the additional object instances are gotten from the second class argument to the union operator. The **union\*** operator has the following syntax:

$$\langle \text{virtual-class} \rangle^* := \text{union}^*( \langle \text{source-class1} \rangle^*, \langle \text{source-classx} \rangle ).$$

The **union\*** operator creates the virtual schema  $\langle \text{virtual-class} \rangle^*$  that is composed of virtual classes  $C_i'$  defined as follows: for all classes  $C_i$  in the source schema  $\langle \text{source-class} \rangle^*$ , derive a virtual class  $C_i'$  by the query  $C_i' := \text{union}(C_i, \langle \text{source-classx} \rangle)$ . Again the following *is-a* relationships hold:

$$(\forall C_i, C_j \text{ in } \langle \text{source-class1} \rangle^*) (\forall C_i', C_j' \text{ in } \langle \text{virtual-class} \rangle^*) (C_i \text{ is-a } C_j \Leftrightarrow C_i' \text{ is-a } C_j').$$

In addition, the virtual classes  $C_i'$  are superclasses of both their sources classes  $C_i$  and of  $\langle \text{source-classx} \rangle$ , i.e.,

$$(\forall C_i \text{ in } \langle \text{source-class1} \rangle^*) (\forall C_i' \text{ in } \langle \text{virtual-class} \rangle^*) (C_i \text{ is-a } C_i' \text{ and } (\langle \text{source-classx} \rangle \text{ is-a } C_i')).$$

The following optimization on the generalization hierarchy will also take place. Since the class  $\langle \text{source-classx} \rangle$  is a subclass of all virtual classes  $C_i'$ , it is sufficient to only maintain the *is-a* relationships between the  $\langle \text{source-classx} \rangle$  and the leaf classes  $C_i'$  of the virtual schema  $\langle \text{source-class1} \rangle^*$ . The *is-a* relationships of the  $\langle \text{source-classx} \rangle$  with all other non-leaf classes  $C_i'$  are derivable by transitive closure.

**Example 15.** *An example of the union\* macro operator is given in Figure 5.13. The query  $C2'^* := \text{union}^*(C2^*, C5)$  is used to derive the virtual schema  $C2'^*$ . The union\* macro operator modifies the subschema  $C2^*$  by adding a set of objects equal to  $\text{extent}(C5)$  to the extent of all classes in  $C2^*$ . As can be seen in the figure,  $C$  is only a direct subclass of all leaf nodes of  $C2'^*$ . All other *is-a* relationships between  $C2'^*$  and  $C5$  can be derived by transitive closure.*

### 5.3.5 The Intersection\* Macro Operator

The **intersection\*** operator removes a subset of object instances from all classes in a subschema, namely, those object instances that are members of both

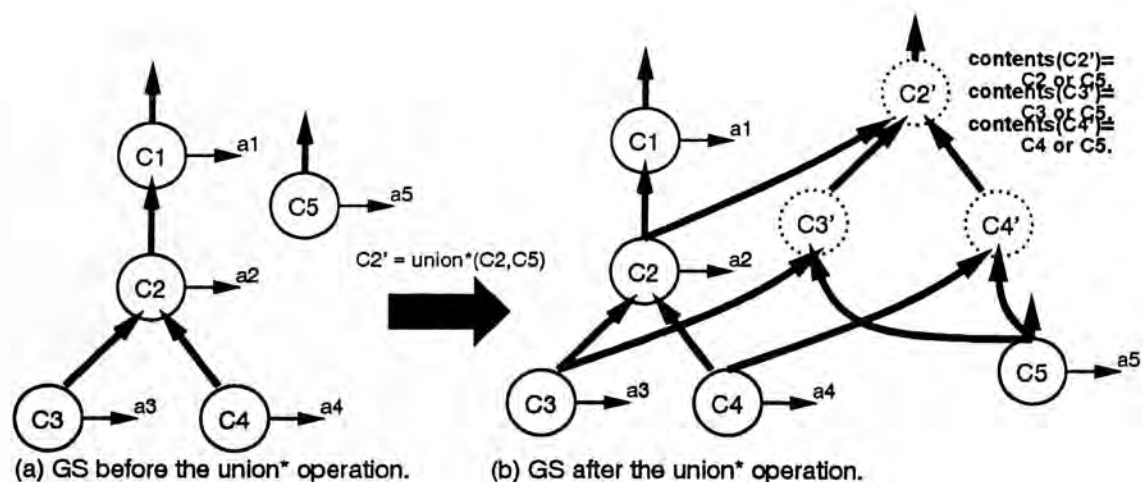


Figure 5.13: The  $\text{union}^*$  Macro Operator.

source classes. This operator corresponds closely to the  $\text{select}^*$  operator with the predicate of selection defined via the expression  $(o \in C_i \text{ and } o \in C_x)$  with  $i$  for  $i=1, \dots, n$ , referring to all classes  $C_i$  in the subschema. The  $\text{intersection}^*$  operator has the following syntax:

$$\langle \text{virtual-class} \rangle^* := \text{intersect}^*( \langle \text{source-class1} \rangle^*, \langle \text{source-classx} \rangle ).$$

The  $\text{intersect}^*$  operator creates the virtual schema  $\langle \text{virtual-class} \rangle^*$  that is composed of virtual classes  $C_i'$  defined as follows: for all classes  $C_i$  in the source schema  $\langle \text{source-class1} \rangle^*$ , derive a virtual class  $C_i'$  by the query  $C_i' := \text{intersect}(C_i, \langle \text{source-classx} \rangle)$ . The following *is-a* relationships hold:

$$(\forall C_i, C_j \text{ in } \langle \text{source-class1} \rangle^*) (\forall C_i', C_j' \text{ in } \langle \text{virtual-class} \rangle^*) (C_i \text{ is-a } C_j \Leftrightarrow C_i' \text{ is-a } C_j').$$

In addition, the virtual classes  $C_i'$  are subclasses of both their sources classes  $C_i$  and of  $\langle \text{source-classx} \rangle$ , i.e.,

$$(\forall C_i \text{ in } \langle \text{source-class1} \rangle^*) (\forall C_i' \text{ in } \langle \text{virtual-class} \rangle^*) (C_i' \text{ is-a } C_i \text{ and } C_i' \text{ is-a } \langle \text{source-classx} \rangle).$$

Again, an optimization of the resulting generalization hierarchy takes place. Since all virtual classes  $C_i'$  are subclasses of the class  $\langle \text{source-classx} \rangle$ , it is sufficient to only maintain the *is-a* relationship between the root  $\langle \text{source-class1} \rangle$  of the virtual schema  $\langle \text{source-class} \rangle^*$  and the source class  $\langle \text{source-classx} \rangle$ .

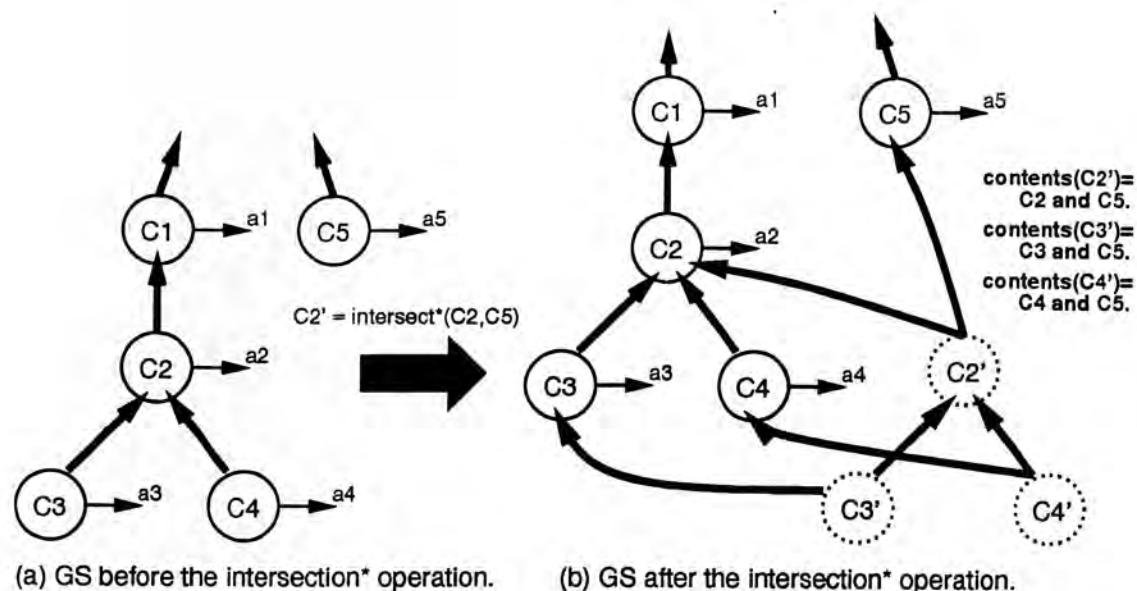


Figure 5.14: The  $\text{intersect}^*$  Macro Operator.

**Example 16.** An example of the  $\text{intersect}^*$  macro operator is given in Figure 5.14. The query  $C2'^* := \text{intersect}^*(C2^*, C5)$  is used to derive the virtual schema  $C2'^*$ . The  $\text{intersect}^*$  macro operator modifies the subschema  $C2^*$  by removing from each class in  $C2^*$  all object instances that are not also members of the class  $C5$ . As can be seen in the figure, only the root class of  $C2'^*$  is made a direct subclass of  $C5$ . All other is-a relationships between  $C2'^*$  and  $C5$  can be derived by transitive closure.

### 5.3.6 The Difference\* Macro Operator

Similarly to the intersection operator, the difference\* operator removes a subset of object instances from all classes in a subschema with the subset of objects equal to the second argument class to the operation. It thus corresponds closely to the select operator, where the predicate of selection is defined via the expression  $(o \in C_i \text{ and not}(o \in C_x))$  with  $i$  for  $i=1, \dots, n$ , referring to all classes  $C_i$  in the subschema. The  $\text{difference}^*$  operator has the following syntax:

$\langle \text{virtual-class} \rangle^* := \text{diff}^*(\langle \text{source-class1} \rangle^*, \langle \text{source-classx} \rangle)$ .

The  $\text{diff}^*$  operator creates the virtual schema  $\langle \text{virtual-class} \rangle^*$  that is composed of virtual classes  $C_i'$  defined as follows: for all classes  $C_i$  in the source schema

$\langle source-class \rangle^*$ , derive a virtual class  $C_i'$  by the query  $C_i' := \text{diff}(C_i, \langle source-class \rangle)$ . The following *is-a* relationships hold:

$$(\forall C_i, C_j \text{ in } \langle source-class \rangle^*) (\forall C_i', C_j' \text{ in } \langle virtual-class \rangle^*) (C_i \text{ is-a } C_j \Leftrightarrow C_i' \text{ is-a } C_j').$$

In addition, the virtual classes  $C_i'$  are subclasses of their sources classes  $C_i$ , i.e.,

$$(\forall C_i \text{ in } \langle source-class \rangle^*) (\forall C_i' \text{ in } \langle virtual-class \rangle^*) (C_i' \text{ is-a } C_i).$$

There are no subclass relationships between the virtual classes  $C_i'$  and the second input class  $\langle source-class \rangle$ .

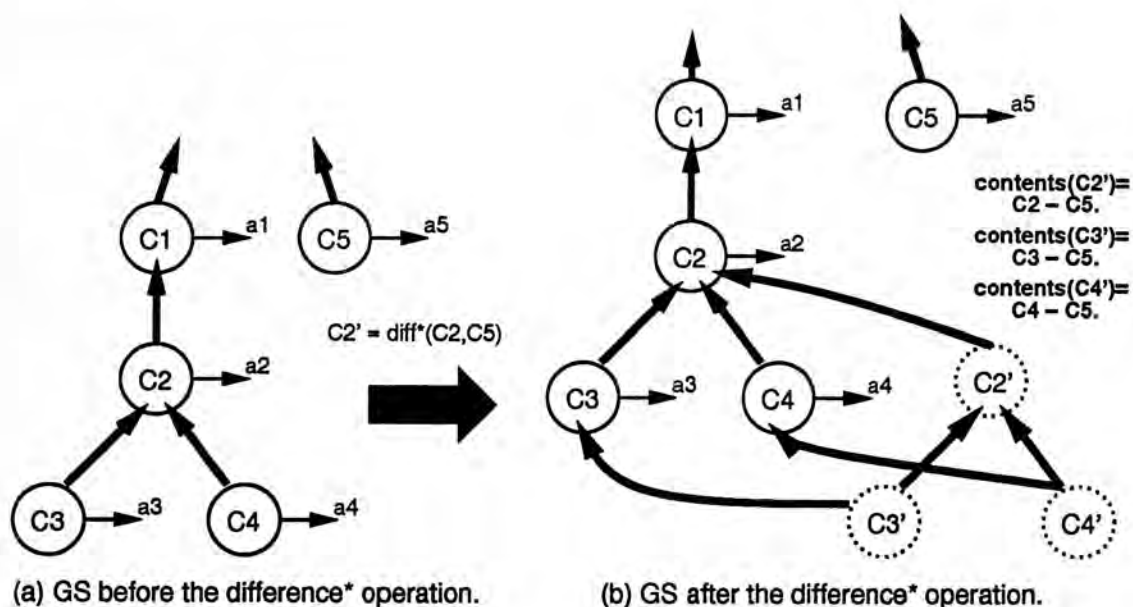


Figure 5.15: The *difference\** Macro Operator.

**Example 17.** An example of the *difference\** macro operator is given in Figure 5.15. The query  $C2'^* := \text{diff}^*(C2^*, C5)$  is used to derive the virtual schema  $C2'^*$ . The *diff\** macro operator modifies the subschema  $C2^*$  by removing all object instances in  $\text{extent}(C5)$  from all classes in  $C2^*$ . Note that this is equivalent to the *select\** operator if I set the selection predicate  $\langle predicate \rangle$  to " $o \notin C5$ ".

In this section, I have developed more complex macro-operators that are composed of the algebra operators. Furthermore, I have shown how the virtual schemata



created by the macro operators are integrated with their source schemata. In the next section, I will discuss how to integrate virtual classes with the complete global schema.

# Chapter 6

## Set Operators in Object-Oriented Databases

### 6.1 Basic Issues

In Chapter 5, I have proposed a basic set of object algebra operators with simple fixed semantics. In this chapter, on the other hand, I will explain that the semantics of the operators proposed in Chapter 5 are just one of a number of possible interpretations. I will focus in particular on describing richer semantics for set operators. This discussion is based on the work published jointly with L. Bic in [Rund92b].

Query languages designed for traditional database systems, such as the relational model, generally support set operators. However, the semantics of these set operators are not adequate for richer data models of newly developed object-oriented database systems, such as [Fish87, Maie86, Abit87, Hull87]. Most such models support a rich class definition facility based on restricting inherited properties, that is, special forms of the specialization and generalization abstractions [Atki87, Hull87, Mylo80, Hamm81]. On the other hand, the potential of set operators has largely been unexplored. The reason for this is that, while set operators on simple elements (values) are well understood, precise semantics for set operators on complex objects have not yet been developed. Such definitions require a clear distinction between the dual notion of a class, which represents a *set* and also provides a *type* description.

Our work fills this gap by presenting a framework for executing set-theoretic operators on the class construct. Commonly supported class derivation mechanisms, such as the specialization abstraction, are *type-oriented* [Atki87]; they perform operations on the type aspect of a class, which then automatically implies a particular set relationship between the original and the derived class. Set operators work in a contrary manner. They perform some operation on the set-aspect of a class and

the particular type relationships are implied. Such implied types, however, do not always have to take on the restricted forms assumed in the literature [Hamm81] and suggested in Chapter 5, as we will show in the following.

This chapter presents a framework for executing set-theoretic operations on complex objects. We consider the four set operators most common in set theory – union, intersection, difference, and symmetric difference. Our approach is to extend set theory to the world of classes (with the term classes as defined for instance in semantic and object-oriented data models) while preserving the well-known set-theoretic semantics. The specification of set operations on classes is based on our distinction between the set and type aspect of a class. Here, we first define the effect of a set operation on the membership of the corresponding class, i.e., we address the set aspect of a class. For this, we utilize the concept of object identity from data modeling research. In addition, the type description of the result class derived by a set operation is specified. Different types of set operations can be distinguished based on the choices of the resulting type descriptions of the derived class. We refer to these set operation types as *collecting*, *extracting*, and *user-specified* depending on whether the derived class inherits *all* properties, only the *common* properties, or a *user-specified* subset of the properties from the two original classes, respectively. Furthermore, we design rules that describe how property characteristics are to be inherited through these set operations. These rules accommodate property characteristics such as multi-valued versus single-valued and required versus optional.

Another contribution of this work is the following. We utilize our distinction between set and type relationships of classes – which are commonly combined and treated as one relationship, the *is-a* relationship – for an analysis of relationship types that hold between classes derived by set operators. This analysis shows that the resulting class relationships are not necessarily *is-a* relationships, as is commonly (but implicitly) assumed in the literature [Hamm81]. In fact, a class derived by a symmetric difference operator will never stand in any *is-a* relationship with its base classes, no matter whether the operator's type is *collecting*, *extracting*, or *user-specified*. Consequently, our framework allows for the inheritance of properties between classes that are not *is-a* related. We know of no other data model that has this capability.

On the other hand, we do not address, in this chapter, class definition abstractions, such as specialization, selection and Cartesian aggregation, since these are common to most data models and have been dealt with in depth in the literature [Rund93, Rund91a, Atki87, Hull87, Mylo80, Hamm81]. Instead, we focus on the set operators, which have for the most part been ignored by data model designers. Existing data models can augment their set of abstractions by our proposed set

operators. In summary, this work provides the designers of a data model with a framework of set operators that allow them to make an explicit and educated choice among them.

This chapter is organized as follows. In Section 6.2 we review the basics of conventional set theory as needed for the remainder of the chapter. Section 6.3 familiarizes the reader with conceptual data modeling terminology. Special emphasis is placed here on the distinction between subset/superset and subtype/supertype relationships of classes. In Section 6.4 we present definitions of set operators on complex objects as well as rules for the inheritance of properties from the two original classes to the derived class. Throughout this section we give pragmatic examples that support the usefulness of our framework. In Section 6.5 we then present a potpourri of class derivation examples using the proposed set operators, followed by a discussion of the possible set and type relationships in Section 6.6. Related research and conclusions are given in Sections 6.7 and 6.8, respectively.

## 6.2 Conventional Set Theory

Set operations in conventional set theory are well understood. They assume values as members and do not address typing and related problems. We briefly survey the basics of set theory, since our goal is to preserve these well-accepted semantics of set operations when extending them to typed objects and classes.

A *set*  $S$  is a collection of objects where objects may be anything including symbols, physical objects or abstract concepts. Objects within a set  $S$  are referred to as *elements*. In set theory, all elements, be they complex like a Person or simple like an integer, are represented by one symbol. In other words, set theory considers all objects to be simple, not typed, and lacking any associated properties. We write ' $s \in S$ ' ( ' $s \notin S$ ' ) to mean that the object  $s$  is (*not*) an *element* of the set  $S$ .

New sets can be formed from existing ones by the following operations: union, intersection, difference, and symmetric difference. In set theory, these operations determine the exact membership of the newly created sets. Let  $S1$  and  $S2$  be any two sets. The *difference* of  $S1$  with respect to  $S2$  is defined as  $S1 - S2 = \{s | s \in S1 \wedge s \notin S2\}$ . The *union* of  $S1$  and  $S2$  is defined as  $S1 \cup S2 = \{s | s \in S1 \vee s \in S2\}$ . The *symmetric difference* of  $S1$  and  $S2$  is defined as  $S1 \Delta S2 = \{s | (s \in S1 \wedge s \notin S2) \vee (s \notin S1 \wedge s \in S2)\}$ . The *intersection* of  $S1$  and  $S2$  is defined as  $S1 \cap S2 = \{s | s \in S1 \wedge s \in S2\}$ .

A set  $S1$  is defined to be a *subset* of the set  $S2$ , written  $S1 \subseteq S2$ , if and only if every element of  $S1$  is also an element of  $S2$ . Formally,  $S1 \subseteq S2$  if and only if  $(\forall s)(s \in S1) \implies (s \in S2)$ . The subset operation is not a mechanism to create a new set out of a given one, instead it only models a *relationship* between two sets. To actually create a subset, operators such as the set difference, union, etc., must be used. It is of course also possible to explicitly create a subset of a given set by selecting some of its elements and grouping them into a new set. Figure 6.1 lists the subset relationships between initial sets and the sets resulting from applying these set operators.

set operation	resulting subset relationships
intersection	$(S1 \cap S2 \subseteq S1)$ and $(S1 \cap S2 \subseteq S2)$
union	$(S1 \subseteq S1 \cup S2)$ and $(S2 \subseteq S1 \cup S2)$
difference	$S1 - S2 \subseteq S1$
symmetric difference	none

Figure 6.1: Conventional Set Operators and Resulting Subset Relationships.

In the remainder of this chapter, we discuss how these operations can be generalized for dealing with classes and complex objects found in data modeling environments. It is our goal for this work that the set operations, in this new context, preserve the subset relationships shown in Figure 6.1.

### 6.3 Characteristics of Object-Oriented Data Models

A key to the solution of well-defined set operations on complex objects is the explicit distinction between the type and set aspect of a class as discussed in Chapter 3. Set operations on collections of untyped elements (mathematical sets) are well understood. Thus we have to study the effect of set operations on the type description of a class while preserving the semantics of the class's set notion. Below we introduce terminology for the data modeling concepts common to object-oriented data models needed for subsequent discussions. This discussion is based on the work presented in [Rund91a] and [Rund93].

### 6.3.1 Entities and Classes

In semantic data modeling, the notions of set, class, and type are not always clearly distinguished. In this section we take the following position. Entities in our data model represent a concrete or abstract concept in the application world. The term entity is used in this section in its most generic form — it may, for example, stand for a row in a relation table [Codd79], an entity (or a relationship) in the entity-relationship model [Chen76], an entity in a semantic data model [Hull87], or an object in an object-oriented model [Mylo80].

We distinguish between *values* and *abstract entities*. Values are taken directly from some predefined base domains, such as integers or strings. Abstract entities (or entities) correspond to abstract concepts or objects from the application domain, for example, a Person or a Hotel-Reservation record. An entity is modeled in the database by an *identity* and a *state* [Khos86, Rund93]. The *identity* is a globally unique identifier of an entity that is independent of the state of the entity. Each time a new entity is created, an identity is assigned to it by the system. A value, on the other hand, does not have the concept of identity or state associated with it. Each value is essentially the string of symbols used to represent it. Thus, when the value is modified (i.e., a symbol is changed), it becomes a *different* element. In this section, we use a pair of angle brackets “<” and “>” to indicate that we are referring to an abstract entity rather than a value:

*<entity reference>*

When it is clear from the context that we are referring to an abstract entity, the angle brackets may be dropped.

To indicate that we are referring to the entity’s identifier, we use the following notation,

*<entity reference>.id.*

The state of an entity corresponds to a collection of one or more property names and associated values. We refer to the properties (attributes) of an entity by:

*<property name > (<entity reference > ).*

The following example illustrates the just introduced notations.

**Example 18.** An *abstract person entity* may be referred by  $\langle person1 \rangle$ . The unique identifier of that *person entity* is referred by  $\langle person1 \rangle.id$ . The Name property of that person entity  $\langle person1 \rangle$  is referred by  $Name(\langle person1 \rangle)$ .

Entities can either be *simple* or *complex*. Simple entities are taken directly from a base domain of the application, while complex entities are built from other entities of the database using database abstractions, such as Cartesian aggregation or a power set grouping [Rund91a]. An example of a simple entity is an abstract object in the real world, for instance, a person or a ship. An example of a complex entity is a hotel-reservation record, which corresponds to the relationship between a person, a hotel room and a date. Using the terminology introduced above, a simple entity is a database entity with identity and a possibly empty set of simple properties, meaning, all its property values correspond to values. A complex entity is a database entity with identity and a non-empty set of *complex* properties, meaning, at least one of its property values is equal to another abstract entity. Examples of a simple and of a complex entity are given below:

**Example 19.** A *simple entity*  $\langle person1 \rangle$  may have the following properties:

$$\begin{aligned} Name(\langle person1 \rangle) &= \text{"Frank"}; \\ Age(\langle person1 \rangle) &= 29; \end{aligned}$$

A *complex hotel-reservation entity*  $\langle reservel \rangle$  may have the following properties:

$$\begin{aligned} Reserved\text{-}Room(\langle reservel \rangle) &= \langle room1 \rangle; \\ Guest(\langle reservel \rangle) &= \langle person1 \rangle; \\ Days\text{-}Reserved(\langle reservel \rangle) &= \{ \langle date1 \rangle, \langle date2 \rangle, \langle date3 \rangle \}; \end{aligned}$$

A *class* is formed by grouping together a collection of similar entities from the application domain. The class notion serves a dual purpose: it does not only represent a collection (*set*) of entities but also provides a generic (*type*) description for all entities belonging to that class. The term *type* refers to the collection of properties associated with all entities belonging to that class. All entities that are members of a class have a value for each property defined for the class; this value is either explicitly inserted by a database user or it is implicitly set to "unknown". Hence, a class imposes a *type* on its members. A property of a class is specified by:

$$\langle property\ name \rangle : \langle domain\text{-}class \rangle [ \langle characteristics \rangle ];$$

The domain class can be any user-defined class of the model or a predefined domain like the set of all integers. Characteristics are general descriptions of properties, for instance, whether a given property is *required* or *optional*. More on this is presented in Section 6.3.3. An example of the just introduced notation is given below.

**Example 20.** The class **Person** is defined by the following declaration:

```
class Person with properties:
  Name : String [identifying, required];
  Age : Integer [single-valued, optional];
end-class Person;
```

The class **Hotel-Reservation** is defined by the following declaration:

```
class Hotel-Reservation with properties:
  Reserved-Room : Room [single-valued, optional];
  Guest : Person [single-valued, required];
  Days-Reserved : Date [multi-valued, required];
end-class Hotel-Reservation;
```

The declarations of the **Date** and **Room** classes are not shown.

Analogous to entities, we distinguish between *value-based* and *abstract* classes. A *value-based class* contains values (entities without identity). For instance, the set of the integers in the range from 1 to 100 is a value-based class. An *abstract class*, on the other hand, consists of abstract entities (entities with identity). For instance, the classes **Person** and **Hotel-Reservation** in Example 20 are both abstract classes. An abstract class can consist of simple entities (like the **Person** class) or of complex entities (like the **Hotel-Reservation** class).

The following notation is used to refer to the properties of a class:

*< class name > . < property name >*.

We refer to the domain class of a property by:

**domain**( *< class name > . < property name >* ).

We use the following predicate to test whether a property is defined for a class or not:



$\langle \text{class name} \rangle . \langle \text{property name} \rangle ?$

**Example 21.** Assume the **Person** class as given in Example 20. We refer to the Name property of the class by **Person.Name** and to the domain of this property by **domain(Person.Name)**. In this example **domain(Person.Name)** is equal to the value-based class *String*. Furthermore, the predicate **Person.Name?** returns true because the Name property is defined for the class **Person**, whereas the predicate **Person.Friend?** returns false because the Friend property is not defined for the class **Person**.

**Definition 1.** We define an equality relation “=” :  $E \times E \rightarrow \{true, false\}$  with  $E$  the set of values and entities in the database as follows. For (abstract) entities  $\langle e1 \rangle$  and  $\langle e2 \rangle$  in  $E$ , “=” is an identity-based equality relation defined by:

$$\langle e1 \rangle = \langle e2 \rangle := \begin{cases} true & \text{if } \langle e1 \rangle .id = \langle e2 \rangle .id \\ false & \text{otherwise.} \end{cases}$$

For values  $e1$  and  $e2$  in  $E$ , this is a value-based equality relation defined by:

$$(e1 = e2) := \begin{cases} true & \text{if } e1 \text{ and } e2 \text{ represent the same string of symbols} \\ false & \text{otherwise.} \end{cases}$$

Each class has an associated membership predicate that determines the set of entities that are members of the class. This membership predicate, also called the *member-of* function, is based on the equality relation defined above, i.e., it utilizes the identity-based equality relation for abstract classes and the value-based equality relation for value-based classes [Rund93]. The predicate that an entity  $\langle e \rangle$  is a *member-of* a class  $C$ , denoted by  $\langle e \rangle \in C$ , evaluates to true if  $\langle e \rangle$  belongs to class  $C$  and to false otherwise.

An entity may take on values for different sets of properties when viewed as a member of different classes. For instance, a person will exhibit different characteristics when viewed as a spouse than as an employee. To refer to the properties of an entity as the participant in a particular class we use the following notation:

$\langle \text{property name} \rangle (\langle \text{entity reference} \rangle \text{ as } \langle \text{class name} \rangle)$ .

**Example 22.** Assume the **Person** class as given in Example 20. Then we can assign the value \$4,000 to the property Salary of the entity  $\langle \text{Jack} \rangle$  in class **Employee** by “Salary( $\langle \text{Jack} \rangle$  as **Employee**) := \$4,000”. The entity  $\langle \text{Jack} \rangle$  does not have

a Salary property when viewed as a member of the **Person** class (by Example 20), and thus the assignment "Salary(<Jack> as **Person**) := \$4,000" is illegal.

We define the *inheritance* of properties in the context of class derivation as follows.

**Definition 2.** Let *C1* and *C2* be two classes with class *C2* being derived from class *C1* using a database abstraction. If there is a property *p* defined in class *C1* that is also defined in class *C2*, then class *C2* is said to have **inherited** the property *p* from the class *C1*.

### 6.3.2 Class Derivation Operations

There are numerous types of class creation abstractions for abstract classes, such as specialization/generalization abstractions [Hull87], the aggregation abstraction, also called "part-of" relationship in object-oriented systems, and groupings found in semantic data models [Hamm81, Rund93]. The most common one is specialization, which is supported by virtually all object-based database systems, notably, SDM [Hamm81], TAXIS [Mylo80], and IFO [Abit87]. Specialization creates a subclass of an existing class in one of three ways: (1) by constraining the property description of an existing class (e.g., Red-Cars are defined as Cars with Color=Red), (2) by specifying an additional property on the derived class (e.g., Grad-Students are defined as Students with the additional property Type-of-employment), and (3) by explicitly collecting some elements to belong to it (e.g., Banned-Ships could be a class of Ships categorized by some criteria external to the data model).

The Cartesian aggregation abstraction is equally supported by most database systems [Smit77, Hamm81, Mylo80, Abit87]. It is an abstraction that allows a relationship between several entities to be viewed as a single aggregate (*complex* entity). Each element in the Cartesian class is taken from the *cross product* of existing classes and a new unique identity is associated with it. For example, the hotel-reservation class, introduced earlier in Example 20, could be modeled as a Cartesian aggregation on the three classes person, hotel room and date.

Abstract classes can also be derived by means of set operations, notably, union, set difference, intersection and symmetric difference. These mechanisms, which are much less common, are the main focus of the research presented in this chapter.

### 6.3.3 Characteristics of Properties

There are two special values, 'undefined' and 'unknown', a property may, in general, take on besides values from its domain class. The value 'undefined' means that the property is not defined for the given entity, while the value 'unknown' indicates that the property is defined but no value has been assigned to it. In particular, if a property  $p$  is not defined for a class  $C$  then the value of the property  $p$  will be 'undefined' for all members of the class  $C$ . Formally, this can be stated as follows:

$$(C.p?=false) \implies (\forall e \in C) (p(e \text{ as } C) = \text{'undefined'}).$$

If, on the other hand, a property  $p$  is defined for a class  $C$  then all members of the class  $C$  will take on some value (not equal to 'undefined') for the property  $p$ . This value will either be 'unknown' or an entity from the property's domain class. Formally, this can be stated as follows:

$$(C.p?=true) \wedge (\text{domain}(C.p)=D) \implies (\forall e \in C) ((p(e \text{ as } C)=\text{'unknown'}) \vee (p(e \text{ as } C) \in D) \vee (p(e \text{ as } C) \subseteq D)).$$

Most object-oriented and semantic data models [Hamm81, Hull87] associate some or all of the following characteristics with each class property:

1. *required* versus *optional*;
2. *identifying* versus *non-identifying*; and
3. *single-valued* versus *multi-valued*.

As indicated in Section 6.3.1, we associate these characteristics with each property in the class declaration (See Example 20).

The first characteristic distinguishes between *required* (mandatory) and *optional* (non-required) properties. Each entity of a class must have a value for all its *required* properties, i.e., a value not equal to 'unknown'. It may or may not have a value for the *optional* ones, i.e., its value may be 'unknown'. This *required/optional* characteristic is redefinable for a given property, i.e., a property  $p$  may be *mandatory* for some classes and *optional* for others.

The *identifying* versus *non-identifying* characteristic corresponds to the concept of a key in relational database theory. All properties of a class characterized

as *identifying* together uniquely identify the entities of the class. In relational terminology, if there is one property characterized as *identifying*, then it is called a single-valued key, and if there are two or more properties characterized as *identifying*, then they are called a composite key. The concept of keys is not as important in object-oriented data models since the underlying concept of object identities allows entities to be identified independently of their values. It can however still be used by users who wish to maintain their own unique values as entity references. For consistency reasons, a user should only designate a property to be *identifying* if it is also characterized as *required* [Rund93].

A property is defined to be either *single-* or *multi-valued* independent of the class for which it is initially introduced. This characteristic is not redefinable, i.e., it cannot be changed by classes that inherit this property. If a property  $p$  is *single-valued* then for any entity  $e$  of class  $C$ ,  $p(e \text{ as } C)$  has to be an element of  $\text{domain}(C.p)$  or be 'unknown'. If a property  $p$  is *multi-valued* then for any entity  $e$  of class  $C$ ,  $p(e \text{ as } C)$  is a subset of  $\text{domain}(C.p)$  or is 'unknown'.

We assume the *uniqueness of property values* throughout the entire schema: if two classes define the same property, then an entity that appears in both classes cannot have two distinct values for it. The property value is either 'unknown' in one of them or the two values are identical. There are various ways of enforcing this assumption in an implementation, for instance, by keeping only one copy (location) for each property value of an entity even if inherited by distinct classes.

For the purposes of subsequent discussions, the following simple *naming convention* is introduced, which guarantees that the names of all properties are unique throughout the entire schema: The name of a property is prefixed by the name of the class for which it is initially defined. Consequently, if a property is inherited from another class then its property name is prefixed by the name of the class in which it was originally defined. For instance, if the **Employee** class and the **Administrator** class both have a newly defined property called Salary, then the system refers to the **Employee's** property as **Employee.Salary** and the **Administrator's** property as **Administrator.Salary**. When no name ambiguities arise then this convention can of course be omitted - as, for example, in Figure 6.3.

#### 6.3.4 Class Relationships

In Chapter 3, we have given basic definitions of fundamental class relationships, such as subset, subtype and *is-a*. In this section, we now refine these definitions as needed for the remainder of this chapter.

Most existing systems ignore the set/type duality of the class construct and hence cannot provide clean semantics for set operations on classes. In the following, we emphasize this dual notion by studying the meaning of class relationships, which in the literature are generally referred to as *subclass/superclass* or *is-a* relationships. We disambiguate their meaning by distinguishing between two types of relationships:

- *subset/superset* relationships, and
- *subtype/supertype* relationships.

These two aspects of a class are not equivalent, i.e., a type relationship does not determine a set relationship, and vice versa. If an entity belongs to a class then this entity will necessarily be described by all properties of the type description of that class (which includes possibly 'unknown' values if some properties are characterized as optional). However, if an entity has all the properties of a class then this does not imply that the entity necessarily belongs to the class. Hence, properties are necessary but not sufficient conditions for a class membership.

The *subset relationship* between two classes is based on the identities of their entities as defined in Section 6.3.1. The *subset relationship* then is defined as follows.

**Definition 3.** The following set relationships can exist between two classes C1 and C2:

1. C1 is a **subset** of C2, denoted by  $C1 \subseteq C2$ , as defined by  $C1 \subseteq C2 \iff (\forall e1) (e1 \in C1) \implies (e1 \in C2)$  with the member-of predicate "∈" as defined in Section 6.3.1.
2. C1 is a **strict subset** of C2, denoted by  $C1 \subset C2$ , as defined by  $C1 \subset C2 \iff (C1 \subseteq C2 \text{ and } ((\exists e) (e \in C2 \text{ and } \text{NOT}(e \in C1))))$ .
3. C1 is **set equivalent** to C2, denoted by  $C1 \equiv^s C2$ , as defined by  $C1 \equiv^s C2 \iff (C1 \subseteq C2 \text{ and } C2 \subseteq C1)$ .
4. C1 is **set inequivalent** with C2, denoted by  $C1 \not\equiv^s C2$ , as defined by  $C1 \not\equiv^s C2 \iff (\text{NOT}(C1 \subseteq C2) \text{ and } \text{NOT}(C2 \subseteq C1))$ .

This definition of set relationships is based on object identity. It disregards the type description associated with the respective classes. Hence, the class C1 may have *more*, *fewer*, or *the same* number of attributes as C2. For example, the class **Students** in Figure 6.3 could be a subset of the class **Employees** (if every Student were working) - in spite of the fact that the elements of these two classes are described by different properties. This stands in contrast to conventional set theory where elements of the sub- and superset always look alike [Rund93].

The *subtype/supertype relationship* is concerned with the type description of classes and consequently with the state of all elements that participate in them. The *subtype* relationship between two classes is based on their type descriptions.

**Definition 4.** The different type relationships that can exist between two classes C1 and C2 are defined as follows:

1. If C1 and C2 are both value-based classes then we have:
  - (a) C1 is a **subtype** of C2, denoted by  $C1 \preceq C2$ , as defined by  $C1 \preceq C2 \iff (C1 \subseteq C2)$ .
  - (b) C1 is a **strict subtype** of C2, denoted by  $C1 \prec C2$ , as defined by  $C1 \prec C2 \iff (C1 \subset C2)$ .
  - (c) C1 is **type compatible** to C2, denoted by  $C1 \equiv^t C2$ , as defined by  $C1 \equiv^t C2 \iff (C1 \preceq C2 \text{ and } C2 \preceq C1)$ .
  - (d) C1 is **type incompatible** with C2, denoted by  $C1 \not\equiv^t C2$ , as defined by  $C1 \not\equiv^t C2 \iff (\text{NOT}(C1 \preceq C2) \text{ and } \text{NOT}(C2 \preceq C1))$ .
2. If C1 and C2 are both abstract classes then we have:
  - (a) C1 is a **subtype** of C2, denoted by  $C1 \preceq C2$ , as defined by  $C1 \preceq C2 \iff (\forall p) (C2.p?=true \implies C1.p?=true)$  and  $(\text{domain}(C1.p) \subseteq \text{domain}(C2.p))$ .
  - (b) C1 is a **strict subtype** of C2, denoted by  $C1 \prec C2$ , as defined by  $C1 \prec C2 \iff (C1 \preceq C2 \text{ and } ((\exists p) (C2.p?=true \text{ and } C1.p?=true \text{ and } (\text{domain}(C1.p) \subset \text{domain}(C2.p)))) \text{ or } ((\exists p) (C1.p?=true \text{ and } \text{NOT}(C2.p?=true))))$ .
  - (c) C1 is **type compatible** to C2, denoted by  $C1 \equiv^t C2$ , as defined by  $C1 \equiv^t C2 \iff (C1 \preceq C2 \text{ and } C2 \preceq C1)$ .
  - (d) C1 is **type incompatible** with C2, denoted by  $C1 \not\equiv^t C2$ , which is defined by  $C1 \not\equiv^t C2 \iff (\text{NOT}(C1 \preceq C2) \text{ and } \text{NOT}(C2 \preceq C1))$ .
3. If C1 is an abstract class and C2 is a value-based class, or vice versa, then C1 is **type incompatible** with C2, denoted by  $C1 \not\equiv^t C2$ .

If both classes are value-based, then the above definition distinguishes four cases depending on the set relationship of the values captured by the classes. If both classes are abstract, the above definition distinguishes four cases based on the type description associated with the classes. If C1 has more or the same number of properties than C2 and the same or more restricted property domains for its

properties than C2, then C1 is a subtype of C2 (case a). If C1 has more properties or more restricted property domains than C2, then C1 is a strict subtype of C2 (case b). If two classes C1 and C2 have identical properties and domains, then they represent *compatible* types (case c). If there is no type relationship between two classes, i.e.,  $C1 \preceq C2$  and  $C1 \succeq C2$  are both false, then we use the symbol  $C1 \not\equiv C2$  to denote their *type incompatibility* (case d). Finally, the two classes are also type incompatible if one is simple and the other is complex.

A subtype has all properties of its supertype and optionally some additional ones. Hence, a subtype has either *more* or *the same* number of properties than its supertype. The domain of the subtype properties may be equal to or contained in those of the corresponding properties of the supertype. For instance, the class Banned-Ships is a subtype of the Ships class and both have the same properties with the same domains. The type relation does not make any assumptions about the corresponding class memberships. Hence, theoretically, a subtype could have *more*, *fewer*, or *the same* number of elements as its supertype, or their instances may even be totally unrelated. The set of given class derivations and their semantics ultimately determine how type and set relationships interact within a given data model as will be shown in Section 6.4.

The term *is-a* relationship has been misused to mean many different things [Brach83]. We can now define the *is-a* relationship in terms of the two just defined class relationships.

**Definition 5.**  $C1 \text{ is-a } C2 \iff C1 \preceq C2 \text{ and } C1 \subseteq C2$ .

Informally, we say that C1 *is-a* C2 if (1) every member of C1 is an member of C2 (the subset relationship) and (2) every property defined for C2 is also defined for C1 (the subtype relationship) [Mylo80].

## 6.4 Set Operations in Object-Oriented Databases

### 6.4.1 Motivation

This section discusses how the set operations can be applied in object-oriented data models to create new classes<sup>1</sup>. We are concerned with set operations on only

---

<sup>1</sup>Set operations can be applied to either two abstract or two value-based classes; the mix of one abstract and one value-based class is however not meaningful.

abstract classes, since set operations on value-based classes correspond to the traditional set operations defined in set theory. These are well understood (see Section 6.2), since the underlying objects are values, i.e., are not typed and don't have any associated properties. Thus typing and related problems such as the inheritance of properties are not addressed in set theory. When dealing with conceptual data models, typing becomes an issue. An important distinction between sets in set theory and abstract classes in data modeling is that a set represents a collection of values whereas a class represents a collection of complex entities. In addition, it also provides their type description. Consequently, a well-defined set-theoretic operation on a class must specify the effect on both the type description and the resulting membership of that class.

The membership of a class derived by a set operation is based on the object identities of the involved entities [Rund93]. The resulting type description, however, is largely based on the properties defined for the original classes. The latter has no correspondence in conventional set theory, where a set is completely described by enumerating its members. Consequently, there is nothing in set theory to dictate the treatment of the type description of the resulting class. Other data models [Hamm81, Su86] have made certain (arbitrary) choices in this regard without giving a convincing argument to support their choice. The often raised point that an entity has a certain property and hence this property has to be reflected in the type description of the class it belongs to is not well founded. First, properties can be optional and, second, when viewed as member of a class the entity may grant access to only some of its properties, as was described in Section 6.3.

Determining the type description of classes derived by set operations is related to the issue of property inheritance. The difference is that the inheritance of properties usually takes place between *two* classes while set operations always deal with three classes [Abit87]. In other words, the definition of set operations on complex objects is similar to the problem of multiple inheritance. However, the literature assumes that the inheritance of properties takes place between classes that stand in an *is-a* relationship to one another [Hull87]. We will demonstrate that this is not necessarily the case for classes derived by set operations. As will be illustrated in Section 6.6, the latter assumption appears to be the reason for the limited use of set operations found in the literature. Below, we address the property inheritance problem by determining what characteristics properties of a derived class should have, once inherited. This leads to the development of general rules for property inheritance.

Definitions that cover the treatment of property values are given next. The following definitions are based on the naming convention and the assumption of the



uniqueness of property values, both of which are discussed in Section 6.3.3. The definition of the COMBINE function describes how a value of a particular property is to be constructed if it is inherited from more than one source. In particular, the values for property  $p$  are combined into one property attribute by collecting all property values that an entity takes on for property  $p$  while it is member of  $C1$  and/or  $C2$ .

**Definition 6.** Let  $p$  be a property. Let  $e$  be a member of the classes  $C1$  and/or  $C2$ . Let  $C3$  be the class derived from  $C1$  and  $C2$  using a set operation. Assume that the class  $C3$  has inherited the property  $p$  from  $C1$  and/or  $C2$ . For simplicity let us assume that if  $p$  is a single-valued property then  $p(e)$  corresponds to a singleton set rather than to an element. Then the function  $\text{COMBINE}: 2^V \times 2^V \rightarrow 2^V$ , where  $V$  is the domain of the property  $p$  or the value 'unknown' or 'undefined', determines the value for the property  $p$  of  $e$  in  $C3$ . It is defined by:

$$p(e \text{ as } C3) := \text{COMBINE}(p(e \text{ as } C1), p(e \text{ as } C2)) :=$$

$$\left\{ \begin{array}{l} p(e \text{ as } C1) \text{ if } (e \in C1 \wedge C1.p? \wedge p(e \text{ as } C1) \neq \text{unknown}) \wedge \\ \quad (e \in C2 \wedge C2.p? \wedge p(e \text{ as } C2) \neq \text{unknown}) \\ p(e \text{ as } C1) \text{ if } (e \in C1 \wedge C1.p? \wedge p(e \text{ as } C1) \neq \text{unknown}) \wedge \\ \quad (e \notin C2 \vee \text{NOT}(C2.p?) \vee (C2.p? \wedge p(e \text{ as } C2) = \text{unknown})) \\ p(e \text{ as } C2) \text{ if } (e \in C2 \wedge C2.p? \wedge p(e \text{ as } C2) \neq \text{unknown}) \wedge \\ \quad (e \notin C1 \vee \text{NOT}(C1.p?) \vee (C1.p? \wedge p(e \text{ as } C1) = \text{unknown})) \\ \text{unknown} \quad \text{otherwise} \end{array} \right.$$

In the first case, the property  $p$  is defined for both classes  $C1$  and  $C2$ , the entity  $e$  is a member of both classes, and it takes on known values for  $p$  as a member of both classes. By the uniqueness of property values assumption described in Section 6.3.3, this implies that  $p(e \text{ as } C1) = p(e \text{ as } C2)$ . Without loss of generality we assign one of the two to the combined property value. In the second case,  $p$  is defined for  $C1$ , the entity  $e$  is a member of  $C1$ , and it takes on a known value for  $p$  as a member of  $C1$ . However, either  $p$  is not defined for  $C2$ , the entity  $e$  is not a member of  $C2$ , or  $e$  does not take on a known value for  $p$  as a member of  $C2$ . Consequently, the value  $p(e \text{ as } C1)$  is inherited. Case three is analogous to case two with the classes  $C1$  and  $C2$  exchanged. If neither of the three cases are met, then the entity does not take on a value for the property  $p$  from the domain of  $p$  in either of the two classes  $C1$  and  $C2$ , and hence its resulting value is 'unknown'.

Next, operations on type descriptions are introduced that select among the set of properties to be inherited by a derived class.

**Definition 7.** Let  $C$  denote the set of all classes and  $P$  the set of all properties. Then the function  $Prop: C \rightarrow 2^P$  is defined by:

$$Prop(C1) := \{ p \mid C1.p? \}$$

with  $C1 \in C$ .

The function  $\sqcup : C \times C \rightarrow 2^P$  is defined by:

$$C1 \sqcup C2 := Prop(C1) \cup Prop(C2)$$

with  $C1, C2 \in C$ .

The function  $\sqcap : C \times C \rightarrow 2^P$  is defined by:

$$C1 \sqcap C2 := Prop(C1) \cap Prop(C2)$$

with  $C1, C2 \in C$ .

Intuitively,  $C1 \sqcup C2$  denotes the collection of *all* properties defined for either  $C1$  or  $C2$ .  $C1 \sqcap C2$ , on the other hand, consists of all properties common to the type description of both classes. We refer to the first operation as *property-collecting* (or *collecting*) and to the second one as *property-extracting* (or *extracting*).

## 6.4.2 Difference Operations

We propose two types of difference operations. The first is derived automatically by the system while the second is determined by the user by specifying the desired type description. The property values of all entities included in the newly derived class will automatically be calculated once the type description of the new class has been established. This is correct for all set-theoretic operations and parallels the situation of set operations in conventional set theory.

**Definition 8.** Let  $P$  be the collection of properties defined by  $P := Prop(C1)$ . The **(automatic) difference** of  $C1$  with respect to  $C2$ , denoted by  $C1 \dot{-} C2$ , is defined by

$$C1 \dot{-} C2 := \{ e \mid e \in C1 \wedge e \notin C2 \} \text{ and}$$

$$Prop(C1 \dot{-} C2) := P \text{ and}$$

$$(\forall e \in C1 \dot{-} C2) (\forall p \in P) ( p(e \text{ as } C1 \dot{-} C2) := p(e \text{ as } C1) ).$$

**Definition 9.** Again  $P := Prop(C1)$ . The **user-specified difference** operation of  $C1$  with respect to  $C2$ , denoted by  $C1 \tilde{-} C2$ , is specified by giving some  $Q \subseteq P$ . It is defined like the automatic difference except for replacing  $P$  with  $Q$ .

There is an important difference between the just presented user-specified set operation and the user-specified subclass mechanism commonly found in the literature [Hamm81]. Here, the user has to specify the type description once - namely during the creation of the derived class. Thereafter, the class can be instantiated automatically by the system according to the semantics of the applied set difference operation. Therefore, the level of user involvement is minimal. This contrasts strongly with the user-specified subclass mechanism where the user has to explicitly insert all entities into the class. The user-specified set operations can be classified as automatic class derivation mechanisms.

Given these set definitions, let us now study rules for the inheritance of property characteristics. We distinguish between single- and multi-valued properties. As described in Section 6.3.3, this characteristic, once defined, is fixed throughout the data model and thus does not change when a property is inherited by another class.

The second characteristic determines whether a property is identifying or not. The following simple rule is sufficient to describe the propagation of this characteristic from the base classes to the derived class.

**Lemma 1.** *Let  $P1$  and  $P2$  be (non-empty) sets of identifying properties for  $C1$  and  $C2$ , respectively. If  $C$  is a class resulting from the difference of  $C1$  relative to  $C2$  as given by Definitions 8 and 9 then  $P1$  will be identifying for  $C$  if inherited.*

The previous rule is self-explanatory. The result class will contain only entities from the class  $C1$ . Hence if a set of properties  $P1$  is sufficient to distinguish between all entities of  $C1$  then it will also be sufficient to distinguish between the ones of a subset of  $C1$ , i.e., the difference class. Next we address the third characteristic, which determines whether a property is required or non-required for a class.

**Lemma 2.** *The table in Figure 6.2 lists the propagation rules for the inheritance of the required/optional characteristics of a class derived by a difference operation.*

The table is to be read as follows. The symbol "required" refers to a required property, "optional" refers to a not required (but existing) property, and "-" means that the particular property is not defined for that class. The third column gives the characteristic of the inherited property in the derived class  $C1 \tilde{-} C2$  or  $C1 \tilde{=} C2$ , based on the characteristics of the corresponding property in  $C1$  and  $C2$  (first and

C1	C2	C1 - C2
optional	-	optional
required	-	required
-	optional	-
-	required	-
optional	required	optional
required	optional	required
optional	optional	optional
required	required	required

Figure 6.2: Inheritance of Property Characteristics for a Difference Class.

second column). The difference operations are not symmetric and hence the figure contains entries for all possible combinations. Every member of a difference class is also a member of C1. Therefore, all properties of C1 as well as their characteristics can be directly inherited by  $C1 \dot{-} C2$  or  $C1 \bar{-} C2$ . This explains why the third column of the table in Figure 6.2 is an exact copy of the first column.

Figure 6.3 demonstrates how the difference operation can be used in a conceptual data model to derive a new class from existing classes. The example in Figure 6.3 will also be used to show the result of the inherited type description as well as the membership of the derived class for all other set operations. The examples are given to show the usefulness of the proposed procedures for the propagation of characteristics.

**Example 23.** Figure 6.3 depicts the classes **Persons**, **Employees** and **Students**. The **Employees** and **Students** classes are subclasses (by *is-a* relationship) of the **Person** class. This *is-a* relationship is depicted by solid dark arrows. The class **Employees** has two required properties, which are the inherited property **Name** and the newly defined property **Salary**. The class **Students** has two required and one optional property, namely, the inherited property **Name** and the newly defined properties **Grade** and **Major**.

The difference operation  $C1 \bar{-} C2$  given in Definition 8 is applied to this conceptual data model. The result is a class that consists of all **Persons** who are **Employees** but not **Students**. Hence, all its elements are also members of the **Employees** class. Consequently, properties required for the **Employees** class, in this case, **Name** and **Salary**, are also required for the difference class. If the **Employees** class had optional properties, then those would still be optional for the newly created class. Properties of the **Students** class are not relevant to the resulting class since no entity of the result class would have any value for them. Hence, we do not include them in the resulting type description, and therefore choose the

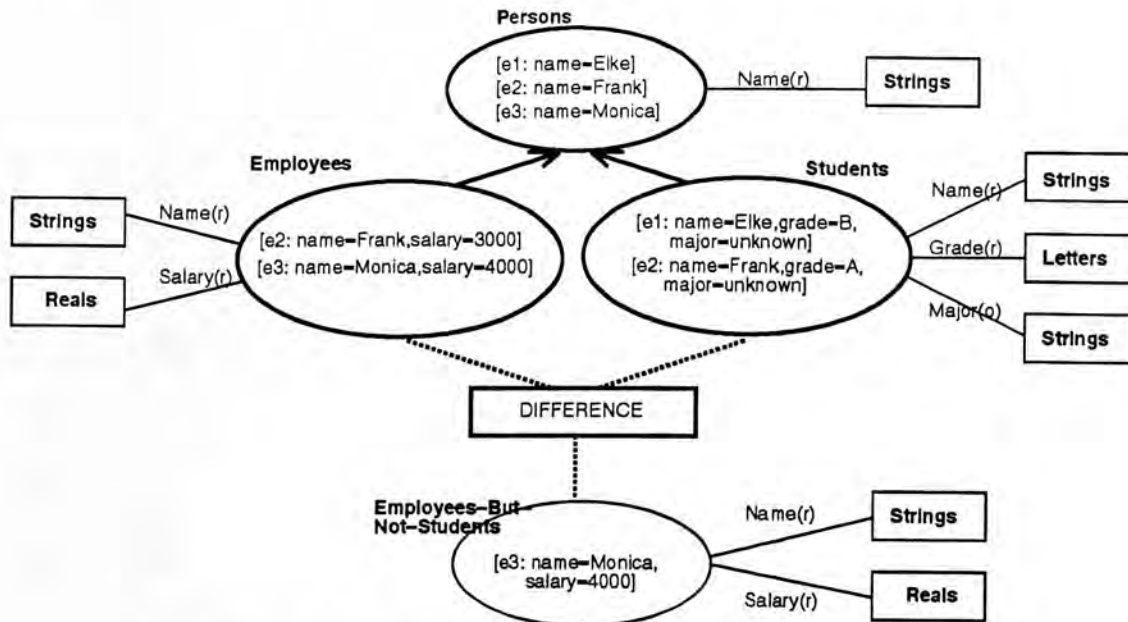


Figure 6.3: Derived Class Created by the Difference Operation.

difference operation  $C1 \dot{-} C2$  as given in Definition 8 over the operation  $C1 \dot{-} C2$  given in Definition 9.

### 6.4.3 Union Operations

Next, we distinguish three types of union operations. The type descriptions of the first two are derived automatically by the system, whereas the third one is determined by the user.

**Definition 10.** Let  $P$  be  $C1 \sqcup C2$ . The first union operation of  $C1$  and  $C2$ , denoted by  $C1 \dot{\cup} C2$ , is called the **collecting union**. It is defined by

$$C1 \dot{\cup} C2 := \{e \mid e \in C1 \vee e \in C2\} \text{ and}$$

$$\text{Prop}(C1 \dot{\cup} C2) := P \text{ and}$$

$$(\forall e \in C1 \dot{\cup} C2) (\forall p \in P) (p(e \text{ as } C1 \dot{\cup} C2) := \text{COMBINE}(p(e \text{ as } C1), p(e \text{ as } C2))).$$

**Definition 11.** The second type of union operation of  $C1$  and  $C2$ , denoted by  $C1 \hat{\cup} C2$ , is called the **extracting union**. The extracting union is defined as in the previous definition except for replacing  $P$  with  $P := C1 \sqcap C2$ .

**Definition 12.** The user-specified union operation of  $C1$  and  $C2$ , denoted by  $C1 \dot{\cup} C2$ , is defined by specifying a collection of properties  $Q$  with  $Q \subseteq C1 \sqcup C2$ . The definition of  $C1 \dot{\cup} C2$  is equivalent to the one in Definition 10 with  $Q$  substituted for the symbol  $P$ .

Note here that  $\dot{\cup}$  contains the other two union operations as special cases, since the user could choose the properties in the two cases as automatically derived by the system for  $\cup$  or  $\hat{\cup}$ .

Next, we study the general rules for property inheritance. As described in Section 6.3.3, the data model distinguishes between single- and multi-valued properties. This characteristic does not change when a property is inherited. Consequently, the rule of inheritance for this characteristic is trivial.

The second type of characteristic is whether a property is identifying or not. The following rule describes the propagation of this characteristic from the base classes to the derived class.

**Lemma 3.** *Let  $P1$  be a non-empty set of identifying properties for  $C1$  and  $P2$  a non-empty set of identifying properties for  $C2$ . If  $C$  is a class resulting from the union of  $C1$  and  $C2$  ( any of the three union types ) then  $P1$  together with  $P2$  will be identifying for  $C$  if both are inherited.*

The rule can best be explained with an example. Assume a situation similar to Figure 6.5 where the entities of the class **Students** are identified by the property **Student-Id** and the **Employee** entities by the property **Employee-Id**. Then, in the collection of **Employees** and **Students** each individual **Student** entity can be distinguished from all other **Student** entities by its **Student-Id** and from **Employees** entities by not having an **Employee-Id**. The converse is true for all **Employee** entities. If both properties are not inherited then some of the entities in the derived class may be indistinguishable to the user but the system is still able to distinguish them based on their object identities. This is so because object identities are globally unique identifiers maintained by the system [Rund93]. The example shows that if a database user intends to use some property as unique identifier (i.e., property is characterized as being identifying) then he or she has to declare it as a required property. Furthermore, the user has to use set operations that propagate this property to the derived class.

**Lemma 4.** *The table in Figure 6.4 lists the general inheritance rules for the required/optional characteristic when deriving a class by one of the three union operations.*

C1	C2	C1 $\cup$ C2
optional	-	optional
-	optional	optional
required	-	optional
-	required	optional
optional	optional	optional
required	optional	optional
optional	required	optional
required	required	required

Figure 6.4: Inheritance of Property Characteristics for a Union Class.

Note that a property can only be required if it has been required for both base classes. In all other cases, it cannot be guaranteed that all entities will take on a value for a property and hence they can only be asserted as optional. Next, an example is given to demonstrate this inheritance mechanism.

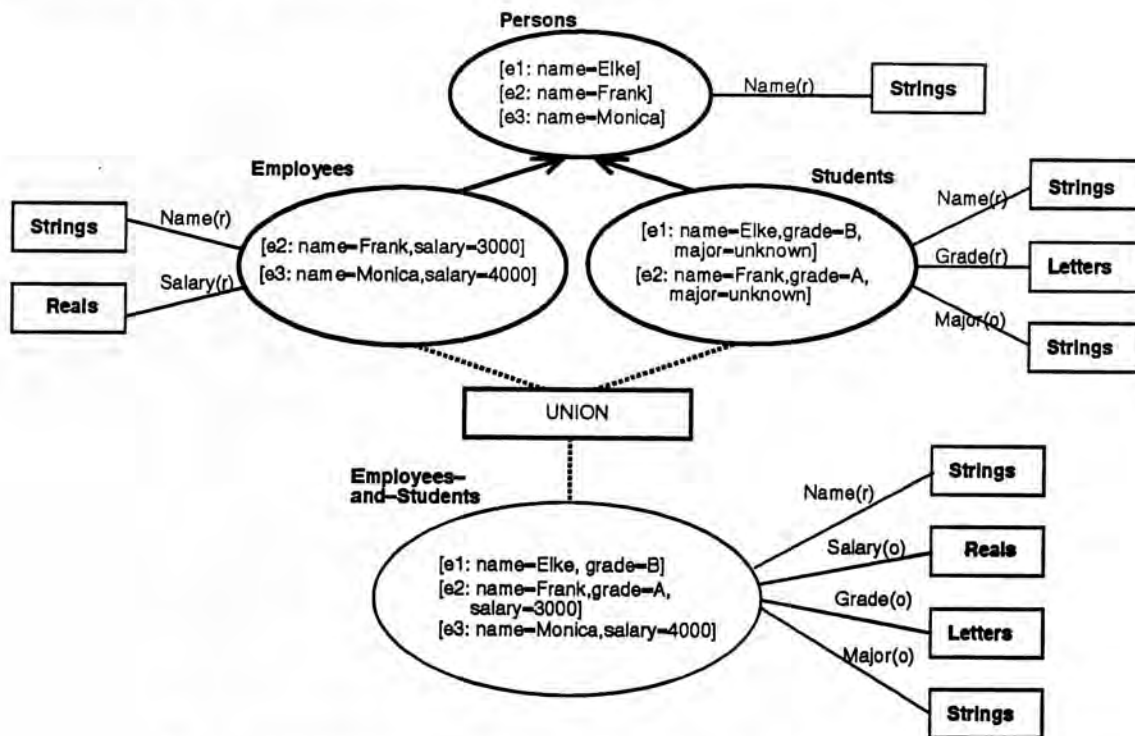


Figure 6.5: Derived Class Created by the Union Operation.

**Example 24.** In this example we describe how the union operation  $C1 \cup C2$  given in Definition 10 can be applied to the conceptual data model presented in Figure 6.3. The result is shown in Figure 6.5. The new class consists of all those elements that are members of either the **Employees** or the **Students** class. Properties that were required for both classes can still be required for the resulting class, e.g., the

inherited property Name. However, properties that were required for only one of these two classes can no longer be required. They can at best be optional in the new class. This is so since, for example, the person e1 does not have a value for the Salary property, even though the Salary property is required for the Employee class. The optional attributes must stay optional.

#### 6.4.4 Intersection Operations

Next we propose three types of intersection operations similar to the three unions. The first two again are automatically derived by the system. The third one is user-specified. The latter contains the other two as special cases.

**Definition 13.** Let  $P$  be  $C1 \sqcup C2$ . The intersection of  $C1$  and  $C2$ , denoted by  $C1 \tilde{\cap} C2$ , is called the **collecting intersection**. It is defined by

$$C1 \tilde{\cap} C2 := \{e | e \in C1 \wedge e \in C2\} \text{ and}$$

$$Prop(C1 \tilde{\cap} C2) := P \text{ and}$$

$$(\forall e \in C1 \tilde{\cap} C2) (\forall p \in P) (p(e \text{ as } C1 \tilde{\cap} C2) := COMBINE(p(e \text{ as } C1), p(e \text{ as } C2))).$$

Recall that if a property  $p$  is defined for both classes  $C1$  and  $C2$  then an entity that takes on values for  $p$  in both classes will have the same value in both cases.

**Definition 14.** Let  $P$  now be  $C1 \sqcap C2$ . The second type of intersection of  $C1$  and  $C2$ , denoted by  $C1 \hat{\cap} C2$ , is defined as in Definition 13 with the new meaning for  $P$ . It is called the **extracting intersection**.

**Definition 15.** The **user-specified intersection** operation of  $C1$  and  $C2$ , denoted by  $C1 \tilde{\cap} C2$ , is defined by specifying a collection of properties  $Q$  with  $Q \subseteq C1 \sqcup C2$ . The definition of  $C1 \tilde{\cap} C2$  is equivalent to the one in Definition 13 with  $Q$  substituted for the symbol  $P$ .

The effect of the intersection operation on the characteristics of properties is evaluated next. Again, the single- and multi-valued property is fixed throughout the data model and therefore does not change when a property is inherited (Section 6.3.3).

The identifying characteristic can be propagated from the base classes to the derived class by the following rule:



**Lemma 5.** *Let  $P1$  and  $P2$  be non-empty sets of identifying properties for  $C1$  and  $C2$ , respectively. If  $C$  is a class resulting from any of the three just defined intersection operations of  $C1$  and  $C2$  then  $P1$  or  $P2$  will be identifying for  $C$  if inherited.*

Again, the rule is self-explanatory. The result class will be a subset of both  $C1$  and  $C2$ . Thus, if  $P1$  is sufficient to distinguish between the entities of  $C1$  then it is sufficient to uniquely identify them when they appear in a subset of  $C1$ , i.e., the derived class. The same is true for  $P2$ .

**Lemma 6.** *Figure 6.6 gives the inheritance rules for the required/optional characteristic of properties defined for an intersection class.*

C1	C2	$C1 \cap C2$
optional	-	optional
-	optional	optional
required	-	required
-	required	required
optional	optional	optional
optional	required	required
required	optional	required
required	required	required

Figure 6.6: Inheritance of Property Characteristics for a Intersection Class.

To summarize, a property can be required for the result class if it is required for at least one of them. In all other cases, the property can only be asserted to be optional. The following example uses the previous definition for property inheritance.

**Example 25.** In Figure 6.7 the intersection operation  $C1 \cap C2$  of Definition 14 has been applied to derive a new class. The new class consists of all **Persons** who are **Employees** and **Students** at the same time. Properties that were required for either of the two classes are also required for the resulting class, i.e., the inherited properties Name, Salary and Grade. This is a sensible rule since all members of the intersection class will be guaranteed to take on values for these properties. The optional attribute Major can still be optional.

### 6.4.5 Symmetric Difference Operations

Next we introduce three types of symmetric difference operations. In all three cases, the entities of the result class will come from exactly one of the two base classes. Consequently, no merging of properties by means of the COMBINE operation is needed – not even for multi-valued properties.

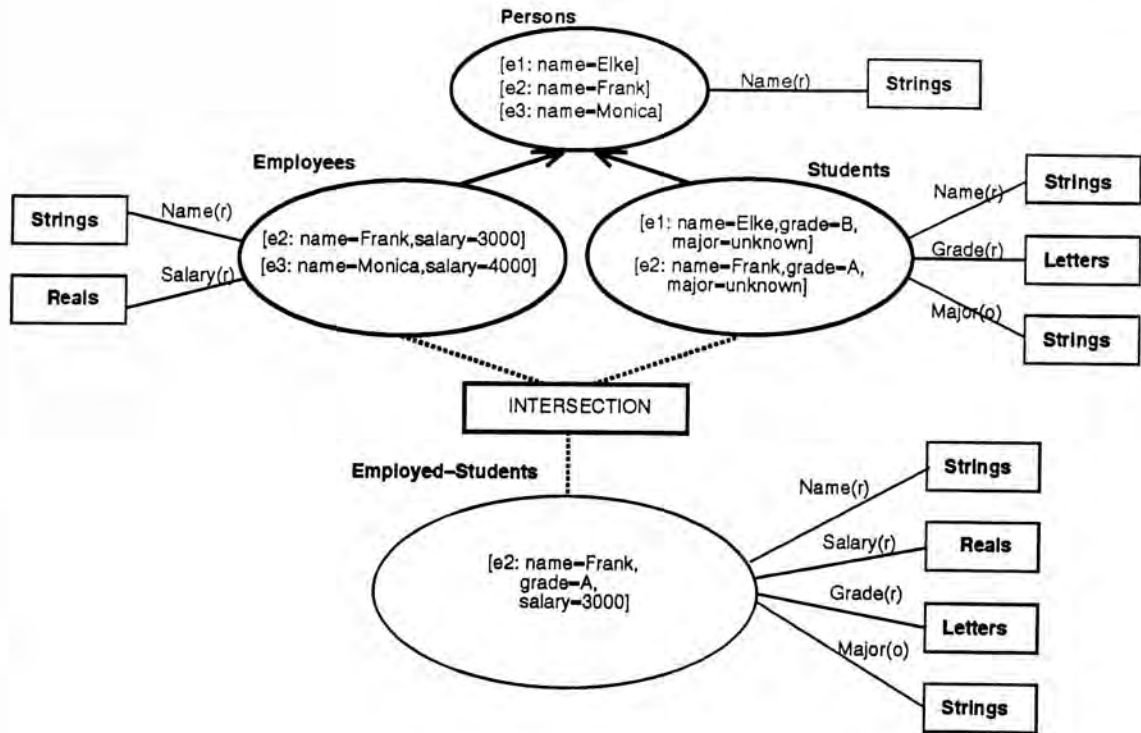


Figure 6.7: Derived Class Created by the Intersection Operation.

**Definition 16.** Let  $P$  be  $C1 \sqcup C2$ . The symmetric difference of  $C1$  and  $C2$ ,  $C1 \dot{\Delta} C2$ , is called the **collecting symmetric difference**. It is defined by

$$C1 \dot{\Delta} C2 := \{e \mid (e \in C1 \wedge e \notin C2) \vee (e \notin C1 \wedge e \in C2)\} \text{ and}$$

$$Prop(C1 \dot{\Delta} C2) := P \text{ and}$$

$$(\forall e \in C1 \dot{\Delta} C2) (\forall p \in P)$$

$$((e \in C1 \wedge C1.p? \Rightarrow p(e \text{ as } C1 \dot{\Delta} C2) := p(e \text{ as } C1)) \wedge$$

$$(e \in C2 \wedge C2.p? \Rightarrow p(e \text{ as } C1 \dot{\Delta} C2) := p(e \text{ as } C2)) \wedge$$

$$(((e \in C1 \wedge \text{NOT}(C1.p?)) \vee (e \in C2 \wedge \text{NOT}(C2.p?))) \Rightarrow p(e \text{ as } C1 \dot{\Delta} C2) := \text{unknown})).$$

Note that in the previous definition for all entities  $e$  of the result class at most one of the three predicates will be true since either  $e \in C1$  or  $e \in C2$  but not both.

**Definition 17.** Let  $P$  be  $C1 \sqcap C2$ . The symmetric difference of  $C1$  and  $C2$ ,  $C1 \hat{\Delta} C2$ , is called the **extracting symmetric difference**. It is defined by

$$C1 \hat{\Delta} C2 := \{e \mid (e \in C1 \wedge e \notin C2) \vee (e \notin C1 \wedge e \in C2)\} \text{ and}$$

$$Prop(C1 \hat{\Delta} C2) := P \text{ and}$$

$$(\forall e \in C1 \hat{\Delta} C2) (\forall p \in P)$$

$$((e \in C1 \implies p(e \text{ as } C1 \hat{\Delta} C2) := p(e \text{ as } C1)) \wedge$$

$$(e \in C2 \implies p(e \text{ as } C1 \hat{\Delta} C2) := p(e \text{ as } C2))).$$

In the previous definition, the properties in  $P$  are defined for both  $C1$  and  $C2$  and hence do not have to be tested.

**Definition 18.** The **user-specified symmetric difference** operation of  $C1$  and  $C2$ , denoted by  $C1 \tilde{\Delta} C2$ , is defined by specifying a collection of properties  $Q$  with  $Q \subseteq C1 \sqcup C2$ . The definition of  $C1 \tilde{\Delta} C2$  is equivalent to the one in Definition 16 with the symbol  $P$  replaced by  $Q$ .

For the same reasons mentioned earlier, the user-specified symmetric difference operation contains the other two automatic symmetric difference operations as special cases.

As described in Section 6.3.3, the single- and multi-valued property characteristic is fixed throughout the data model. Therefore, when a property is inherited it simply keeps its characteristic.

The second type of characteristic is whether a property is identifying or not. The following rule is sufficient to describe the propagation of this characteristic from the base classes to the derived class.

**Lemma 7.** *Let  $P1$  and  $P2$  be non-empty sets of identifying properties for  $C1$  and  $C2$ , respectively. If  $C$  is a class resulting from a symmetric difference of  $C1$  and  $C2$ , then  $P1$  together with  $P2$  will be identifying for  $C$  if both are inherited by  $C$ .*

The rule of inheritance described in the previous rule can be justified by an argument similar to the one given for the union operation (rule 5).

**Lemma 8.** *The inheritance of property characteristics for a derived symmetric difference class is defined by the rules described in the table of Figure 6.8.*

Again, a property can only be required in the new class if it has been required for both classes. In all other cases, it cannot be guaranteed that all entities of the result class will take on values for these properties. This is shown in the next example.

C1	C2	$C1 \Delta C2$
optional	-	optional
-	optional	optional
required	-	optional
-	required	optional
optional	optional	optional
optional	required	optional
required	optional	optional
required	required	required

Figure 6.8: Inheritance of Property Characteristics for a Symmetric Difference Class.

**Example 26.** The symmetric difference operation  $C1 \Delta C2$  of Definition 16 is used in Figure 6.9. It results in a derived class that consists of all **Persons** who are **Employees** but not **Students**, or **Students** but not **Employees**. Only properties that are required for both classes are required by rule 8. All others are optional. This is so since members of the result class will be exactly of one of the two types. For instance, the Salary property required for the entities of the **Employees** class could not be required in the result class since Student members of the latter would not have a value for it, and vice versa. The person e1 in Figure 6.9 for example, does not have a value for the Salary property.

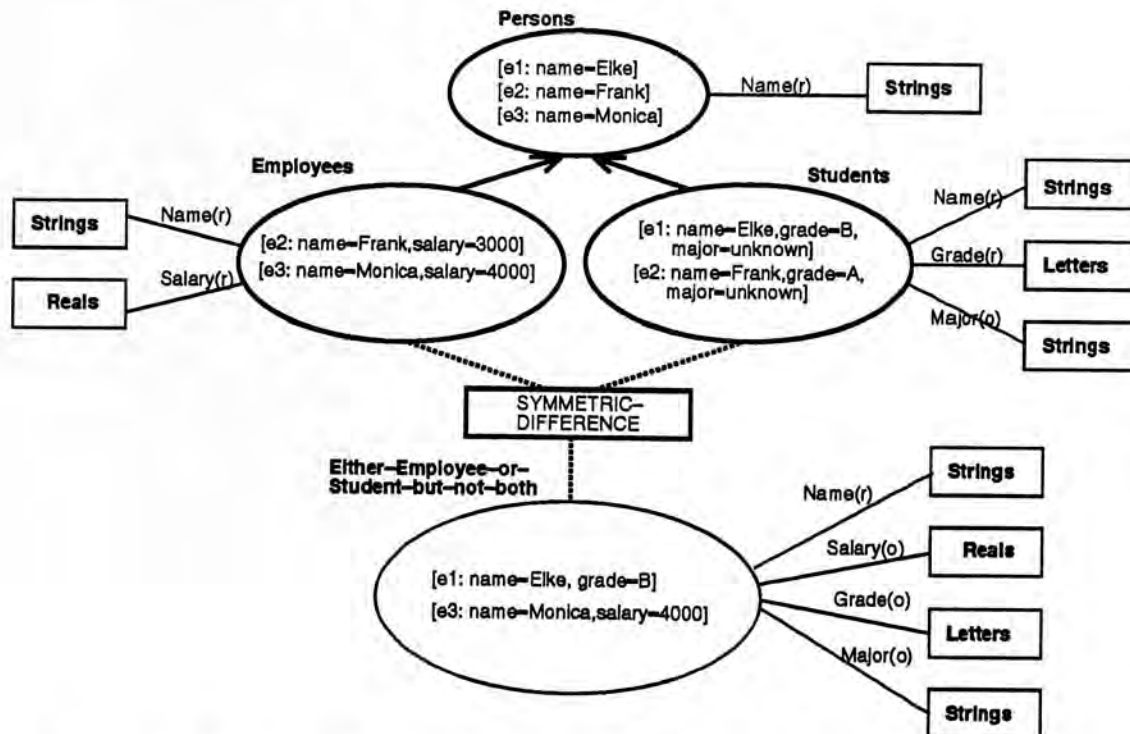


Figure 6.9: Derived Class Created by the Symmetric Difference Operation.

## 6.5 Examples of Class Derivation

In this section, we present examples that demonstrate how the set-theoretic operations defined in this section can be used for class derivation. The examples presented throughout Section 6.4 were used to explain the propagation of property characteristics from the existing to the derived class, and therefore have all been based on *collecting* set operations. We now present an example of class creation using *extracting* set operations.

**Example 27.** Figure 6.10 presents the extracting intersection and the extracting union operations on the Student/Employees schema example. The extracting intersection operation,  $\mathbf{Employees} \hat{\cap} \mathbf{Students}$ , results in the class **Employed-Students** with the type description “**Employed-Students.Name** [required]” and the content  $\{ [e2: \text{Name}=\text{Frank}] \}$ . The corresponding extracting union operation,  $\mathbf{Employees} \hat{\cup} \mathbf{Students}$ , results in the class **Employees-and-Students-1** with the same type description and the content  $\{ [e1: \text{Name}=\text{Elke}], [e2: \text{Name}=\text{Frank}], [e3: \text{Name} = \text{Monica}] \}$ .

In the examples thus far we have not yet depicted a property for an entity where the property value is ‘unknown’. In Figure 6.10 we repeat the *collecting* union operation on the Student/Employee schema from Figure 6.5. But this time, all its ‘unknown’ values are presented as well. Next, we present an example to compare *extracting* and *collecting* set operations.

**Example 28.** The example presented in Figure 6.10 contains the classes **Employees-and-Students-1** and **Employees-and-Students-2** derived by the *extracting* union and *collecting* union operations, respectively. Both contain the same entities; however, each class has a different type description imposed on its members. Depending on the intended use of the derived class, one class derivation operation may be preferred over the other. For instance, if we want to use this class to determine which student to assign a teaching assistant job, then it may be of advantage to know which of the students is already employed and what their respective salary is. Consequently, the *collecting* union would be chosen over the *extracting* union for this modeling situation.

An example of a *user-specified* set operation is given next.

**Example 29.** Assume the database designer is interested in **Students** who are employed but are good students nonetheless. In this case, the user may not be interested in details of their employment status. However, in order to determine

how good a student is, the Grade property would be relevant. Hence, the *user-specified* intersection operation  $\text{Employees} \cap \text{Students}$  combined with the chosen type specification of “Name” and “Grade” is the appropriate choice for creating the desired class. The derived class **Good-Employed-Students** (not shown in the figure) then has the content  $\{ [e2: \text{Name}=\text{Frank}, \text{Grade}=\text{A}] \}$ .

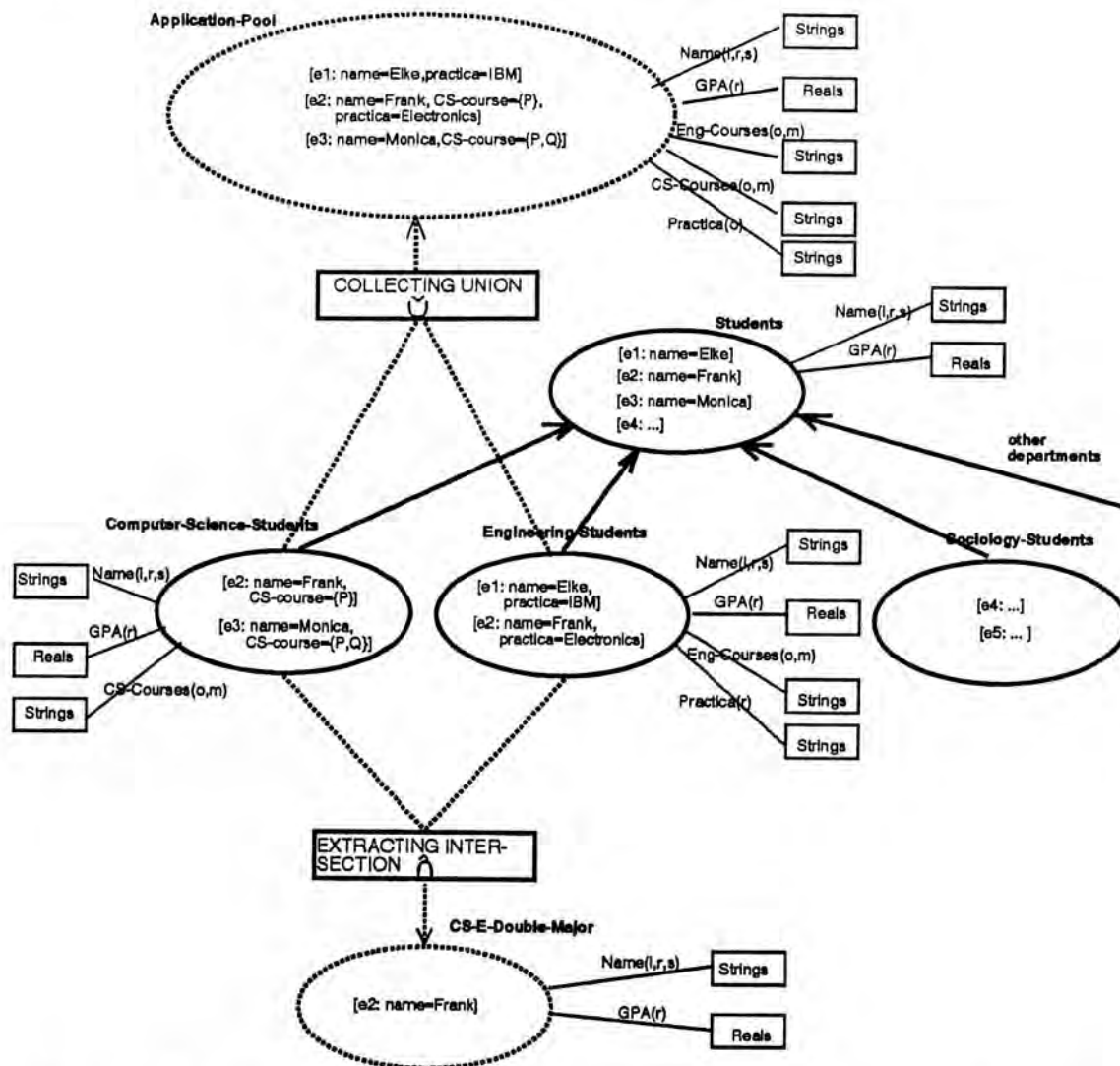


Figure 6.11: Examples of Set Operations Creating *Is-A* Incompatible Classes.

In the literature [Hamm81], the *extracting* union operation and the *collecting* intersection operation are generally assumed. Therefore, in the following examples we present an *extracting* intersection operation and a *collecting* union operation to demonstrate their usefulness.

**Example 30.** Figure 6.11 displays groups of Students organized according to their major, e.g, **Computer-Science-Students**, **Engineering-Students**, **Sociology-Students**, etc. They are specializations of the **Student** class. The *extracting* intersection operation is applied to the **Computer-Science-Students** and **Engineering-Students** to determine which students are double majors in Computer Science and Engineering, namely,  $\text{CS-E-Double-Major} = \text{Computer-Science-Students} \hat{\cap} \text{Engineering Students}$ . The database modeler is not interested in characteristics of students particular to each major but only in the fact that they are double majors. Hence in this case the **CS-E-Double-Major** class is built using the extracting intersection operation.

**Example 31.** Figure 6.11 also demonstrates the use of the *collecting* union operation. Assume that a company offers a new fellowship award that is open to Computer Science and Engineering students. Then the database user may form the class of all Computer Science and Engineering students using the union operation, i.e.,  $\text{Applicant-Pool} = \text{Computer-Science-Students} \dot{\cup} \text{Engineering Students}$ . In order to make a selection among the students, though, additional information on the students is needed. Consequently, the *collecting* union operation is chosen over the *extracting* one in order to preserve all information available on the students and to make it directly accessible from the **Applicant-Pool** class.

The last example utilizes the symmetric difference operation, a common operation in set theory, which generally has not been used for class creation in object-oriented data models.

**Example 32.** Consider a group of patients that have Aids and a group of patients that tested positive for the HIV virus, called **Aids** and **HIV-Positive**, respectively. It is suspected that there is a relationship between testing positive for the HIV virus and having Aids. Then the doctors are interested in determining whether this relationship is true, i.e., whether every HIV-infected patient has Aids and vice versa. In addition, they may want to run a set of experiments on those patients who do not obey this correlation. The desired set of patients is determined by the symmetric difference operation on the classes **Aids** and **HIV-Positive**, namely,  $\text{Aids} \hat{\Delta} \text{HIV-Positive}$ . We use the collecting  $\hat{\Delta}$  operation, since the doctors may be interested in studying all information that is available on the patients in either group. An empty set indicates a perfect correlation between the two groups.

## 6.6 An Analysis of Set Operations and the Class Relationships of Resulting Derived Classes

In this section we investigate what combinations of subset and subtype relationships as defined in Section 6.3.4 could occur within a data model supporting the set operations proposed in Section 6.4.

↓types sets→	$C1 \subset C2$	$C1 \equiv^s C2$	$C1 \supset C2$	$C1 \equiv^t C2$
$C1 \prec C2$	<i>is-a</i>	<i>is-a</i>	*	-
$C1 \equiv^t C2$	<i>is-a</i>	<i>is-a</i>	<i>is-a</i>	-
$C1 \succ C2$	*	<i>is-a</i>	<i>is-a</i>	-
$C1 \not\equiv^t C2$	-	-	-	-

**Legend:**

- "*is-a*" : *is-a* compatible relationship
- "\*" : *is-a* incompatible relationship
- "-" : neither of the two

Figure 6.12: Compositions of Set and Type Relationships.

For this, we first study all possible combinations of set and type relationships between two classes as defined in Definitions 3 and 4. In Figure 6.12, we display these combinations in a tabular manner. The combinations of set and type relationships that result in *is-a* compatible relationships as defined in Definition 5 are indicated by the label "*is-a*". Combinations of set and type relationships that result in *is-a* incompatible relationships are indicated by the label "\*". By *is-a* incompatible we mean having the set relationship between class C1 and class C2 in the reverse order from the type relationship between C1 and C2, e.g., C1 is a subset of C2 but, at the same time, C1 is a supertype of C2. Finally, the combinations of set and type relationships that are neither of the two, i.e., they don't result in *is-a* relationships but they are also not incompatible with the *is-a* relationship definition, are left unmarked. We can combine the results of either two rows or two columns of Figure 6.12 to gain additional information. For instance, the row with ( $C1 \prec C2$ ) and the row with ( $C1 \equiv^t C2$ ) can be combined to get a row for ( $C1 \preceq C2$ ).

Before analyzing the consequences of performing a set operation on abstract classes in terms of the subset/subtype combinations described in Figure 6.12, we need to differentiate between possible different effects of user-defined set operations on the type description of the resulting class. While the collecting and the extraction set operations determine automatically and exactly the type description of the derived class, the user-specified set operations leave it up to the user to decide on the desired type description of the derived class. This resulting type description could range from no properties to all properties of the original classes being inherited. For this reason,



we distinguish six possible cases of user-specified set operations depending on the choice of the desired type description. The different choices for the type description of the resulting class, denoted by  $Q$  in Definitions 8, 12, 15, and 18, result in distinct class relationships. Let  $P1 := Prop(C1)$  and  $P2 := Prop(C2)$ . Let  $\hat{P} := C1 \sqcup C2$  and  $\check{P} := C1 \sqcap C2$ . Recall that  $Q \subseteq \hat{P}$ . Then the choices for  $Q$  are listed below and are also shown graphically in Figure 6.13:

- case 1:  $P1 \subset Q$  and  $P2 \subset Q$ ;
- case 2:  $P1 = Q$ ;
- case 3:  $P1 \subset Q$  and  $\text{NOT}(P2 \subseteq Q)$ ;
- case 4:  $P1 \supset Q$  and  $\text{NOT}(P2 \supseteq Q)$ ;
- case 5:  $P1 \supset Q$  and  $P2 \supset Q$ ;
- case 6:  $P1 \not\equiv Q$  and  $P2 \not\equiv Q$ .

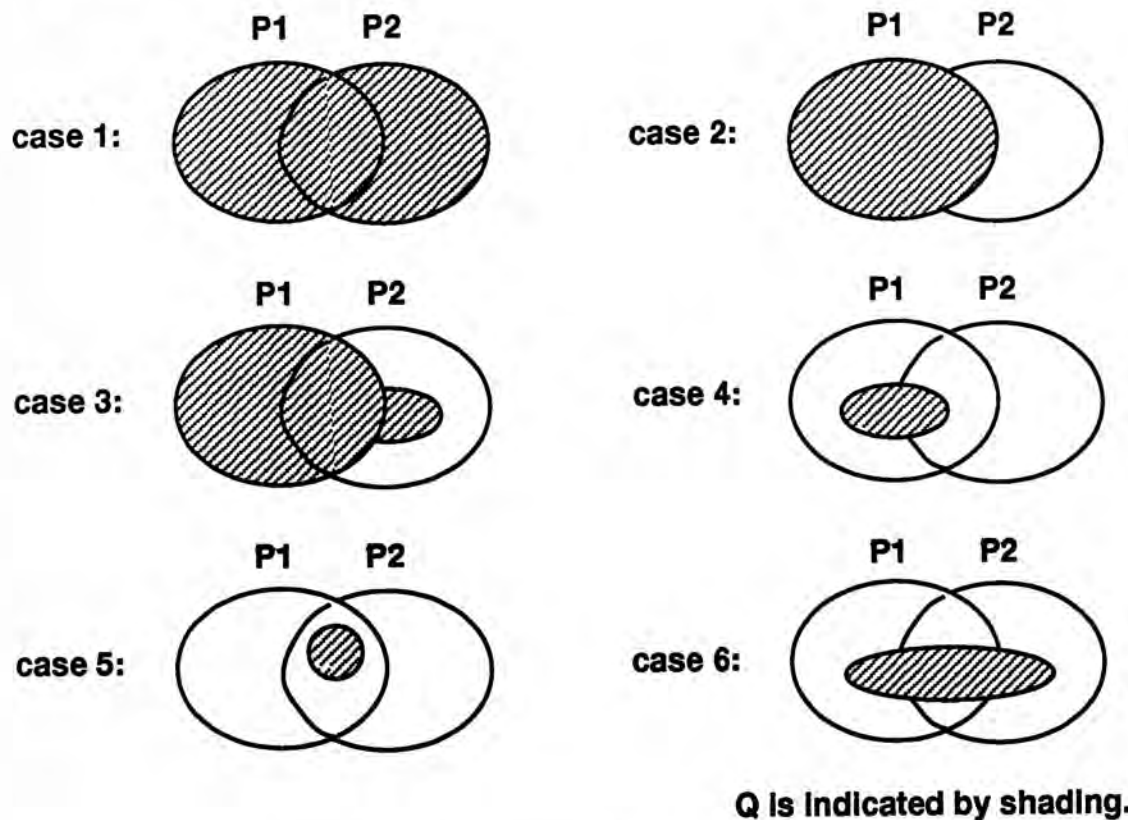


Figure 6.13: Six Cases of Type Descriptions.

Case 1 with  $(P1 \subset Q)$  and  $(P2 \subset Q)$  implies  $(Q = \hat{P})$ . Case 1 therefore models the situation of the first type of an automatic set operation ( the *collecting* type ) that collects all properties of  $C1$  and  $C2$ . In case 2,  $Q$  contains all properties defined

for C1 and none defined for C2 only. In case 3, Q contains all properties defined for C1 and some (but not all) properties defined for C2. Q of case 4 contains a subset of P1 and at least one of its elements is not in P2. The three set operations (excluding the difference operation) are symmetric and hence the analogous situation obtained by exchanging C1 and C2 is also covered by cases 2, 3 and 4. Case 5 with  $(P1 \supset Q)$  and  $(P2 \supset Q)$  implies  $(Q \subseteq \hat{P})$ . Case 5 includes  $Q = \hat{P}$ , i.e., the second type of an automatic set operation that *extracts* all properties common to both C1 and C2 as a special case. Case 6 illustrates the case where there is no set relationship between P1 and Q (and P2 and Q). In order for that to occur, Q must contain some (but not all) elements of P1 that are not in P2, and some (but not all) elements of P2 that are not in P1.

Finally, the analysis below shows the consequences of performing a set operation: both sets and types of the derived and original classes obey certain relationships as given in Figure 6.14. In particular, this figure lists the set and type relationships that hold between the two source classes C1 and C2 and the result class R derived by applying a set operation on them. In the fifth column in Figure 6.14 we mark the type of the resulting set and type relationship combination as defined in Figure 6.12. An "*is-a*" ("*\**") indicates cases that match (contradict) the requirements of an *is-a* relationship. For all other cases the fifth column is left unmarked. Cases marked with the asterisk each model a situation that could not appear in a database built by applying only specialization and generalization operations. Our framework, on the other hand, allows for the inheritance of properties between classes that are not necessarily *is-a* related. We know of no other data model that has this capability.

The first block represents the two set difference operations. Row 1 shows the results of the automatic difference (Definition 8), while rows 2 to 7 show the possible cases for the user-defined difference operation (Definition 9). Similarly the three other blocks show the results for the union, the intersection, and the symmetric difference operations, respectively. Note that in the table in Figure 6.14 we ignore degenerate cases. For instance, an example of a special situation for rows 24 to 31 is  $(C1 \subseteq C2)$  since it implies that  $(R \subseteq C2)$ . Another example is  $(C1 = \emptyset)$ ; it implies that  $(R = \emptyset)$ , which by default implies that  $(R \subseteq C1)$  and  $(R \subseteq C2)$ .

In the literature [Hamm81], the *extracting* union operation (row 9) is chosen over all other types of union operations. This table shows clearly the reason for this choice. First, the *extracting* union operation is an automatic operation that requires no further human interaction. Second, the *extracting* union operation results in an *is-a* relationship between the original and the derived classes. Similarly, our analysis reveals why the *collecting* intersection operation definition (row 16) was chosen over the one in row 17 in the literature. The reason again is that row 16 results in *is-a*

#	set operation	set relationships	type relationships	
1	$R = C1 \dot{\sim} C2$	$R \subseteq C1 \wedge$	$R \equiv^t C1$	<i>is-a</i>
2	$R = C1 \dot{\sim} C2$	$R \subseteq C1$	case 1: n/a	
3		"	case 2: $R \equiv^t C1$	<i>is-a</i>
4		"	case 3: n/a	
5		"	case 4: $R \succ C1$	*
6		"	case 5: $R \succ C1$	*
7		"	case 6: n/a	
8	$R = C1 \dot{\cup} C2$	$R \supseteq C1 \wedge R \supseteq C2$	$R \prec C1 \wedge R \prec C2$	*
9	$R = C1 \dot{\cup} C2$	$R \supseteq C1 \wedge R \supseteq C2$	$R \succ C1 \wedge R \succ C2$	<i>is-a</i>
10	$R = C1 \dot{\cup} C2$	$R \supseteq C1 \wedge R \supseteq C2$	case 1: $R \prec C1 \wedge R \prec C2$	*
11		" "	case 2: $R \equiv^t C1 \wedge R \not\equiv^t C2$	
12		" "	case 3: $R \prec C1 \wedge R \not\equiv^t C2$	*
13		" "	case 4: $R \succ C1 \wedge R \not\equiv^t C2$	
14		" "	case 5: $R \succ C1 \wedge R \succ C2$	<i>is-a</i>
15		" "	case 6: $R \not\equiv^t C1 \wedge R \not\equiv^t C2$	
16	$R = C1 \hat{\cap} C2$	$R \subseteq C1 \wedge R \subseteq C2$	$R \prec C1 \wedge R \prec C2$	<i>is-a</i>
17	$R = C1 \hat{\cap} C2$	$R \subseteq C1 \wedge R \subseteq C2$	$R \succ C1 \wedge R \succ C2$	*
18	$R = C1 \hat{\cap} C2$	$R \subseteq C1 \wedge R \subseteq C2$	case 1: $R \prec C1 \wedge R \prec C2$	<i>is-a</i>
19		" "	case 2: $R \equiv^t C1 \wedge R \not\equiv^t C2$	
20		" "	case 3: $R \prec C1 \wedge R \not\equiv^t C2$	
21		" "	case 4: $R \succ C1 \wedge R \not\equiv^t C2$	*
22		" "	case 5: $R \succ C1 \wedge R \succ C2$	*
23		" "	case 6: $R \not\equiv^t C1 \wedge R \not\equiv^t C2$	
24	$R = C1 \hat{\Delta} C2$	$R \not\subseteq C1 \wedge R \not\subseteq C2$	$R \prec C1 \wedge R \prec C2$	
25	$R = C1 \hat{\Delta} C2$	$R \not\subseteq C1 \wedge R \not\subseteq C2$	$R \succ C1 \wedge R \succ C2$	
26	$R = C1 \hat{\Delta} C2$	$R \not\subseteq C1 \wedge R \not\subseteq C2$	case 1: $R \prec C1 \wedge R \prec C2$	
27		" "	case 2: $R \equiv^t C1 \wedge R \not\equiv^t C2$	
28		" "	case 3: $R \prec C1 \wedge R \not\equiv^t C2$	
29		" "	case 4: $R \succ C1 \wedge R \not\equiv^t C2$	
30		" "	case 5: $R \succ C1 \wedge R \succ C2$	
31		" "	case 6: $R \not\equiv^t C1 \wedge R \not\equiv^t C2$	

Figure 6.14: Set Operations and Resulting Set, Type and Class Relationships.

relationships whereas the others don't. No violation of the *is-a* relationship can occur in the fourth block of Figure 6.14 since no type relationships hold. The complexity and diversity of the resulting relationships may partly be the reason for the lack of "user-specified" set operations in the literature.

Subset/superset relationships between classes resulting from the set operations proposed in this section (shown in Figure 6.14) are comparable with those in set theory. The subset relationships between base classes and the classes derived by the proposed set operations are identical to those that hold between base sets and derived sets in set theory. In other words, the semantics of traditional set operations have been preserved while utilizing the richness of the framework provided by advanced data models. Moreover, the type descriptions associated with the resulting classes does not have any effect on the resulting set relationships. To preserve the semantics of traditional set operations also means that the proposed set operations, when applied to classes that have value-based domains, return a result equivalent to that generated by traditional set operations when applied to the corresponding sets. This is indeed the case as can easily be seen from the definitions in Sections 6.3.4 and 6.4. In fact, all proposed variations on set operations collapse into one when applied to 'value-based domain classes' since the handling of types (based on which they are distinguished) becomes irrelevant.

These results are shown graphically in Figure 6.15. This figure presents class derivation using the proposed set operations as well as the subset/superset relationships between the original and the derived classes. The dotted lines in the figure indicate the derivation of the new classes from C1 and C2. The set operation symbols, like  $\cup$ , are used as generic operators, representing all types of the corresponding operations proposed in this section. Subset relationships are denoted by solid directed arcs with the arrow pointing from the subset to the superset.

We wish to emphasize that the use of set operations for class creation results in class relationships that would not exist in a database schema built solely by specialization and generalization abstractions.

## 6.7 Related Research on Set Operators

Early work on the problem of extended set operations by Childs [Chil68] has been published in the Proceedings of the IFIP Congress in 1968. Childs's work concerns the definition of a machine-independent data structure, called the Set-Theoretic Data Structure, which allows for fast execution of principal set operations

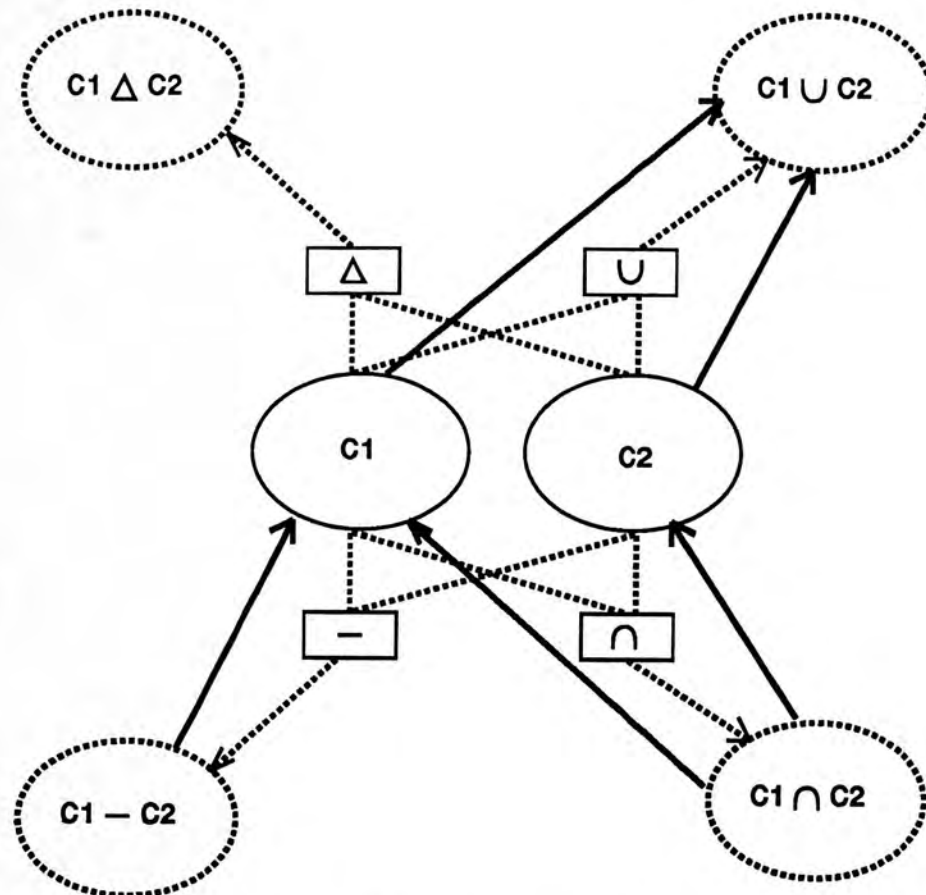


Figure 6.15: Set Relationships of Derived Classes.

on arbitrarily complex sets. This work however is not in the context of conceptual data models, and therefore does not address concepts, such as object identity, classes, complex objects, properties and the inheritance of property characteristics.

Some data models, in particular, most object-oriented models, define only type-oriented class operations. We use the term type-oriented to mean that these operations are applied to the type description of the original class and that the resulting membership of the new class is derived automatically. Others [Mylo80, Abit87] also include some limited repertoire of set operations on classes. Most of these approaches however are ad-hoc, as discussed below.

Hammer and McLeod [Hamm81] were among the first to propose different types of derivation mechanisms for subclasses. In SDM, they list four subclass connections, namely, attribute-defined, user-controllable, set-operator defined, and existence subclasses. However, their approach towards set-operator defined classes is limited in as much as only one of all possible interpretations is chosen for each set operation, namely, the *collecting* difference, the *extracting* union, and the *collecting* intersection. Our presentation in Section 6.6 makes it apparent that the ones preserving *is-a* relationships were selected. Our analysis in Section 6.6 also explains why the symmetric difference operation has not been utilized as a class derivation mechanism in SDM: it never results in an *is-a* relationship, as shown in Figure 6.14. Most other existing data models [Maie86] do not even consider the use of set operations.

SAM\* by Su [Su86] consists of seven different abstractions, referred to as association types, that construct new classes out of existing ones. One abstraction, called generalization, creates a more general concept type out of existing ones. This generalization corresponds to a form of union operation, however, it is not clear how the type description of the resulting concept type is formed. This ambiguity arises from the fact that SAM\* is value- rather than object-based. No set operations other than this union are considered in SAM\*.

Property characteristics, such as mandatory, single- or multi-valued, and others, have been proposed by several researchers [Hamm81, Mylo80]. Property inheritance has been studied extensively in the context of type-oriented class creation operations, such as specialization and generalization. However, to our knowledge no one discusses the effect of class derivations by set operations on property characteristics. Inheritance of properties and their characteristics is generally studied only between *is-a* related classes, i.e., for generalization and specialization abstractions.

## 6.8 Conclusions

The contributions of the work presented in this chapter are summarized below. First, the work presents sound definitions for set operations on the class construct. We show that the semantics of set theory are preserved by these definitions since the resulting set relationships between classes correspond to those of set operations in set theory. A class derivation mechanism would not be well-defined without the specification of the exact treatment of characteristics of inherited properties (especially, when inherited from more than one class). Consequently, we develop rules that regulate the inheritance of properties and their associated characteristics. These rules take care of required versus optional, identifying versus non-identifying, and single- versus multi-valued properties. In summary, this work provides the designers of a data model with a framework of set operations that allow them to make an explicit and educated choice among them.

We distinguish between *is-a*, subset, and subtype relationships. Our analysis of class relationships resulting from applying set operations sheds some light on the implicit assumptions concerning *is-a* relationships in the literature. Specifically, it is usually taken for granted that an *is-a* relationship must exist between base classes and a derived class. This assumption is unjustified since, for instance, the symmetric difference operation can never result in an *is-a* relationship (as shown in Figure 6.14). This problem has been avoided in other approaches by simply ignoring the existence of that set operation. As far as we know, the symmetric difference operation has never been utilized as a class derivation mechanism. Our approach, which allows for the symmetric difference operation, results in a data model where property inheritance proceeds along not necessarily *is-a* related class relationships.

Due to the generality of our approach, results of this work apply to any object-oriented data model that supports the class construct. We have ignored behavioral abstractions (methods) associated with classes of object-oriented systems [Fish87, Maie86], since their inclusion would not aid the understanding of the presented concepts. We believe, however, that much of this work can be extended to also include the behavioral aspect of classes.

# Chapter 7

## Automatic Schema Integration

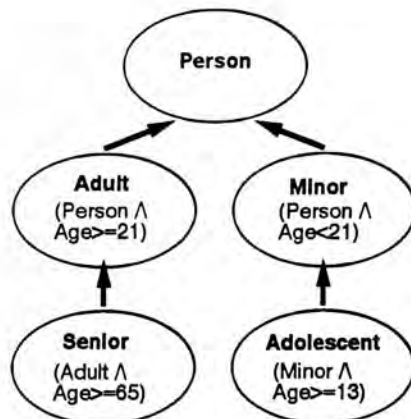
### 7.1 Towards One Comprehensive Global Schema

#### 7.1.1 Motivation

*MultiView* supports the integration of virtual classes created for different view schemata into one comprehensive global schema. Class integration is concerned with finding the most appropriate location in the schema graph for a given virtual class with the term ‘appropriate’ meaning correct in terms of property inheritance and subset relationships between classes (Definition 9). The derivation specification of a new class explicitly states some subsumption relationship between the source and the result class (as indicated in Section 5.2). For example, a virtual class created by the **select** operator is a subclass of its source class. I hence could place the result class of a selection as *direct subclass* of its source class. This placement is not semantically incorrect, but it results in an incomplete schema graph that does not capture all existing class relationships (Definition 9). Namely, there may be additional subsumption relationships between the derived class and other classes in the schema that are not directly derivable from the class derivation. It is the task of class integration to find these class relationships and to explicitly represent them in the schema graph. These ideas are illustrated with the example below (for which the class derivations used in Figure 7.1.a have been borrowed from ([Abit91], pg. 242)).

**Example 33.** *Figure 7.1.a depicts the specifications for deriving the virtual classes Adult, Minor, Senior, and Adolescent. Figure 7.1.a also depicts the resulting virtual class hierarchy that incorporates these four classes. This hierarchy that has been generated using the simplistic strategy of inferring the class placement directly from the definitions of the virtual classes [Abit91]. In this case, the simple placement strategy did result in a valid class hierarchy (Definition 9). In Figure 7.1.b, I present alternative derivations for the semantically equivalent set of classes. Note that the simple placement strategy now generates a different virtual class hierarchy (Figure*

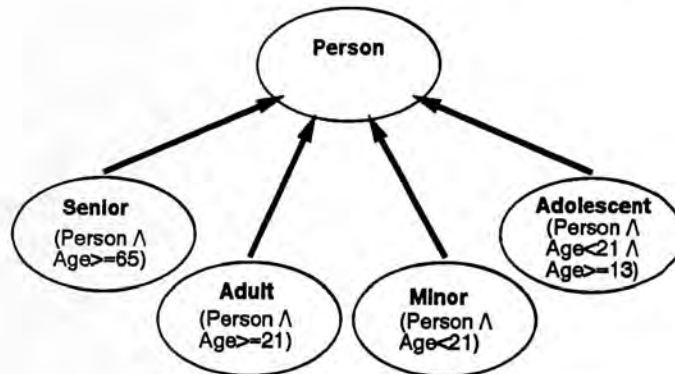




(a) Determine Class Placement Directly from the Class Derivation.

Class Derivations:

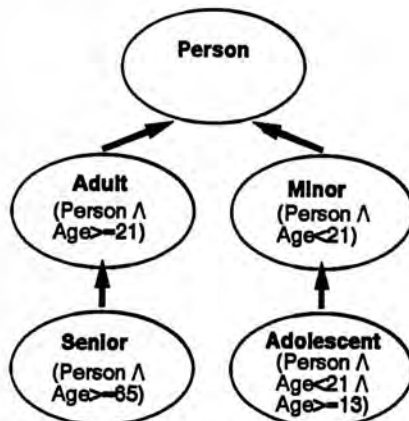
- (1) class Adult := SELECT (P:Person)  
where (P.Age ≥ 21)
- (2) class Minor := SELECT (P:Person)  
where (P.Age < 21)
- (3) class Senior := SELECT (A:Adult)  
where (A.Age ≥ 65)
- (4) class Adolescent := SELECT (M:Minor)  
where (M.Age ≥ 13)



(b) Determine Class Placement Directly from the Class Derivation.

Class Derivations:

- (1) class Adult := SELECT (P:Person)  
where (P.Age ≥ 21)
- (2) class Minor := SELECT (P:Person)  
where (P.Age < 21)
- (3) class Senior := SELECT (P:Person)  
where (A.Age ≥ 65)
- (4) class Adolescent := SELECT (P:Person)  
where (P.Age < 21 ∧ P.Age ≥ 13)



(c) Use Subsumption Inferencing to Determine Final Class Placement.

Class Derivations:

( same derivation as b. )

Figure 7.1: Complete Classification Versus Partial Classification.

7.1.b). This virtual class hierarchy is still consistent in the sense that it does not represent any incorrect is-a relationships (Definition 9). On the other hand, it is not complete since some subclass relationships, like, for instance, the relationship (Senior is-a Adult), are not explicitly captured in the structure of the schema graph. Clearly, the schema graph in Figure 7.1.b is less informative than the one in Figure 7.1.a. A classifier could determine these subclass relationships by comparing the class derivation predicates. For instance, predicate  $(Person \wedge Age < 21)$  of the class **Minor** clearly subsumes the predicate  $(Person \wedge Age < 21 \wedge Age \geq 13)$  of the class **Adolescent**. A complete classification of classes derived using the class derivations given in Figure 7.1.b would result in the schema graph shown in Figure 7.1.c. Note that the schema graph shown in Figure 7.1.c is identical to the one in Figure 7.1.a, with the exception of some of the class derivation predicates.

With the above example I want to convey that our goal is complete classification (Figure 7.1.c) rather than partial classification driven exclusively by the view-defining query (Figure 7.1.b).

Explicit capture of class relationships between stored and derived classes in the form of a global schema graph brings numerous advantages. It is for instance useful for sharing property functions and object instances among classes without unnecessary duplication. Since virtual classes are defined in terms of existing classes, they often use property functions of these base classes in their type description. Type inheritance between base and virtual classes thus becomes an issue - and this is exactly what is supported by global schema integration.

In addition, class integration serves data modeling purposes. Recall that one of the functions of an object schema is to explicitly model class relationships rather than having to recompute these relationships, whenever needed. Class integration follows this philosophy by organizing the concepts and objects of the application domain in a systematic manner such that they are more easily comprehensible by the users of the system.

Disadvantages of ignoring class integration would not only be the less informative class hierarchy but possibly also performance penalties. A known subclass relationship between two classes can be exploited by a query optimizer during query processing. For instance, if I know that C2 is a subclass of C1 then the union of the two classes C1 and C2 would be equal to C1. In this example, computationally expensive query processing has been replaced by a simple check of the existence of a subclass relationship among two classes. Moreover, insertion of an object instance into a class C1 automatically implies that the instance is also inserted in all superclasses of C1. If this class C1 is not placed at its optimal location (i.e., the

lowest possible place in the schema graph), then the membership predicate of other classes in the class hierarchy would have to be checked to determine whether the new instance is also a member of that class.

Last but not least, the comprehensive global schema graph is a necessary basis for the third subtask of *MultiView*, namely, for the formation of view schema graphs composed of both base and virtual classes. If the virtual classes are not integrated with the classes in the global schema, then a view schema would correspond to a collection of possibly 'unrelated' classes rather than a generalization schema graph as defined in Definition 15.

### 7.1.2 Towards a Simple Classification Algorithm of Virtual Classes

*Classification* is the process of taking a new [class] description and putting it where it belongs in the [class] hierarchy [Schm83]. A class is in the "right place" if it is below all classes that subsume it and if it is above all classes that it subsumes. I thus need to define a boolean function *subsumes()* that given two classes C1 and C2 will determine whether the first subsumes the second:

$$\textit{subsumes}(C1, C2) = \begin{cases} \textit{true} & \textit{if } C1 \textit{ is - a } C2 \\ \textit{fail} & \textit{otherwise} \end{cases}$$

Class C1 is said to subsume class C2 if and only if the set denoted by C1 *necessarily* includes the set denoted by C2. While exact details of the *subsumes()* function are dependent on the object model, I characterize its general features [Schm83]. The *subsumes(C1,C2)* function returns true if and only if the following holds:

- type description (both defined and inherited properties)
  - For each property function p1 defined for C1, there must be an equivalent function p2 defined for C2, where a property function could be a storable value, an object pointer, or a complex method. Since I assume for simplicity that property functions have a unique property identifier, two property functions are determined to be equivalent by comparing these identifiers rather than by comparing the actual function body, the later being a potentially undecidable problem.
- declarative constraints on property functions
  - The domain for each function p1 defined for C1 must subsume the domain of the equivalent function p2 defined for C2.

- Constraints for a property function  $p_1$  defined for  $C_1$  must include the constraints for the equivalent function  $p_2$  defined for  $C_2$ , where constraints could be cardinality restrictions, access modes, and the like.
- membership constraints
  - Membership constraints are predicates that restrict the set content of a class, i.e., this could be a subset-predicate for base classes or a derivation query for virtual classes.

Since a comparison of arbitrary expressions (possibly involving functions) is in general undecidable, one would either have to limit the expressiveness of the derivation specification such as to be computable, or, I could require a canonical predicate expression that can be broken into decidable subexpressions. In the later case, I would base the classification on the comparison of this partial information.

Clearly, more work is needed in each of these three issues. Details of a *subsumes()* function for the KL-ONE knowledge representation schema are for instance given in [Schm83]. It would of course also be interesting to develop efficient *subsumes()* functions for some of the newly emerging object-oriented data models. However, this investigation is beyond the scope of this dissertation.

In general, the classification problem is not decidable since it may involve the comparison of arbitrary functions and predicates. Therefore, our classification algorithm is sound but not complete. The *subsumes()* function being *sound* means that if the function returns true for a pair of classes then the two classes are necessarily *is-a* related. Put differently, any subsumption relationship discovered by the classifier is legitimate. Second, the *subsumes()* function is *total*, i.e., it always terminates and returns either true or fail. However, the *subsumes()* function is not *complete*, i.e., the function is not guaranteed to discover a relationship between two classes even if one exists. In other words, I cannot guarantee that all subsumption relationships are discovered. For instance, if two classes have property functions with equivalent semantics but different property identifiers, then the *subsumes()* function will fail even though these two classes may indeed be equivalent. In the worst case, if some *is-a* relationship is not discovered, then the virtual class is placed higher in the class hierarchy than would theoretically be possible. This would still be a correct but possibly not the most informative class arrangement. For example, Figure 7.1.b represents the result of such a partial classification whereas Figure 7.1.c shows the result of a complete classification.

Once I have developed the *subsumes()* function for a particular object model, then the basic algorithm for finding the correct position for a class VC in a schema

$G=(V,E)$  can be summarized as follows. First, the classifier determines the subsumption relationships between VC and all other classes in the global schema using the *subsumes()* function. VC then is placed below all its direct parent classes and above all its direct children classes (Definition 9). As I will show in later sections, this simplified classification algorithm does not correctly account for the type inheritance underlying the schema graph. In fact, I will describe two problems, called the type inheritance mismatch problem and the *is-a* incompatibility problem, that this simple class integration algorithm does not properly address. In the remainder of this chapter, I then present an algorithm for automatic classification that solves both problems.

### 7.1.3 Manual Placement Versus Automatic Classification

There are two obvious but not necessarily mutually exclusive approaches towards global schema integration:

1. I can require the view definer to manually place a newly defined virtual class into the global schema graph, and
2. I could develop an algorithm for automatic classification.

Clearly, class integration is required by the view methodology rather than being of direct modeling interest to the view definer. The view definer would have to be knowledgeable about all classes in the global schema, even those that are not related to his or her view schema, in order to correctly perform class integration. Also, as the size of the schema graph grows, class integration becomes a more and more involved process. For these reasons, I do not want to force the view definer to have to take care of this task. Instead, I suggest automatic classification. This would decrease the view creation time, and more importantly, it may allow a non-database expert to specify an application-specific view on his or her own.

Automatic class integration does not only simplify the task of the view definer, but it also prevents the introduction of inconsistencies into the schema. A view definer could easily place a class into an incorrect location or insert incorrect generalization arcs. A class VC is for instance placed incorrectly, if it is below a class C that is its subtype and/or its subset. The former would mean that the schema graph would incorrectly capture the fact that VC inherits property functions from C. The later, which corresponds to a mistake in set membership inclusion, would incorrectly represent VC as being a subset of C. A consequence of these errors would be an inconsistent global schema graph in terms of property inheritance and subset

relationships – and, since view schemata are defined on top of the global schema, also inconsistent view schemata. This inconsistency would not only confuse the user of the database system but it may also result in inefficiencies in terms of query processing and it may potentially lead to incorrect results.

Besides the assertion of blatantly incorrect *is-a* relationships, there is also the possibility of adding redundant or omitting required class relationships (Definition 9). If the view definer omits *is-a* arcs that are *required* to describe the complete semantics of the view schema, then this would leave some of the type inheritance that is actually taking place unexposed. If the view definer inserts *redundant is-a* arcs, then this may obscure the structure of the schema graph and it may result in inefficient query processing. The manual classification approach thus requires the development of a consistency checker that verifies the correctness of the user-inserted classes and arcs. This consistency checker would be equivalent in flavor (and in complexity) to the automatic classifier. Therefore I have opted to automate this process of classification. This decision does not necessarily prevent manual specification, if so desired. In fact, the automatic classifier could be used to guide the user during the view specification process, or, it could serve as basis of a consistency checker for manually-specified class placement.

To summarize, automatic classification provides a means of enforcing semantics and of checking for consistency of the class hierarchy. Therefore, automatic classification is a superior alternative to manual classification of the taxonomy.

#### 7.1.4 Road Map for the Rest of the Chapter

This section has outlined the basics issues of class integration. The rest of this chapter is organized as follows. Section 7.2 characterizes the two class integration problems, respectively. In Sections 7.3, 7.4 and 7.5, I present a solution to the first problem, while in Section 7.6, I present a solution to the second problem. In Section 7.7, the class integration algorithm is refined for virtual classes derived by the object algebra operators defined in Chapter 5. The comparison of *MultiView's* class integration approach with other work in the literature is given in Chapter 2.

## 7.2 Two Problems of Schema Integration

In Section 7.1 I have described a simple solution approach to (partial) class integration that has been advocated repeatedly in the literature [Schm83, Tana88,

Abit91]. In this section I will show that this simple classification approach does not appropriately handle classification in all cases. Based on our distinction between the type and the set content of a class as two independent concepts [Rund92b], we can characterize the two problems of class integration as follows:

1. the problem of inheritance mismatch in the type hierarchy, and
2. the problem of composing *is-a* incompatible subset and subtype hierarchies into one class hierarchy.

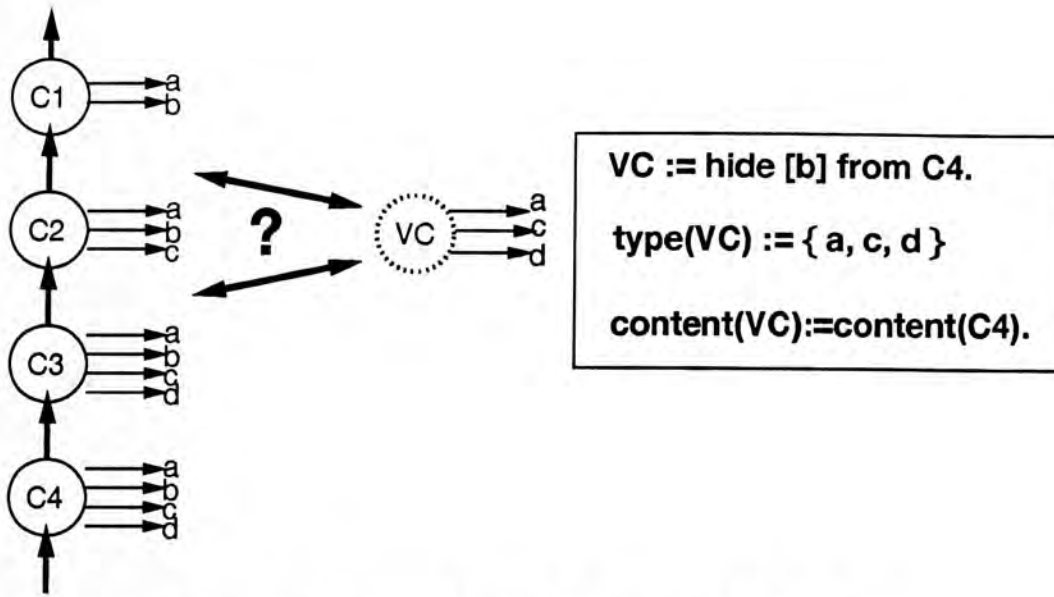
### 7.2.1 The Type Inheritance Problem

The first problem is concerned with constructing a type hierarchy that assures the proper inheritance of property functions after the insertion of new classes. As I will demonstrate below, in some cases there may be no correct placement for a class  $C$  in a given schema graph  $G$ . In such situations, I have to reorganize the schema graph such as to allow for the insertion of  $VC$ , while preserving the type inheritance for all existing classes. This problem is best explained with an example as is done in the following.

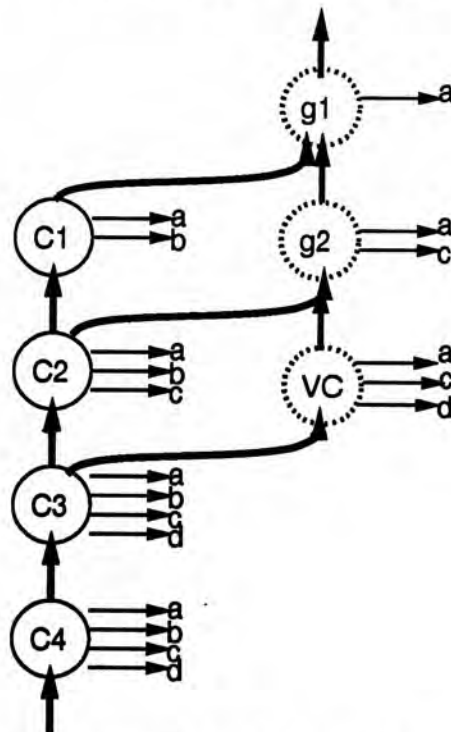
**Example 34.** *This example explains the type hierarchy mismatch problem based on Figure 7.2. Figure 7.2.a depicts the schema graph  $G$  and the virtual class  $VC$  derived by the query " $VC := \text{hide } [b] \text{ from } C_4$ ." Clearly,  $VC$  is a supertype (and superclass) of both  $C_4$  and  $C_3$ , and therefore must be placed above  $C_3$  and  $C_4$  in the class hierarchy. I cannot determine any subtype relationships between  $C_2$  and  $VC$ , i.e., neither  $(C_2 \preceq VC)$  nor  $(VC \preceq C_2)$  hold. This is so because even though  $C_2$  and  $VC$  share some common properties, each of them also has properties that are not defined for the other. Therefore,  $VC$  can be placed neither below nor above  $C_2$  in  $G$ . The same is true for  $C_1$  and  $VC$ . As a consequence, there is no correct location for  $VC$  in  $G$ .*

*In Figure 7.2.b, I present a solution to this problem. It is based on the idea of integrating intermediate classes into  $G$  that filter out the properties that are common to the existing classes in  $G$  and to the new class  $VC$ , so that they can be inherited by both. The intermediate class  $g_1$ , for instance, filters out the property function "a" so that it can be inherited by both the base class  $C_1$  and by  $VC$ . As shown in Figure 7.2.b, this extended class hierarchy now allows for the consistent integration of  $VC$  into  $G$ .*

In the example above, I assume that the set contents of all classes are identical, i.e.,  $\text{content}(C_1) = \text{content}(C_2) = \text{content}(C_3) = \text{content}(C_4)$ . This then clearly



(a) Problem: No proper place to integrate VC into G.



(b) Solution: Create intermediate classes for preserving type inheritance.

Figure 7.2: The Type Inheritance Problem.



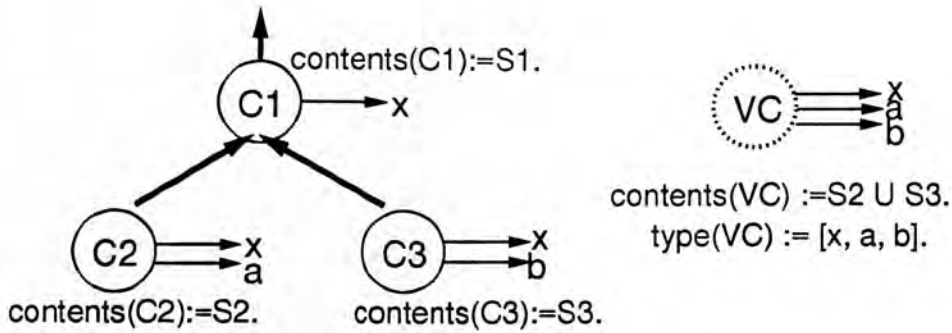
shows that the type inheritance mismatch is a problem of the type hierarchy alone - not dependent on the characteristics of the set hierarchy. In a later section, I will present a general solution to this problem based on work in type lattice classification.

### 7.2.2 The *is-a* Incompatibility Problem

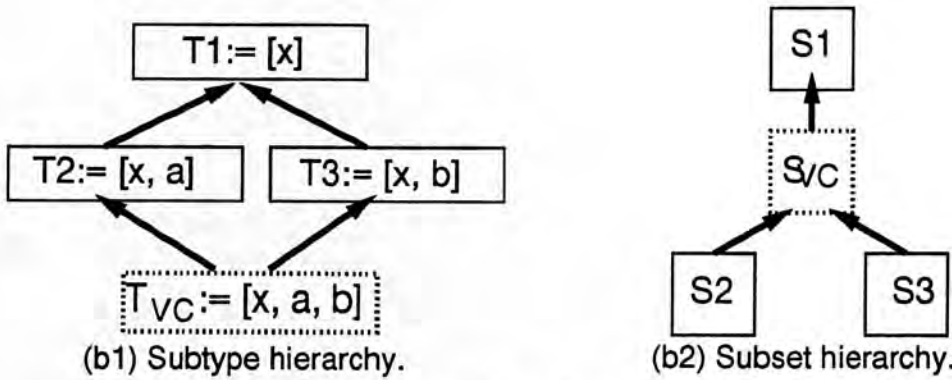
The second class integration problem results from the fact that I am combining the subset and the subtype relationships among classes into one relationship, called the *is-a* or subclass relationship. Differences between the subtype and the subset hierarchies underlying the class hierarchy may lead to conflicts when trying to compose these two hierarchies into one graph. More precisely, if a class's set content is lower in the corresponding set hierarchy and the class's type is higher in the corresponding type hierarchy, then there is a conflict in where to place the class in the combined class hierarchy. I can again solve this problem by reorganizing the schema graph such as to allow for the proper insertion of VC, while preserving the semantics of all existing classes. Below I explain the *is-a* incompatibility problem based on an example.

**Example 35.** *This example explains the problem caused by the incompatibility between subtype and subset relationships underlying a class hierarchy based on Figure 7.3. For this example, I assume that the type and the set content of a class  $C_i$  are denoted by the symbols  $T_i$  and  $S_i$ , respectively. Figure 7.3.a depicts the schema graph  $G$  and the virtual class  $VC$  with  $\text{content}(VC) := \text{content}(C2) \cup \text{content}(C3) = S2 \cup S3$  and  $\text{type}(VC) := \text{type}(C2) \sqcup \text{type}(C3) = T2 \sqcup T3 = [x, a, b]$ . When integrating  $VC$  into  $G$ , I must first determine the subtype and the subset relationships among all classes. As shown in Figure 7.3.b2,  $\text{type}(VC)$  is a subtype of  $T2$  and of  $T3$  and therefore must be placed below both of them in the subtype hierarchy. On the other hand, as shown in Figure 7.3.b1,  $\text{content}(VC)$  is a superset of  $S2$  and of  $S3$  and therefore must be placed above both of them in the subset hierarchy. This clearly represents a conflict since in the final class hierarchy  $VC$  needs to be above  $C2$  and  $C3$  as dictated by the subset relationships and below  $C2$  and  $C3$  as dictated by the subtype relationships. As a consequence, there is no correct location for  $VC$  in  $G$ . I say that  $VC$  is *is-a* incompatible with respect to  $G$ .*

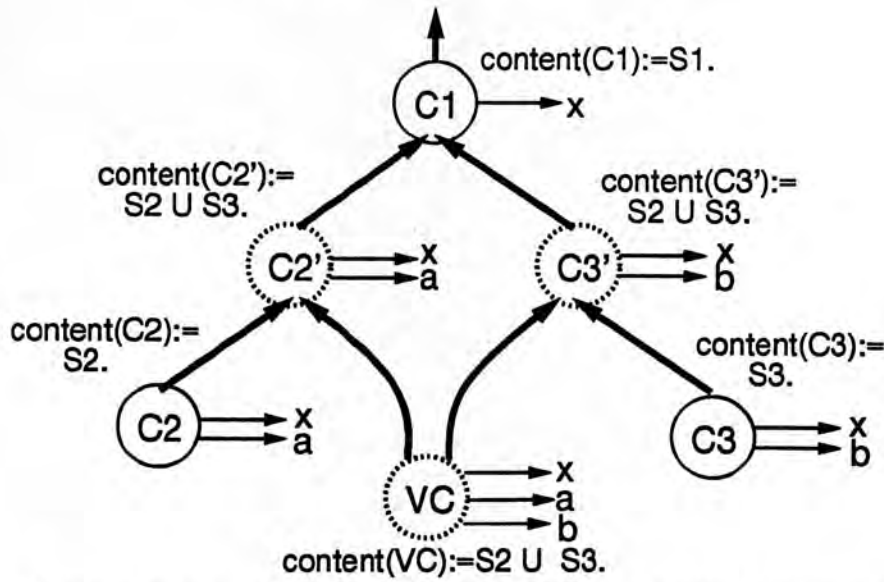
*In Figure 7.3.c, I present a solution to the *is-a* incompatibility problem based on the creation of additional intermediate classes. For this example, I create the two intermediate classes  $C_2'$  and  $C_3'$ . Both  $C_2'$  and  $C_3'$  have the types of their respective source classes  $C_2$  and  $C_3$ , such that  $VC$  can correctly inherit their combined type  $[x, a, b]$ . On the other hand,  $C_2'$  and  $C_3'$  both have a larger set content (namely, their*



(a) Integrating the is-a incompatible class VC into G.



(b) The Incompatibility Problem.



(c) Solving the Is-a Incompatibility Problem by Creating Intermediate Classes.

Figure 7.3: The *Is-a* Incompatibility Problem.

*set content is equal to the one of VC), so that VC can indeed be placed below them in the class hierarchy without violating the set hierarchy constraints.*

Note that in Example 35, the problem for class integration is not caused by the type hierarchy itself, but by composing a subclass hierarchy out of the subtype and the subset hierarchy (see Figures 7.3.b1 and 7.3.b2). If I assume that the set contents of all classes in Figure 7.3 were identical, then a correct position can be found for VC without intermediate class creation. In this case, the resulting class hierarchy would be equal to the type hierarchy graph shown in Figure 7.3.b1. In the remainder of this work, I present a class integration algorithm that solves the two problems characterized in this section.

## 7.3 A Solution to the Type Inheritance Problem

In this section, I present an approach to class integration that solves the type inheritance problem introduced in Section 7.2. This problem refers to the fact that in general there is not always a placement for a class C in a given schema graph G that results in correct type inheritance (see also Example 34). I thus am interested in a class hierarchy structure where the correct placement of a new class can always be enforced. Our solution to this problem is based on type lattice theory [Bouz84, Missi87]. More precisely, I present an algorithm that solves the problem by inserting additional intermediate classes that reorganize the schema graph such as to allow for the insertion of VC, while preserving the type inheritance for all existing classes. In the following, I assume that there are no conflicts between the subset and the subtype hierarchies underlying the class generalization graph. This problem of *is-a* incompatibility introduced in Section 7.2 will be dealt with in Section 7.6.

### 7.3.1 The Type Closure and Class Closure Properties

In our object model (as well as in most others), a property is defined exactly once and, if used elsewhere, it is inherited from this original definition class. A direct consequence of this is the fact that if two types  $t1$  and  $t2$  share some common properties, then they must have ultimately inherited them from the same type. As defined in Definition 4, this lowest common supertype of  $t1$  and  $t2$  must have all attributes common to  $t1$  and  $t2$  and no others. I then define the type closure property of a type hierarchy as stated below.

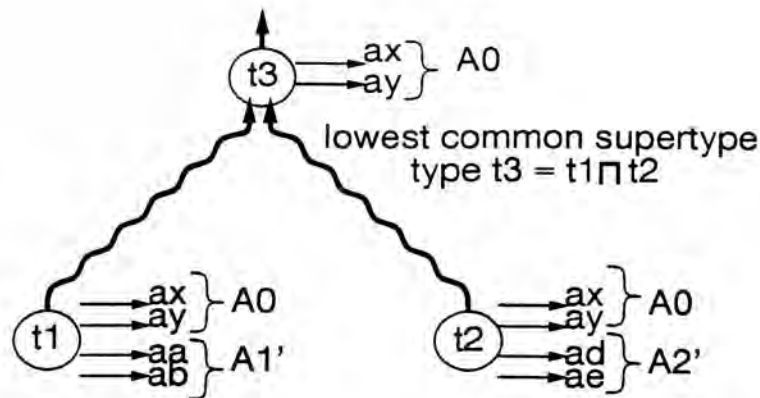


Figure 7.4: Lowest Common Supertype of Two Types in a Type Hierarchy.

**Definition 22. [Type Closure]** A type specialization hierarchy  $T$  is said to be closed under “ $\cap$ ” if and only if for any two types  $t1$  and  $t2$  in  $T$ , there exists a third type  $t$  in  $T$  which has exactly all properties that are common to  $t1$  and  $t2$ , i.e.,  $t = t1 \cap t2$  with the “ $\cap$ ” function defined in Definition 4.

Since a class hierarchy  $G$  has an underlying type hierarchy  $T$ , I now extend Definition 22 from types to classes.

**Definition 23. [Class Hierarchy Closure]** A class hierarchy  $G$  is said to be closed under  $\cap$  if and only if for any two classes  $C1$  and  $C2$ , there must exist a third class  $C3$  in  $G$  with

1.  $type(C3) = type(C1) \cap type(C2)$  and
2.  $content(C3) \supseteq contents(C1) \cup contents(C2)$ .

Definition 23 is a direct consequence of Definition 22. By Definition 22, for any two classes  $C1$  and  $C2$ , there must exist a third class  $C3$  in  $G$  with the type description of  $C3$  equal to the lowest common supertype of  $type(C1)$  and  $type(C2)$ . Since  $C3$  is a supertype of both  $C1$  and  $C2$ ,  $C3$  must be a superclass of  $C1$  and  $C2$  in the class hierarchy. This then implies the second condition; namely, the subset relationships  $C3 \supseteq C1$  and  $C3 \supseteq C2$  imply  $C3 \supseteq C1 \cup C2$ . If two types (classes) don't share any common attributes, then their common lowest supertype (superclass) is the general root type (class) of the hierarchy. It is fairly easy to see that if a class hierarchy is closed then there is no problem with the type inheritance.

For the remainder of this work, I assume that the type hierarchy and the class hierarchy are closed under the “ $\cap$ ” operator. I then will show that the lattice structure of the type hierarchy and of the class hierarchy can be maintained when

inserting new classes. Put differently, I will show how to assure correct type inheritance when inserting new classes. This approach is effective in the design of views for complex applications, since it is generally difficult to proceed linearly top-down (by first introducing the most general types and then repeatedly specializing them into several subtypes, etc.) or bottom-up (by introducing the most specific types and then repeatedly finding generalizations of existing types). On the contrary, when a new virtual class is being specified for a view schema, then the view definer has no knowledge about which (super)-classes will be required by other views later on. It therefore is more natural to introduce types at the intermediate level and to either specialize downward or generalize upward, whenever needed. Intermediate types generated by the system that are not of interest to the view definer can be dropped (hidden) at the end of the view definition process, while all others are kept explicitly.

### 7.3.2 Using the Closure Property for Class Integration

In this section, I want to determine how to keep a schema graph  $G$  closed after the insertion of a new virtual class  $VC$ , i.e., how to maintain its lattice structure. This is done by coercing the generation of the lowest common superclasses required by the closure property of the schema graph (Definitions 22 and 23).

I shortly explain the requirements of the closure property on class integration using an example before presenting a more formal treatment. In Figure 7.5.a, I depict a closed schema graph  $G$  and a virtual class  $VC$  to be inserted into  $G$ . Figure 7.5.b then demonstrates an attempt to integrate  $VC$  into  $G$ . Note that the resulting schema graph  $G'$  is not necessarily closed, since the lowest common superclasses  $g_i$  for the virtual class  $VC$  and each of the existing classes  $C_i$  in  $G$  may not exist in  $G'$ . Hence, in order for  $G'$  to be closed, I have to insert all required lowest common superclasses. In the example in Figure 7.5.c, for instance, I had to insert the lowest common superclasses  $g_1 = C3 \sqcap VC$  and  $g_2 = C7 \sqcap VC$ . On the other hand, I did not have to insert the lowest common superclass of  $C2$  and  $VC$ , since  $C2 \sqcap VC$  is equal to  $C1$  which obviously already exists in  $G$ .

**Lemma 9. [Necessity]** *Given a closed schema graph  $G=(V,E)$  and a new class  $VC$  that is to be integrated into  $G$ . Assume that the result of this integration is  $G'=(V',E')$ . By Definition 22, the following types  $t'_i$  must exist in  $G'$ :*

$$(\forall t_i \in G)(\exists t'_i \in G')(t'_i = t_i \sqcap \text{type}(VC))$$

*in order for  $G'$  to be closed.*

*By Definition 23, the following classes  $g_i$  must exist in  $G'$ :*

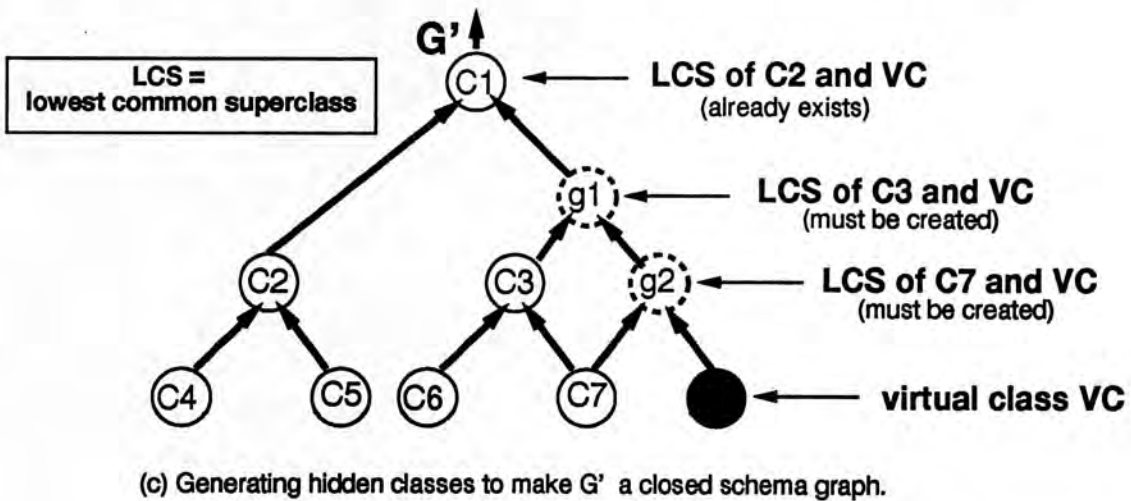
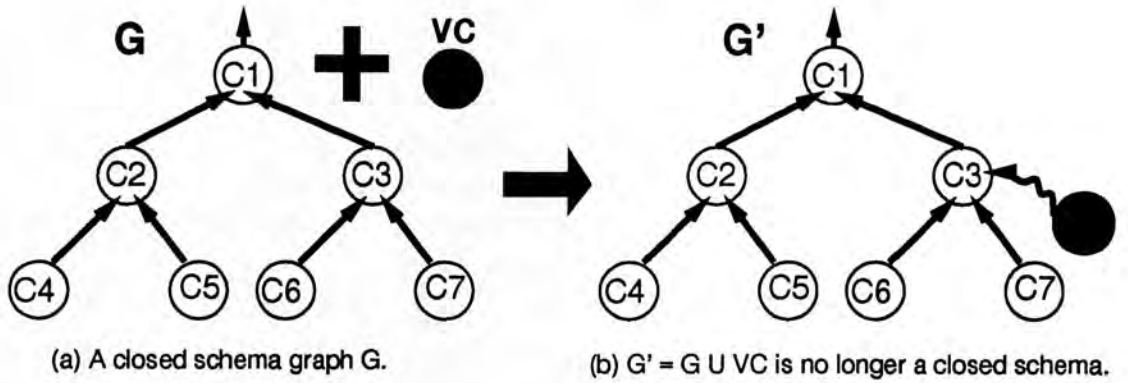


Figure 7.5: The Necessity of Creating Intermediate Classes.

$$(\forall C_i \in G)(\exists g_i \in G')(type(g_i) = type(C_i \sqcap VC) \wedge content(g_i) \supseteq content(C_i) \cup content(VC)).$$

in order for  $G'$  to be closed.

**Proof:** The first part of Lemma 9 follows directly from Definition 22, which states that for any pair of types  $t1$  and  $t2$  in  $G$ , their lowest common supertype  $t = t1 \sqcap t2$  must also exist in  $G$ . As a matter of course, this condition must hold for all pairs consisting of  $VC$  and any of the existing types. The second part of Lemma 9 follows directly from Definition 23, which states that for every pair of classes  $C1$  and  $C2$  in  $G$  their lowest common superclass  $C3$  must also exist in  $G$ . ■

Lemma 9 indicates the **necessity** of certain types (classes) to be created, when inserting a new class  $VC$  into a schema  $G$ . The creation of these new classes  $g_i = C_i \sqcap VC$  with  $C_i \in G$  may recursively cause the creation of additional classes, namely, the lowest common superclasses defined by  $(g'_i = C_i \sqcap g_i)(\forall C_i \in G)$ . Missikoff and Scholl [Missi89] prove that the first set of types  $g_i$  as specified in Lemma 9 is **sufficient** to guarantee the closure of the resulting type hierarchy. This work is directly applicable to our research, and as shown below I extend this sufficiency criteria from the type hierarchy to the class hierarchy.

**Lemma 10. [Sufficiency for Types]** *Given a closed type hierarchy  $TG=(TV,TE)$  and a new type  $t_{VC}$  to be integrated into  $TG$ . I denote the result of this integration by  $TG'=(TV',TE')$ . Then the integration of the types  $t'_i$  into  $TG'$  defined by*

$$GG = \{t'_i \mid t'_i = t_i \sqcap t_{VC} \text{ and } t_i \in TG \}$$

and

$$TV' = TV \cup GG \cup t_{VC}$$

will result in a closed type hierarchy  $TG'=(TV',TE')$ .

Lemma 10 states that integrating the newly generated types  $t'_i$  into  $G$  does not cause the generation of additional new types. It indicates that the types  $t'_i$  obtained by the first iteration are **sufficient** for assuring the closure of the resulting schema graph  $TG'$ . In particular, it suggests that the computation of the new types  $t'_i$  can be done in a single pass - without recursive iteration over the newly generated types  $t'_i$ . The proof for Lemma 10 can be found in [Missi89], and thus is not repeated here.

I am interested in extending this result from the type hierarchy to the class hierarchy. Note I now deal with a class, which is a two-facet concept consisting of both an associated type and a set content. I am hence concerned with two tasks:

First, the adjustment of the underlying type hierarchy such as to make it closed, and second, the determination of the correct set contents for these newly generated classes  $g_i$  such as to maintain the consistency of the schema graph.

**Lemma 11. [Sufficiency for Classes]** *Given a closed schema graph  $G=(V,E)$  and a new class  $VC$  to be integrated into  $G$  (Definition 22). The result of this integration defined by*

$$V'=V \cup VC \cup GG$$

with

$$GG = \{g_i \mid g_i = C_i \sqcap VC \text{ and } \text{content}(g_i) = \text{content}(C_i \cup VC) \text{ and } C_i \in G \}$$

represents a closed schema graph  $G'=(V',E')$ .

**Proof:** I divide the proof into the following three cases: (a) both classes in  $V$ , (b) one class in  $V$  and one in  $GG$ , and (c) both classes in  $GG$ .

**case a.** I show that for  $C_i \in V$  and  $C_j \in V$ , the class  $C_i \sqcap C_j$  already exists in  $V'$ .

This is true by assumption, since  $G=(V,E)$  being a closed schema graph implies that  $C_i \sqcap C_j$  in  $V$ . And,  $V'$  is a superset of  $V$ .

**case b.** I show that for  $g_i \in GG$  and  $C_j \in V$ , the class  $g_i \sqcap C_j$  exists in  $V'$ .

$$\begin{aligned} &g_i \sqcap C_j \\ &= (C_i \sqcap VC) \sqcap C_j && \text{by definition of } GG \\ &= (VC \sqcap C_i) \sqcap C_j && \text{by commutativity of } \sqcap \\ &= VC \sqcap (C_i \sqcap C_j) && \text{by associativity of } \sqcap \\ &= VC \sqcap C_k && \text{by } G \text{ closed, } \exists C_k = C_i \sqcap C_j \in V \\ &= g_k \in V' && \text{by definition of } GG \end{aligned}$$

This demonstrates that the newly generated classes  $g_i$  do not cause the generation of new classes when combining them with existing classes  $C_j$ , since the classes  $g_i \sqcap C_j$  are already in  $GG$ .

**case c.** I show that for  $g_i \in GG$  and  $g_j \in GG$ , the class  $g_i \sqcap g_j$  already exists in  $V'$ .

$$\begin{aligned} &g_i \sqcap g_j \\ &= (C_i \sqcap VC) \sqcap (C_j \sqcap VC) && \text{by definition of } GG \\ &= (VC \sqcap VC) \sqcap (C_i \sqcap C_j) && \text{by commutativity and associativity of } \sqcap \\ &= VC \sqcap (C_i \sqcap C_j) && \text{by idempotence of } \sqcap \\ &= VC \sqcap C_k && \text{by } G \text{ closed, } \exists C_k = C_i \sqcap C_j \in V \\ &= g_k \in V' && \text{by definition of } GG \end{aligned}$$



This demonstrates that newly generated classes  $g_i$  do not cause the generation of new classes when combining them with other newly generated classes  $g_j$ , since the classes  $g_i \sqcap g_j$  are already in  $V'$ .

The three cases together then prove that  $G'$  is closed. ■

Lemma 11 states that the computation of the intermediate classes  $g_i$  required for the closure of the schema graph  $G'$  can be done in a single pass – without recursive iteration over the newly generated types  $g_i$ .

### 7.3.3 Minimizing the Generation of Intermediate Classes

Next, I discuss how to limit the number of intermediate classes  $g_i$  generated for assuring the closure of a schema graph after class integration. This work is again based on [Missi89].

**Definition 24.** *Given a type hierarchy  $TG=(TV,TE)$  and a new type  $t_{VC}$  to be integrated into  $TG$ . Let  $\equiv_{t_{VC}}$  be an equivalence relationship defined by*

$$(\forall t_i, t_j \in TV)((t_i \equiv_{t_{VC}} t_j) \iff (t_i \sqcap t_{VC} = t_j \sqcap t_{VC})).$$

*Also I define the set of required lowest common supertypes in  $G$  with respect to  $\equiv_{t_{VC}}$  by*

$$GG = \{t'_i \mid t'_i = t_i \sqcap t_{VC} \wedge t_i \in G\}$$

*I divide the set of types  $TV$  of  $TG$  into equivalence groups  $G_i$  for  $i = 1, \dots, |GG|$  with*

$$G_i = \{t_j \mid ((t_{VC} \sqcap t_j) = t'_i) \wedge (t_j \in TV)\}$$

*with  $t'_i \in GG$  some fixed type per group  $G_i$ .*

Definition 24 defines two types to be equivalent with respect to VC if and only if both types have the same lowest common supertype with respect to VC. An equivalence group  $G_i$  then is composed of all types  $t_i$  that have the same lowest common supertype  $t'_i$  with respect to VC.

**Theorem 1.** *Given a closed type hierarchy  $TG=(TV,TE)$  and a new type  $t_{VC}$  to be integrated into  $TG$ . Let  $\{G_i\}$  denote the set of equivalence groups of  $G$  with respect to  $\equiv_{t_{VC}}$ . For each equivalence group  $G_i$ , there is one type  $t_i \in G_i$  that is **minimal and unique** in  $G_i$ .*

The proof of correctness for Theorem 1 can be found in [Missi89]. For a type  $t_i$  in  $G_i$  to be **minimal** means that it is a supertype of all other types in the equivalence group  $G_i$ . For  $t_i$  in  $G_i$  to be **unique** and **minimal** means that it is the only type that is a supertype of all other types in  $G_i$ , i.e., it is root type of the subgraph representing  $G_i$ . I denote this **unique** and **minimal** member  $t_i$  of  $G_i$  by  $\text{rep}(G_i)$ . Next, I extend Definition 24 and Theorem 1 from types to classes.

**Definition 25.** Given a schema graph  $G=(V,E)$  and a new class  $VC$  to be integrated into  $G$ . Let  $\equiv_{VC}$  be an equivalence relationship defined by

$$(\forall C_i, C_j \in V)((C_i \equiv_{VC} C_j) \iff (\text{type}(C_i \sqcap VC) = \text{type}(C_j \sqcap VC))).$$

Also I define the set of required lowest common supertypes in  $G$  with respect to  $\equiv_{VC}$  by

$$GG = \{t_i \mid t_i = \text{type}(C_i \sqcap VC) \wedge C_i \in G\}$$

I divide the set of classes  $V$  of  $G$  into equivalence groups  $G_i$  for  $i = 1, \dots, |GG|$  with

$$G_i = \{C_j \in V \mid (\text{type}(VC \sqcap C_j) = t_i) \wedge (C_j \in V)\}$$

with  $t_i$  some fixed type per group  $G_i$ .

**Theorem 2.** Given a closed schema graph  $G=(V,E)$  and a new class  $VC$  to be integrated into  $G$ . Let  $\{G_i\}$  denote the set of equivalence groups of  $G$  with respect to  $\equiv_{VC}$ . For each equivalence group  $G_i$ , there is one member class  $C_i \in G_i$  that is **minimal** and **unique** in  $G_i$ .

The proof of correctness for Theorem 2 can be directly derived from Theorem 1, and thus is omitted here. For a class  $C_i$  to be **minimal** in  $G_i$  means that it is a supertype and a superset of all other classes in the equivalence group  $G_i$ . For  $C_i$  in  $G_i$  to be **unique** and **minimal** means that it is the only class that is a superclass of all other classes in  $G_i$ , i.e., it is root class of the subgraph representing  $G_i$ . I denote this **unique** and **minimal** member  $C_i$  of  $G_i$  by  $\text{rep}(G_i)$ .

**Example 36.** In this example, I explain the concepts introduced in Theorems 1 and 2 based on Figure 7.6. In Figure 7.6, the two classes  $C_i$  and  $C_k$  are equivalent with respect to  $VC$ , since both have the same lowest common supertype  $[a,b]$ :  $\text{type}(C_i \sqcap VC) = [a,b] \sqcap [a,b,x] = [a,b]$  and  $\text{type}(C_k \sqcap VC) = [a,b,c] \sqcap [a,b,x] = [a,b]$ . All classes with this same lowest common supertype are equivalent and thus form an equivalence group. For instance, the classes  $C_i, C_l, C_k, C_n$ , and  $C_j$  have the same lowest common supertype  $[a,b]$ , and they form the equivalence group  $G_i$ . The class  $K_j$  does not belong to the equivalence group  $G_i$ , because  $\text{type}(K_j \sqcap VC) = [a,b,x] \neq [a,b] = \text{type}(\text{rep}(G_i))$ .

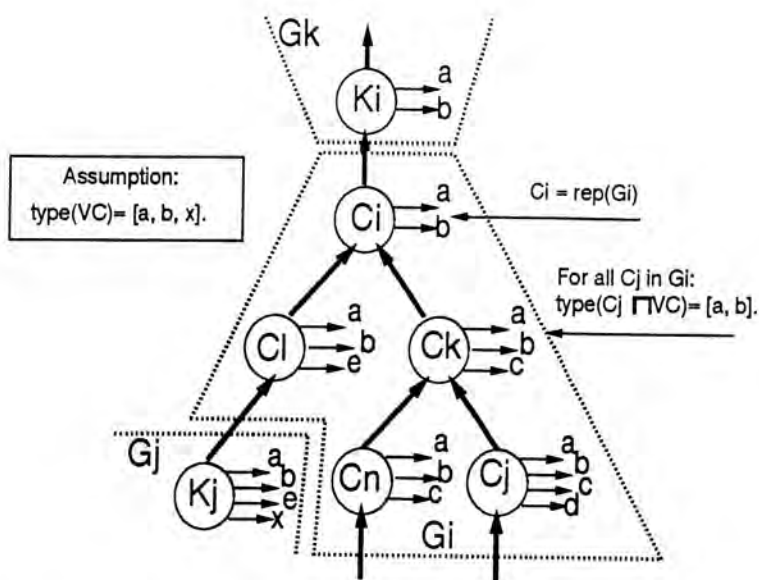


Figure 7.6: Partitioning of the Schema Graph  $G$  Using the Equivalence Relation

It is straightforward to see that the equivalence function  $\equiv_{VC}$  partitions  $G$  into a set of non-overlapping subgraphs  $G_i$ . This is so since for all  $C_i \in V$ ,  $C_i \cap VC$  is equal to exactly one type  $t_i$ . This type  $t_i$  then determines the membership of  $C_i$  in one and only one group  $G_i$ .

**Lemma 12.** *Given a closed schema graph  $G = (V, E)$  and a class  $VC$  that is to be integrated into  $G$ . Then the result of this integration defined by*

$$V' = V \cup VC \cup GG$$

and

$$GG = \{g_i \mid \text{type}(g_i) = \text{type}(\text{rep}(G_i) \cap VC) \text{ and}$$

$$(\text{content}(g_i) = \text{content}(\text{rep}(G_i)) \cup \text{content}(VC))\}^1 \text{ and}$$

$$(G_i \text{ an equivalence group in } G \text{ with respect to } \equiv_{VC}) \}.$$

corresponds to a closed schema graph  $G' = (V', E')$ .

**Proof:** By Lemma 11, the following intermediate classes  $g_i$  must exist in  $G$  (or must be created in  $G'$ ):

<sup>1</sup>When dealing with an *is-a* compatible schema graph, then  $VC$  being a subtype of these classes would also be a subset of  $\text{rep}(G_i)$ . Hence,  $\text{content}(\text{rep}(G_i)) \cup \text{content}(VC) = \text{content}(\text{rep}(G_i))$

$(\forall C_i \in G)(\exists g_i \in G') (type(g_i) = type(C_i \sqcap VC) \text{ and } content(g_i) \supseteq content(C_i) \cup content(VC)).$

By Definition 25, all classes in an equivalence group  $G_i$  have the same lowest common supertype  $t_i$  with respect to VC. Therefore, I only need to create one intermediate class  $g_i$  for each equivalence group  $G_i$ . More precisely, the following intermediate classes  $g_i$  must be created:

$(\forall G_i \in G)(\exists g_i \in G') (type(g_i) = type(rep(G_i) \sqcap VC) \text{ and } content(g_i) = content(rep(G_i) \cup VC)).$  ■

**Lemma 13.** *Given a closed schema graph  $G=(V,E)$  and a class VC that is to be integrated into G. Assume that this integration of VC into G forces the creation of intermediate classes  $GG = \{ g_i \mid i = 1, \dots, m \}$ . If an intermediate class  $g_i \in GG$  already exists in G, then  $g_i$  is a member of the equivalence group  $G_i$  of G with  $(\forall C_i \in G_i)(type(C_i \sqcap VC) = type(g_i))$ . In fact, the type of  $g_i$  would be equal to the type of  $rep(G_i)$ .*

**Proof:** Lemma 13 can be explained as follows.  $g_i \in GG$  means that  $g_i$  is a lowest common superclass for some group  $G_i$ . For all classes  $C_i$  in  $G_i$ ,  $type(C_i \sqcap VC) = type(g_i)$ . This observation together  $(g_i \in G)$  imply that  $g_i \in G_i$ . Obviously,  $g_i$  corresponds to the smallest type in  $G_i$ . Hence,  $type(g_i) = type(rep(G_i))$ . ■

From Lemma 13 I can conclude that the existence of a class C in G with a type equal to  $g_i$ 's type can be determined by checking whether  $type(rep(G_i)) = type(C)$  for each  $G_i$  of G. Hence, I can assure that if an intermediate class  $g_i \in GG$  exists in G, then I can easily find it and thus won't unnecessarily create redundant ones.

Next, I show that all classes that are members of the same equivalence group  $G_i$  correspond to a connected subgraph of G with  $C_i = rep(G_i)$  the root of the subgraph.

**Theorem 3.** *Given a schema graph  $G=(V,E)$  and a new class VC to be integrated into G. For each equivalence group  $G_i$  of G, the classes  $C_j \in G_i$  form a connected subgraph of G. More formally, for all  $C_j \in G_i$ , all classes  $C_k \in V$  with  $(C_j \text{ is-a } * C_k)$  and  $(C_k \text{ is-a } * rep(G_i))$  must also be members of  $G_i$ .*

**Proof (By contradiction):** By Theorem 2, for all  $C_j \in G_i$ ,  $(C_j \text{ is-a } * rep(G_i))$  holds because  $rep(G_i)$  is minimal in  $G_i$ . In other words, each class  $C_j \in G_i$  is a subclass of the unique representative  $rep(G_i)$  of  $G_i$ . Next, I show that if there is a class  $C_k$  between  $C_j \in G_i$  and  $rep(G_i)$ , then  $C_k \in G_i$ . The argument below is based on the situation depicted in Figure 7.7.

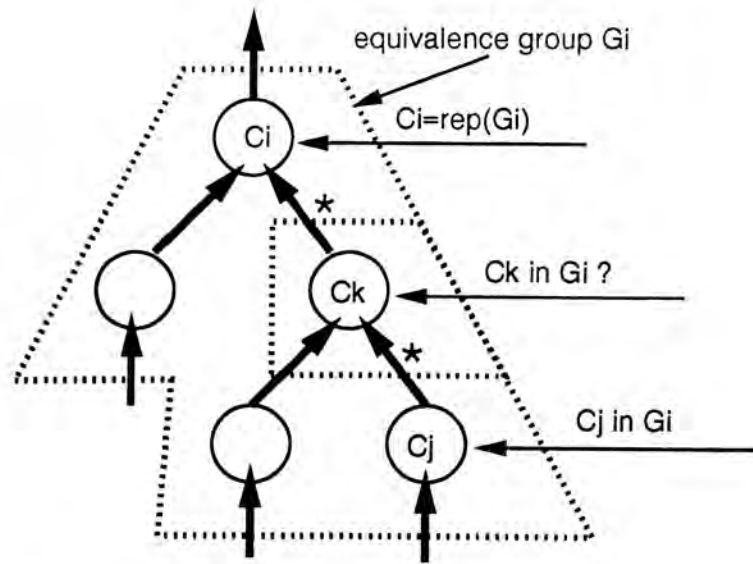


Figure 7.7: An Equivalence Group  $G_i$  Forms a Connected Subgraph of  $G$ .

Assumption: Let  $C_k \in V$  be a class with  $(C_j \text{ is-a } * C_k)$  and  $(C_k \text{ is-a } * \text{rep}(G_i))$  and  $C_k \notin G_i$ .

(a)  $(C_j \text{ is-a } * C_k)$  implies  $(C_j \preceq C_k)$ .  $(C_j \preceq C_k)$  and  $(\text{type}(C_j \sqcap VC) = \text{type}(g_i))$  imply  $((C_k \sqcap VC) \succeq g_i)$ .

(b)  $(C_k \text{ is-a } * \text{rep}(G_i))$  implies  $(C_k \preceq \text{rep}(G_i))$ . And  $(C_k \preceq \text{rep}(G_i))$  and  $(\text{type}(\text{rep}(G_i) \sqcap VC) = \text{type}(g_i))$  imply  $((C_k \sqcap VC) \preceq g_i)$ .

(c) By (a) and (b), I have  $((C_k \sqcap VC) \preceq g_i)$  and  $((C_k \sqcap VC) \succeq g_i)$ . This then implies  $((C_k \sqcap VC) = g_i)$ . I thus have shown  $C_k$  to be a member of the equivalence group  $G_i$ . This is a contraction to the assumption  $C_k \notin G_i$ . ■

### 7.3.4 Interconnecting Intermediate Classes

Let  $f()$  denote the function defined by  $f(t_j) = (t_j \sqcap VC) = g_j$ . Then, the reverse function  $f^{-1}()$  is defined by  $f^{-1}(g_j) = t_j$  with  $t_j$  the canonical representative of the group  $G_i$ . Lattice properties that lead to the interconnection of these intermediate classes are discussed next based on ([Missi89], Lemma 4.4 and Theorem 4.2).

**Definition 26.** Given a schema graph  $G=(V,E)$  and a class  $VC$  to be inserted into  $G$ . By Theorem 3, the equivalence relation  $\equiv_{VC}$  defines a partition  $GG$  of equivalence groups  $G_i$  on  $G$  of size  $m$ . I define  $G^*=(V^*,E^*)$  to be a schema graph with  $V^*=GG$

and  $E^*$  the set of edges  $e = \langle G_i, G_j \rangle$  with  $G_i, G_j \in V^*$  and  $(\exists C_i \in G_i) (\exists C_j \in G_j) ((C_i \text{ is-a } C_j) \text{ in } G)$ .

$G^*$  corresponds to the set of class representatives modulo  $\equiv_{VC}$ . Put differently,  $G^*$  is a hypergraph on  $G$  since each node in  $G^*$  is equal to an equivalence group  $G_i$ . For  $C_i, C_j \in G$ , I say that  $C_j$  is a parent\* of  $C_i$ , denoted by  $\text{parent}^*(C_i) = C_j$ , if  $C_j = \text{rep}(G_j)$  and there is an edge  $e = \langle G_i, G_j \rangle$  in  $G^*$ . In [Missi89], it is shown that  $G^*$  is isomorphic to the graph  $GG$  defined above.

**Theorem 4.** *Given a closed schema graph  $G = (V, E)$  and a class  $VC$  to be inserted into  $G$ . The integration of a class  $C$  with  $\text{type}(C) = \text{type}(VC)$  and arbitrary set content results in a closed schema graph  $G' = (V', E')$  if I add the set of classes  $GG = \{g_i\}$  as defined in Lemma 9. In addition, the following is-a edges must be established:*

1. *Each class  $g_i \in GG$  has a single child in  $G$  which is the canonical representative of the group to which  $f^{-1}(g_i)$  belongs. Put differently, I add the edge  $e = \langle \text{rep}(G_i), g_i \rangle$  between each  $\text{rep}(G_i)$  and the corresponding generated class  $g_i = \text{rep}(G_i) \sqcap VC$ .*
2. *For all  $g_i, g_j$  in  $GG$ , I add the edge  $e = \langle g_i, g_j \rangle$  if and only if  $C_i = \text{rep}(G_i)$  and  $C_j = \text{rep}(G_j)$  and  $C_j$  is a direct parent\* of  $C_i$  in  $G^*$ .*

Note that the creation of intermediate classes  $g_i$  coerced by the type lattice problem is driven by the closure requirements of the type hierarchy. No requirements are made on the set content of these classes, since these types are hidden and not necessarily of interest to the database user. Therefore, I can specify the set aspect of these new intermediate classes as needed. By setting the content of the new intermediate class  $g_i$  equal to the content of the representative class  $\text{rep}(G_i)$  of  $G_i \cup VC$ , I assure that  $g_i$  is the superset of all classes in  $G_i$ . By Theorem 2,  $g_i$  for  $G_i$  is also the supertype of all classes in  $G_i$ . This implies that  $g_i$  is indeed a superclass of all classes in  $G_i$ . Hence, the largest class in  $G_i$ , namely,  $\text{rep}(G_i)$  should be the direct child of  $g_i$ , which is stated in part 1 of Theorem 4. By part 2 of Theorem 4, two newly generated types  $g_i$  and  $g_j$  are only subtypes of one another if the corresponding representative classes  $C_i$  and  $C_j$  of the equivalence groups  $G_i$  and  $G_j$  are also subtypes of one another. If  $C_i$  and  $C_j$  are subtypes of one another in  $G$  then they must also be subsets of one another. Since the contents of  $g_i$  and  $g_j$  are equal to the contents of  $C_i$  and  $C_j$ , I can imply that  $g_i$  is a subclass of  $g_j$ . This then justifies part two of Theorem 4. A more detailed proof of Theorem 4 with respect to the type hierarchy can be found in [Missi89].

**Example 37.** *Figure 7.8.a shows a schema graph  $G$  with a partition induced by the class  $VC$ . The partition consists of the five equivalence groups  $G_i, G_j, G_k, G_l,$*

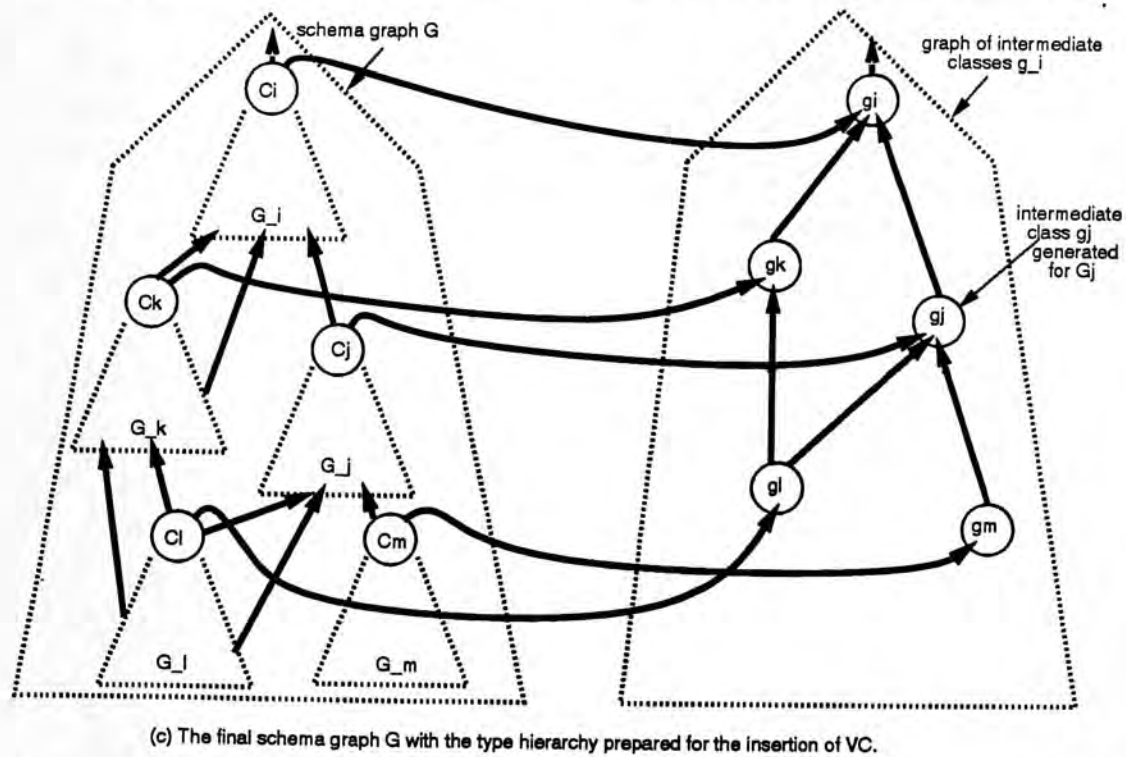
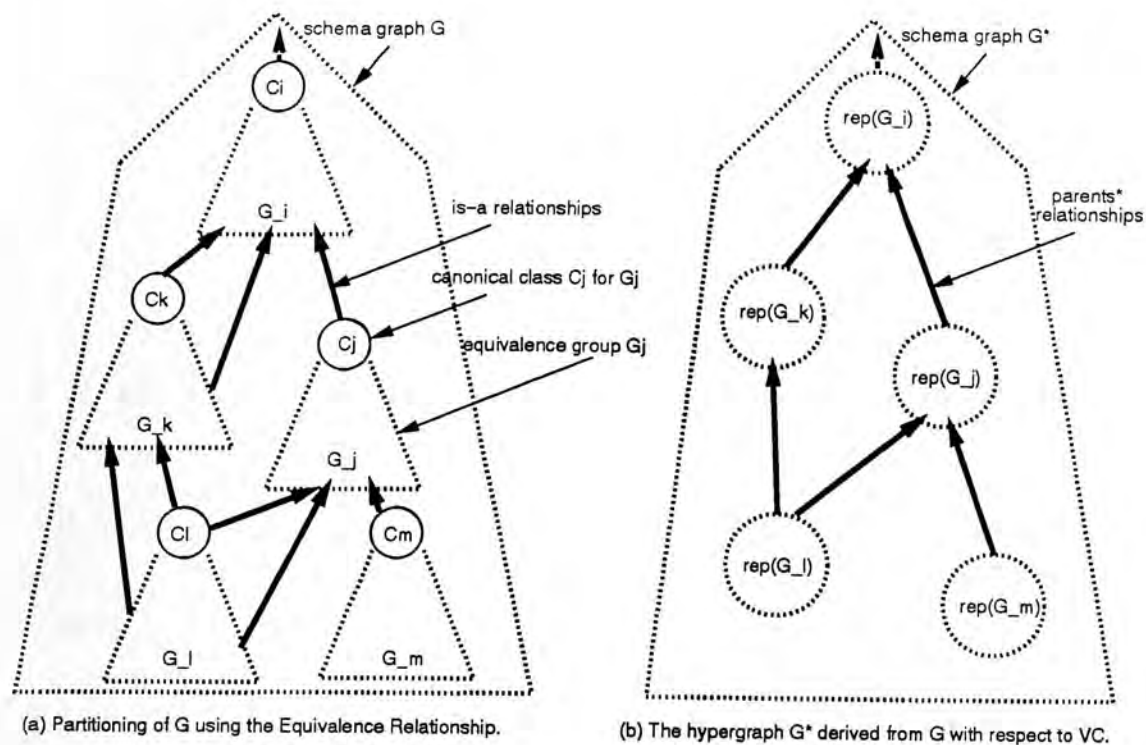


Figure 7.8: Connecting Intermediate Classes with Classes in the Schema Graph.

and  $G_m$ , which have the representatives  $C_i, C_j, C_k, C_l$ , and  $C_m$ , respectively. The matching hypergraph  $G^*$  of  $G$ , in which each node in  $G^*$  corresponds to an equivalence group  $G_i$  in  $G$ , is depicted in Figure 7.8.b. Figure 7.8.c then shows the schema graph that results from integrating a class  $C$  with  $\text{type}(C) = \text{type}(VC)$  into  $G$ . First, for each equivalence group  $G_i$  in  $G$ , I create an intermediate class  $g_i$  as shown on the right hand side of Figure 7.8.c. By Theorem 4.a, each representative class  $C_i$  of  $G_i$  is connected to the respective class  $g_i = C_i \cap VC$ . This is depicted by the dark arrows going from the left to the right of Figure 7.8.c. By Theorem 4.b, the newly generated classes  $g_i$  are connected with one another as dictated by the relationships of the corresponding representatives classes  $C_i$  (as shown in Figure 7.8.b). For instance, the edge ( $g_i$  is-a  $g_j$ ) is inserted into  $GG$  because the relationship ( $C_i$  is-a  $* C_j$ ) holds in  $G$ .

### 7.3.5 Class Integration After Type Preparation

In previous sections, I have demonstrated how to prepare a schema graph hierarchy for the insertion of a new class  $VC$ . Next, I need to handle the actual integration of  $VC$  into  $G$ . Contrary to the flexibility in arbitrary determining the set content of the intermediate classes  $g_i$  generated for type hierarchy preparation, the type and the set content of the virtual class  $VC$  are predetermined. Hence, the integration of the class  $VC$  itself must obey both the subtype and the subset relationships of  $VC$  with all other classes in the schema graph. The integration of  $VC$  into  $G$  can be characterized as follows.

**Lemma 14.** *Given a closed schema graph  $G=(V,E)$  and a class  $VC$  to be integrated into  $G$ . The integration of a class  $VC$  into  $G$  results in a closed schema graph  $G'=(V',E')$  if*

1. first, I extend the class hierarchy  $G$  to include  $\text{type}(VC)$  using Theorem 4, and
2. second, I add  $VC$  into its correct location in  $G'$  by connecting it to its direct parents and direct children in  $G'$ .

By Theorem 4, step 1 of Lemma 14 results in an extension of  $G$  that correctly incorporates a class with  $\text{type}(VC)$  into its class hierarchy. Once, a class with its type equal to  $VC$  is present in  $G$ , class integration of  $VC$  becomes a matter of finding the correct location for  $VC$  in  $G$ . Details on finding the correct position of  $VC$  given a prepared type hierarchy are presented in a later section. I will use the lemma above for implementing the integration process as explained in Section 7.5.



I have discussed the creation of intermediate classes and the associated arcs required to consistently integrate a virtual class VC into a schema graph G. The discussion and hence the solution are driven by the type inheritance problem (Section 7.2). I now need to assure that the edges created are also sufficient in terms of capturing the subset relationships between all classes. It is easy to see that all suggested edges are correct and non-redundant. It is not necessarily clear whether they are also sufficient in capturing all subset relationships. Edges could not be missing between the original graph G and the virtual graph GG consisting of intermediate classes  $g_i$ , since I assumed G to be complete and showed GG to be complete. It is relatively straightforward to show that no additional edges from G to GG can exist. At present, I am not able to prove that no additional edges from GG to G need be added to complete all information on the combined schema graph. If this is the case, then I would apply the algorithm described in Section 7.4 to add these extra direct-parent relationships for each intermediate class  $g_i$ . Since this can be done in linear time, this will not change the complexity of the type hierarchy preparation algorithm.

### 7.3.6 The Type Hierarchy Preparation Algorithm

In the previous sections, I have described the theory underlying the preparation of a class hierarchy for the insertion of a new class. Based on these results, I now develop an algorithm to solve this problem. This algorithm creates the additional intermediate classes required by the insertion of a new type into a schema graph. This algorithm, a direct extension of ([Missi89], page 77-80), handles both the type and set content of the class concept while Missikoff's work focuses on type classification.

For the following, I assume that the graph G is represented by a table with sorted rows (one for each class) and with the following four columns, the name of the class C, the set of parents of C in G, a label "\*" to mark members of  $G^*$ , and the set of parents\* of C in  $G^*$ . The former two are given initially and the later two are generated by the algorithm. I also assume that the rows are sorted according to the generalization relationship. The notation  $f()$  is again used to denote the function  $f(C) = C \sqcap VC$ .

The Compute- $G^*(G, VC)$  procedure in Figure 7.9 computes the hypergraph  $G^*$ . In particular, it computes the representative class  $rep(G_i)$  for each equivalence group  $G_i$  with respect to VC. By Theorem 2, a class  $C_i$  is a representative of an equivalence group  $G_i$  if and only if it is the highest class in the group  $G_i$ . This means that all its parents must belong to different equivalence groups. This is exactly what is tested by statement (2) of the procedure. These unique representatives  $C_i = rep(G_i)$  are

**Algorithm outline:** Generation of Intermediate Classes.

**Input:**

A schema  $G = (V, E)$  with possibly multiple inheritance.

A class  $VC$  to be integrated into  $G$ .

**Output:**

$G$  augmented by all intermediate classes required for the integration of  $VC$  into  $G$ .

**Algorithm:**

```

procedure Generate-Intermediate-Classes( $G, VC$ )
begin
  (1) Compute- $G^*(G, VC)$ ;
  (2) for all  $C \in G^*$  do
    if  $(C \cap VC) \neq \text{type}(C)$  then
       $\text{type}(g) = C \cap VC$ ;
       $\text{contents}(g) = \text{contents}(C) \cup \text{contents}(VC)$ ;
       $V = V \cup \{g\}$ ;
      for all  $p \in f(\text{parents}^*(C))$ 
        add the edge  $(g \text{ is-a } p)$  to  $G$ .
      endfor
      add the edge  $(C \text{ is-a } g)$  to  $G$ .
      if  $\text{parents}(g) \cap \text{parents}(C) \neq \{\emptyset\}$  then
        for all  $p \in \text{parents}(g)$  remove the edge  $(C \text{ is-a } p)$  endif
      endif
    endif
  endfor
end procedure

procedure Compute- $G^*(G, VC)$ 
begin
  for all  $C \in G$  do
    (1)  $\text{parents}^*(C) = \text{parents}(C)$ ;
    (2) if  $(\forall C_k \in \text{parents}(C)) (C \cap VC \neq C_k \cap VC)$  then
      mark  $C$  by the label “*”;
    (3) for all  $C_k \in \text{parents}(C)$  do
      if  $C_k$  is not marked then
         $\text{parents}^*(C) = \text{parents}^*(C) - \{C_k\}$ ;
        for all  $C_j \in \text{parents}^*(C_k)$  do
          if  $\text{parents}^*(C) = \{\emptyset\}$  then  $\text{parents}^*(C) = \{C_j\}$ ;
          else if  $(\forall C_i \in \text{parents}^*(C)) (C_j \text{ is not a supertype of } C_i)$ 
            then  $\text{parents}^*(C) = \text{parents}^*(C) \cup \{C_j\}$ ;
          endif
        endif
      endif
    endif
  endfor
end procedure

```

Figure 7.9: The Type-Hierarchy-Preparation Algorithm.

marked by the label "\*" and they are said to belong to  $G^*$ . By Lemma 12, each of them will trigger the generation of a new intermediate class  $g_i$  with  $\text{type}(g_i) = C_i \sqcap VC$ .

The Compute- $G^*(G, VC)$  procedure also finds the parents of each unique representative class  $C_i$  in the hypergraph  $G^*$ , denoted by  $\text{parents}^*(C_i)$ . These  $\text{parents}^*(C_i)$  correspond to the canonical representatives of the equivalence groups  $G_i$  that the class  $C_i$  is directly *is-a* related to. Put differently,  $\text{parents}^*(C_i)$  corresponds to the set of all parents of  $C_i$  in  $G^*$ . For a class  $C$ , if any of its parents  $C_k$  are marked then these parents in  $G$  are also its  $\text{parents}^*$  in  $G^*$ . However, if a parent  $C_k$  of  $C$  is not marked, then I need to find the  $\text{parent}^*$  of  $C$  higher in the graph. If I could assume that the  $\text{parents}^*$  of all classes above the current class  $C$  have already been determined before attempting to calculate  $\text{parents}^*$  of  $C$ , then I could simply set  $\text{parents}^*(C)$  equal to  $\text{parents}^*$  of  $C_k$ . Indeed, this is assured by scanning the list of classes ordered according to the generalization relationship among classes from left to right.

Next, the Generate-Intermediate-Classes( $G, VC$ ) procedure (Figure 7.9) is applied to construct all required intermediate classes  $g_i$  and interconnects them with one another and with the classes in  $G$ . The type of the new class  $g_i$  is determined by the lowest common supertype operator  $\sqcap$  while the set membership of  $g_i$  is determined by setting the content of  $g_i$  equal to the content of the respective  $\text{rep}(G_i) \cup VC$ . Note that this represents an important extension to the algorithm in [Missi89] since I generate a complete class (rather than just a type).

For each class  $C$  in  $G^*$ , I check whether the condition  $(C \sqcap VC \neq \text{type}(C))$  evaluates to false. This would mean that an intermediate class  $g$  with  $\text{type}(g) = C \sqcap VC$  (i.e., the required type) exists in  $G$  and therefore need not be generated. In this case, the class  $C$  is already properly connected since the original schema graph is assumed to be closed. Thus the if-statement is skipped. If, on the other hand, the condition  $(C \sqcap VC \neq \text{type}(C))$  evaluates to true, then the intermediate class  $g_i$  with  $\text{type}(g) = C \sqcap VC$  does not yet exist in  $G$  and therefore needs to be created. In this case I associate a new class  $g_i$  with  $\text{type}(g_i) = C \sqcap VC$  and  $\text{content}(g_i) = \text{content}(C)$  with  $C$ . In addition, I create an edge  $e = \langle C, g_i \rangle$  between  $C$  and  $g_i$  to make  $C$  the unique child of  $g_i$ . I also add edges from  $g_i$  to all its parents in the hypergraph  $G^*$ . These  $\text{parents}^*$  of  $g_i$  could be other classes  $g_k$  in  $G^*$  or existing classes  $C_i$  in  $G$ . This is done by creating the edges  $e = \langle g_i, p \rangle$  with  $p = f(\text{parents}^*(C))$ .

In order to avoid redundant arcs, the procedure removes edges  $e = \langle C, p \rangle$  from the class  $C$ , if  $C$  shares any parents with its newly generated intermediate class

$g_i$ , i.e., if  $p \in \text{parents}(g_i)$ .  $g_i$  becomes a direct parent of  $C$  and thus these arcs are redundant.

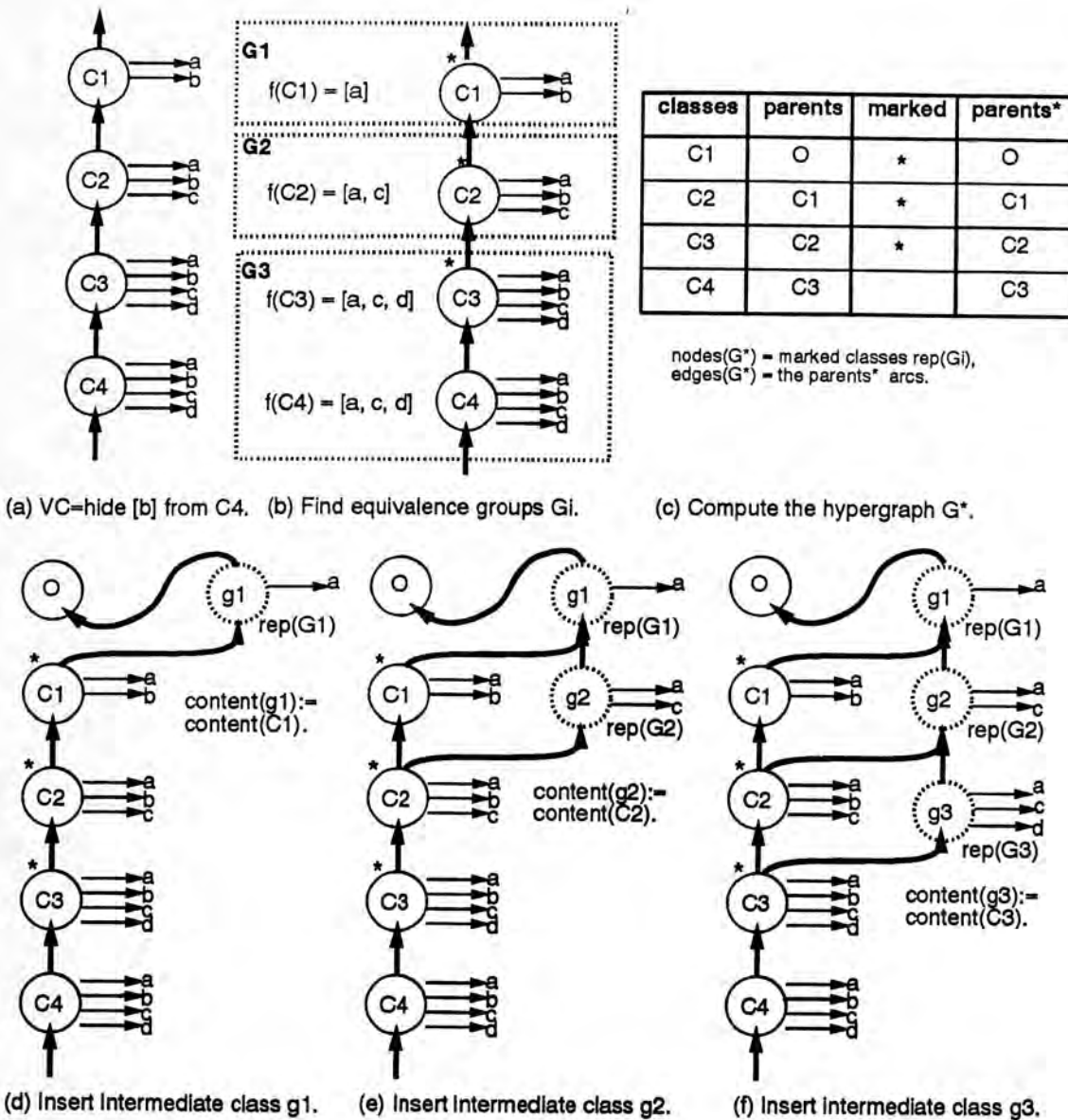


Figure 7.10: An Example of Class Hierarchy Preparation.

**Example 38.** Figure 7.10 demonstrates how to apply the type classification algorithm given in Figure 7.9 to prepare a class hierarchy  $G$  for class insertion. Figure 7.10.a shows the schema  $G$  before type classification. Assume that the virtual class  $VC$  is derived by the view derivation "VC = hide [b] from  $C_4$ ". Then the type of  $VC$  is defined by  $\text{type}(VC)=[a,c,d]$  and the object membership of  $VC$  is defined by

$\text{content}(VC) = \text{content}(C4)$ . The *Generate-Intermediate-Classes*( $G, VC$ ) procedure first calls the *Compute- $G^*$* ( $G, VC$ ) procedure to compute the hypergraph  $G^*$ . There are three equivalence groups with respect to  $VC$ , namely,  $G_1$  with the lowest common supertype of  $[a]$ ,  $G_2$  with the lowest common supertype of  $[a, c]$ , and  $G_3$  with the lowest common supertype of  $[a, c, d]$ . The outer for-loop of the *Compute- $G^*$* ( $G, VC$ ) procedure steps through the list of all classes in  $G$ . When processing the first class  $C1$ , step (2) of the for-loop marks the class  $C1$ , since its parent is the general root of the schema. Step (3) checks whether all parents of  $C1$  are marked, and since they are,  $\text{parents}^*(C1) = \text{parents}(C1)$  is not modified. The second iteration of the for-loop marks the class  $C2$ , since  $C2$  and  $C1$  are in the different equivalence groups  $G_2$  and  $G_1$ , respectively. Step (3) checks whether the parent of  $C2$ , which is  $C1$ , is marked. Since  $C1$  is indeed marked,  $\text{parents}^*(C2) = \{C1\}$  is not modified. The third iteration of the for-loop marks the class  $C3$ , since its only parent  $C2$  is again in a different equivalence group. Since the parent of  $C3$  is marked, the  $\text{parents}^*(C3)$  set is not modified. The fourth and last iteration of the for-loop does not mark the class  $C4$ , since its parent  $C3$  is in the same equivalence group  $G_3$  as  $C4$ . Step (3) then checks whether the parent of  $C4$ , which is  $C3$ , is marked. Since  $C3$  is marked,  $\text{parents}^*(C4) = \{C3\}$  is not modified. This completes the computation of the hypergraph  $G^*$  on  $G$ . The equivalence groups  $G_1, G_2$  and  $G_3$  and their marked representatives  $\text{rep}(G_1)=C1$ ,  $\text{rep}(G_2)=C2$ , and  $\text{rep}(G_3)=C3$  are shown in Figure 7.10.b. The hypergraph  $G^*$ , i.e., the marked classes  $C_i = \text{rep}(G_i)$  and their  $\text{parents}^*$  relationships, are shown in Figure 7.10.c.

The second step of the *Generate-Intermediate-Classes*( $G, VC$ ) procedure now generates the intermediate classes  $g_i$  for each equivalence group  $G_i$ . For the first marked class  $C1$ , it checks whether a class  $g_1$  with  $\text{type}(g_1) = (C1 \sqcap VC)$  exists in the schema. Since it does not,  $g_1$  is created with  $\text{type}(g_1) = C1 \sqcap VC = [a]$  and  $\text{content}(g_1) = \text{content}(C1)$  as shown in Figure 7.10.d. I add edges from  $g_1$  to the generated intermediate classes of all its  $\text{parents}^*$  in  $G^*$ . Since  $\text{parents}^*(C1) = \{O\}$  and  $O \sqcap VC = O$ , this corresponds to the edge  $\langle g_1, O \rangle$ . The edge  $\langle C1, g_1 \rangle$  becomes redundant and is removed. I also add the edge  $\langle C1, g_1 \rangle$  to connect  $g_1$  to its canonical representative in the original schema graph  $G$  (Figure 7.10.d). The next marked class in the list in Figure 7.10.c is  $C2$ . I check whether a class  $g_2$  with  $\text{type}(g_2) = (C2 \sqcap VC)$  exists in the schema.  $\text{Type}(g_2) = [a, c] \neq [a, c, d] = \text{type}(C2)$  implies that a class with the required type of  $g_2$  does not exist in the schema. Hence, I create the class  $g_2$  with  $\text{type}(g_2) = (C2 \sqcap VC) = [a, c]$  and  $\text{content}(g_2) = \text{content}(C2)$  as shown in Figure 7.10.e. I add edges from  $g_2$  to the intermediate classes  $g_i$  that are  $g_2$ 's  $\text{parents}^*$  in  $G^*$ . Since  $\text{parents}^*(C2) = \{C1\}$  and  $C1 \sqcap VC = g_1$ , this corresponds to the edge  $\langle g_2, g_1 \rangle$ . The edge  $\langle C2, g_2 \rangle$  is also added to connect  $g_2$  to its canonical representative in  $G$ . The resulting schema is depicted

in Figure 7.10.e. Lastly, the algorithm processes the marked class  $C3$  in a similar manner. The final result, the graph  $G$  prepared for the insertion of the virtual class  $VC$ , is given in Figure 7.10.f.

In [Missi89], the algorithm has been shown to be of quadratic complexity  $O(m^2)$  with  $m$  the number of edges in  $G$ . Our extension of the algorithm, namely, the generation of a complete class (rather than just a type) does not influence this complexity. This analysis assumes of course that the *subsumes()* function can be calculated in constant time.

## 7.4 The Class Placement Algorithm

In this section, I discuss the integration of a class into a given class hierarchy assuming that the type hierarchy has been properly prepared for class insertion as described in Section 7.3. The proposed placement algorithm handles both single- and multiple-inheritance schema graphs. I prove the correctness of the algorithm and show the algorithm to be of linear complexity (assuming a *subsumes()* function of constant complexity). For the following, I assume the existence of a function *subsumes*( $C1, C2$ ) that determines whether the *is-a* relationship ( $C2$  *is-a*  $C1$ ) exists. As described in Section 7.1, this function is computed by comparing the type description and the membership predicate of the two classes. This is different from checking whether the edge  $e = \langle C2, C1 \rangle$  exists in a schema graph  $G$ .

### 7.4.1 The General Class Placement Algorithm

By Definition 8, a schema graph captures all *direct is-a* relationships between pairs of classes  $C_1$  and  $C_2$ , namely, ( $C_1$  *is-a*  $C_2$ ), by directed graph edges  $e = \langle C_1, C_2 \rangle$ . Put differently, each class  $C_i$  in a schema graph  $G$  is connected to its direct sub- and super-classes via graph edges. *Indirect is-a* relationships ( $C_1$  *is-a*  $C_2$ ) are derivable via the transitive closure on the graph edges. The integration of a new virtual class  $VC$  into the schema graph  $G=(V,E)$  thus requires the identification of the direct *is-a* relationships between the virtual class  $VC$  and all other classes in the global schema  $G$  as defined below.

**Definition 27.** *Given a schema graph  $G=(V,E)$  and a class  $VC$ . Then I defined the set of all classes in  $G$  that directly subsume  $VC$ , i.e., the direct superclasses of  $VC$ , by*

$DIRECT-PARENTS_{VC} :=$

$\{C_i \mid (VC \text{ is-a } C_i) \wedge (\nexists C_j \in V)(j \neq i)((VC \text{ is-a}^* C_j) \wedge (C_j \text{ is-a}^* C_i))\}.$

Similarly, I define the set of all classes in  $G$  that  $VC$  directly subsumes, i.e., the direct subclasses of  $VC$ , by

$DIRECT-CHILDREN_{VC} :=$

$\{C_i \mid (C_i \text{ is-a } VC) \wedge (\nexists C_j \in V)(j \neq i)((C_i \text{ is-a}^* C_j) \wedge (C_j \text{ is-a}^* VC))\}.$

**Example 39.** Definition 27 is explained based on the schema graph given in Figure 7.12. In this figure, the labels **sup** and **sub** are associated with a node  $C_i$  that is a superclass or a subclass of  $VC$ , respectively.

The  $DIRECT-PARENTS_{VC}$  set contains all classes that fulfill the following conditions: (1) they are superclasses of  $VC$ , i.e., they have the **sup** label, and (2) there are no other classes below them in the schema graph that are also superclasses of  $VC$ . The latter means that they are the **lowest** possible classes still marked by the **sup** label. In Figure 7.12, the members of the  $DIRECT-PARENTS_{VC}$  set, which are  $C3$  and  $C9$ , are marked by horizontal strips.

The  $DIRECT-CHILDREN_{VC}$  set contains all classes that fulfill the following conditions: (1) they are subclasses of  $VC$ , i.e., they have the **sub** label, and (2) there are no other classes above them in the schema graph that are also subclasses of  $VC$ . The latter means that they are the **highest** possible classes still marked by the **sub** label. In Figure 7.12, the members of the  $DIRECT-CHILDREN_{VC}$  set, which are  $C16$  and  $C25$ , are marked by vertical strips.

Based on Definition 27, the algorithm for finding the correct position for the class  $VC$  in the schema  $G=(V,E)$  can be summarized as follows. First, I find all classes in  $G$  that are direct superclasses of  $VC$ , namely, the set of classes  $DIRECT-PARENTS_{VC}$  as defined in Definition 27. Next, I find all classes in  $G$  that are direct subclasses of  $VC$ , namely, the set of classes  $DIRECT-CHILDREN_{VC}$  as defined in Definition 27.  $VC$  then is placed directly below all classes in the  $DIRECT-PARENTS_{VC}$  set and directly above all classes in the  $DIRECT-CHILDREN_{VC}$  set by adding new *is-a* edges to the schema graph  $G$ . The just described algorithm is given in Figure 7.11.

The Class-Placement algorithm shown in Figure 7.11 has the following steps. Step (1) of the algorithm computes the  $DIRECT-PARENTS_{VC}$  set. As explained later, this is done by a depth-first downwards traversal of  $G$  starting from the root of

**Algorithm outline:** Placement of A Virtual Class into the Global Schema.

**Input:**

A schema  $G = (V,E)$  with possibly multiple inheritance.

A class  $VC$  to be integrated into  $G$ .

**Output:**

The schema  $G$  with  $VC$  integrated into  $G$ .

**Data Structures:**

variables  $DIRECT-PARENTS_{VC}$ ,  $DIRECT-CHILDREN_{VC}$ : set of classes

**Algorithm:**

```

procedure Class-Placement-Algorithm( $G,VC$ )
begin
  (1) Compute  $DIRECT-PARENTS_{VC}$ .
  (2) if  $C_i \in DIRECT-PARENTS_{VC}$  with ( $C_i = VC$ ) then STOP
      else  $V := V \cup VC$ ;
      endif
  (3) Compute  $DIRECT-CHILDREN_{VC}$ .
  (4) Update-Edges $_{VC}(G, VC, DIRECT-PARENTS_{VC}, DIRECT-CHILDREN_{VC})$ ;
end procedure

procedure Update-Edges $_{VC}(G,VC,DIRECT-PARENTS_{VC},$ 
 $DIRECT-CHILDREN_{VC})$  begin
  (4.1) for all  $p \in DIRECT-PARENTS_{VC}$  do
      create-edge( $\langle VC,p \rangle$ );
      end for
  (4.2) for all  $c \in DIRECT-CHILDREN_{VC}$  do
      create-edge( $\langle c,VC \rangle$ );
      end for
  (4.3) for all  $p \in DIRECT-PARENTS_{VC}$  do
      for all  $c \in DIRECT-CHILDREN_{VC}$  do
          if edge  $\langle c,p \rangle$  exists then delete-edge( $\langle c,p \rangle$ ) endif
      end for
      end for
end procedure

```

Figure 7.11: The Class-Placement Algorithm.



G. If it finds a class in the  $\text{DIRECT-PARENTS}_{VC}$  set that is equal to  $VC$ , then  $VC$  exists already in  $G$  and the algorithm terminates (step (2)). This equality of classes is determined by comparing their type descriptions and membership characteristics and not necessarily their class names. If a mismatch in class names exists, then the view designer may attach the new name of  $VC$  as synonym with the original name of the existing class  $C_i$ . For simplicity, I assume that  $(C_i=VC)$  is defined by  $\text{subsumes}(C_i, VC)=\text{true}$  and  $\text{subsumes}(VC, C_i)=\text{true}$ . If  $VC$  was not integrated in  $G$  to being with, then the else-branch of step (2) will now add the class  $VC$  to the set of classes  $V$  of  $G$ . Step (3) of the algorithm computes the  $\text{DIRECT-CHILDREN}_{VC}$  set again by depth-first traversal of  $G$ . Lastly, step (4) updates the edges  $E$  of  $G$  so that the new class  $VC$  is now properly connected with the classes of  $G$ . This edge computation is described in detail in the Update-Edges procedure in Figure 7.11. The first for-loop creates edges between  $VC$  and all classes in the  $\text{DIRECT-PARENTS}_{VC}$  set, i.e., it connects  $VC$  with its direct superclasses. The second for-loop creates edges between  $VC$  and all classes in the  $\text{DIRECT-CHILDREN}_{VC}$  set, i.e., it connects  $VC$  with its direct subclasses. Finally, the third for-loop removes all edges from  $G$  that have become redundant due to the introduction of the edges listed above. Namely, all edges that directly connect classes in the  $\text{DIRECT-PARENTS}_{VC}$  set with classes in the  $\text{DIRECT-CHILDREN}_{VC}$  set have become redundant and thus are removed.

**Example 40.** *In this example, I demonstrate the Class-Placement algorithm given in Figure 7.11 on the example schema graph shown in Figure 7.12. The first step of the algorithm finds the  $\text{DIRECT-PARENTS}_{VC}$  set by depth-first downwards traversal of  $G$  starting from the root  $C0$ . The search stops for a given branch if either a leaf node  $C_i$  is reached (e.g., for the class  $C4$ ) or if the condition  $\text{subsumes}(C_i, VC)$  no longer holds (e.g., for the class  $C3$ ). All classes at the borderline of this search space that still fulfill the superclass condition are put into the  $\text{DIRECT-PARENTS}_{VC}$  set, in this case, the classes  $C3$  and  $C9$ . The algorithm does not find any class  $C_i$  that is equal to  $VC$  (step (2)). The third step of the algorithm then computes the  $\text{DIRECT-CHILDREN}_{VC}$  set by depth-first downwards traversal of  $G$ . The search stops for a given branch if either a leaf is reached (e.g., the classes  $C15$  and  $C8$ ) or if a class node  $C_i$  is reached for which the subclass condition  $\text{subsumes}(VC, C_i)$  is true (e.g., for the classes  $C16$  and  $C25$ ). All classes of the later category are placed into the  $\text{DIRECT-CHILDREN}_{VC}$  set. Lastly, the fourth step of the algorithm updates the edges  $E$  of  $G$  so that the new class  $VC$  is now properly connected with the classes of  $G$ . First, I create the edges  $e1 = \langle VC, C3 \rangle$  and  $e2 = \langle VC, C9 \rangle$  to connect  $VC$  with all classes in the set  $\text{DIRECT-PARENTS}_{VC} = \{C3, C9\}$ . Then, I create the edges  $e3 = \langle C18, VC \rangle$  and  $e4 = \langle C25, VC \rangle$  to connect all classes in the set  $\text{DIRECT-CHILDREN}_{VC} = \{C18, C25\}$  with  $VC$ . There is one direct edge between classes in  $\text{DIRECT-CHILDREN}_{VC} = \{C18, C25\}$  and those in  $\text{DIRECT-PARENTS}_{VC} = \{C3, C0\}$ ,*

namely, the edge  $\langle C25, C9 \rangle$ . Due to introduction of the edges  $\langle C25, VC \rangle$  and  $\langle VC, C9 \rangle$ , this edge has become redundant and is thus removed.

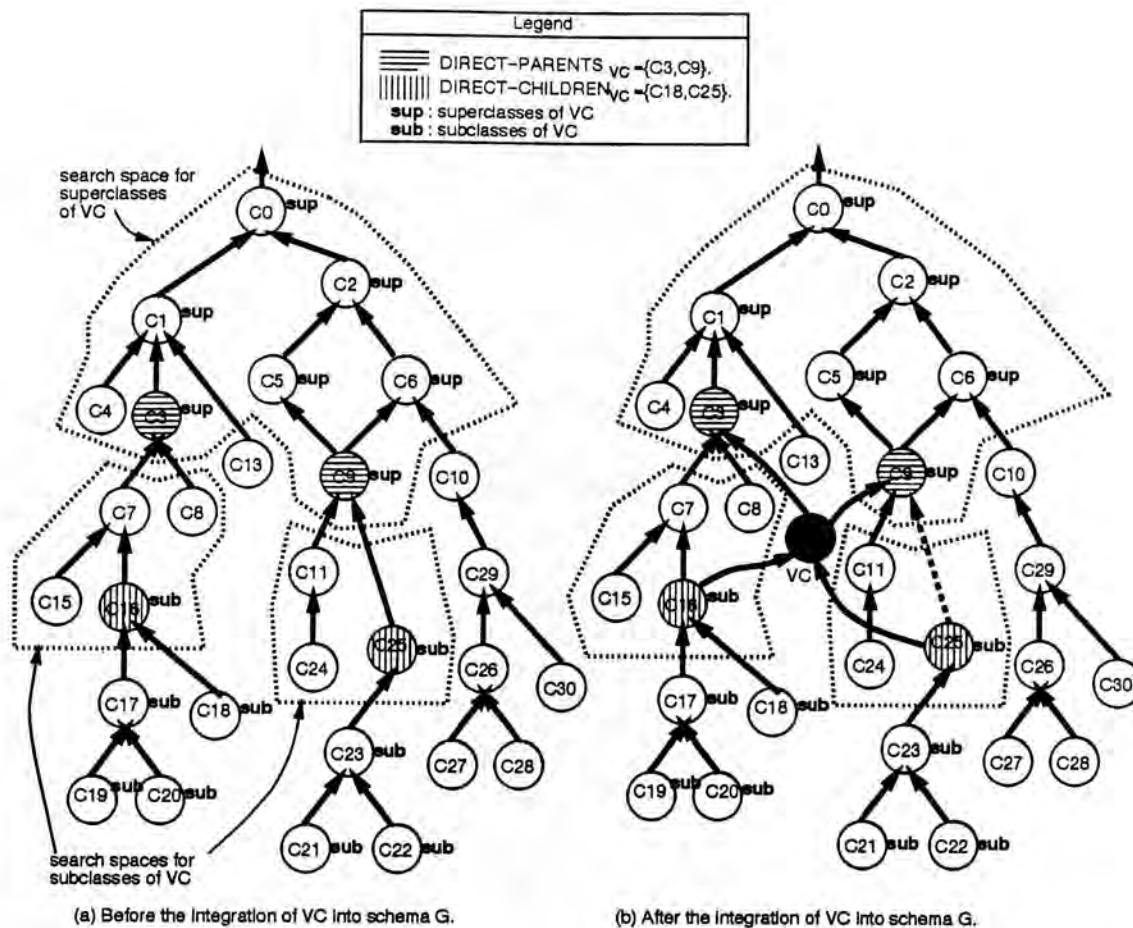


Figure 7.12: An Example of the Class-Placement Algorithm.

**Theorem 5. (Correctness)** Given the schema  $G = (V, E)$  and a class  $VC$ , the Class-Placement algorithm shown in Figure 7.11 integrates  $VC$  into  $G$  with the resulting  $G$  representing a correct schema graph as defined in Definition 8.

**Proof:** Definition 27 defines the direct superclasses and subclasses of the class  $VC$  in a set of classes  $S$  as  $\text{DIRECT-PARENTS}_{VC}$  and  $\text{DIRECT-CHILDREN}_{VC}$ , respectively. For this proof, I assume that the Class-Placement algorithm (steps 1 and 3 in Figure 7.11) indeed calculates these two sets of classes correctly – as I will show later in this section. Then I only need to show the correctness of the fourth step, namely, of the manipulation of the graph edges to prove the correctness of the overall algorithm.

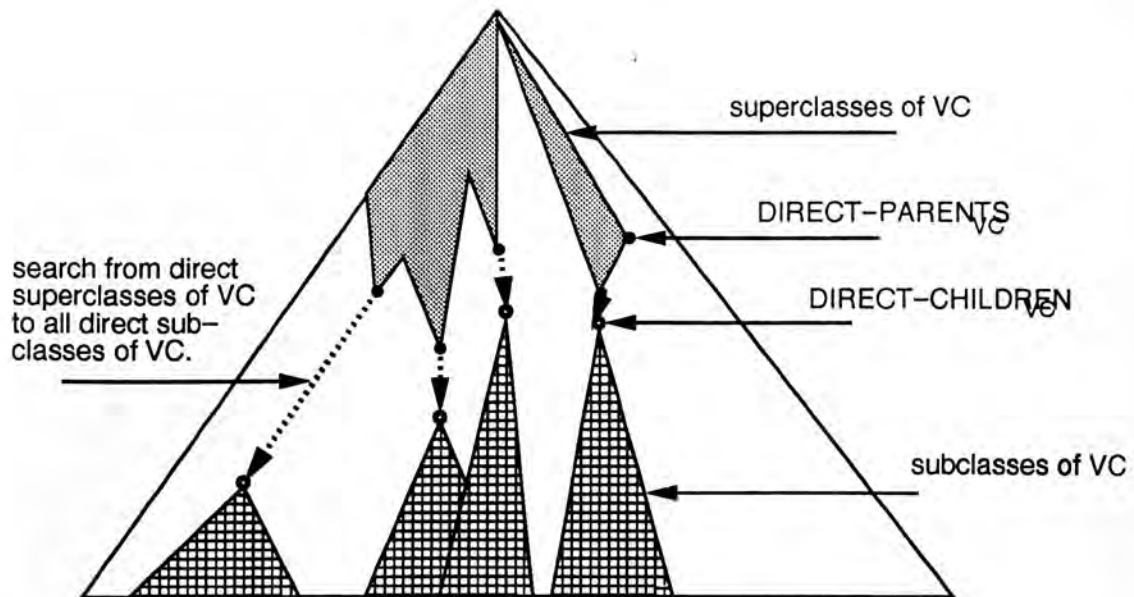


Figure 7.13: Search Space for Class Placement.

By Definition 8, a schema graph  $G=(V,E)$  is **correct** if it represents a set of classes  $S = \{C_i | i = 1, \dots, n\}$  and their *is-a* relationships in the following manner:

$$(1) V := S,$$

$$(2) E := \{ e = \langle C_i, C_j \rangle \mid (C_i \text{ is-a }^d C_j) \text{ in } S \text{ and } \forall C_i, C_j \in V \}.$$

From the point of view of a class  $C_i$ , this is equivalent to each class  $C_i$  having an *is-a* edge  $e$  in  $G$  to all its direct super- and subclasses in  $S$ . More formally,  $(\forall C_i \in V)$

$$(\forall p_j \in V) ((C_i \text{ is-a }^d p_j) \text{ in } S \implies e_{ij} = \langle C_i, p_j \rangle \text{ in } E) \wedge$$

$$(\forall ch_k \in V) ((ch_k \text{ is-a }^d C_i) \text{ in } S \implies e_{ki} = \langle ch_k, C_i \rangle \text{ in } E)$$

I assume that the input schema graph  $G=(V,E)$  is correct as defined above. If the new class  $VC$  is equal to one of the existing classes in  $G$ , then the Class-Placement algorithm in Figure 7.11 sets  $G':=G$  by **step 2**.  $G':=G$  is by assumption correct.

If the new class  $VC$  is not equal to any of the existing classes in  $G$ , then the Class-Placement algorithm constructs a new graph  $G'=(V',E')$  by adding  $VC$  to  $G$ . It is easy to see that the resulting graph  $G'$  will have the following characteristics:

$$(1) V' := V \cup VC,$$

$$(2) E' := E \cup DP \cup DC - DPC \text{ with}$$

$$DP := \{e_{VC,j} = \langle VC, p_j \rangle \mid p_j \in \text{DIRECT-PARENTS}_{VC}\}, \text{ and}$$

$$DC := \{e_{k,VC} = \langle ch_k, VC \rangle \mid ch_k \in \text{DIRECT-CHILDREN}_{VC}\}, \text{ and}$$

$$DPC := \{e_{kj} = \langle ch_k, p_j \rangle \mid ch_k \in \text{DIRECT-CHILDREN}_{VC} \wedge p_j \in \text{DIRECT-PARENTS}_{VC}\}.$$

I now show that  $G'$  is correct, namely, that  $G'$  contains graph edges for all direct *is-a* relationships and no others, by examining the following four cases:

**Part I:**  $(C_i \in V) \wedge (C_i \notin \text{DIRECT-PARENTS}_{VC}) \wedge (C_i \notin \text{DIRECT-CHILDREN}_{VC})$ .

**Part II:**  $(C_i \in \text{DIRECT-PARENTS}_{VC})$ .

**Part III:**  $(C_i \in \text{DIRECT-CHILDREN}_{VC})$ .

**Part IV:**  $(C_i = VC)$ .

**Part I:**  $(C_i \in V) \wedge (C_i \notin \text{DIRECT-PARENTS}_{VC}) \wedge (C_i \notin \text{DIRECT-CHILDREN}_{VC})$ .

**Part I.a:** The introduction of a new class to  $G$  can make an existing *direct is-a* relationship between two classes  $C_i$  and  $C_j$  redundant (indirect) if and only if the new class is placed between  $C_i$  and  $C_j$ , such as to provide an alternative *is-a* path of length greater or equal to two to the existing *is-a* edge between  $C_i$  and  $C_j$ . For the addition of  $VC$  to make an existing *is-a* edge redundant would imply that  $(C_i \in \text{DIRECT-PARENTS}_{VC})$  and  $(C_j \in \text{DIRECT-CHILDREN}_{VC})$ , or, vice versa. This is a contradiction to the assumption of **Part I**. Hence, the introduction of  $VC$  will not affect any of the existing *is-a* relationships of  $C_i$ . It can easily be seen that for  $(C_i \notin \text{DIRECT-PARENTS}_{VC}) \wedge (C_i \notin \text{DIRECT-CHILDREN}_{VC})$  no *is-a* edges are added or removed by the algorithm.

**Part I.b:** The introduction of a new class creates new *direct is-a* relationships only between classes that are the direct parents or the direct children of the new class. For the addition of  $VC$ , this would imply that it creates a new *direct is-a* relationship only for the classes  $(C_i \in \text{DIRECT-PARENTS}_{VC})$  or  $(C_i \in \text{DIRECT-CHILDREN}_{VC})$ . This is a contradiction to the assumption of **Part I**. Hence, the introduction of  $VC$  will not create a new *direct is-a* relationship for  $C_i$ . As stated

above, it can easily be seen that the algorithm does not add any new *is-a* edges for  $(C_i \notin \text{DIRECT-PARENTS}_{VC}) \wedge (C_i \notin \text{DIRECT-CHILDREN}_{VC})$ . ■

**Part II:**  $(C_i \in \text{DIRECT-PARENTS}_{VC})$ .

**Part II.a:** By Definition 27,  $C_i$  acquires a new direct subclass in  $G'$ , namely,  $VC$ . Therefore, the edge  $e = \langle VC, C_i \rangle$  has to be added to  $E'$ . It can easily be seen that **step 4.1** of the algorithm adds this edge  $\langle VC, C_i \rangle$  for all  $C_i \in \text{DIRECT-PARENTS}_{VC}$ .

**Part II.c:** Assume that there is a class  $C_j$  in  $V$  with which  $C_i$  had a direct *is-a* relationship in  $G$ . If this class  $C_j$  was a superclass of  $C_i$ , then it could not have been affected by the introduction of  $VC$ . It can easily be seen that no edges are added or removed by the algorithm for classes  $C_j$  that are neither in the  $\text{DIRECT-PARENTS}_{VC}$  nor in the  $\text{DIRECT-CHILDREN}_{VC}$  set. If this class node  $C_j$  was a subclass of  $C_i$ , then it can become indirect if  $VC$  has been placed between  $C_i$  and  $C_j$ . The later is only possible for  $C_j \in \text{DIRECT-CHILDREN}_{VC}$ . In this case, the edge  $e_{ji} = \langle C_j, C_i \rangle$  with  $(C_j \in \text{DIRECT-CHILDREN}_{VC})$  is redundant, since the integration of  $VC$  into  $G$  adds the edges  $e_2 = \langle C_j, VC \rangle$  and  $e_3 = \langle VC, C_i \rangle$  to  $E$ . By transitivity,  $e_2 = \langle C_j, VC \rangle$  and  $e_3 = \langle VC, C_i \rangle$  imply the (indirect) *is-a* relationship  $(C_j \text{ is-a } * C_i)$ . Hence, the edge  $e_{ji} = \langle C_j, C_i \rangle$  must be removed from  $E$ . See Figure 7.14 for an example of the creation of redundant edges. It can easily be seen that **step 4.3** of the algorithm removes all edges  $\langle C_j, C_i \rangle$  for  $C_j \in \text{DIRECT-CHILDREN}_{VC}$  and for  $C_i \in \text{DIRECT-PARENTS}_{VC}$ .

**Part II.d:** Assume that there are classes  $C_j$  in  $V$  with which  $C_i$  had no direct *is-a* relationship in  $G$ . The introduction of the extra class  $VC$  into the schema  $G$  will not add any a direct *is-a* relationship between these two classes  $C_j$  and  $C_i$ , since neither  $C_i$  nor  $C_j$  are equal to  $VC$ . It can easily be seen that no new edges are added for a class  $C_i \in \text{DIRECT-PARENTS}_{VC}$ , except for those connecting it to the new class  $VC$  done in **step 4.1**. ■

**Part III:**  $(C_i \in \text{DIRECT-CHILDREN}_{VC})$ .

The argument for **Part III** is similar to the one for **Part II**, except for dealing with a direct subclass rather than a direct superclass of  $VC$ . The proof is thus omitted here. ■

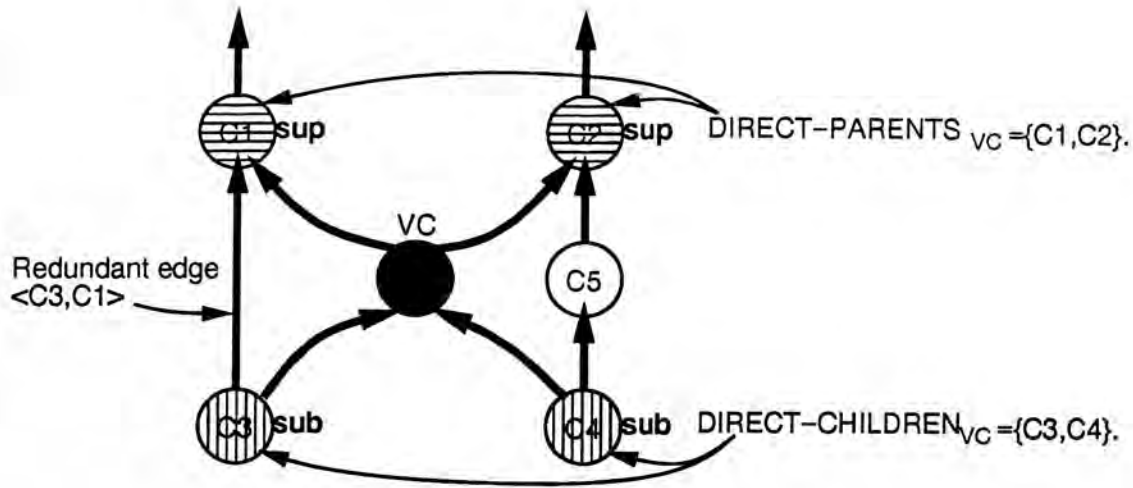


Figure 7.14: Removal of Redundant Edges During Class Placement.

**Part IV:** ( $C_i = VC$ ).

As explained above (and in Definition 8), a schema graph  $G' = (V', E')$  is **correct** if each class  $C_i$  in  $G'$  has an *is-a* edge  $e$  in  $E'$  to all its direct super- and subclasses in  $G'$ . Hence, there need to be edges in  $G'$  to connect  $VC$  to all its direct superclasses, which by Definition 27 are equal to all classes in the  $DIRECT-PARENTS_{VC}$  set. It can easily be seen that **step 4.1** of the algorithm adds exactly these required edges  $\langle VC, p_j \rangle$  for all  $p_j \in DIRECT-PARENTS_{VC}$ . In addition, there need to be edges in  $G'$  to connect  $VC$  to all its direct subclasses, which by Definition 27 are equal to all classes in the  $DIRECT-CHILDREN_{VC}$  set. It can easily be seen that **step 4.2** of the algorithm adds exactly these edges  $\langle ch_k, VC \rangle$  for all  $ch_k \in DIRECT-CHILDREN_{VC}$ .

**Lemma 15. (Correctness)** *Given the schema  $G = (V, E)$  and a class  $VC$ . All classes  $C_i$  in  $G$  that are direct children of  $VC$  are subclasses of all classes  $C_k$  in  $G$  that are direct parents of  $VC$ .*

**Proof:** Lemma 15 follows directly from the transitive closure property of the *is-a* relationship. For a class  $C_i$  to be a direct subclass of a class  $VC$  in  $G$  implies, by transitivity, that  $C_i$  will also be subclass of all superclasses of  $VC$ . More formally,

$$(\forall C_i \in V) (\forall C_k \in V) ((C_i \text{ is-a }^d VC) \wedge (VC \text{ is-a }^d C_k) \implies (C_i \text{ is-a }^* C_k)).$$

From Lemma 15, I can derive the lemma given below.

**Lemma 16.** *Given the schema  $G = (V, E)$  and a class  $VC$ . The search for the classes  $C_i$  in  $G$  that belong to the  $DIRECT-CHILDREN_{VC}$  set has to consider only classes that are in subgraphs of  $G$  rooted at some class  $C_k$  that is a member of the  $DIRECT-PARENTS_{VC}$  set of  $G$ .*

Lemma 16 follows directly from Lemma 15 since classes in the  $DIRECT-CHILDREN_{VC}$  set are subclasses of  $VC$  and classes in the  $DIRECT-PARENTS_{VC}$  set are superclasses of  $VC$ . From Lemma 16, I can conclude that the search for the  $DIRECT-CHILDREN_{VC}$  set starts where the search for the  $DIRECT-PARENTS_{VC}$  set ends. See the example below for a demonstration of this idea.

**Example 41.** *In Example 40 I have discussed the application of the Class-Placement algorithm given in Figure 7.11 to the schema graph shown in Figure 7.12. I now want to show how Lemma 16 can be used to limit the search space of the algorithm. The algorithm first searches for the  $DIRECT-PARENTS_{VC}$  set of  $G$  by depth-first downwards traversal of  $G$  starting from root  $C0$ . Thereafter, it computes the  $DIRECT-CHILDREN_{VC}$  set also by a depth-first downwards traversal of  $G$ . However, rather than starting again from the root node  $C0$  of  $G$ , the search starts from the  $DIRECT-PARENTS_{VC}$  set of  $G$  (Lemma 16). For instance, the search of direct subclasses continues with the subgraphs rooted at the class nodes  $C3$  and  $C9$ . Other subgraphs, e.g., the ones rooted at the nodes  $C13$  and  $C10$ , do not need to be explored at all. In Figure 7.11, I have encircled the parts of the graph  $G$  traversed for either the search for the  $DIRECT-PARENTS_{VC}$  set or for the search for the  $DIRECT-CHILDREN_{VC}$  set by dotted lines. Note that these two search spaces do not overlap.*

## 7.4.2 Computing The Direct Parents Set

In this section, I describe an algorithm for the computation of the  $DIRECT-PARENTS_{VC}$  set. This then represents a solution for step (1) of the Class-Placement algorithm given in Figure 7.11. Based on Definition 27, I can make the following observation about the elements in the  $DIRECT-PARENTS_{VC}$  set.

**Lemma 17.** *Given the schema  $G = (V, E)$  and a class  $VC$ , then the following properties hold for the classes  $C_i$  that are direct superclasses of  $VC$  in  $G$ , i.e., that are in the  $DIRECT-PARENTS_{VC}$  set,*

- I. *All classes  $C_i$  in  $DIRECT-PARENTS_{VC}$  are subclasses of the root class  $C0$  of the schema:*

$$(\forall C_i \in DIRECT-PARENTS_{VC})(subsumes(C0, C_i)=true).$$

II. For all classes  $C_i$  in  $DIRECT-PARENTS_{VC}$ , none of its subclasses  $C_k$  in  $G$  subsumes  $VC$ :

$$(\forall C_k \in V) ((C_k \text{ is-a } * C_i) \implies (\text{subsumes}(C_k, VC) = \text{false})).$$

III. For all classes  $C_i$  in  $DIRECT-PARENTS_{VC}$ , all of its superclasses  $C_k$  in  $G$  also subsume  $VC$ :

$$(\forall C_k \in V) ((C_i \text{ is-a } * C_k) \implies (\text{subsumes}(C_k, VC) = \text{true})).$$

**Proof:**

**Part I:** Schema root.

**Part I** is true by default, since the root class of a schema is by definition a supertype and a superset of all classes in the schema. ■

**Part II:** Classes below the direct superclasses of  $VC$ .

By Definition 27, for a class  $C_i$  to be a direct superclass of  $VC$ , i.e.,  $C_i \in DIRECT-PARENTS_{VC}$ , means that the following holds:

$$(1) (VC \text{ is-a } C_i), \text{ and}$$

$$(2) (\nexists C_k \in V)(k \neq i)((VC \text{ is-a } * C_k) \wedge (C_k \text{ is-a } * C_i)).$$

For all classes  $C_k$  in  $G$  that are subclasses of the classes  $C_i$  in the  $DIRECT-PARENTS_{VC}$  set in  $G$ , I have the *is-a* relationship:

$$(C_k \text{ is-a } * C_i).$$

By Definition 27, this implies that none of the subclasses  $C_k$  of  $C_i$  can satisfy the subsumption relationship, i.e.,

$$((VC \text{ is-a } * C_k) = \text{false})$$

since, otherwise, there would exist classes  $C_k$  that contradict the definition of the  $DIRECT-PARENTS_{VC}$  set and thus it contradicts the initial assumption that the classes  $C_i$  are members of the  $DIRECT-PARENTS_{VC}$  set. I can conclude that  $\forall C_k \in subclasses(C_i)$  the condition  $subsumes(C_k, VC)$  fails. ■

**Part III:** Classes above the direct superclasses of  $VC$ .

By Definition 27, for all classes  $C_i$  in  $DIRECT-PARENTS_{VC}$ , I have

$$(VC \text{ is-a } ^d C_i).$$



By the Definition 8 of a schema graph, the classes  $C_k$  above the direct superclasses  $C_i$  of VC in G have the following *is-a* relationships with their subclasses  $C_i$ :

$$(C_i \text{ is-a } * C_k).$$

By transitivity, this implies the desired *is-a* relationship, namely,

$$(\forall C_i \in \text{DIRECT-PARENTS}_{VC}) (\forall C_k \in \text{superclasses}(C_i))$$

$$(((VC \text{ is-a } ^d C_i) \wedge (C_i \text{ is-a } * C_k)) \implies (VC \text{ is-a } * C_k)).$$

The later is equivalent to the desired subsumption relationship, namely,

$$\text{subsumes}(C_k, VC) = \text{true}. \quad \blacksquare$$

Lemma 17 provides us with conclusive information on the shape of the search space for the DIRECT-PARENTS<sub>VC</sub> set. Namely, by **Part I** of Lemma 17, I know that the root class C0 of G qualifies as superclass for any VC. All other superclasses of VC are also located in the upper half of the schema graph (above all classes that are not superclasses of VC) (**Part II**). Once I find a class  $C_i$  which fulfills the condition  $\text{subsumes}(C_i, VC)$  but for which none of its children fulfill the superclass condition of VC, then the class  $C_i$  is a direct parent of VC (**Part III**). In term of the schema graph this means that the DIRECT-PARENTS<sub>VC</sub> set corresponds to all classes that are at the *lower* borderline of classes that still are subsumed by VC. In summary, I can conclude that the search for direct parents of VC should start with the root class C0 of G (**Part I**). It should continue downwards while the encountered classes are still superclasses of VC (**Part II**). Once I find a class  $C_i$  for which none of its children is a superclass of VC, then the class  $C_i$  is a direct parent of VC (**Part III**) and I am done with the search for this branch.

I associate the label *success* of type Boolean with each class C to delimit this search to the upper part of the schema. The label *success()* determines whether the class C (or any of its subclasses on this branch) has been identified as a member of the DIRECT-PARENTS<sub>VC</sub> set. This label is used to propagate upwards the fact whether or not a 'successful' class has been located in a given subgraph.

As explained in the previous section, this algorithm is based on the depth-first traversal of the schema graph. However, since I allow for schema graphs with multiple inheritance, I need to assure that a subgraph of G does not get processed more than one. For this, I use a labeling scheme that marks classes C that have been processed by the label  $\text{processed}(C) = \text{true}$ . These labeled classes are not processed again.

**Algorithm outline:** Compute Direct-Parents of A Class.

**Input:**

A schema  $G = (V,E)$  with multiple inheritance with  $C_0$  the root.  
A class  $VC$  to be integrated into  $G$ .

**Output:**

$DIRECT-PARENTS_{VC}$ : a set of classes.

**Data Structures:**

label processed(class) : Boolean;  
label success(class) : Boolean;  
function subsumes(class1,class2) : Boolean;

**Algorithm:**

```

procedure Find-Direct-Parents (G,VC,DIRECT-PARENTSVC)
begin
  DIRECT-PARENTSVC := ∅;
  for all classes  $C_i$  do processed( $C_i$ ) := false; end for
  for all classes  $C_i$  do success( $C_i$ ) := false; end for
  Process-Node (root-of-schema);
end procedure
procedure Process-Node (C)
begin
  processed(C) := true;
  for all K in children(C) do begin
    if ((processed(K)=false) and (subsumes(K,VC)))
    then Process-Node (K);
    endif
    if (success(K)) then success(C) := true endif;
  end for
  if (success(C)=false)
  then begin
    DIRECT-PARENTSVC := DIRECT-PARENTSVC ∪ C;
    success(C) := true;
  endif
end procedure

```

Figure 7.15: An Algorithm for Computing the  $DIRECT-PARENTS_{VC}$  Set.

The algorithm for computing the  $\text{DIRECT-PARENTS}_{VC}$  set of  $VC$ , called the Find-Direct-Parents algorithm, is depicted in Figure 7.15. It proceeds as follows. The algorithm traverses the graph  $G$  depth-first starting from schema root. It goes down as far as possible while the function  $\text{subsumes}(C_i, VC)$  holds. If a class  $K$  is found that has already been processed, then the algorithm backtracks since the given branch needs to be explored but once. The search will also backtrack if the condition  $\text{subsumes}(K, VC)$  does not hold, since then neither the current class (nor any of its subclasses) will qualify as  $\text{DIRECT-PARENTS}_{VC}$  (Lemma 17). After all children of a class  $C$  have been processed, the algorithm proceeds as follows. If one or more of its children (or their subclasses) have been found to be 'successful', i.e., they were added to the  $\text{DIRECT-PARENTS}_{VC}$  set, then the class  $C$  need no longer be added to the  $\text{DIRECT-PARENTS}_{VC}$  set. If however none of the children did qualify as  $\text{DIRECT-PARENTS}_{VC}$ , then  $C$  is the lowest class on this subtree that still subsumes the virtual class  $VC$ . Hence,  $C$  is added to the  $\text{DIRECT-PARENTS}_{VC}$  set.

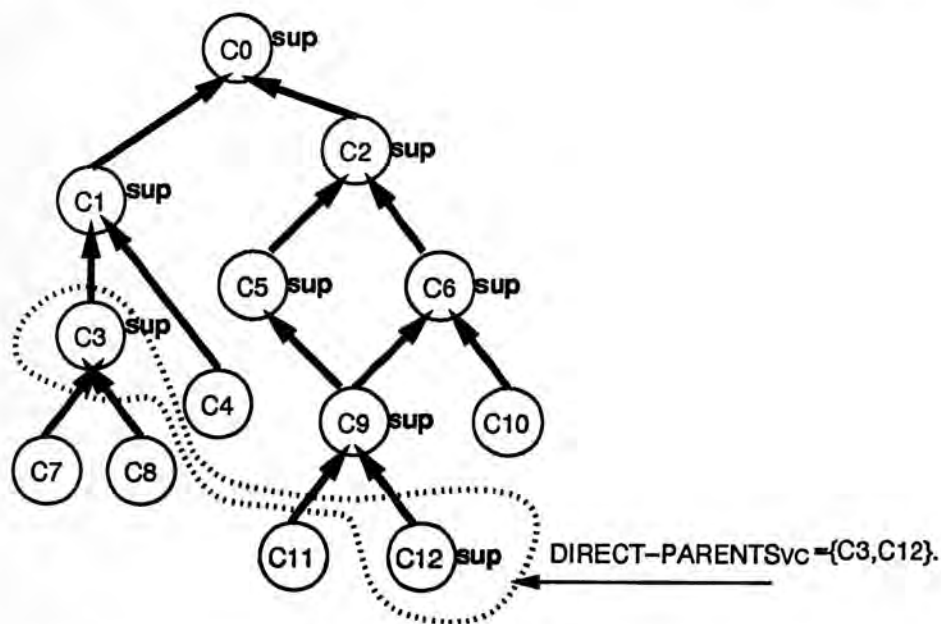


Figure 7.16: An Example of the Find-Direct-Parents Algorithm.

**Example 42.** In Figure 7.16, I demonstrate the Find-Direct-Parents algorithm given in Figure 7.15. In the figure, the label **sup** for a class  $C_i$  indicates that the condition  $\text{subsumes}(C_i, VC)$  evaluates to true, i.e.,  $C_i$  is a **superclass** of  $VC$ . After initialization, the algorithm starts the traversal of the schema graph  $G$  with the root node  $C_0$ . It processes the first child of  $C_0$ , which is  $C_1$ . Since the first if-statement is true, it then recursively processes the first child of  $C_1$ , which is  $C_3$ . For both children of  $C_3$ , which are  $C_7$  and  $C_8$ , the if-statement evaluates to false. Hence, the label

*success(C3)* remains false. The fourth if-statement then adds *C3* into the *DIRECT-PARENTS<sub>VC</sub>* set. The search now backtracks to complete the processing of *C1*. Since the child *C3* of *C1* carries the label *success(C3)=true*, the class *C1* is not added to the *DIRECT-PARENTS<sub>VC</sub>* set. The search now backtracks to continue processing of *C0*, which corresponds to exploring the right subgraph in the just described manner.

**Theorem 6. (Correctness)** Given the schema  $G = (V,E)$  and a class *VC*. The Find-Direct-Parents algorithm in Figure 7.15 will find all classes of *G*, called *DIRECT-PARENTS<sub>VC</sub>*, that are direct superclasses of *VC*.

**Proof:** I prove Theorem 6 in several steps.

**Part I:** The Find-Direct-Parents algorithm in Figure 7.15 will call the Process-Node function on each superclass of *VC* exactly once.

A schema  $G = (V,E)$  is a *connected* directed acyclic graph. This assures that a general depth-first search starting from the root will meet all leaf nodes exactly once (See a standard algorithm book, e.g., [Aho74], pg. 176 - 178). By Lemma 17, the shape of the search space for superclasses of *VC* in *G* is such that all indirect superclasses of *VC* are above all direct parents of *VC* which are above all other classes in the schema graph *G*. The Find-Direct-Parents algorithm traverses *G* in a depth-first manner starting from the root of *G* until it reaches class *C<sub>i</sub>* for which the condition *subsumes(C<sub>i</sub>,VC)* no longer holds, i.e., it will process all indirect and all direct parents of *C<sub>i</sub>*; and it backtracks as soon as it reaches other undesirable classes. The Find-Direct-Parents algorithm in Figure 7.15 recursively continues the search downwards if and only if the first if-statement evaluates to true, i.e., the condition “((*processed(K)=false*) and (*subsumes(K,VC)*))” is true. This means that the search continues only if the class *K* has not been processed before and if it is indeed a superclass of *VC*. This assures that the search stops after all superclasses of *VC* have been found and also that the same class is processed at most once.

**Part II:** When traversing downwards the graph, the algorithm encounters a class *C* with *processed(C)=true*, then *success(C)=true* must also hold.

First, the label *processed(C)* is set to true for classes *C* that have been processed before. Second, once a class *C<sub>i</sub>* is found to be a direct parent of *VC*, then the label *success(C<sub>i</sub>)* is set to true by the third if-statement. In addition, the second if-statement sets the *success* label of parents to true whenever the *success* label of a child is true. Hence, the *success* label of all parents of the class *C* is also changed to true when backtracking. The directed depth-first search will completely finish the processing of a class *C* (and all its successors) before possibly encountering the same class again via an alternative path (due to multiple inheritance). This can easily be

shown by using the facts that (1) there are no directed cycles in the schema graph and (2) the search starting at a class  $C$  will recursively process only its subclasses in the schema graph but no other class until terminating the processing of class  $C$ . Thus, the search encounters a class with the label  $processed(C)=true$  if and only if its label  $success(C)=true$  either due to being a direct parent or due to having backtracked passed this class.

**Part III:** The algorithm adds the class  $C_i$  of  $V$  into the  $DIRECT-PARENTS_{VC}$  set if and only if the class  $C_i$  is a direct superclass of  $VC$ .

**Part III.a:** If a class  $C_i$  of  $V$  is a direct superclass of  $VC$  then the algorithm will add  $C_i$  into the  $DIRECT-PARENTS_{VC}$  set.

Assume that the class  $C_i$  of  $V$  is a direct superclass of  $VC$ . Note that a class  $C_i$  is added into the  $DIRECT-PARENTS_{VC}$  set only by the last if-statement of the algorithm. For this if-statement to be executed for a class  $C_i$ , I must show that the label  $success(C_i)$  must be false.

First, if the class  $C_i$  has no children, then the for-loop over all children  $K$  of  $C_i$  will not be executed. Hence, the label  $success(C_i)$ , which is modified within this for-loop, is not changed and remains 'false'. If  $success(C_i)=false$ , then the last if-statement evaluates to true and adds  $C_i$  to the  $DIRECT-PARENTS_{VC}$  set.

Second, if the class  $C_i$  has some children  $K$ , then the for-loop over all children  $K$  of  $C_i$  will be executed. However, none of the children  $K$  of  $C_i$  will be a superclass of  $VC$  (by Lemma 17) and hence the function  $subsumes(K,VC)$  will fail for all of them. Therefore, the first if-statement within the for-loop will be skipped for all  $K$ . Hence, the label  $success(K)=false$  will not be modified, and the second if-statement within the for-loop will also be skipped for all  $K$ . This again implies that the label  $success(C_i)$  is not modified and remains 'false'. In this case, the last if-statement will add  $C_i$  to the  $DIRECT-PARENTS_{VC}$  set.

In summary, I have demonstrated that a class  $C_i$  that is a direct superclass of  $VC$  will always be added to  $DIRECT-PARENTS_{VC}$  set by the algorithm.

**Part III.b:** If the algorithm adds the class  $C_i$  of  $V$  into the  $DIRECT-PARENTS_{VC}$  set then the class  $C_i$  is a direct superclass of  $VC$ .

Assume that the algorithm has added the class  $C_i$  of  $V$  to the  $DIRECT-PARENTS_{VC}$  set. Note that there is only one statement, namely, the last if-statement of the algorithm, that would add a class  $C_i$  into the  $DIRECT-PARENTS_{VC}$  set. For this if-statement to be executed, the label  $success(C_i)$  must have been false.

### 7.4.3 Computing The Direct Children Set

In this section, I present algorithms for computing the set of direct children of a new class VC in a schema graph G. An algorithm similar to the one for computing the direct parents set (Figure 7.15) can be used. The algorithm traverses the schema graph downwards in a depth-first manner, until it finds a class  $C_i$  that fulfills the condition  $subsumes(VC, C_i)$ . This class  $C_i$  is a *direct* subclass of VC, since any class above it that is also subsumed by VC would have been found before reaching  $C_i$ . See Figure 7.17 for a precise formulation of the just outlined algorithm.

**Algorithm outline:** Compute The Direct-Children classes of A Class.

**Input:**

A schema  $G=(V,E)$  with single inheritance and a class VC to be integrated into G.  
 DIRECT-PARENTS<sub>VC</sub>: a set of direct-parent classes of VC in G.

**Output:**

DIRECT-CHILDREN<sub>VC</sub>: a set of direct-children classes of VC in G.

**Data Structures:**

function  $subsumes(class1, class2)$  : Boolean;

**Algorithm:**

```

procedure Find-Direct-Children1(G, VC, DIRECT-PARENTSVC)
return DIRECT-CHILDRENVC
begin
  DIRECT-CHILDRENVC :=  $\emptyset$ ;
  for all classes  $C_i$  in DIRECT-PARENTSVC do Process-Node ( $C_i$ ); end for
end procedure
procedure Process-Node (class C)
begin
  if ( $subsumes(VC, C)$ )
  then DIRECT-CHILDRENVC := DIRECT-CHILDRENVC  $\cup$  C;
  else for all classes K in children(C) do Process-Node (K); end for
  endif
end procedure

```

Figure 7.17: An Algorithm for Computing the DIRECT-CHILDREN<sub>VC</sub> Set for Single Inheritance.

The Find-Direct-Children1 algorithm in Figure 7.17 works on a schema  $G = (V,E)$  with single inheritance. Such a schema G with single inheritance corresponds to a graph structure in the form of a tree, and hence a depth-first traversal of G will

not traverse the same subgraph twice. Hence, the *processed* label introduced earlier to mark already processed classes is not needed for graphs  $G$  with single inheritance.

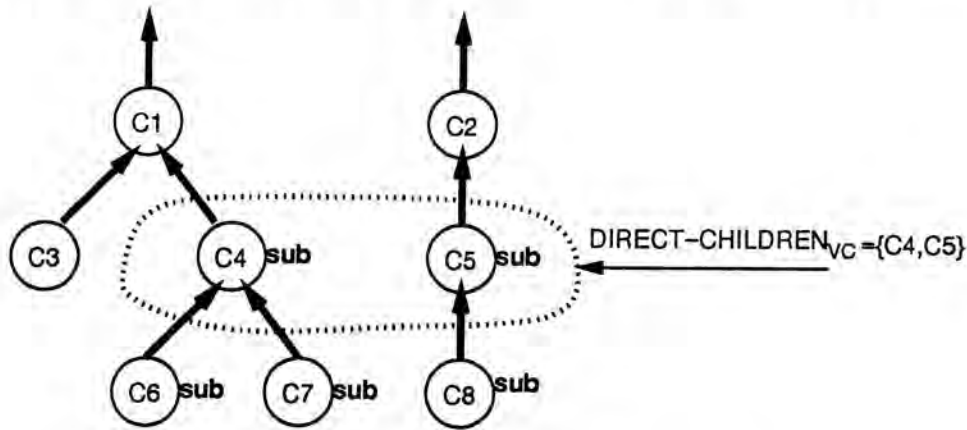


Figure 7.18: An Example of the Find-Direct-Children1 Algorithm.

**Example 43.** In this example, I apply the Find-Direct-Children1 algorithm given in Figure 7.17 to find the direct subclasses of the class  $VC$  in the schema graph  $G$  shown in Figure 7.18. I assume that the graph shown in Figure 7.18 corresponds to a subgraph of the complete schema graph  $G$  and that the  $DIRECT-PARENTS_{VC}$  set has been determined to be  $\{C1, C2\}$ . The search starts by traversing  $G$  downwards starting from the classes  $\{C1, C2\}$ . The algorithm first processes the class  $C1$  using the *Process-Node()* function. Since the first if-statement fails, the execution falls into the else-branch. Next, the for-loop iterates over the children of  $C1$ , which are  $C3$  and  $C4$ . For the first iteration of the for-loop with  $K=C3$ , the if-statement fails and the execution continues with the else-branch. However, since  $C3$  has no children, the for-loop in the else-branch is not executed. The algorithm then processes the second child of  $C1$ , which is  $K=C4$ . The first if-statement evaluates to true, i.e.,  $C4$  is indeed a subclass of  $VC$ , and the then-branch adds  $C4$  to the  $DIRECT-CHILDREN_{VC}$  set. The else-branch is skipped, and hence none of the classes below  $C4$  will be processed. The search along this branch is completed. The algorithm backtracks to process the next class in the  $DIRECT-PARENTS_{VC}$  set,  $C2$ , in the just described manner.

**Theorem 8. (Correctness)** Given the schema  $G = (V, E)$  with single inheritance and a class  $VC$ , then the Find-Direct-Children1 algorithm in Figure 7.17 will find all classes of  $G$ , called  $DIRECT-CHILDREN_{VC}$ , that are direct subclasses of  $VC$ .

**Proof:** A schema  $G = (V, E)$  is a *connected* directed acyclic graph. This assures that a general depth-first search starting from the root will meet all leaf nodes exactly

<sup>2</sup>I assume that the function *subsumes* is computable for our object model and by the terms 'all direct subclasses' I mean all direct subclasses recognizable by the *subsumes* function.

once (See a standard algorithm book, e.g., [Aho74], pg. 176 - 178). By Lemma 16, it is sufficient to start the the search for the  $DIRECT-CHILDREN_{VC}$  set from the classes in the  $DIRECT-PARENTS_{VC}$  set on downwards rather than from the root of the schema graph. By Definition 27, the  $DIRECT-CHILDREN_{VC}$  set contains all classes that fulfill the following conditions: (1) they are subclasses of VC and (2) there are no other classes above them in the schema graph that are also subclasses of VC. The later means that they are the **highest** possible classes in the schema graph that are still subsumed by VC. Due to the depth-first traversal of the schema graph G, the Find-Direct-Children1 algorithm finds the 'highest' classes of G that are subsumed by VC first. In fact, the search terminates the search for a given branch of the tree G, whenever the condition " $subsumes(VC,C)$ " evaluates to true. By Definition 27, these highest subclasses C of VC are indeed the direct children of VC. These classes are thus added to the  $DIRECT-CHILDREN_{VC}$  set by the then-branch of the if-statement, and the search along this branch of the schema graph is terminated. ■

**Theorem 9. (Complexity)** *Given a schema  $G = (V,E)$  with single inheritance and a class VC, then the Find-Direct-Children1 algorithm in Figure 7.17 has a worst case complexity of  $O(|E|)$  with  $|E|$  the number of edges in the schema.*

**Proof:** I assume a set representation (e.g., for the sets  $DIRECT-CHILDREN_{VC}$  and  $DIRECT-PARENTS_{VC}$ ) that allows for the manipulation of the set by adding or deleting elements and for the checking its membership for a particular element in  $O(1)$  time, e.g., a vector representation of G. The proof is then similar to those for the standard depth-first search (See a standard algorithm book, e.g., [Aho74], pg. 176 - 178). The functions to initialize the search require  $O(|V|)$  steps if a list of class nodes is available. The time spend in the Process-Node (C) function for a given class C is proportional to the number of out-going edges (or the number of children of C). The function Process-Node (C) is called at most once for a given class C, since for a graph G with single inheritance, a depth-first traversal will by default encounter each node in the tree exactly once (See [Aho74], pg. 176 - 178). Therefore the worst case complexity is  $O(\max(|V|,|E|))$ . Note that in a connected directed graph the number of edges  $|E|$  in the schema is always larger or equal to the number of classes  $|V|$  in the schema.  $|E| \geq |V|$  implies that  $complexity(\text{Find-Direct-Children1}) := O(\max(|V|,|E|)) := O(|E|)$ . ■

The Find-Direct-Children1 algorithm in Figure 7.17 works on a schema  $G = (V,E)$  with single inheritance. If I want to apply the algorithm to a schema G with multiple inheritance, which corresponds to a DAG structure rather than to a tree structure, then I need to assure that the depth-first traversal will not traverse the same subgraph of G more than once. Hence, I introduce the label *processed* into



the Find-Direct-Children1 algorithm in Figure 7.17 to mark classes that have been processed before. The resulting algorithm is shown in Figure 7.19.

**Algorithm outline:** Compute The Direct-Children classes of A Class.

**Input:**

A schema  $G=(V,E)$  with multiple inheritance and class  $VC$  to be integrated into  $G$ .  
 $DIRECT-PARENTS_{VC}$ : a set of direct-parent classes of  $VC$  in  $G$ .

**Output:**

$DIRECT-CHILDREN_{VC}$ : direct- and possibly indirect-children of  $VC$  in  $G$ .

**Data Structures:**

label  $processed(class)$  : Boolean;  
 function  $subsumes(class1,class2)$  : Boolean;

**Algorithm:**

```

procedure Find-Direct-Children2( $G,VC,DIRECT-PARENTS_{VC}$ )
return  $DIRECT-CHILDREN_{VC}$ 
begin
   $DIRECT-CHILDREN_{VC1} := \emptyset$ ;
  for all classes  $C_i$  in  $V$  do  $processed(C_i) := false$ ; end for
  for all classes  $C_i$  in  $DIRECT-PARENTS_{VC}$  do
     $processed(C_i) := true$ ;
    Process-Node ( $C_i$ );
  end for
end procedure
procedure Process-Node (class  $C$ )
begin
  if ( $subsumes(VC,C)$ )
  then  $DIRECT-CHILDREN_{VC} := DIRECT-CHILDREN_{VC} \cup C$ ;
  else begin
    for all classes  $K$  in  $children(C)$  do begin
      if ( $processed(K)=false$ )
      then begin
         $processed(K):=true$ ;
        Process-Node ( $K$ );
      endif
    end for
  endif
end procedure

```

Figure 7.19: An Algorithm for Computing the  $DIRECT-CHILDREN_{VC}$  Set for Multiple Inheritance.

**Theorem 10. (Correctness)** Given the schema  $G = (V,E)$  with possibly multiple inheritance and a class  $VC$ , then the Find-Direct-Children2 algorithm in Figure 7.19

will find all classes of  $G$ , called  $DIRECT-CHILDREN_{VC}$ , that are direct subclasses of  $VC$ . In addition, the algorithm might select some classes  $C_i$  that are indirect subclasses of  $VC^3$ .

**Proof:**

**Part I:** The Find-Direct-Children2 algorithm finds all  $DIRECT-CHILDREN_{VC}$  of  $VC$ .

Theorem 8 shows that the Find-Direct-Children1 algorithm finds all  $DIRECT-CHILDREN_{VC}$  of  $VC$  for a graph  $G$  with single-inheritance. The same reasoning can be used here to argue that the Find-Direct-Children2 algorithm finds all direct subclasses of  $VC$  for a graph  $G$  with multiple-inheritance. Namely, by depth-first traversal, each relevant branch of  $G$  is explored (once) and all classes  $C_i$  for which the condition " $subsumes(VC,C)$ " evaluates to true are collected in the set  $DIRECT-CHILDREN_{VC}$ .

**Part II:** The Find-Direct-Children2 algorithm may also find indirect subclasses of  $VC$ .

In a graph  $G$  with multiple inheritance, a class node  $C$  may have more than one parent. Hence, I may arrive at the same node  $C$  of  $G$  more than once during a depth-first traversal. If I arrive at such a node  $C_i$  from one parent while one of its other parents is also subsumed by  $VC$ , then the class  $C_i$  will be redundant in the  $DIRECT-CHILDREN_{VC}$  set. An example of this situation is depicted in Figure 7.20. ■

**Example 44.** The schema graph in Figure 7.20 is identical to the one in Figure 7.18, except for the addition of the arc  $e = \langle C6, C3 \rangle$ , which turns the schema into a graph with multiple inheritance. The Find-Direct-Children2 algorithm given in Figure 7.19 proceeds as described in Example 43. However, when processing the class  $C3$ , the algorithm now also processes  $C6$ , the new child of  $C3$ . Since  $C6$  is the first class on this branch that fulfills the  $subsumes(VC, C6)$  condition,  $C6$  is added to the  $DIRECT-CHILDREN_{VC}$  set. Next,  $C4$  and  $C5$  are also added to the  $DIRECT-CHILDREN_{VC}$  set as explained in Example 43. It can easily be seen that  $C6$  is not a direct child of  $VC$ , since there is another class,  $C4$ , in the graph that is a direct child of  $VC$  and that is a superclass of  $C6$ . Hence,  $C6$  is redundant in the  $DIRECT-CHILDREN_{VC}$  set.

**Theorem 11. (Complexity)** Given a schema  $G = (V, E)$  with multiple inheritance and a class  $VC$ , then the Find-Direct-Children2 algorithm in Figure 7.19 has a worst case complexity of  $O(|E|)$  with  $|E|$  the number of edges in the schema.

<sup>3</sup>Again, I assume that the function  $subsumes$  is computable.

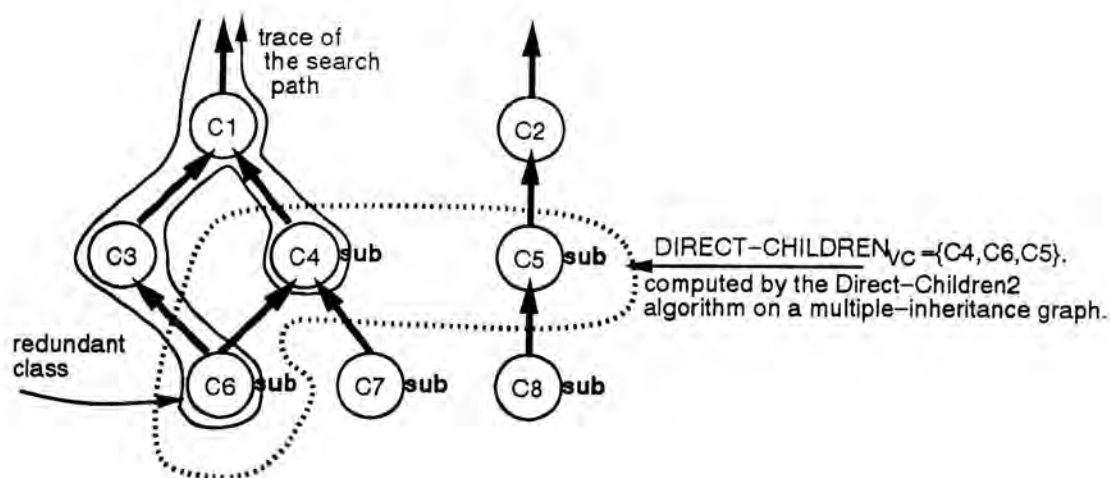


Figure 7.20: An Example of the Find-Direct-Children2 Algorithm.

**Proof:** The Find-Direct-Children2 algorithm in Figure 7.19 is identical to the Find-Direct-Children1 algorithm in Figure 7.17, except for the introduction of the *processed* label to delimit the search on a graph with multiple inheritance. Hence, the proof proceeds similar to Theorem 9. The only difference is the reason for why the function Process-Node (C) is called at most once for a given class C given below. Namely, the Find-Direct-Children2 algorithm marks all classes by the label *processed* the first time they are called. When a class C is encountered for a second time, then its label *processed*(C) will be true. For a class C with the label *processed*(C)=true, the if-statement evaluates to false and the Process-Node (C) function call is skipped. Hence, the Process-Node (C) function is called but once for each class C. ■

**Lemma 18.** Given the schema  $G = (V, E)$  and a class VC, then the following properties hold for classes  $C_i$  that are either direct or indirect subclasses of VC in G:

I. For all  $C_i$  that are direct subclasses of VC, i.e., they are true members of the  $DIRECT-CHILDREN_{VC}$  set, the following holds:

$$(\forall C_k \in \text{parents}(C_i))(\text{subsumes}(VC, C_k) = \text{false}).$$

II. For all  $C_i$  that are indirect subclasses of VC, i.e., they are redundant members of the  $DIRECT-CHILDREN_{VC}$  set, the following holds:

$$(\exists C_k \in \text{parents}(C_i))(\text{subsumes}(VC, C_k) = \text{true}).$$

**Proof:**

**Part I:** For direct subclasses of VC.

Assume that the class  $C_i$  is a direct subclass of VC. Then the following must hold by Definition 27:

- (1)  $(C_i \text{ is-a } VC)$ , and
- (2)  $(\exists C_k \in V)$  with  $(C_i \text{ is-a } * C_k) \wedge (C_k \text{ is-a } * VC)$ .

The second condition can be rewritten as " $(\exists C_k \in \text{parents}(C_i))$  with  $(C_k \text{ is-a } * VC)$ ", since if there is a class  $C_k$  with the above property then there is at least one direct superclass (parent) of  $C_i$  with the same property. This, of course, is equivalent to the condition that " $(\forall C_k \in \text{parents}(C_i)) (\text{subsumes}(VC, C_k) := \text{false})$ " given in Lemma 18.I.

**Part II:** For indirect subclasses of VC.

For a class  $C_i$  to be an indirect – but not a direct – subclass of VC means that the following holds:

- (1)  $(C_i \text{ is-a } VC)$ , and
- (2)  $(\exists C_k \in V)$  with  $(C_i \text{ is-a } * C_k) \wedge (C_k \text{ is-a } * VC)$ .

The first condition indicates that  $C_i$  is a subclass of VC and the second condition is a negation of the requirement that the subclass relationship must be *direct*. The second condition can be rewritten as " $(\exists C_k \in \text{parents}(C_i))(C_k \text{ is-a } * VC)$ ", since if there is a class  $C_k$  with the above property then there is at least one direct superclass (parent) of  $C_i$  with the same property. This is equivalent to the condition listed in Lemma 18.II. ■

Next, I demonstrate these definitions by an example.

**Example 45.** As discussed in Example 44, the Find-Direct-Children2 algorithm finds the  $\text{DIRECT-CHILDREN}_{VC}$  set  $:= \{ C4, C6, C5 \}$  for the schema graph in Figure 7.20. By Lemma 18.I, the class C4 is a true member of the  $\text{DIRECT-CHILDREN}_{VC}$  set, since its only parent C1 is not subsumed by VC. Similarly, the class C5 is a true member of the  $\text{DIRECT-CHILDREN}_{VC}$  set, since its parent C2 is not subsumed by VC. However, by Lemma 18.II, the class C6 is not a direct subclass of VC, since one of its parents, namely, C4, is also subsumed by VC. Since there is at least one class between VC and the class C6, namely, its parent C4, the class C6 is a redundant member of the  $\text{DIRECT-CHILDREN}_{VC}$  set (Lemma 18.II).

Based on Lemma 18, I can derive an algorithm that removes all redundant classes (indirect subclasses of VC) from the  $\text{DIRECT-CHILDREN}_{VC}$  set generated

by the Find-Direct-Children2 algorithm. Namely, for each class  $C_i$  in the *DIRECT-CHILDREN<sub>VC</sub>* set, I test the condition outlined in Lemma 18. If the condition in Lemma 18.a is true, then the class  $C_i$  is indeed a *direct child* of VC and should remain in the *DIRECT-CHILDREN<sub>VC</sub>* set. If the condition in Lemma 18.b is true, then the class  $C_i$  is not a *direct child* of VC and should be removed from the *DIRECT-CHILDREN<sub>VC</sub>* set. This algorithm has been implemented by the Remove-Redundant-Classes procedure shown in Figure 7.21.

**Algorithm outline:** Remove all indirect subclasses of VC from *DIRECT-CHILDREN<sub>VC</sub>*.

**Input:**

A schema  $G=(V,E)$  with multiple inheritance and class VC to be integrated into G.  
Set *DIRECT-CHILDREN<sub>VC</sub>* with all direct and some indirect subclasses of VC.

**Output:**

*DIRECT-CHILDREN<sub>VC</sub>* with all direct children of VC.

**Data Structures:**

variable redundant: Boolean;  
function *subsumes*(class1,class2) : Boolean;  
function *remove-from-set*(class,set-of-classes);

**Algorithm:**

```

procedure Remove-Redundant-Classes(G,VC,DIRECT-CHILDRENVC)
return DIRECT-CHILDRENVC
begin
  for all  $C_i$  in DIRECT-CHILDRENVC do begin
    redundant := false;
    for all K in parents( $C_i$ ) do
      if (subsumes(VC,K)) then redundant:=true; endif
    end for
    if (redundant=true) then remove-from-set( $C_i$ ,DIRECT-CHILDRENVC);
    endif
  end for
end procedure

```

Figure 7.21: An Algorithm for Removing all Indirect Subclasses from the *DIRECT-CHILDREN<sub>VC</sub>* Set.

**Example 46.** The schema graph in Figure 7.22 is identical to the one in Figure 7.20. Example 44 explained how all direct and possibly some indirect subclasses of VC are found, i.e., *DIRECT-CHILDREN<sub>VC</sub>* := {  $C_4$ ,  $C_6$ ,  $C_5$  }, while in this example I show how all indirect classes are removed from this set. The algorithm proceeds as follows. The first iteration of the first for-loop with  $C_i=C_4$  checks whether

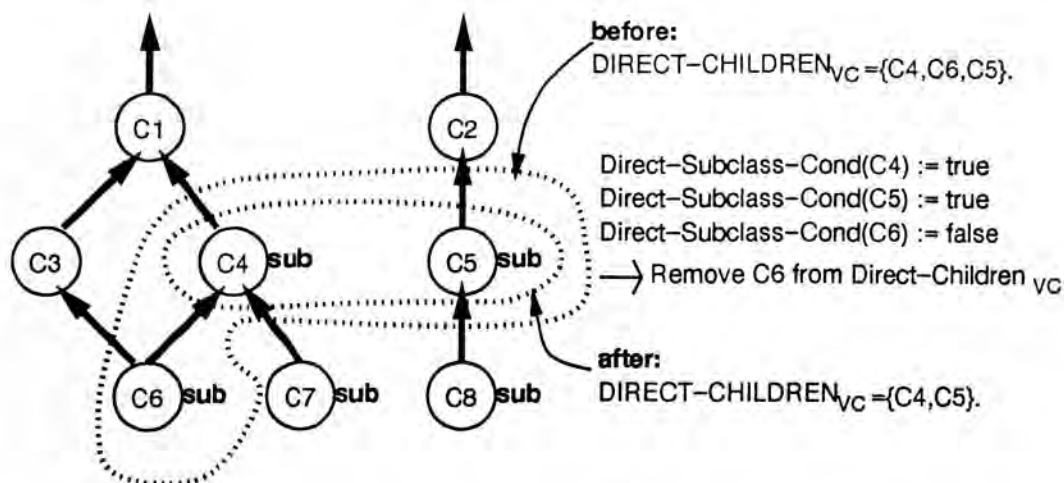


Figure 7.22: An Example of the Remove-Redundant-Classes Algorithm.

the redundancy condition given in Lemma 18.1 applies for  $C4$ . The second for-loop iterates over all parents of  $C4$  to check whether any of them are subsumed by  $VC$ . Since  $C1$ , the only parent of  $C4$ , is not, the variable  $redundant := false$  is not modified.  $C4$  is found to be not redundant. The second iteration of the first for-loop with  $C_i = C6$  checks whether  $C6$  is redundant. The second for-loop iterates over all parents of  $C6$ , which are  $C3$  and  $C4$ . For  $K = C3$ , the first if-statement fails. However, for  $K = C4$ , the first if-statement succeeds and the variable  $redundant$  is set to true. Consequently, the second if-statement evaluates to true and removes  $C6$  from the  $DIRECT-CHILDREN_{VC}$  set.  $C6$  has been found to be redundant. Lastly, the third iteration of the for-loop with  $C_i = C5$  determines that  $C5$  is not redundant. In summary, the classes  $C4$  and  $C5$  remain in the  $DIRECT-CHILDREN_{VC}$  set.

Below, I show that the Remove-Redundant-Classes procedure in Figure 7.21 implements the algorithm for removing all redundant members (indirect subclasses) and none of the true members (direct subclasses) of the  $DIRECT-CHILDREN_{VC}$  set as described in the previous paragraph.

**Theorem 12. (Correctness)** Given the schema  $G = (V, E)$  with possibly multiple inheritance, a class  $VC$ , and the set  $DIRECT-CHILDREN_{VC}$  set composed of all direct subclasses of  $VC$  and possibly some indirect subclasses of  $VC$ . Then the Remove-Redundant-Classes algorithm in Figure 7.21 removes all of the indirect and none of the direct subclasses of  $VC$  from the set  $DIRECT-CHILDREN_{VC}$ .

**Proof:** For a class  $C_i$  in the set  $DIRECT-CHILDREN_{VC}$ , the Remove-Redundant-Classes algorithm checks for the condition described in Lemma 18, namely, the condition  $(\forall C_k \in parents(C_i))(subsumes(VC, C_k))$  for  $C_i$  a subclass of  $VC$ . The flag

*redundant* remains false if and only if the condition of the if-statement never evaluates to true. This is equivalent to  $(\forall C_k \in \text{parents}(C_i))(subsumes(VC, C_k)=false)$ , which is equal to the first condition listed in Lemma 18.I. Hence, by Lemma 18.I, the class  $C_i$  is a direct subclass of VC and is thus not removed from the  $\text{DIRECT-CHILDREN}_{VC}$  set by the second if-statement. The flag *redundant* is set to true if and only if the condition of the if-statement evaluates to true at least once. This is equivalent to the second condition listed in Lemma 18.II, namely, there exists a class  $C_k$  in  $\text{parents}(C_i)$  for which  $(subsumes(VC, C_k)=true)$ . By Lemma 18.II, the class  $C_i$  is an indirect subclass of VC and thus has to be removed from the  $\text{DIRECT-CHILDREN}_{VC}$  set. It can easily be seen that this is done by the second if-statement. After running this test for all classes in the  $\text{DIRECT-CHILDREN}_{VC}$  set, all indirect subclasses of VC have been removed and thus  $\text{DIRECT-CHILDREN}_{VC}$  is left with the direct subclasses of VC. ■

**Algorithm outline:** Find  $\text{Direct-Children}_{VC}$  For a Graph G with Multiple Inheritance.

**Input:**

Schema  $G = (V, E)$  with multiple inheritance and class VC to be integrated into G.  
 $\text{DIRECT-PARENTS}_{VC}$ : set of all direct parents of VC.

**Output:**

$\text{DIRECT-CHILDREN}_{VC}$ : set of all direct children of VC.

**Algorithm:**

```

procedure Find-Direct-Children(G, VC, DIRECT-PARENTSVC)
return DIRECT-CHILDRENVC
begin
  DIRECT-CHILDRENVC :=
    Find-Direct-Children2(G, VC, DIRECT-PARENTSVC);
  DIRECT-CHILDRENVC :=
    Remove-Redundant-Classes(G, VC, DIRECT-CHILDRENVC);
end procedure

```

Figure 7.23: An Algorithm for Finding All Direct-Children for Multiple Inheritance.

**Theorem 13. (Correctness)** *Given the schema  $G = (V, E)$  with possibly multiple inheritance, a class VC, and a set  $\text{DIRECT-PARENTS}_{VC}$  consisting of all direct parents of VC in G. Then the algorithm Find-Direct-Children in Figure 7.23 finds all classes of G, called  $\text{DIRECT-CHILDREN}_{VC}$ , that are direct subclasses of VC.*

**Proof:** By Theorem 10, I know that the Find-Direct-Children2 algorithm will find all classes of G that are direct subclasses of VC plus possibly some classes  $C_i$  that are

indirect subclasses of  $VC$  in  $G$ . By Theorem 12, I know that the Remove-Redundant-Classes algorithm removes all indirect subclasses of  $VC$  and none of the direct subclasses of  $VC$  from  $DIRECT-CHILDREN_{VC}$ . Hence, all direct subclasses of  $VC$  will remain in the  $DIRECT-CHILDREN_{VC}$  set. ■

**Theorem 14. (Complexity)** *Given the schema  $G = (V, E)$  and a class  $VC$ , then the Find-Direct-Children algorithm in Figure 7.23 has a worst case complexity of  $O(|E|)$  with  $|E|$  the number of edges in the schema.*

**Proof:** By Theorem 9, the complexity of the first procedure, called Find-Direct-Children2, is  $O(\max(|V|, |E|)) = O(|E|)$ . The complexity of the second procedure, called Remove-Redundant-Classes, is  $O(\sum_{C_i \in DIRECT-CHILDREN_{VC}} (\#parents(C_i)))$  with  $\#parents(C_i)$  equal to the number of outgoing *is-a* edges of  $C_i$  in  $G$ , denoted by  $\#outgoing-edges(C_i)$ . The sum  $\sum_{C_i \in DIRECT-CHILDREN_{VC}} (\#outgoing-edges(C_i))$  is of course smaller or equal to  $|E|$ . In summary, the complexity of the Find-Direct-Children algorithm is  $complexity(\text{Find-Direct-Children}) := complexity(\text{Find-Direct-Children2}) + complexity(\text{Remove-Redundant-Classes}) = O(|E| + |E|) = O(|E|)$ . ■

#### 7.4.4 Summary

Figure 7.11 in Section 7.4.1 presents the general algorithm for integrating a virtual class  $VC$  into a global schema  $G$ . In Sections 7.4.2 and 7.4.3, I present algorithms for subtasks (1) and (3) of the Class-Placement algorithm, respectively. The reader is referred to Example 40 for an example of the complete algorithm. The complexity of the class placement algorithm given in Figure 7.11 is linear, since both the computation of the direct parent and the direct children set can be done in linear time as shown in Sections 7.4.2 and 7.4.3, respectively.

### 7.5 Putting Everything Together: An Algorithm for Class Integration

In this section, I present the general solution for classification based on the algorithms developed in previous sections (see also Figure 7.9). In Section 7.2, I have demonstrated that the introduction of a class with a new type into a schema graph may require the creation of yet additional classes. The Generate-Intermediate-Classes( $G, VC$ ) procedure addresses this problem by preparing the type hierarchy for



insertion of a virtual class by adding all intermediate classes required to preserve the correct type lattice underlying the schema graph. Thereafter, I need to determine the correct location for VC in this prepared schema graph. The Class-Placement( $G, VC$ ) procedure presented in Section 7.4 solves exactly this problem. Finally, putting these two algorithms together as done in Figure 7.9 results in a class integration algorithm that solves the general classification problem (Theorem 14).

**Input:**

A schema  $G = (V, E)$  and a class VC.

**Output:**

VC integrated into G (with possibly additional intermediate classes).

**Algorithm:**

**procedure** Integration( $G, VC$ )

**begin**

// Prepare G by adding all intermediate classes required for integration of VC.

Generate-Intermediate-Classes( $G, VC$ );

// Insert VC into this modified schema graph G.

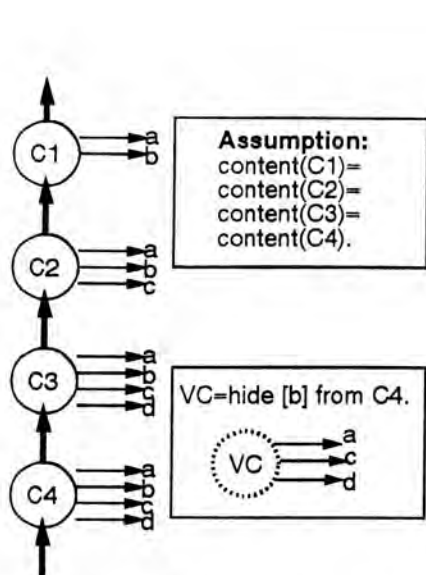
Class-Placement( $G, VC$ );

**end procedure**

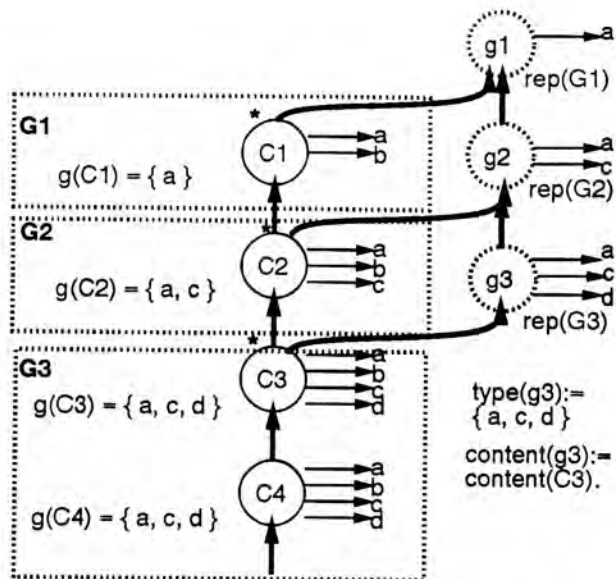
Figure 7.24: The Complete Class Integration Algorithm.

Below, I present two examples that demonstrate the classification algorithm given in Figure 7.24.

**Example 47.** *Figure 7.25 demonstrates how the classification algorithm given in Figure 7.24 inserts a class VC into a class hierarchy G. Figure 7.25.a shows G before type classification. For this example, I assume that all four classes have the same object instances as members, i.e.,  $\text{content}(C1) = \text{content}(C2) = \text{content}(C3) = \text{content}(C4)$ . The virtual class VC is derived by the view derivation "VC = hide [b] from C4". Then the type of VC is defined by  $\text{type}(VC) := [a, c, d]$  and the object membership of VC is defined by  $\text{content}(VC) := \text{content}(C4)$ . I first run the Generate-Intermediate-Classes( $G, VC$ ) procedure to modify the class hierarchy G such as to prepare its underlying type hierarchy for the insertion of the new class VC. As explained in Example 47 and shown in Figure 7.25.b, this procedure generated the three intermediate classes  $g_1, g_2$ , and  $g_3$ . Next, I apply the Class-Placement( $G, VC$ ) procedure to add VC to the now prepared schema graph G (Figure 7.25.c). By traversing G downwards starting from the root, the algorithm establishes the set  $\text{DIRECT-PARENTS}_{VC} := \{g_3\}$ .  $g_3$  is a direct parent of VC since (1)  $g_3$  subsumes VC with  $\text{type}(g_3) = [a, c, d] = \text{type}(VC)$  and  $\text{content}(g_3) = \text{content}(C3) = \text{content}(C4) = \text{content}(VC)$ , and (2) none of  $g_3$ 's children does subsume VC. The algorithm checks whether VC is contained in the  $\text{DIRECT-PARENTS}_{VC}$  set. Since  $\text{type}(g_3) = \text{type}(VC)$  and  $\text{content}(g_3)$*



(a) Given a schema G and virtual class VC defined by "VC= hide [b] from C4".



(b) Prepare the schema graph G for insertion of VC by inserting the intermediate classes  $g_i$  into G.

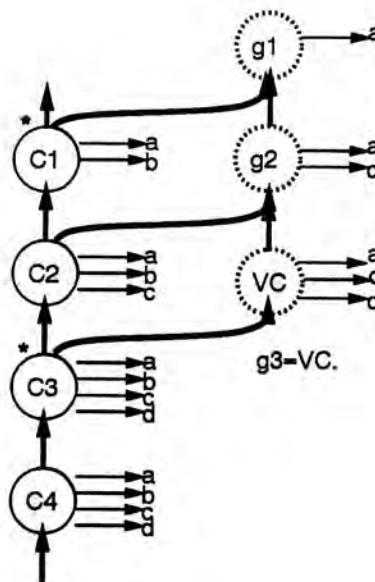
**Class-Integration Algorithm:**

**Search:**

$DIRECT-PARENTS_{VC} = \{g3\}$ .  
 VC in  $DIRECT-PARENTS_{VC}$   
 therefore stop and rename  $g3$  by VC.

**Edge manipulation:**

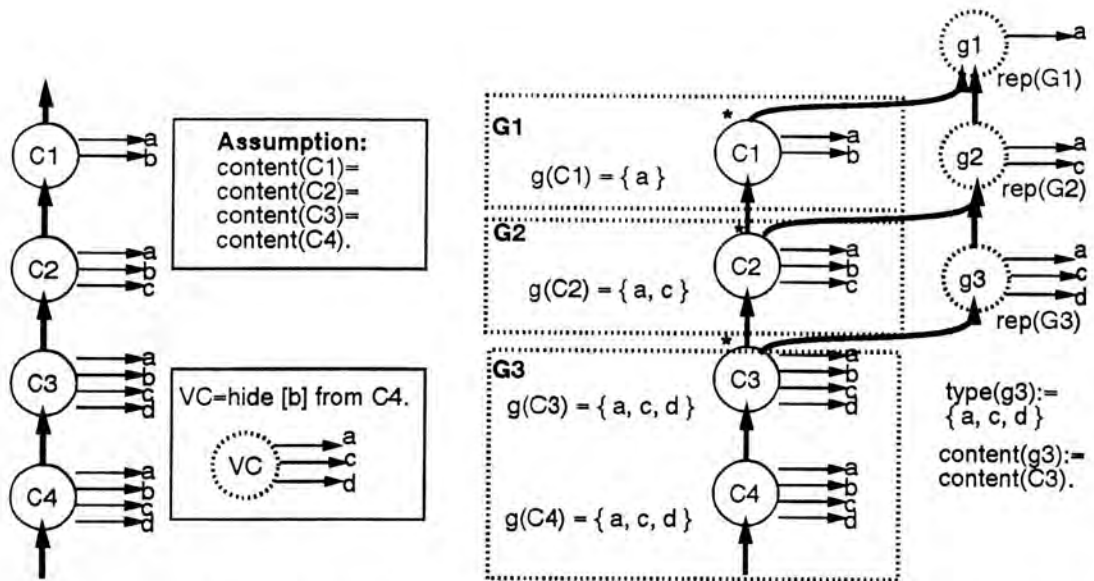
None.



(c) Class Integration of VC into G: tracing through the algorithm.

(d) Class Integration of VC into G: the resulting schema graph G.

Figure 7.25: An Example of Complete Classification (Identical Set Contents).



(a) Given a schema G and virtual class VC defined by "VC= hide [b] from C4".

(b) Prepare the schema graph G for insertion of VC by inserting the intermediate classes gi into G.

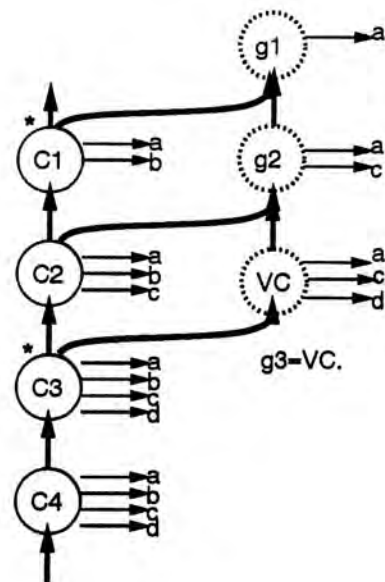
**Class-Integration Algorithm:**

**Search:**

$\text{DIRECT-PARENTS}_{VC} = \{g3\}$ .  
 VC in  $\text{DIRECT-PARENTS}_{VC}$   
 therefore stop and rename g3 by VC.

**Edge manipulation:**

None.



(c) Class Integration of VC into G: tracing through the algorithm.

(d) Class Integration of VC into G: the resulting schema graph G.

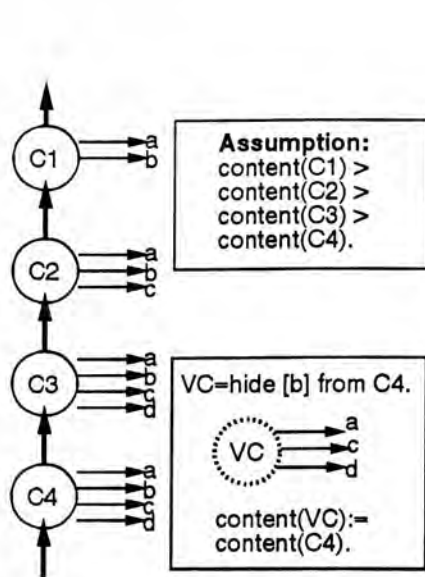
Figure 7.25: An Example of Complete Classification (Identical Set Contents).

$= \text{content}(VC)$ , the two classes  $VC$  and  $g_3$  are found to be equivalent and no new class needs to be added. The intermediate class  $g_3$  is simply renamed by the name of  $VC$  (Figure 7.25.d). Since the class  $g_3$  is already properly integrated in the schema graph, no further manipulation of the graph edges is needed.

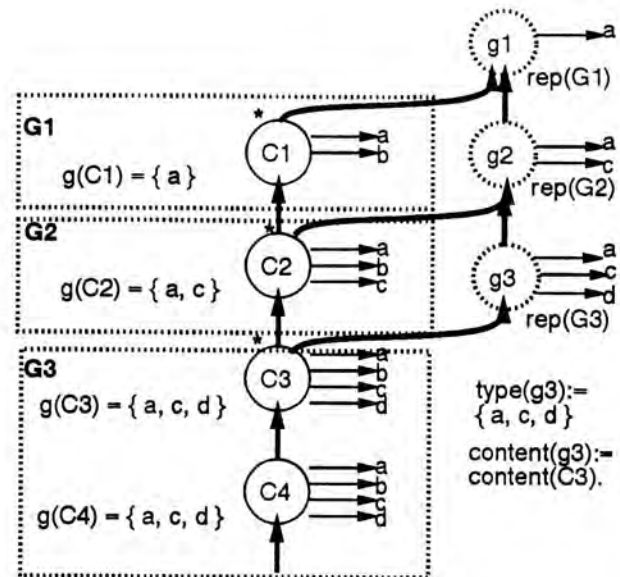
**Example 48.** Figure 7.26 presents another example of the classification algorithm given in Figure 7.24. The example schema graph in Figure 7.26.a is identical to the one in Figure 7.25.a with the only exception that all four classes  $C_i$  now have distinct object memberships, that is,  $\text{content}(C1) \supset \text{content}(C2) \supset \text{content}(C3) \supset \text{content}(C4)$ .  $VC$  is again derived by the view derivation “ $VC = \text{hide } [b] \text{ from } C_4$ ”. First, the type hierarchy preparation task done by the *Generate-Intermediate-Classes*( $G, VC$ ) procedure proceeds as already explained in Example 47. Then, when applying the *Class-Placement*( $G, VC$ ) algorithm to add  $VC$  to the prepared schema graph  $G$ , the  $\text{DIRECT-PARENTS}_{VC}$  set is again set equal to  $\{g_3\}$ . The algorithm checks whether  $VC$  is contained in the  $\text{DIRECT-PARENTS}_{VC}$  set, which this time is not true for the following reason. While the type of  $VC$  is equal to the type of  $g_3$  with  $\text{type}(g_3) = [a, c, d] = \text{type}(VC)$ , the two contents are distinct with  $\text{content}(g_3) = \text{content}(C3) \supset \text{content}(C4) = \text{content}(VC)$ , in short,  $\text{content}(g_3) \neq \text{content}(VC)$ . The algorithm thus initiates the search for the  $\text{DIRECT-CHILDREN}_{VC}$  set starting from  $g_3$  on downwards the class hierarchy.  $C3$  is not subsumed by  $VC$ , since  $\text{content}(C3) \supset \text{content}(C4) = \text{content}(VC)$ . However, since  $\text{content}(C4) = \text{content}(VC)$ , the class  $C4$  is subsumed by  $VC$  and I have  $\text{DIRECT-CHILDREN}_{VC} := \{C4\}$ . Lastly, I need to adjust the edges of the schema graph in order to insert  $VC$  directly below the  $\text{DIRECT-PARENTS}_{VC}$  set and directly above the  $\text{DIRECT-CHILDREN}_{VC}$  set. For this reason, the edges  $\langle VC, g_3 \rangle$  and  $\langle C4, VC \rangle$  are added between  $VC$  and its direct parent and its child, respectively (Figures 7.26.c and 7.26.d). The resulting schema graph is shown in Figure 7.26.d.

## 7.6 A Solution to the Subset/Subtype Incompatibility Problem

In Section 7.2, I have identified a class integration problem that arises due to the conflict between subset and subtype relationship requirements of a class. I have also shown an example of how this problem may be solved (Example 35). In this example, the *is-a* incompatibility problem was solved by creating additional intermediate classes (besides those required for a closed type hierarchy) that restructure the schema graph such as to allow for the correct insertion of the virtual class. In the following, I will present a general solution to the *is-a* incompatibility problem. More



(a) Given a schema G and virtual class VC defined by "VC= hide [b] from C4".



(b) Prepare the schema graph G for VC by inserting the intermediate classes  $g_i$  into G.

**Class-Integration Algorithm:**

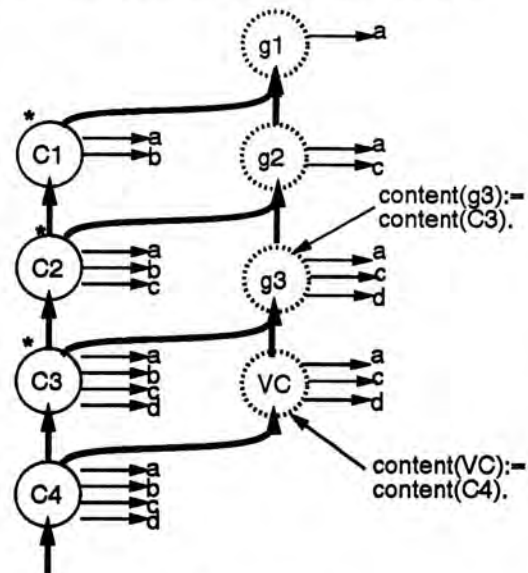
**Search:**

DIRECT-PARENTS<sub>VC</sub> = { g3 }.  
 VC is not in DIRECT-PARENTS<sub>VC</sub>  
 DIRECT-CHILDREN<sub>VC</sub> = { C4 }.

**Edge manipulation:**

Insert <VC,g3>  
 Insert <C4,VC>

(c) Determine the correct position for VC in G.



(d) Insert VC into G.

Figure 7.26: An Example of Complete Classification (Distinct Set Contents).

precisely, I will show that the algorithm that solves the type inheritance mismatch problem (Section 7.3) also successfully addresses the *is-a* incompatibility problem for the object algebra proposed in this dissertation.

It is straightforward to see that virtual classes derived by most of the object algebra operators suggested in this dissertation are not subject to the subset/subtype incompatibility problem. This can intuitively be explained by the fact that the object algebra operators generate virtual classes that are compatible with the existing *is-a* hierarchy, meaning, either the derived class is both a subset and a subtype of the source class, or it is both a superset and a supertype of the source class. I leave it as an exercise to the reader to verify this observation.

Note that there is one exception to this, namely, the **hide** operator may indeed generate an *is-a* incompatible class as shown in Figure 7.28. The **hide** operator creates a virtual class VC with the same content as the source class C and a more generalized type than C. Due to its type, VC must be placed higher in the schema graph above C. In order to develop a solution for the *is-a* incompatibility problem in the context of the **hide** operator I consider the following three cases: (1) a class with a type equal to  $\text{type}(VC)$  already exists, (2) no class with a type equal to  $\text{type}(VC)$  exists but VC can consistently be integrated, and (3) no class with a type equal to  $\text{type}(VC)$  exists and none can be created.

In the first case, if a class  $C_k$  with  $\text{type}(C_k)=\text{type}(VC)$  already exists in the graph G, then VC can always be integrated - without conflict - by simply making it a subclass of  $C_k$ . This is so because the **hide** operator guarantees that the content of VC is smaller or equal to the content of any of the classes above the source class C of VC. Hence,  $C_k \supseteq VC$ . This then implies VC *is-a*  $C_k$ , i.e., VC can always consistently be integrated into G by making it a subclass of  $C_k$ . I explain this situation with the example below.

**Example 49.** *In Figure 7.27, the virtual class VC is derived from the class  $C_3$  using the **hide** operator. By the definition of the **hide** operator,  $\text{content}(VC)=\text{content}(C_3)$ , and thus the set content of VC is always smaller or equal to the set content of any of the classes above  $C_3$ . Since  $VC \succ C_3$ , VC needs to be integrated into G above  $C_3$ . There is one class in G above  $C_3$  that has the same type as VC, namely,  $C_1$ . Therefore, VC can be integrated into G by making it a subclass of  $C_1$ .*

In the second case, a class  $C_k$  with  $\text{type}(C_k)=\text{type}(VC)$  does not exist in the graph G but VC can be consistently integrated. By assumption, this case does not represent a problem. An example of this situation is discussed below.

**Example 50.** *In Figures 7.28.a and 7.28.b, I assume that the set contents of all classes are identical. Then the set relationships can be ignored; this reduces the *is-a**

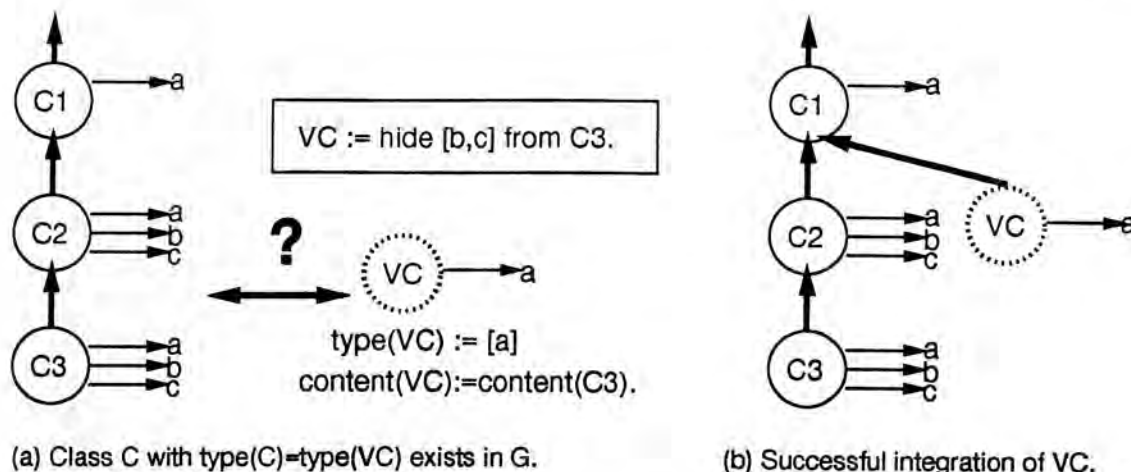


Figure 7.27: The *is-a* Incompatibility Problem with VC's Type in the Schema.

*incompatibility problem to the type inheritance mismatch problem, which fortunately I already know how to solve. In Figure 7.28.b, VC is integrated into G by placing it directly below C1 and above C2.*

Lastly, I take a look at the third case in which no consistent integration of VC can take place. For instance, if I drop the simplifying assumption of identical sets in the previous example, then the fact that a class  $C_k$  with  $\text{type}(C_k)=\text{type}(VC)$  does not exist in G results in conflicts caused by a mismatch between the subset and the subtype hierarchy. In this case, a subset/subtype conflict may occur since VC's content may be smaller than the content of any of the other classes above its source class. VC should be **below** all of them in terms of the set relationships and **above** some of them due to the subtype relationships.

**Example 51.** *An example of this situation is given in Figure 7.28.c, which is equal to Figure 7.28.b except for the fact that the set contents of all classes are now distinct. In this example, VC is required to be **above** C2 due to their subtype relationship and **below** C2 due to their subset relationship. Clearly, there is no consistent position for VC in G. The problem in Figure 7.28.c can be solved by introducing an intermediate class C2' with  $\text{type}(C2')=\text{type}(VC)$  and  $\text{content}(C2')=\text{content}(C2)$ . VC can then be integrated by placing it directly below C2' as depicted in Figure 7.28.e.*

It is interesting to note that the solution for the *is-a* incompatibility problem proposed in the above example is exactly what the type lattice preparation algorithm presented in Section 7.3 would do. This is also demonstrated in Figures 7.28.d and 7.28.e by displaying detailed steps of the type lattice preparation algorithm for the example. I now argue that this is not coincidence but rather that it is always the case. This can be explained as follows. Our solution to the type inheritance

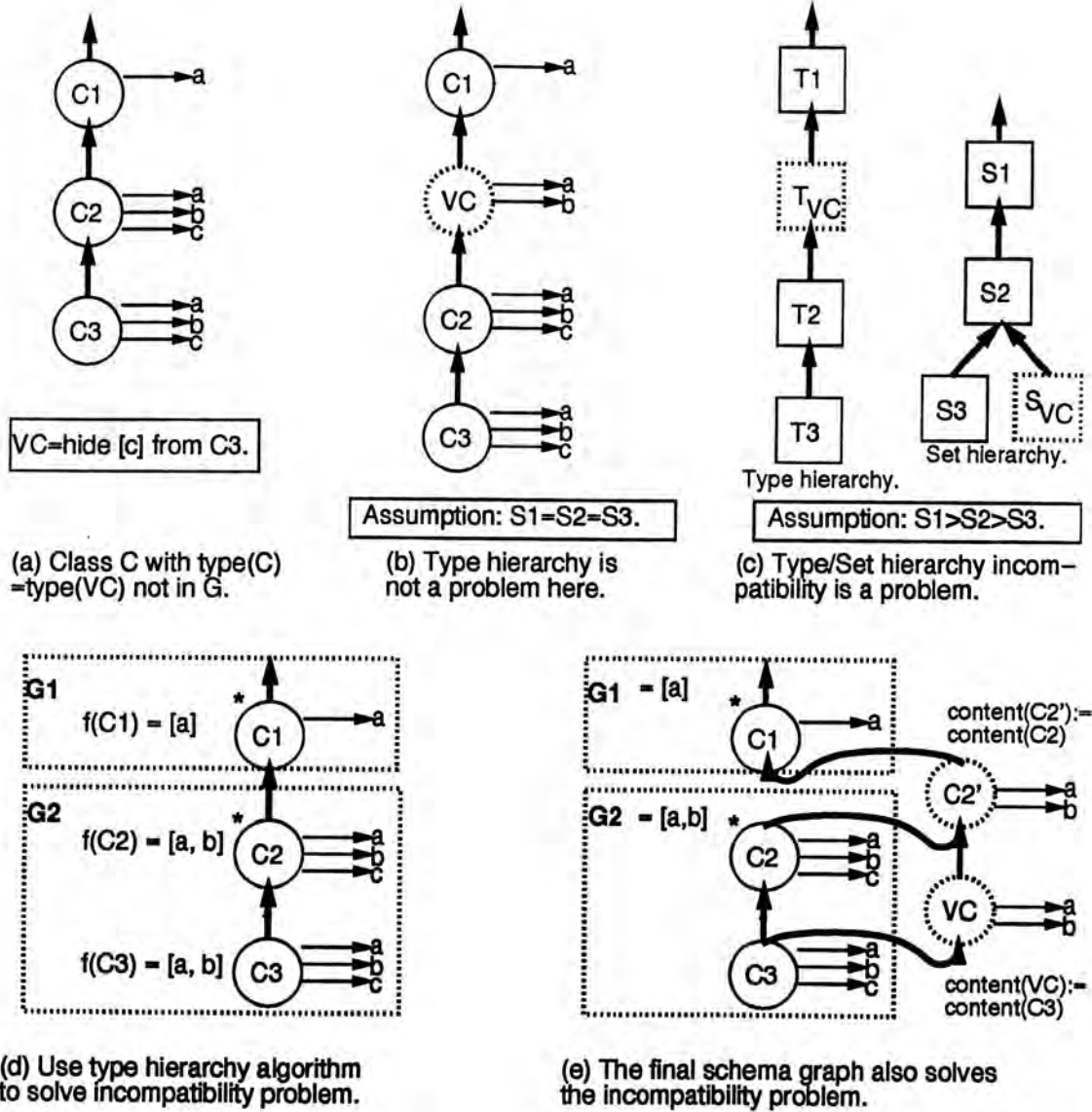


Figure 7.28: An Example of Solving the *is-a* Incompatibility Problem.



problem introduces intermediate classes, one for each equivalence group of  $G$  with respect to  $VC$  (Section 7.3). Since  $\text{type}(\langle \text{hide-source-class} \rangle \sqcap VC) = \text{type}(VC)$ , this then implies that a class with a type equal to  $VC$  is created. Consequently, application of the type hierarchy preparation will always generate a class with the required type of  $VC$ . In other words, the type hierarchy preparation reduces the problem to the situation shown in Figure 7.27. As discussed earlier, in this case *is-a* incompatibility is no longer a problem. In short, I thus have shown that the type hierarchy preparation algorithm solves the *is-a* incompatibility problem for the general case (see also Figures 7.28.d and 7.28.e). For this reason, I need not develop an independent solution to address the *is-a* incompatibility problem. Lastly, I would like to note that if one were to use a different class derivation language than the object algebra proposed in this dissertation, then one might have to develop a general algorithm to solve the *is-a* incompatibility problem (See for example Figure 7.3). Further investigation of this problem is however beyond the scope of this dissertation.

## 7.7 Customizing the Classification Algorithm for the Object Algebra

In this section, we customize the general class integration algorithm presented in this dissertation for each of the six object algebra operators proposed in Section 5.2. I demonstrate that in the majority of cases the type hierarchy preparation procedure, which has quadratic complexity, need not be run. For the following, I refer to the simple class placement procedure defined in Section 7.4 by algorithm A and to the more complex algorithm defined in Section 7.5 that first generates the appropriate intermediate types and then places the virtual class in the modified schema graph by algorithm B. Recall that  $\text{complexity}(A)$  is linear and  $\text{complexity}(B)$  is quadratic, assuming a *subsumes()* function with constant complexity.

The table in Figure 7.29 presents a summary of the results of customizing class integration for each derivation operator, while a more detailed discussion is given next. The first column lists the object algebra operators, the second whether or not type generation may be required, the third indicates the final location of the virtual class in the schema graph, and the fourth and fifth columns indicate the algorithm and complexity needed to accomplish the class integration of the virtual class generated by the respective operator.

derivation operators	intermediate types	final location	algorithm	complexity
<b>select</b>	no	below SC	A	$O( E )$
<b>refine</b>	no	directly below SC, no children	direct	$O(1)$
<b>hide</b>	yes	above SC	B	$O( E ^2)$
<b>union</b>	no	above SC	A	$O( E )$
<b>intersect</b>	yes	below SC	B	$O( E ^2)$
<b>diff</b>	no	below SC	A	$O( E )$

**Legend**

algorithm A = class placement (given correct type hierarchy)  
algorithm B = type hierarchy preparation and class placement

Figure 7.29: Class Integration Customized for each Object Algebra Operator.

The **refine** operator defined in Section 5.2 generates a virtual class VC with a more specialized type description than a given source class SC by adding new property functions to the type description of SC. VC has the same set content as SC. Having a more specialized type description and the same set content as SC, VC becomes a subclass of SC and thus must be placed below SC in the schema graph. In spite of the generation of a new type (and a new property function), I do not need to add intermediate classes to the schema during class integration. This is so, since all property functions added to the **refined** class are distinct from existing functions. Hence the equivalence partition of  $G$  with respect to SC is equal to the equivalence partition of  $G$  with respect to VC (see Definition 25 for a definition of the equivalence partition). By Theorem 4, this implies that all necessary types do already exist in the schema graph. I thus have shown that algorithm B is not needed for the class integration of a **refined** class.

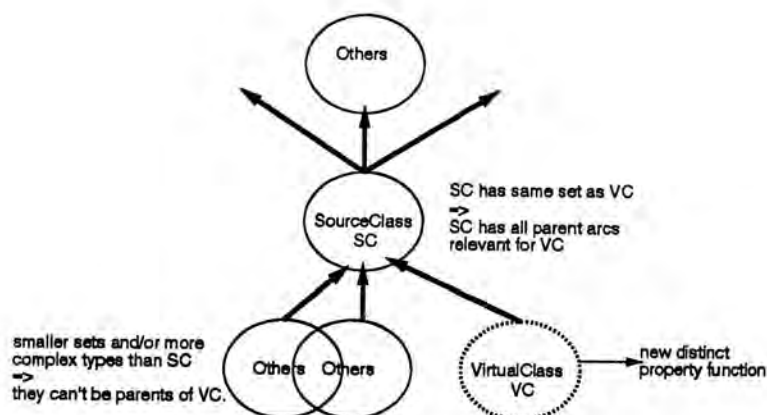


Figure 7.30: Linear Class Integration Time for the **refine** Operator.

Next, I show that the integration of a virtual class VC generated by the **refine** operator can be reduced to a simple  $O(1)$  algorithm requiring no search – rather than having to apply the linear class-placement procedure in algorithm A. As shown in Section 5.2, the **refined** virtual class is always a (direct or indirect) subclass of its source class. Based on the example graph in Figure 7.30, I can explain why VC cannot be placed any lower in the class hierarchy. All classes below the source class SC have a restricted set membership and/or a refined type description of SC. Since the content of VC is equal to the content of SC, none of these children of SC can become a parent of VC. Based on Figure 7.30, I can also explain why the virtual class VC will not have any direct parents besides its source class SC. Since  $\text{content}(\text{SC}) = \text{content}(\text{VC})$  and the types of SC and VC are equivalent with the exception of the new property function of VC, any parent of VC will already be a parent of SC. VC would consequently inherit these parent relationships through SC.

Lastly, I argue that the virtual class VC cannot have any subclasses of its own. VC has a type description distinct from the types of all existing classes due to the newly defined property function and therefore it would be incorrect for any of the existing classes to inherit this new property. I can thus conclude that the **refined** virtual class VC always has to be placed as direct subclass of its source class SC with no children of its own, i.e.,  $\text{direct-parents}(\langle \text{refined-virtual-class} \rangle) := \{ \langle \text{source-class} \rangle \}$  and  $\text{direct-children}(\langle \text{refined-virtual-class} \rangle) := \{ \}$ .

The **hide** operator defined in Section 5.2 modifies the type description of a class SC by hiding some of its property functions. It thus generates a new class VC with a more general type description and the same set content as SC. As for instance demonstrated in Example 7.2, I am forced to generate intermediate classes since VC may have a new type description (composed of existing property functions) that does not yet exist in the object schema. Therefore, the integration of a virtual class generated by the **hide** operator may require the application of the full-blown algorithm B.

The **select** operator is a set-manipulating operator that generates a virtual class VC by selecting a subset of object instances from a given class SC. The final location of VC is below SC in the schema graph, but not necessarily as a direct subclass of SC. It is easy to see that I do not need to generate any intermediate classes, since VC has the same type as SC and SC is already properly integrated into the schema graph. Therefore, simple class placement using algorithm A is sufficient for the **select** operator.

The **union** operator defined in Section 5.2 builds a new class VC by combining the members of two source classes SC1 and SC2 into one set. The type description of the virtual class then is set equal to the lowest common supertype of the two sources classes as defined in Definition 4. Clearly, this makes VC a supertype and a superset of both its source classes, and thus VC is placed above them. Even though manipulating the type description of the two source classes, the integration of a class derived using the **union** operator does not require the introduction of intermediate classes. The reason for this can be explained as follows. The type of VC is equal to  $\text{type}(\text{VC}) := \text{SC1} \sqcap \text{SC2}$ , which corresponds to the lowest common supertype of SC1 and SC2. By Theorem 23, this lowest common supertype has to exist for all pairs of classes in a closed schema graph. Since SC1 and SC2 were members of the schema graph before the inserting of VC, a class with a type equal to the lowest common supertype of SC1 and SC2 must already have existed in the schema graph. Consequently, the simple class placement algorithm A with linear complexity is sufficient.

The **intersect** operator defined in Section 5.2 builds a new class VC by constructing a set of object instances that are members of two classes SC1 and SC2. The type description of the virtual class is equal to the greatest common subtype of the two source classes as defined in Definition 3. VC is placed in the subgraph below both SC1 and SC2. The integration of a class derived using the **intersect** operator may require the introduction of intermediate classes since its new type did not necessarily exist before in the schema graph. I therefore need to use algorithm B with quadratic complexity.

The **difference** operator generates a virtual class VC consisting of a set of object instances that are members of the first but not of the second source class. This is effectively equivalent to the **select** operator, since "VC := **diff**(SC1,SC2)" is equivalent to "VC := **select** (e:SC1) **where** (e not in SC2)". Therefore, the virtual class has the same type as the first source class. And, since this type has already been properly integrated into the schema graph, no generation of intermediate classes is needed. In short, algorithm A is sufficient for the **difference** operator.

# Chapter 8

## The View Definition Language

### 8.1 Introduction to the View Definition Language

In this section, I discuss the second task of *MultiView*, namely, the definition of a view schema on top of the augmented global schema. I divide this view specification task into two subtasks:

1. the selection of view classes, and
2. the generation of view relationships between the view classes.

For the first subtask, I define a view definition language that can be utilized by the view definer for the specification of view schemata as discussed in this section. Note that the view definition language (Figure 8.1) is concerned only with the manipulation of view classes and not with view *is-a* relationships. Rather than specifying *is-a* arcs manually, *MultiView* solves the second subtask by automatically generating the set of view *is-a* arcs that has to be inserted in order to make the view schema *valid* (Section 9). This automatic generation of view *is-a* arcs is preferable over their manual entry since it simplifies the task of the view designer and guarantees the consistency of the resulting view schema. The view definition language is discussed in this section, while the algorithms for automatic view generation are the topic of Section 9.

The view definition language consists of two types of operators: the first group discussed in Section 8.2 either initiates or terminates a transaction on a view schema while the second group discussed in Section 8.3 modifies a given view schema.

```

<view-definition> ::= <view-creation>; | <view-modification>;
<view-creation> ::=
  DEFINE-VIEW <view-name>
    { <view-ops> } <view-manipulation>
  END-VIEW
<view-modification> ::=
  MODIFY-VIEW <view-name>
    { <view-ops> } <view-manipulation>
  END-VIEW
<view-ops> ::= <class-name> := <class-derivation-operator>; |
  <view-schema-manipulation>
<view-schema-manipulation> ::=
  ADD-CLASS(<class-name>);
  | ADD-CLASS-DAG(<class-name>);
  | ADD-VIEW-SCHEMA(<view-name>);
  | REMOVE-CLASS(<class-name>);
  | REMOVE-CLASS-DAG(<class-name>);
  | REMOVE-VIEW-SCHEMA(<view-name>);
  | RENAME-CLASS(<old-class-name>) by (<new-class-name>);
<view-manipulation> ::= SAVE-VIEW; | DELETE-VIEW;

```

Figure 8.1: The BNF Syntax of the View Definition Language.

## 8.2 Commands for View Schema Creation and Deletion

The following operators work on a complete view schema by either initiating or terminating a transaction on a particular view schema: **DEFINE-VIEW**, **MODIFY-VIEW**, **SAVE-VIEW**, **DELETE-VIEW** and **END-VIEW**. The **DEFINE-VIEW** command initializes a new empty view schema and assigns a unique view identifier to it. This operation is executed prior to any other operators. Hence, the creation of virtual classes or the modification of a view schema VS can be done only in the context of a view definition transaction of the particular view schema. Within this transaction, which is marked by a **DEFINE-VIEW** command at the beginning and the **END-VIEW** command at the end, changes can be made to this one view schema only.

The **MODIFY-VIEW** command is similar to **DEFINE-VIEW**, except it is applied to an already defined view schema rather than creating a new one. It thus prepares an existing view schema VS for modification by the operators described in the next section. All operators specified within this view definition transaction, i.e., after this **MODIFY-VIEW** command and before the terminating **END-VIEW** command, will modify

only VS and no other view schema. Since the existing view schema VS already has a unique identifier, no new view identifier is allocated.

Once the view definers want to conclude the view definition phase, they issue the `SAVE-VIEW` command. This command establishes a view table for the view schema which lists all classes that are part of this view [Rund92a]. In addition, the system determines the set of view *is-a* arcs that have to be inserted into this view schema and of course also into the view table (Section 9).

Lastly, a view definer can remove a view schema with the `DELETE-VIEW` command. This command not only deletes the view table and view *is-a* arcs, but it also removes all virtual classes from the global schema that were created for the definition of that view, whenever possible. Virtual classes can no longer be removed when they are already (directly or indirectly) utilized by other views.

### 8.3 View Schema Manipulation Commands

The view schema manipulation operators (`<view-schema-manipulation>` in Figure 8.1) assume that a view schema VS has already been created and opened for manipulation by either a `DEFINE-VIEW` or a `MODIFY-VIEW` command. They then modify this designated view VS by either adding or deleting view classes. The `"ADD-CLASS(<class-name>)"` command adds a class with the name `<class-name>` in GS to the view schema VS. The `"ADD-CLASS-DAG(<class-name>)"` command adds all classes to the view schema VS that are classes in the subschema of GS rooted at the class with the name `<class-name>`. Finally, the `"ADD-VIEW-SCHEMA <view-name>"` command adds all classes of the view schema with the view identifier `<view-name>` to the current view schema VS.

The next three commands do the same as the ones above but rather than adding they are deleting the respective classes. The `"REMOVE-CLASS <class-name>"` command removes the class with the name `<class-name>` from VS. Recall that if a virtual class is created during a transaction of modifying the view schema VS, then it is automatically inserted into the view schema VS. If during the view creation process this virtual class is removed from the view schema VS, then the class is also deleted from the global schema. The `"REMOVE-CLASS-DAG <class-name>"` command removes the subschema of GS rooted at the class with the name `<class-name>` from VS. The `"REMOVE-VIEW-SCHEMA <view-name>"` command removes the existing view schema with the identifier `<view-name>` from VS.



Lastly, the "RENAME-CLASS <old-class-name> by <new-class-name>" command renames an existing view class of the view schema VS by replacing its name <old-class-name> by the new name <new-class-name>. I assume scoping here; hence this is a local change that is only visible from within the current view schema.

## 8.4 Examples of View Schema Definition

Below, I demonstrate the above mentioned commands of the view definition language based on the example views shown in Figure 8.2.

**Example 52.** *In this example, I discuss the definition of the view VS1 in Figure 8.2.d on top of the global schema GS depicted in Figure 8.2.a. Below, I give one possible view creation script for the specification of VS1.*

### View Creation Script For VS1:

```
DEFINE-VIEW VS1
class Minor = select (P:Person) where (P.Age<21);
ADD-CLASS (Person);
ADD-CLASS (TeenageGirl);
SAVE-VIEW;
END-VIEW
```

I start the view definition transaction by issuing the DEFINE-VIEW VS1 command, which creates an empty view schema with the identifier VS1. I then define and insert the virtual class **Minor** into the global schema GS. As discussed above, **Minor** is also automatically added to the view schema VS1. Then the commands ADD-CLASS(**Person**) and ADD-CLASS(**TeenageGirl**) are issued to insert the classes **Person** and **TeenageGirl** into VS1. VS1 now has the classes(VS1) = { **Minor**, **Person**, **TeenageGirl** }. Lastly, VS1 is saved with the command SAVE-VIEW. *MultiView* then automatically creates the view *is-a* arcs for VS1 as depicted in Figure 8.2.d.

**Example 53.** *The second view schema VS2 in Figure 8.2.e is defined on top of the global schema GS depicted in Figure 8.2.b. A possible view creation script for VS2 is given below.*

### View Creation Script For VS2:

```
DEFINE-VIEW VS2
```

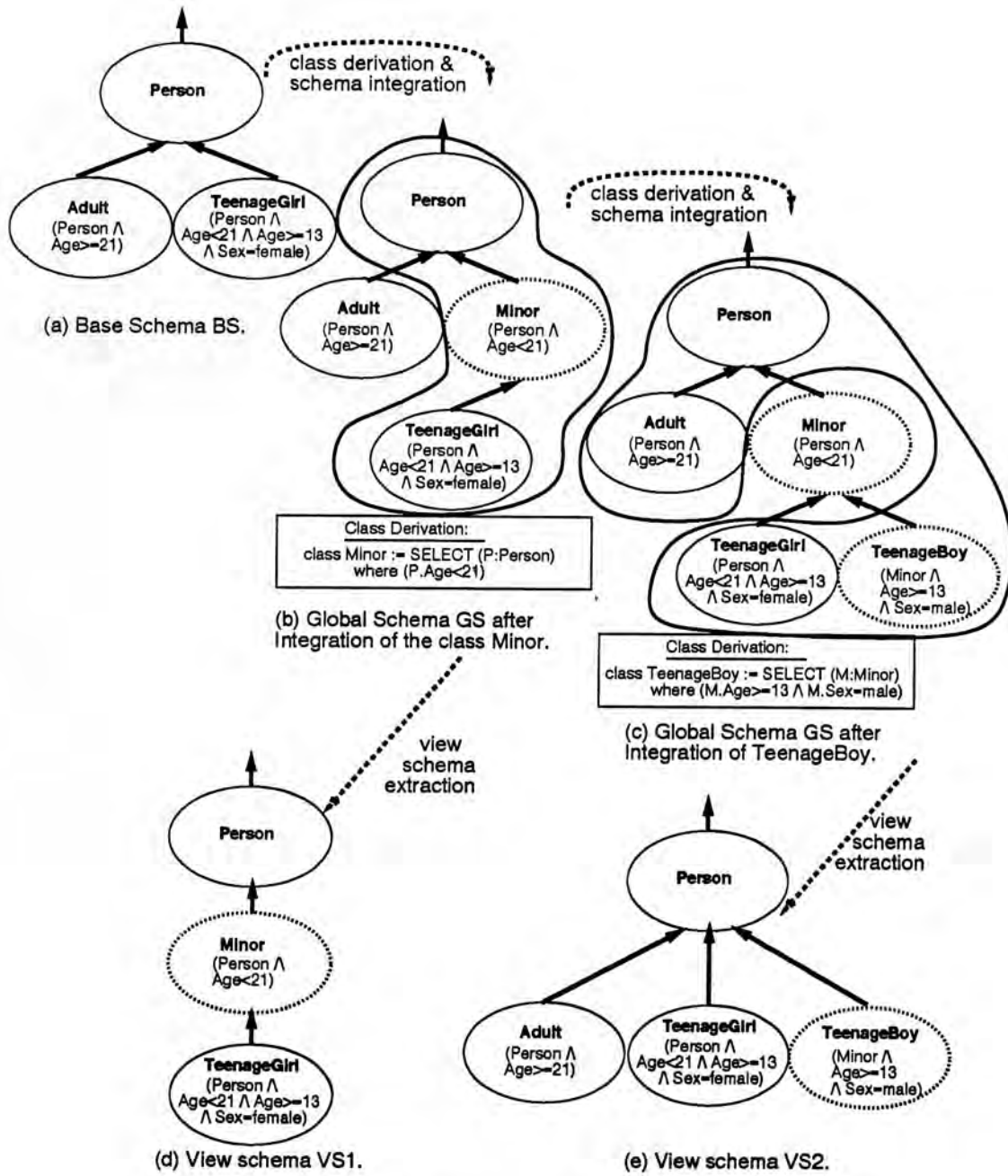


Figure 8.2: An Example of View Specification.

```
class TeenageBoy = select (M:Minor)
  where (M.Age>=13) and (M.Sex=male);
ADD-VIEW-SCHEMA (BS);
SAVE-VIEW;
END-VIEW
```

First, the DEFINE-VIEW VS2 command creates a new view schema with the view identifier VS2. I then define the virtual class **TeenageBoy** (Figure 8.2.c) and integrate it into GS (Figure 8.2.c). The three classes of the base schema are added to VS2 using the command ADD-VIEW-SCHEMA(BS). The selected view classes are shown in Figure 8.2.e. When VS2 is saved, the *is-a* arcs shown in Figure 8.2.e are derived automatically by *MultiView* [Rund92a].

Important to note here is that the restructuring of the underlying global schema GS due to the creation of VS2 did not have any effect on the existing view VS1. I have shown in [Rund92a] that this *view independence* property of *MultiView* is in general true.

# Chapter 9

## Algorithms for Automatic View Generation

### 9.1 Motivation and Problem Definition

Unlike for relational views, generalization relationships in object-oriented views must be validated so that they are consistent with the global schema. As explained in Section 4, I solve the problem of consistent view specification by automating the specification of the view schema class hierarchy rather than requiring manual entry of the *view is-a* arcs by the view definer. Automatic view generation offers numerous additional advantages, some of which are detailed below:

1. simplify the view specification process for the users by automating tedious tasks,
2. guarantee the consistency of the view schema (and thus the correctness of query processing on the view),
3. prevent the introduction of redundant subclass relationships into the view,
4. reduce execution times for query processing on the view,
5. assure the completeness of the view semantics by guaranteeing the presence of all required subclass relationships, and
6. possibly reduce the computer memory required to store the view specification.

One important observation is that I need not be concerned with determining the subclass relationships between two classes by comparing their type descriptions and set membership predicates, a potentially np-complete problem [Rund92b]. Instead, I assume that the subclass relationships have been calculated a priori for each pair of classes and that the result of this evaluation has been entered into the global schema graph. Put differently, a valid view schema can be derived from a given valid global schema (Definition 19) by simply exploiting the (syntactic) graph structure

of the global schema rather than by requiring the (semantic) comparison of the class specifications for each pair of classes. A solution for generating a valid global schema using a classification algorithm has been given in Chapter 7.

The problem I address in this section can thus be formally stated as a graph-theoretic problem. Let  $GS = (V, E)$  be a global schema. Assume that a subset of classes  $VV \subseteq V$  of  $GS$  has been marked by the view identifier  $\langle VS \rangle$ , i.e., these marked classes have been selected to belong to the view schema  $VS$  via the operations given in Section 8.2. I wish to develop an algorithm that automatically determines a set  $VE$  of *is-a* edges among classes in  $VV$ , such that  $VS = (VV, VE)$  is a valid view schema (Definition 19). This graph-theoretic formulation of the problem then allows us to apply standard graph algorithms theory to solve the view generation problem.

In the remainder of this chapter, I present two algorithms that automate the view schema creation process. Both algorithms automatically generate a *complete*, *minimal* and *consistent* view schema from the user selected view classes. The first algorithm presented in Section 9.2 has linear complexity but it assumes that the global schema has a tree structure (a schema with single inheritance). The second algorithm presented in Section 9.3 overcomes the restricting assumption of a tree class hierarchy (a schema with multiple inheritance), but it does so at the cost of a larger complexity. I also provide proofs of correctness, complexity analysis, and examples of these algorithms.

## 9.2 View Generation for Global Schemata with Single Inheritance

First I describe an algorithm for the automatic generation of a view schema, which assumes that the global schema  $GS$  has a tree structure. The proposed algorithm is based on a simple graph traversal of  $GS$ , and thus has linear complexity (Figure 9.1). The algorithm traverses  $GS$  in a breadth-first manner from the root down to the leaves. For each node  $C_i$  in  $GS$  that is marked by  $\langle VS \rangle$  it searches all branches in the subtree rooted at  $C_i$ . An *is-a* edge is inserted into  $VS$  between the parent  $C_i$  of a subtree and all subclasses of  $C_i$  that (1) are also marked by  $\langle VS \rangle$  and (2) are the closest to  $C_i$  in the tree. More formally, if  $(C_1, C_2 \in VS)$  and  $(C_1 \text{ is-a}^* C_2)$  in  $GS$  and  $(\nexists C_i \text{ in } VS)((C_1 \text{ is-a}^* C_i) \text{ and } (C_i \text{ is-a}^* C_2))$ , then the Edge-Creation algorithm inserts the edge  $(C_1 \text{ is-a } C_2)$  into  $VS$ . By Definition 19, this newly inserted edge is a *required* edge. This edge is guaranteed not to be *redundant*, since in a tree-structured graph there is at most one path between any

two nodes. For a given parent node  $C_i$ , this search process stops when the algorithm either finds a marked child or a leaf node of  $GS$ . The algorithm terminates when all marked classes  $C_i$  have been used as parent nodes once.

**Example 54.** *Figure 9.2 presents an example of applying the Edge-Creation algorithm to generate a view schema. Figures 9.2.a1 and 9.2.a2 show  $GS$  and the selected nodes of  $VS$ , respectively. First the root class  $C_1$  of  $GS$  is put onto the ParentQueue. The Parent  $C_1$  is popped from the queue. Then all children of Parent are inserted into the ChildrenQueue, and the Child  $C_2$  is popped from the queue. This results in the state listed below (and in Figure 9.2.a1):*

ParentQueue= $\langle \rangle$ , Parent= $C_1$ , ChildrenQueue= $\langle C_3 \rangle$ , Child= $C_2$ .

*The if-statement then finds that the Child  $C_2$  is in not  $VS$  and inserts all children of  $C_2$  into the ChildrenQueue.  $VS$  stays unchanged as shown in Figure 9.2.a2. A new iteration of the inner while-loop results in the following search state (corresponding to Figure 9.2.b1):*

ParentQueue= $\langle \rangle$ , Parent= $C_1$ , ChildrenQueue= $\langle \rangle$ , Child= $C_3$ .

*Again the Child  $C_3$  is not in  $VS$ , therefore I insert the children  $\{ C_4, C_7 \}$  of  $C_3$  into the ChildrenQueue.  $VS$  still stays unchanged as shown in Figure 9.2.b2. The state generated by the next iteration of the inner while-loop is shown below (and in Figure 9.2.c1):*

ParentQueue= $\langle \rangle$ , Parent= $C_1$ , ChildrenQueue= $\langle C_7 \rangle$ , Child= $C_4$ .

*This time the Child  $C_4$  is in  $VS$ . Hence I insert an edge from the Child to the Parent into  $VS$ , namely, the edge ( $C_4$  is-a  $C_1$ ). The resulting  $VS$  is depicted in Figure 9.2.c2. I also add the Child  $C_4$  to the ParentQueue. The next iteration results in the state given below (and in Figure 9.2.d1):*

ParentQueue= $\langle C_4 \rangle$ , Parent= $C_1$ , ChildrenQueue= $\langle \rangle$ , Child= $C_7$ .

*The Child  $C_7$  is again in  $VS$ , hence I insert the edge ( $C_7$  is-a  $C_1$ ) into  $VS$  (Figure 9.2.d2). I also add the Child  $C_7$  to the ParentQueue. Since the ChildrenQueue is empty, I start with the outer while-loop by popping a new Parent  $C_4$  off the ParentQueue. The children  $\{ C_5, C_6 \}$  of  $C_4$  are added to the ChildrenQueue, and the new state is given below (and in Figure 9.2.e1):*

ParentQueue= $\langle C_7 \rangle$ , Parent= $C_4$ , ChildrenQueue= $\langle C_6 \rangle$ , Child= $C_5$ .

*Since the current Child  $C_5$  is in  $VS$ , I insert an edge between  $C_5$  and the new Parent  $C_4$  into  $VS$  (as shown in Figure 9.2.e2). I also add the Child  $C_5$  to the*

**Assumption:**

Global Schema  $GS$  is tree-structured (No multiple inheritance).

**Input:**

Global Schema  $GS = (V, E)$  and View Schema  $VS = (VV, VE)$

with  $VV \subseteq V$  marked by the view identifier  $\langle VS \rangle$  and  $VE = \emptyset$ .

**Output:**

Determine a set of *is-a* edges  $VE$  on  $VV$ ,

such that  $VS = (VV, VE)$  is a *valid* view schema on  $GS$ .

**Data Structures:**

ParentQueue is a queue to hold nodes of  $GS$ .

ChildrenQueue is a queue to hold nodes of  $GS$ .

Parent and Child are variables that hold one class each.

**Algorithm A1: Creation of *Is-A* Arcs for A View Schema.**

**algorithm** Edge-Creation( $GS, VS$ ) **is**

  Push the root of  $GS$  onto ParentQueue.

**while** (Parent = pop(ParentQueue)) **do**

    Push all children of Parent in  $GS$  onto ChildrenQueue.

**while** (Child = pop(ChildrenQueue)) **do**

**if** Child is in  $VS$  **then**

        insert isa(Child, Parent) into edges( $VS$ );

        Push Child onto ParentQueue;

**else**

        Push all children of Child in  $GS$  onto ChildrenQueue;

**endif**

**endwhile**

**endwhile**

**end algorithm;**

Figure 9.1: The Edge-Creation Algorithm.

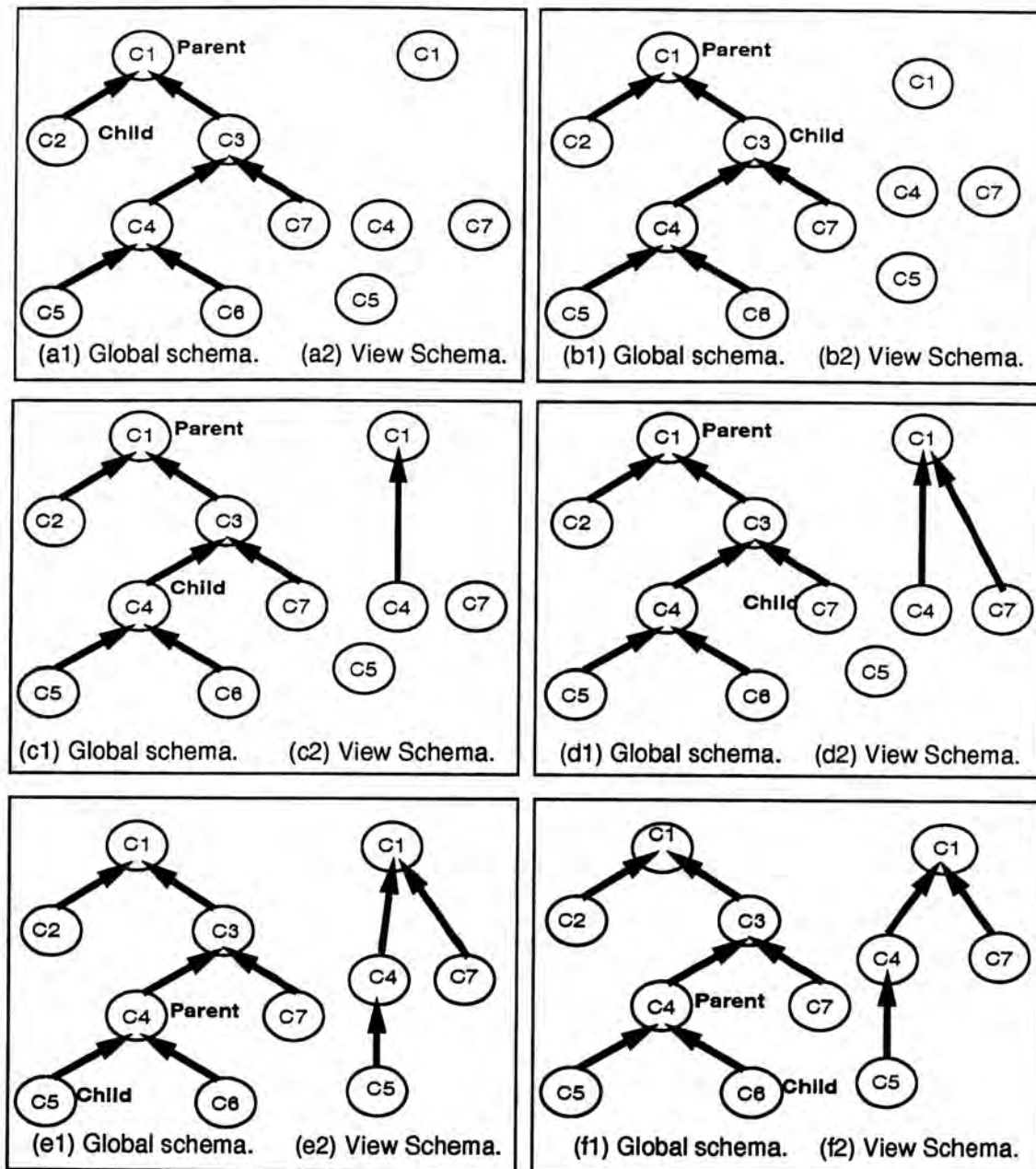


Figure 9.2: Example Snapshots of the Edge-Creation Algorithm.



*ParentQueue*. The new state after the pop of the *ChildrenQueue* is given below (and in Figure 9.2.f1):

$\text{ParentQueue} = \langle C_7, C_5 \rangle$ ,  $\text{Parent} = C_4$ ,  $\text{ChildrenQueue} = \langle \rangle$ ,  $\text{Child} = C_6$ .

$C_6$  is not in *VS*. However,  $C_6$  is a leaf of *GS* and therefore no new children are added to the *ChildrenQueue*. The *ChildrenQueue* is again empty. I start with the outer while-loop and pop a new Parent  $C_7$  from the *ParentQueue*. No more edges are created since both  $C_7$  and  $C_5$  are leaf nodes. The view schema shown in Figure 9.2.f2 is the final result.

**Theorem 15. (Correctness)** The Edge-Creation algorithm generates a valid view schema  $VS = (VV, VE)$  assuming the global schema  $GS = (V, E)$  does not have multiple inheritance.

**Proof (By Contradiction):** For the resulting view schema *VS* to be valid, I need to show that the Edge-Creation algorithm creates *all required* and *no redundant* and *no incompatible is-a* edges for *VS*.

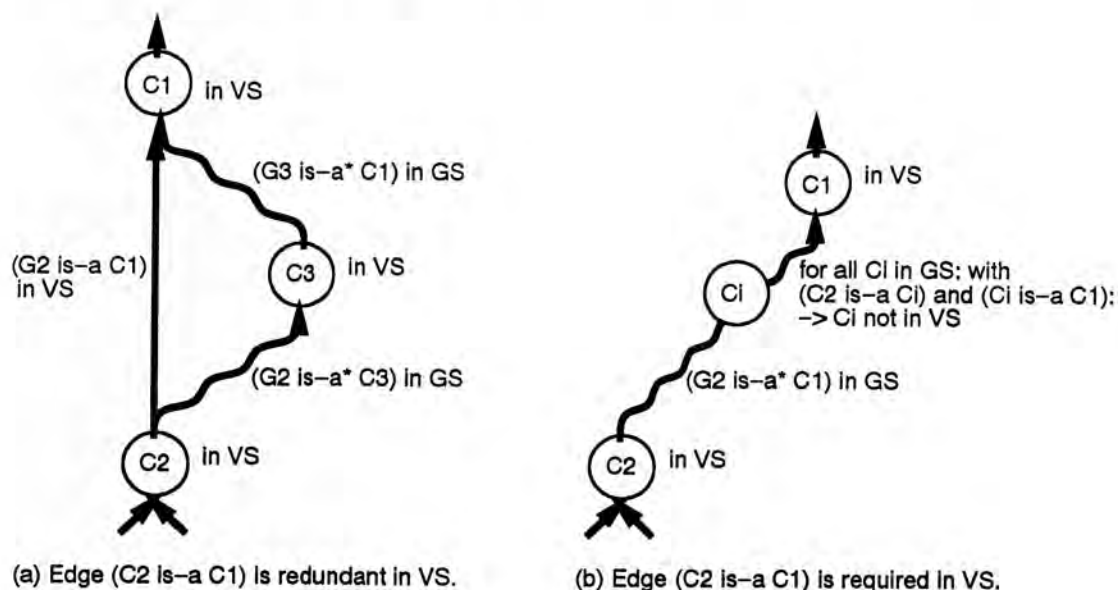


Figure 9.3: Redundant and Required Edges.

**Part I:** The Edge-Creation algorithm creates *no redundant is-a* edges.

Assume that the Edge-Creation algorithm inserted an edge ( $C_2$  is-a  $C_1$ ) into *VS* that is *redundant* in *VS*. (See Figure 9.3.a). By Definition 17, this means that:

(1) the classes  $C_1$  and  $C_2$  are in  $VS$ , and (2) there exists another class  $C_3$  in  $VS$  with  $C_3 \neq C_1 \neq C_2$ , such that  $(C_1 \text{ is-a } * C_3)$  in  $GS$  and  $(C_3 \text{ is-a } * C_2)$  in  $GS$ . For the Edge-Creation algorithm to insert the edge  $(C_2 \text{ is-a } C_1)$  into  $VS$ , the state must have been  $\text{Parent}=C_1$  and  $\text{Child}=C_2$ . Since the schema is a tree, there is exactly one path from  $C_1$  to  $C_2$ . Therefore, the Edge-Creation algorithm must have traversed this path from the Parent  $C_1$  down to Child  $C_2$  without finding any marked node  $C_k$  on the path. If the Edge-Creation algorithm would have found a marked node  $C_k$ , then it would have stopped the search for  $\text{Parent}=C_1$  along this branch. This is a contradiction to the assumption that the inserted edge  $(C_2 \text{ is-a } C_1)$  is redundant, i.e., that there is a marked node  $C_3$  between  $C_2$  and  $C_1$ . Hence, the assumption is incorrect. ■

**Part II:** The Edge-Creation algorithm creates *all required is-a* edges.

Assume that the Edge-Creation algorithm did not insert an edge  $(C_2 \text{ is-a } C_1)$  into  $VS$  even though the edge  $(C_2 \text{ is-a } C_1)$  was *required* in  $VS$  (See Figure 9.3.b). By Definition 16, for the edge  $(C_2 \text{ is-a } C_1)$  to be *required* in  $VS$  means that: (1) the classes  $C_1$  and  $C_2$  are in  $VS$ , and (2)  $C_1$  and  $C_2$  are *is-a* related in  $GS$  by  $(C_1 \text{ is-a } * C_2)$ , and (3)  $\forall C_i$  in  $GS$  with  $(C_2 \text{ is-a } * C_i)$  and  $(C_i \text{ is-a } * C_1)$ , the class  $C_i \notin VS$ .  $C_1 \in VS$  implies that at some point during the algorithm, the Edge-Creation algorithm will make this marked node a parent, i.e.,  $\text{Parent}=C_1$ . Thereafter, the Edge-Creation algorithm would search all branches of the subtree rooted at  $C_1$  for marked children. For the Edge-Creation algorithm not to discover the node  $C_2$ , it must find some other marked node  $C_i$  before finding  $C_2$  (i.e., above  $C_2$  and below  $C_1$ ). But by Definition 19, there is no such node  $C_i$  between  $C_2$  and  $C_1$  that is marked. Hence, the assumption leads to a contradiction. ■

**Part III:** The Edge-Creation algorithm creates *no incompatible is-a* edges.

Assume that the Edge-Creation algorithm inserted an edge  $(C_2 \text{ is-a } C_1)$  into  $VS$  that is *incompatible* in  $VS$ . By Definition 18, this means that  $C_2$  and  $C_1$  are not *is-a* related in  $GS$ , i.e., there is no path in  $GS$  between  $C_2$  and  $C_1$ . For the Edge-Creation algorithm to insert the edge  $(C_2 \text{ is-a } C_1)$  into  $VS$ , the state must have been  $\text{Parent}=C_1$  and  $\text{Child}=C_2$ . Since the algorithm the Edge-Creation algorithm is traversing  $GS$  rather than  $VS$ , this situation can only occur if the node  $C_2$  was found by traversing  $GS$  downwards from  $C_1$  on some path in  $GS$ . Hence, the node  $C_2$  is a *subclass (child)* of  $C_1$  in  $GS$ , denoted by  $(C_2 \text{ is-a } * C_1)$ . This is a contradiction to the initial assumption. ■

**Theorem 16. (Complexity)** *The complexity of the Edge-Creation algorithm is linear in the number of nodes in  $GS$ , i.e.,  $O(|GS|)$ , assuming the class hierarchy of the global schema is a tree.*

**Proof:**  $GS$  being a tree implies that there is exactly one unique path to reach each node from the root. Hence, each node (except the root node) is placed into the ChildrenQueue exactly once. It is inspected in constant time by checking its annotation to determine whether it belongs to the view schema, and then it is removed from the ChildrenQueue. Now, it is either completely discarded, or, it is placed in the ParentQueue in order to be compared to its children. Each of these comparisons with the same node as parent can be charged to the respective children rather than to the parent node. The complexity thus is  $O(|GS|)$ . ■

More intuitively, I can show that the complexity of the Edge-Creation algorithm as follows. The algorithm essentially corresponds to a simple graph traversal of the global schema  $GS$  that concurrently does some book-keeping about the encountered marked view classes. Graph traversal of a graph  $GS$  has the complexity  $O(nodes(GS) + edges(GS))$  [Aho72]. However, since the graph  $GS$  is a tree, I know that  $edges(GS) = nodes(GS) - 1$ . Therefore, the complexity of the algorithm is  $O(nodes(GS) + edges(GS)) = O(nodes(GS))$ .

### 9.3 View Generation for Global Schemata with Multiple Inheritance

A schema with multiple inheritance corresponds to a DAG rather than a tree structure. In this case the Edge-Creation algorithm presented in the previous section does no longer guarantee the creation of a valid view schema.

**Lemma 19.** *For a global schema  $GS$  with multiple inheritance, the Edge-Creation algorithm in Figure 9.1 generates a view schema with all required but possibly also redundant is-a arcs.*

**Proof:** Part II of the proof for Theorem 15, which shows that *all required* edges are added to  $VS$ , and part III, which shows that *no inconsistent* edges are added to  $VS$ , also apply here. However, part I of Theorem 15, which shows that no *redundant* edges will be added to  $VS$ , is no longer applicable. In a schema with multiple inheritance, some classes have more than one parent, and hence there may be more than one path connecting pairs of classes. In this case, the algorithm may insert redundant edges into the view as demonstrated by the example in Figure 9.4. There are two different paths to reach  $C_4$  from  $C_1$  in  $GS$  in Figure 9.4. The Edge-Creation algorithm searches along the first path and creates the ( $C_4$  is-a  $C_1$ ) arc (Figure 9.4.c). It then searches along the second path and discovers the ( $C_4$  is-a  $C_3$ ) arc (Figure 9.4.d). It turns out

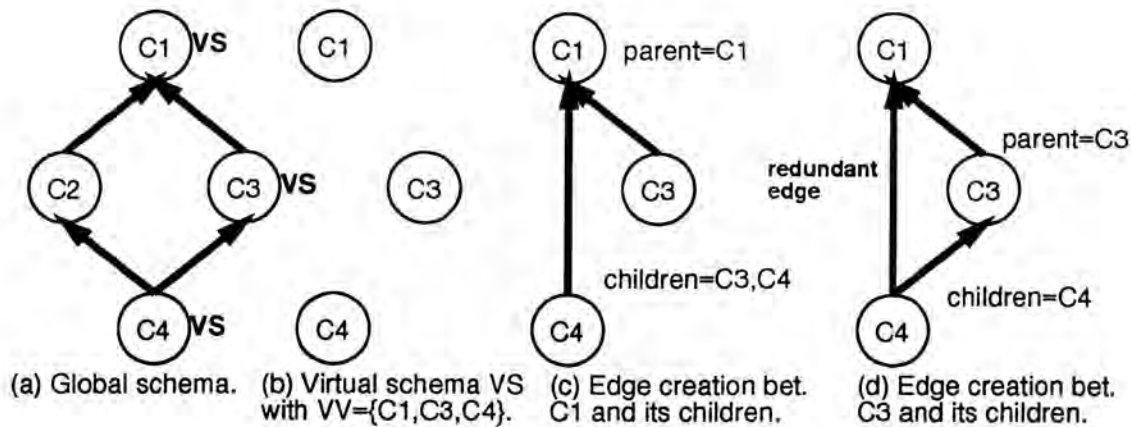


Figure 9.4: An Example of Creating Redundant View *Is-A* Arcs.

that the edge  $\langle C_4, C_1 \rangle$  is redundant, since there is an alternative path between  $C_4$  and  $C_1$  in the view consisting of the edges  $\langle C_4, C_3 \rangle$  and  $\langle C_3, C_1 \rangle$ . ■

In order to create a valid view schema for a global schema with multiple inheritance, I need to remove all redundant arcs from the schema graph. Below I present an algorithm which solves this problem based on the transitive closure property of the *is-a* relationship. Recall that an *is-a* arc between two nodes  $C_1$  and  $C_2$  is *redundant* if and only if there is a path of length larger or equal to one between these two nodes that goes through another class  $C_3$ . It is fairly straightforward to observe that I can reformulate our problem of view generation as the graph-theoretic problem called transitive reduction [Aho72]. Namely, the requirement of keeping all *required* and removing all *redundant* edges from the view schema VS is equivalent to finding “a graph  $G'$  for a given directed acyclic graph  $G$  such that (1) there is a directed path from vertex  $u$  to vertex  $v$  in  $G'$  whenever there is a path from  $u$  to  $v$  in  $G$  and (2) there is no graph with fewer arcs than  $G'$  satisfying the first condition” [Aho72]. This reduced graph  $G'$  is called the transitive reduction of  $G$ . The algorithm for the removal of redundant arcs using transitive reduction is given in Figure 9.5.

**Theorem 17. (Correctness)** *Given a view schema  $V=(VV, VE)$  with all required and possibly some redundant is-a edges, the Edge-Reduction algorithm generates a valid view schema  $V'=(VV', VE')$  with  $VV'=VV$ .*

**Proof:** Due to our reformulation of the view generation problem as the transitive reduction problem, the proof of Theorem 17 can be directly based on the proof for the transitive reduction algorithm [Aho72]. Details of the proof can be found in [Rund92b]. ■

Input:

$VS=(VV,VE)$  a view schema with the classes  $VC_i$  ( $i=1,\dots,n$ ) with  $VE$  containing all required and some redundant edges of  $VS$ .  
 $M_{view}$  is the incidence matrix for the input  $VS$ .

Output:

$M_{required}$  the incidence matrix for the valid view  $VS=(VV,VE)$ .

Data Structures:

$M_{view}$ ,  $M_{consistent}$ ,  $M_{redundant}$ ,  $M_{required}$ ,  $A$ ,  $B$  and  $C$  are Boolean matrices of size  $n=|VS|$ .

Algorithm A2: Removal of Redundant Edges.

```

procedure Transitive-Closure (A) return B is
  B := A;
  for i,j,k from 1 to |VS| do
    B[i,j] := B[i,j] or (A[i,k] and A[k,j]);
  endfor
end procedure
procedure Bool-Multiply (A,B) return C is
  for i,k from 1 to |VS| do
    C[i,k] :=  $\bigvee_{j=1}^n$  ( A[i,j] and B[j,k] );
  endfor
end procedure
procedure Graph-Subtract (A,B) return C is
  for i,j from 1 to |VS| do
    if (A[i,j]=1 and B[j,k]=0)
    then C[i,j] := 1;
    else C[i,j] := 0;
    endif
  endfor
end procedure
algorithm Edge-Reduction(VS) is
  0. Let  $M_{view}$  be the incidence matrix for the input view  $VS$ .
  1.  $M_{consistent}$  := Transitive-Closure(  $M_{view}$  );
  2.  $M_{redundant}$  := Bool-Multiply(  $M_{view}$ ,  $M_{consistent}$  );
  3.  $M_{required}$  := Graph-Subtract(  $M_{view}$ ,  $M_{redundant}$  );
  4. Let  $M_{required}$  be the incidence matrix for the output view  $VS$ .
end algorithm

```

Figure 9.5: The Edge-Reduction Algorithm for Removal of Redundant Arcs.

In the place of a detailed proof, I now give an intuitive argument for why the Edge-Reduction algorithm solves the view generation problem.

First, I explain why the transitive closure procedure used in step 1 finds all required and all redundant edges of the schema graph. By assumption, all required edges are already present in the input schema graph. The transitive closure then finds direct and/or indirect relationships between all pairs of classes in the schema graph and represents them in the form of an edge. The resulting incidence matrix  $M_{consistent}$  thus contains all required and all redundant edges of VS, i.e., all consistent edges.

Next, the Boolean matrix multiplication used in step 2 finds all edges that are redundant and not required. Namely, it finds all arcs  $e = \langle C_i, C_k \rangle$  for which there exists another class  $C_j$  in the graph such that there is an arc  $e_2 = \langle C_i, C_j \rangle$  in VS and a path  $p = \langle C_j, C_k \rangle$  in the transitive closure of VS. Put differently, Boolean matrix multiplication of  $M_{view}$  with  $M_{consistent}$  finds all edges  $\langle C_i, C_k \rangle$  for which there exists  $\langle C_i, C_j \rangle = 1$  in  $M_{view}$  and  $\langle C_j, C_k \rangle = 1$  in  $M_{consistent}$  for some  $j$ . The resulting incidence matrix  $M_{redundant}$  thus contains all redundant edges of VS.

Lastly, the graph subtraction used in step 3 removes all edges of the matrix  $M_{redundant}$  (i.e., all redundant edges) from the matrix  $M_{consistent}$  (i.e., from all required and redundant edges). Clearly, the resulting incidence matrix  $M_{required}$  will contain all edges required for VS and no others. Next, I present an example which illustrates these steps.

**Example 55.** *In Figure 9.6, I present an example of applying the Edge-Reduction algorithm to remove redundant edges from the view VS depicted in Figure 9.6.a. The incidence matrix  $M_{view}$  for the view VS is shown in Figure 9.6.b. Next, I apply the Transitive-Closure procedure to  $M_{view}$  to find the class relationships between all pairs of classes in the view. The resulting schema graph that contains all required and all redundant edges is shown in Figure 9.6.c and its corresponding incidence matrix  $M_{consistent}$  in Figure 9.6.d. Note that in this example all redundant edges did already exist in the original view schema, and hence no additional edges were created by this last step.*

*I now apply Boolean multiplication to  $M_{view}$  and  $M_{consistent}$  to find all redundant edges. For instance, the edge  $\langle C_5, C_1 \rangle$  is redundant because  $\langle C_5, C_3 \rangle$  in  $M_{view}$  and  $\langle C_3, C_1 \rangle$  in  $M_{consistent}$ . Figures 9.6.e and 9.6.f present the resulting incidence matrix  $M_{redundant}$  and the corresponding schema graph, respectively.*

Finally, I subtract the matrix of all redundant edges in Figure 9.6.e from the matrix of all consistent edges in Figure 9.6.d. The result achieved by this graph-subtraction procedure is shown in Figures 9.6.g and 9.6.h, respectively. Clearly, the view VS shown in Figure 9.6.h contains all required and no redundant and no inconsistent edges.

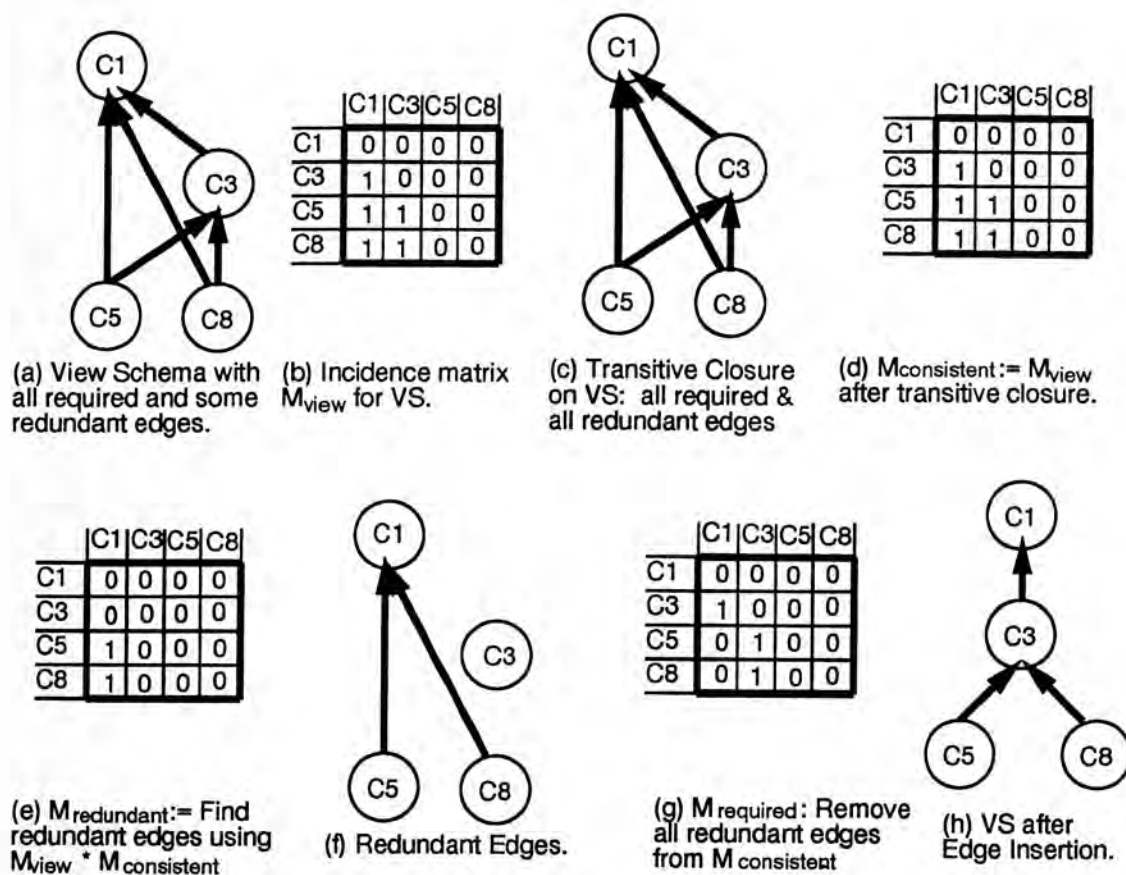


Figure 9.6: An Example of Removing Redundant Edges Using the Edge-Reduction Algorithm.

**Theorem 18. (Complexity)** *The worst-case complexity of the Edge-Reduction algorithm is  $O(|VS|^3)$  with  $|VS|$  the number of classes in the view schema VS.*

**Proof:** The complexity of the Edge-Reduction algorithm can be easily determined by inspection of the algorithm description given in Figure 9.5, since the Transitive-Closure and the Bool-Multiply procedures involve the computation of three nested for-loops of size  $|VS|$  each. ■

Theoretically, the complexity of these two algorithms has been shown to be  $O(n^{2.37}(\log n)^2)$  time for large  $n$ . Since the size of a schema graph is generally not large, a straightforward implementation of these algorithms as shown in Figure 9.5 with cube complexity is preferable [Aho74].

Now I put the Edge-Creation algorithm together with the Edge-Reduction algorithm to create an algorithm for automatic view generation of schema graphs with multiple inheritance.

**Theorem 19. (Correctness)** *Given a global schema  $GS=(V,E)$  with possibly multiple inheritance and a set of view classes  $VV \subseteq V$ , the View-Generation algorithm (Figure 9.7) generates a valid view  $VS=(VV,VE)$ .*

**Proof:** The proof of correctness of View-Generation algorithm is straightforward. First, it applies the Edge-Creation algorithm to create *all required* and possibly some redundant edges of the view schema. Theorem 15 shows the correctness of this edge creation algorithm. Then, it applies the Edge-Reduction algorithm to remove *all redundant* edges. The correctness of this edge removal step has been shown by Theorem 17. The result thus is a view schema with all required and no redundant edges, i.e.,  $VS$  is valid. ■

**Input:**

Global Schema  $GS = (V,E)$  and View Schema  $VS=(VV,VE)$   
with  $VV \subseteq V$  marked by the view identifier  $\langle VS \rangle$  and  $VE=\emptyset$ .

**Output:**

The View Schema  $VS=(VV,VE)$  with  $VS$  valid.

**Algorithm A3:** Complete Valid View Schema Generation 1

```

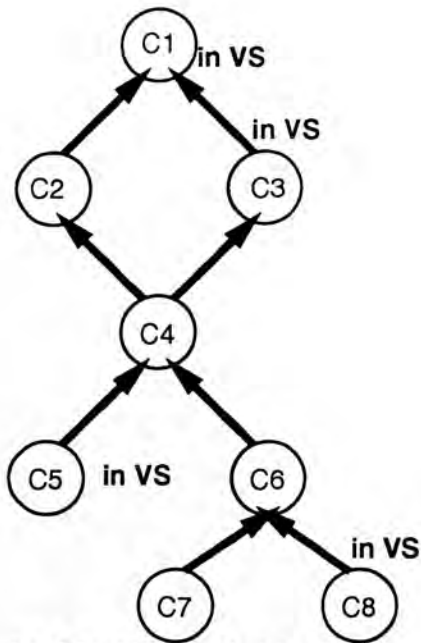
algorithm View-Generation( $GS, VS$ ) is
  Edge-Creation( $GS, VS$ );
  Edge-Reduction( $VS$ );
end algorithm;
```

Figure 9.7: The View-Generation1 Algorithm.

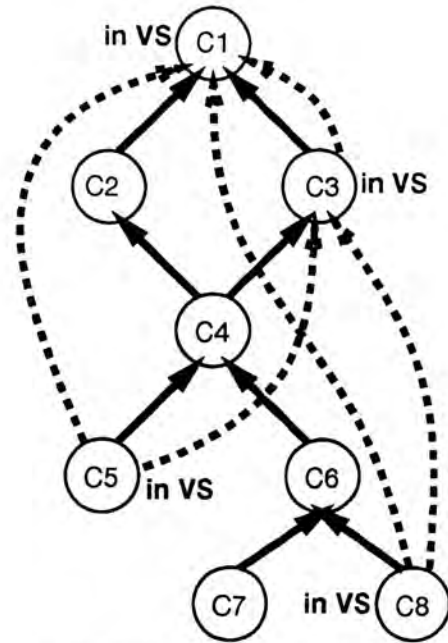
The View-Generation algorithm is explained based on the example.

**Example 56.** *In this example, I discuss the application of the View-Generation algorithm to the example schema depicted in Figure 9.8. Figure 9.8.a shows the global schema  $GS$  with multiple inheritance. A subset of the nodes of  $GS$  are marked by the view identifier  $\langle VS \rangle$ . Figures 9.8.b and 9.8.c demonstrate the computation of the redundant and the required edges of  $VS$  using the Edge-Creation algorithm. Thereafter, the incidence matrix  $M_{view}$  of  $VS$  is shown in Figure 9.8.d. Finally, the*

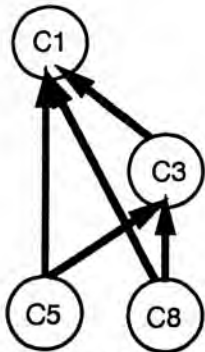




(a) GS with multiple inheritance and VS with  $VV=\{C1, C3, C5, C8\}$ .



(b) Apply Edge-Generation algorithm.



(c) VS generated by Edge-Generation.

	C1	C3	C5	C8
C1	0	0	0	0
C3	1	0	0	0
C5	1	1	0	0
C8	1	1	0	0

(d)  $M_{view}$

	C1	C3	C5	C8
C1	0	0	0	0
C3	1	0	0	0
C5	1	1	0	0
C8	1	1	0	0

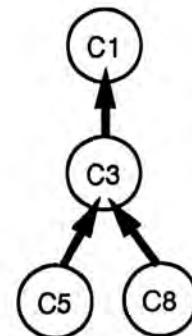
(e)  $M_{consistent}$

	C1	C3	C5	C8
C1	0	0	0	0
C3	0	0	0	0
C5	1	0	0	0
C8	1	0	0	0

(f)  $M_{redundant}$

	C1	C3	C5	C8
C1	0	0	0	0
C3	1	0	0	0
C5	0	1	0	0
C8	0	1	0	0

(g)  $M_{required}$



(h) Valid VS.

Figure 9.8: Example of the View-Generation1 Algorithm for Creating a Valid View Schema.

three incidence matrices  $M_{consistent}$ ,  $M_{redundant}$ , and  $M_{required}$  are shown in Figures 9.8.e, 9.8.f, and 9.8.g, respectively. Figure 9.8.g depicts the final virtual schema  $VS$  that is generated by the Edge-Reduction procedure.

Next, I modify the Edge-Creation procedure proposed in Section 9.2 for graphs with single inheritance such as to run efficiently on a DAG graph structure. For this purpose, I suggest two changes to the Edge-Creation procedure in Figure 9.1. First, I mark each class that belongs to the view  $\langle VS \rangle$  as 'parent-processed', so that it will be used as a parent node exactly once. The algorithm then finds for each (marked) parent class  $P$  all its children that also belong to  $\langle VS \rangle$ . In order to avoid retraversing nodes  $C$  during this search, I mark all classes traversed for parent  $P$  by the label  $child\text{-}processed(C)='P'$ .

**Theorem 20. (Correctness)** *Given a  $GS=(V,E)$  with multiple inheritance, then the Edge-Creation-DAG algorithm in Figure 9.9 generates a complete and consistent view schema  $VS=(VV,VE)$ .*

**Proof:** The Edge-Creation-DAG algorithm in Figure 9.9 is a simple extension of the one in Figure 9.1. It can easily be seen that all statements that have been added (indicated by underlining them in Figure 9.9) take care of avoiding the reprocessing of nodes due to the DAG structure of the global schema graph. Since the functionality of the algorithm is not modified, the correctness of the algorithm can be directly derived from the correctness of the Edge-Creation algorithm in Theorem 15. ■

Theorem 16 shows that the Edge-Creation algorithm defined in Figure 9.1 is of linear complexity when applied to a tree-structured schema graph. For an arbitrary DAG, this complexity does no longer hold. As discussed above, I have adjusted the Edge-Creation algorithm for a graph with multiple inheritance (Figure 9.9). It is straightforward to show that the algorithm's complexity is quadratic in the number of nodes and edges in  $GS$  [Rund92b].

**Theorem 21. (Complexity)** *The complexity of the Edge-Creation-DAG algorithm in Figure 9.9 is quadratic in the number of nodes and edges in  $GS$ , i.e.,  $O(nodes(GS)*edges(GS))$ , assuming the class hierarchy of the global schema is a DAG.*

**Proof:** For a detailed proof see [Rund92b], while below I give the intuitive reasoning. First, each node in  $GS$  that belongs to the view  $\langle VS \rangle$  is processed exactly once as parent node. And, there are at most  $O(|GS|)$  such nodes. Second, for each parent node  $P$  that belongs to the view  $\langle VS \rangle$  I traverse the graph downwards to find all view classes  $C$  closest to  $P$ . Such a graph traversal can be done in  $O(nodes(GS) + edges(GS))$  time. Therefore, the total complexity is  $O(nodes(GS) * (nodes(GS) + edges(GS))) = O(|GS| * edges(GS))$ . ■

**Assumption:**

Global Schema  $GS$  is a DAG (Multiple inheritance).

**Input:**

Global Schema  $GS = (V, E)$  and View Schema  $VS = (VV, VE)$   
with  $VV \subseteq V$  marked by the view identifier  $\langle VS \rangle$  and  $VE = \emptyset$ .

**Output:**

Determine a set of *is-a* edges  $VE$  on  $VV$ ,  
such that  $VS = (VV, VE)$  is a *consistent* and *complete* view on  $GS$ .

**Data Structures:**

ParentQueue is a queue to hold nodes of  $GS$ .

ChildrenQueue is a queue to hold nodes of  $GS$ .

Parent and Child are variables that hold one class each.

child-processed( $\langle \text{class} \rangle$ )=P indicates whether  $\langle \text{class} \rangle$  has been  
processed as child for the parent P.

parent-processed( $\langle \text{class} \rangle$ ) is a flag for whether  $\langle \text{class} \rangle$  has been  
processed as parent.

**Algorithm A4: Creation of Is-A Arcs for A View Schema.**

```

algorithm Edge-Creation-DAG( $GS, VS$ ) is
  Push the root of  $GS$  onto ParentQueue.
  while (Parent = pop(ParentQueue)) do
    Push all children of Parent in  $GS$  onto ChildrenQueue.
    while (Child = pop(ChildrenQueue)) do
      if child-processed(Child)  $\neq$  parent then
        if Child is in  $VS$  then
          insert isa(Child, Parent) into edges( $VS$ );
          if parent-processed(Child)=false then
            Push Child onto ParentQueue;
          end if
          parent-processed(Child)=true;
        else
          Push all children of Child in  $GS$  onto ChildrenQueue;
        endif
      endif
      child-processed(Child)=parent;
    endwhile
  endwhile
end algorithm;

```

Figure 9.9: The View Schema Creation Algorithm for DAGs.

Let  $n$  and  $m$  denote the number of edges and vertices of a graph. The traversal of a graph is linear time in the number of edges and vertices, i.e.,  $O(n+m)$ . If the graph has a small number of edges, say  $O(n)$ , then the Edge-Creation-DAG algorithm would be of quadratic complexity in the number of nodes, i.e.,  $O(m \times n) = O(n^2)$ . Otherwise, if the number of edges of a graph is large, say  $O(n^2)$ , then the complexity of the Edge-Creation-DAG algorithm would be  $O(m \times n) = O(n^3)$ .

**Theorem 22. (Complexity)** *The worst-case complexity of the View-Generation algorithm is  $O(\min(|GS| * \text{edges}(GS), |VS|^3))$  with  $|GS|$  the number of classes in the global schema  $GS$  and  $|VS|$  the number of classes in the view schema  $VS$ .*

**Proof:** Theorem 21 shows that the graph reduction from  $GS$  to  $VS$  as defined in Figure 9.9 has the complexity of  $O(|GS| * \text{edges}(GS))$ . Theorem 18 shows that the complexity of the redundant edge removal is of the order  $O(|VS|^3)$ . Consequently, the total complexity for the View-Generation algorithm is  $O(\min(|GS| * \text{edges}(GS), |VS|^3)) = O(|GS|^3)$ . ■

The Edge-Reduction algorithm removes all redundant edges from a view schema. Consequently, I need no longer be concerned with preventing the creation of redundant edges during the first stage of the view schema creation algorithm. I could therefore replace the Edge-Creation algorithm used for the step by a simple transitive closure algorithm that does the following: Given a graph  $G=(V,E)$ , it creates a new graph  $G^*=(V,E^*)$  with  $(\forall C_i, C_j \in V) ((C_i \text{ is-a } C_j) \in E^* \iff (C_i \text{ is-a } * C_j) \in E)$ . As can easily be seen, this generates all required but also *all* redundant *is-a* edges.

Using transitive closure for edge creation will lead to a simpler implementation of the view creation algorithm, since the Edge-Reduction procedure is already implemented using transitive closure. In addition, I generate the incidence matrix for  $GS$ , rather than  $VS$ , and all subsequent computations are performed on the matrix representation. Hence, rather than first manipulating the schema graph to insert edges using the Edge-Creation algorithm and then manipulating the schema graph to remove redundant edges using the Edge-Reduction algorithm, this algorithm works directly on the matrix representation. Only at the end, after all required edges have been identified, the algorithm inserts these edges into the graph. Furthermore, I will show below that this solution is more efficient for a schema graph with a large number of edges, since theoretically it can accomplish the same task in  $O(n^{2.37}(\log n)^2)$  time – in contrast to the worst case complexity of  $O(n^3)$  for the Edge-Creation-DAG algorithm [Aho74]. The detailed View-Generation2 algorithm, which uses the transitive closure algorithm in place of the original Edge-Creation procedure, is presented in Figure 9.10.

**Input:**

Let  $GS=(V,E)$  be a global and  $VS=(VV,VE)$  a view schema defined on  $GS$  with  $VV \subseteq V$  and  $VE = \emptyset$ .

**Output:**

$VS=(VV,VE)$  a valid view schema.

**Data Structures:**

$A, M_{global}$ : Boolean matrices of size  $n = |GS|$ .

$B, M_{consistent}, M_{redundant}, M_{required}$ : Boolean matrices of size  $n = |VS|$ .

**Algorithm A4: Complete Valid View Schema Generation 2**

```

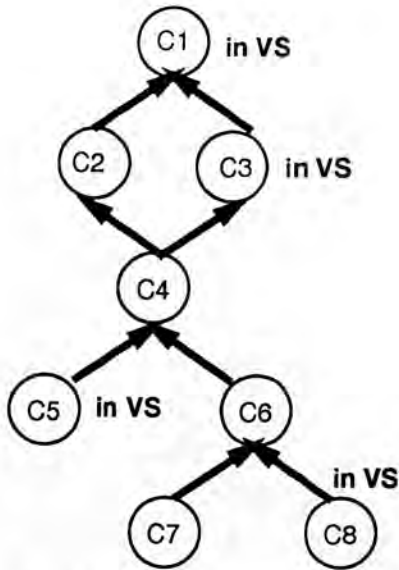
procedure Reduce-Matrix (A) return B is
  for  $i, j$  from 1 to  $|A|$  do
    if  $(C_i \in VS)$  and  $(C_j \in VS)$  then
       $B[C_i, C_j] := A[C_i, C_j]$ ;
    endif
  endfor
end procedure

```

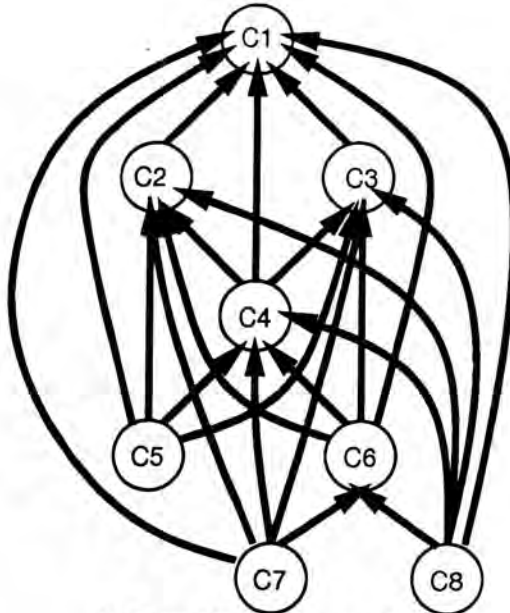
**algorithm** View-Generation2( $GS, VS$ ) **is**

0. Let  $M_{global}$  be the incidence matrix for the global schema  $GS$ .
  1.  $M_{global} := \text{Transitive-Closure}( M_{global} )$ ;
  2.  $M_{consistent} = \text{Reduce-Matrix}( M_{global} )$ ;
  3.  $M_{redundant} := \text{Bool-Multiply}( M_{view}, M_{consistent} )$ ;
  4.  $M_{required} := \text{Graph-Subtract}( M_{view}, M_{redundant} )$ ;
  5. Let  $M_{required}$  be the incidence matrix for the view  $VS$ ;
- end algorithm**;

Figure 9.10: The View-Generation2 Algorithm.



(a) GS with multiple inheritance and VS with  $VV=\{C1,C3,C5,C8\}$ .



(c) Transitive Closure on GS.

	C1	C2	C3	C4	C5	C6	C7	C8
C1	0	0	0	0	0	0	0	0
C2	1	0	0	0	0	0	0	0
C3	1	0	0	0	0	0	0	0
C4	0	1	1	0	0	0	0	0
C5	0	0	0	1	0	0	0	0
C6	0	0	0	1	0	0	0	0
C7	0	0	0	0	0	1	0	0
C8	0	0	0	0	0	1	0	0

(c) Initialize incidence matrix A for GS.

	C1	C2	C3	C4	C5	C6	C7	C8
C1	1	0	0	0	0	0	0	0
C2	1	0	0	0	0	0	0	0
C3	1	0	0	0	0	0	0	0
C4	1	1	1	0	0	0	0	0
C5	1	1	1	1	0	0	0	0
C6	1	1	1	1	0	0	0	0
C7	1	1	1	1	0	1	0	0
C8	1	1	1	1	0	1	0	0

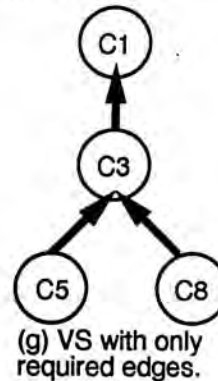
(d) Matrix A holds transitive closure for is-a arcs in GS.

	C1	C3	C5	C8
C1	0	0	0	0
C3	1	0	0	0
C5	1	1	0	0
C8	1	1	0	0

(e) Reduce A to B: B holds the transitive closure for VS.

	C1	C3	C5	C8
C1	0	0	0	0
C3	1	0	0	0
C5	0	1	0	0
C8	0	1	0	0

(f) Find all required edges for B.



(g) VS with only required edges.

Figure 9.11: An Example of the View-Generation2 Algorithm for Creating a Valid Schema.

Next, I explain the View-Generation2 algorithm for creating a valid view schema based on the example presented in Figure 9.11.

**Example 57.** *The steps of the View-Generation2 algorithm for valid view schema creation are:*

1. *The input to the View-Generation2 algorithm is the global schema  $GS$  with a subset of nodes marked by the identifier  $\langle VS \rangle$  shown in Figure 9.11.a.*
2. *Step 0 initializes the incidence matrix  $M_{global}$  to represent the global schema  $GS$  in a matrix format (Figure 9.11.b).*
3. *Step 1 then computes the transitive closure on the is-a relationships of the classes in  $GS$ . The transitive closure of  $GS$  in a graph and incidence matrix representation are shown in Figure 9.11.c and 9.11.d, respectively.*
4. *Step 2 then reduces the incidence matrix  $M_{global}$  for  $GS$  down to the incidence matrix  $M_{view}$  for  $VS$  by selecting all classes that belong to the virtual schema  $VS$  and all arcs of  $GS$  among these classes.  $M_{global}$  and  $M_{view}$  are shown in Figure 9.11.d and 9.11.e, respectively.*
5. *Steps 3 and 4 finds all required edges of the view schema  $VS$  (Figure 9.11.f). A detailed example of how this is computed using transitive reduction can be found in Figure 9.6. In particular, Figure 9.6.b is equal to the input to the procedure shown in Figure 9.11.e and Figure 9.6.f is equal to the result given in Figure 9.11.g.*
6. *Note that  $M_{view}$  will always contain all required and all redundant edges of  $VS$ , i.e., by default,  $M_{view} = M_{consistent}$ . Therefore, the transitive closure on  $M_{view}$  need not be computed in this context, as has been done in Figure 9.5.*
7. *Step 5 then inserts all arcs into  $VS$  that have a value of one in the incidence matrix  $M_{required}$  (Figure 9.11.g). The resulting graph is the final virtual schema  $VS$ .*

Having given the intuition behind the steps of the View-Generation2 algorithm, I sketch a proof for its correctness.

**Theorem 23.** *The View-Generation2 algorithm presented in Figure 9.10 generates a valid view schema.*

**Proof:** The View-Generation2 algorithm implements the above described steps and thus creates a valid schema. The correctness of each step is shown below:

1. First, the View-Generation2 algorithm initializes the incidence matrix  $M_{global}$  to store the graph  $GS$  in a matrix representation. This is a standard matrix representation for a graph.

2. Step 1 of the View-Generation2 algorithm computes the transitive closure on the *is-a* relationships in  $M_{global}$  and finds all pairs of classes that are *is-a\** related in  $GS$ . The proof of correctness for this can be derived from a standard algorithms book (e.g., [Aho74]) and thus is not given here. The result is the following:

$$M_{global}(C_i, C_j) = \begin{cases} 1 (true) & \text{if } (C_1 \text{ is-a } C_2) \text{ in } GS \\ 0 (false) & \text{otherwise} \end{cases} \quad (9.1)$$

Using the terminology defined in Section 3.5, this is equivalent to representing *all required* and *all redundant* but *no incompatible is-a* relationships of the global graph.

3. Step 2 of the View-Generation2 algorithm reduces the incidence matrix  $M_{global}$  to the incidence matrix  $M_{consistent}$  by extracting all entries  $M_{global}(C_i, C_j)$  with  $C_i, C_j \in VS$ , i.e.,  $M_{consistent}(C_i, C_j) := M_{global}(C_i, C_j)$  for  $C_i, C_j \in VS$ . Since  $M_{global}$  represents the transitive closure for  $GS$ ,  $M_{consistent}$  will now represent the transitive closure for  $VS$  as explained below.

The transitive closure of *is-a* edges means that for each class  $C_i \in GS$ , the matrix  $M_{global}$  will hold the edges to all other classes  $C_j$  with which  $C_i$  is *is-a* related, i.e.,  $(\forall j \text{ with } 1 \leq j \leq |GS|) ((C_i \text{ is-a } C_j) \implies M_{global}(C_i, C_j) = 1)$ . I can consequently conclude the following: (for each  $C_i \in VS$ )  $(\forall j \text{ with } 1 \leq j \leq |GS|) ((C_j \in VS) \text{ and } (C_i \text{ is-a } C_j)) \implies M_{global} = M_{consistent}(C_i, C_j) = 1$ .

The matrix representation of  $M_{consistent}$  thus corresponds to the following:

$$M_{consistent}(C_i, C_j) = \begin{cases} 1 (true) & \text{if } (C_1 \text{ is-a } C_2) \text{ in } VS \\ 0 (false) & \text{otherwise} \end{cases} \quad (9.2)$$

This clearly corresponds to *all required* and *all redundant* but *no incompatible is-a* relationships of the view graph  $VS$ .

4. Steps 3 and 4 correspond to Edge-Reduction algorithm, the result of which is the incidence matrix  $M_{required}$ .  $M_{required}$  represents all required *is-a* relationships of the view  $VS$  and no others. This step has already been shown correct in Theorem 17.
5. Step 5 then inserts all required edges into  $VS$ : For each pair of classes  $(C_i, C_j)$  with  $M_{required}[C_i, C_j] = 1$ , it inserts the edge  $(C_i \text{ is-a}^d C_j)$  into  $VS$ . This is a standard conversion of a matrix representation into its corresponding graph format. The correctness of this step can easily be seen, since  $M_{required}$  is the incidence matrix representing all required *is-a* relationships of the view  $VS$  and no others.



**Theorem 24. (Complexity)** *The worst-case complexity of the View-Generation2 algorithm is  $O(|GS|^3)$  with  $|GS|$  the number of classes in the global schema  $GS$ .*

**Proof:** I break the proof of complexity into the following steps:

- The View-Generation2 algorithm first initializes the incidence matrix  $M_{global}$  by storing a value at each position  $M_{consistent}(i,j)$ . The complexity thus is the size of the matrix, i.e.,  $O(|GS|^2)$ .
- The Transitive-Closure procedure involves the computation of three nested for-loop of size  $|GS|$  each. The body of the three for-loops is of constant time. Hence, this computation has the complexity of  $O(|GS|^3)$ .
- The Reduce-Matrix function inspects each entry of matrix  $M_{consistent}$  exactly once, and thus has complexity of  $O(|GS|^2)$ .
- The complexity of the Edge-Reduction procedure (steps 3 and 4) has been shown to be  $O(|VS|^3)$  in Theorem 18.
- Lastly, the insertion of all required edges is done by inspecting each field in the matrix  $M_{required}$  once. The complexity thus is the size of the matrix, i.e.,  $O(|VS|^2)$ .

I can now combine these complexities to get the total complexity of the View-Generation2 algorithm. Note that the two most time-consuming steps of the algorithm are the transitive closure on  $GS$  and the longest path computation on  $VS$ , while all other steps are of quadratic or lesser complexity. Since the number of classes in  $GS$  is larger or equal to the number of classes in  $VS$ , I can conclude that the complexity of the first computation outweighs the complexity of the second. Hence, the total complexity for the View-Generation2 algorithm is  $O(|GS|^3)$ . ■

Theoretically, the complexity of both the transitive closure and Boolean multiplication algorithms has been shown to be  $O(n^{2.37}(\log n)^2)$  time for large  $n$  [Aho74]. Since the size of a schema graph is generally not large, the straightforward implementation of these algorithms as shown in Figure 9.10 with cube complexity is preferable.

Note that the View-Generation2 algorithm has to be executed once for each view schema, namely, after the initial specification of the view schema. Furthermore, the first part of the the View-Generation2 algorithm (the transitive closure on the global schema) will not have to be computed for each and every view schema. Instead it can be computed once a-priori (after creating the base schema). Thereafter it is updated only when new virtual classes are added to the global schema. Hence, the

complexity for view schema generation in most instances only equals the complexity of the second part of the View-Generation2 algorithm, i.e.,  $\text{complexity}(\text{View-Generation2}) = \text{complexity}(\text{Edge-Reduction}) = O(|VS|^3)$ , which may be considerably smaller than the complete complexity shown in Theorem 24.

# Chapter 10

## Algorithms for Enforcing View Consistency

In this chapter, I describe two algorithms that I have developed to solve different aspects of view consistency. In Section 10.1, I present an algorithm, called View-Validity Checker, for checking the *is-a* validity of a manually specified view. This represents an alternative to the approach of automatically generating a valid view generalization hierarchy as proposed in Section 9. In Section 10.2, I introduce the Closed-View Generator. This algorithm checks whether a given view is (*type-*)*closed* or not. In addition it transforms a view that is found to be not closed into a (*type-*)*closed* view schema.

### 10.1 The View Validity Checker

In Section 9, I have described algorithms for the automatic generation of a valid view generalization hierarchy. These algorithms are based on the validity criterion of a view schema given in Definition 19. Namely, the algorithm enforces the resulting view schema to be *minimal* and *complete* and *consistent*. Note that under certain circumstances it may be desirable to relax this strict requirement such as to allow the view definer more freedom in modeling the view schema. For instance, the view definer may want to explicitly create redundant hierarchies in the view schema, if this provides a more intuitive classification of the application concepts. Similarly, he or she may not desire to capture complete information about all class relationships between the view classes in the view. Of course, I cannot allow the view definer to insert inconsistent edges into the view schema, since they would represent an error in the model and thus would result in an error during query processing. In the terminology introduced in Section 3, a less restrictive yet acceptable definition of a view schema corresponds to a *consistent* – but not necessarily *complete* or *minimal* – view (Definition 8) rather than a *valid* view (Definition 19).

To support the manual specification of the view schema hierarchy, I need to extend the view definition language presented in Section 8 with operators to manipulate generalization edges. Since this follows directly the spirit of the other operators introduced in that section, it is not further discussed in this dissertation.

More importantly, manual view specification may lead to the introduction of inconsistent class relationships into the view schema. I therefore need to develop a view consistency checker that verifies the consistency of the view schema. Below, I propose such a view consistency checker that can determine the consistency of an edge with a table lookup of  $O(1)$  (Figure 10.1).

The view consistency checker proposed in Figure 10.1 is directly based on the view generation algorithm presented in Section 9. First I apply the view generation algorithm on a temporary view schema  $VS'=(VV',VE')$  with  $VV'=VV$  in order to determine the *valid* view schema. Recall that this algorithm also generates a number of intermediate boolean matrices called  $M_{consistent}$ ,  $M_{redundant}$ ,  $M_{required}$  (Figure 9.5). As explained in Section 9, the boolean matrix  $M_{consistent}$  models all consistent view *is-a* relationships, i.e., they are either required and/or redundant, while  $M_{redundant}$  models all redundant and  $M_{required}$  all required view *is-a* relationships. I exploit these boolean matrices for generating one table, called consistency-status, that describes the consistency status of the edge between each pair of classes in the view. For each edge  $e = \langle Ci, Cj \rangle$  that is required, i.e.,  $M_{required}(Ci, Cj)=1$ , I store the value *required* into consistency-status( $Ci, Cj$ ). For each edge  $e = \langle Ci, Cj \rangle$  that is consistent but not required, i.e.,  $M_{required}(Ci, Cj)=0$  and  $M_{redundant}(Ci, Cj)=1$ , I store the value *redundant* into consistency-status( $Ci, Cj$ ). Finally, all other edges are marked as *inconsistent*. Given this table of consistency-status information, determining the consistency of an edge becomes a simple matter of table-lookup as described in the Edge-Consistency procedure in Figure 10.1.

The Edge-Consistency procedure can be used in a number of different ways. First, it can verify the consistency of an edge that is incrementally added during view specification. Second, it could be applied in a batch fashion to all edges in the view after completion of manual view specification. Furthermore, it could also be used to inform the user of any missing required edges by traversing the table and checking for each entry  $M_{required}(Ci, Cj)=1$ , whether the corresponding edge  $\langle Ci, Cj \rangle$  does indeed exist in the view.

**Example 58.** *On the left-hand side of Figure 10.2, I reiterate the steps of automatically deriving a view VS from the global schema. On the right-hand side, I describe how the consistency-status table is updated concurrently to reflect the consistency status of each edge. Figure 10.2.a presents the global schema as well as a set of*

**Input:**

Global Schema  $GS = (V, E)$  and View Schema  $VS = (VV, VE)$   
with  $VV \subseteq V$  marked by the view identifier  $\langle VS \rangle$  and  $VE = \emptyset$ .

**Output:**

The View Schema  $VS = (VV, VE)$  with  $VS$  valid.

**Data Structures:**

enumeration type status is  $\{required, redundant, inconsistent\}$ .  
matrix consistency-status of type status and of size  $n = |VS|$ .  
 $A, B$  and  $C$  are boolean matrices of size  $n = |VS|$ .  
 $M_{view}, M_{consistent}, M_{redundant}, M_{required}$  are boolean matrices of size  $n = |VS|$ .

**Algorithm A5: View Consistency Checking Algorithm**

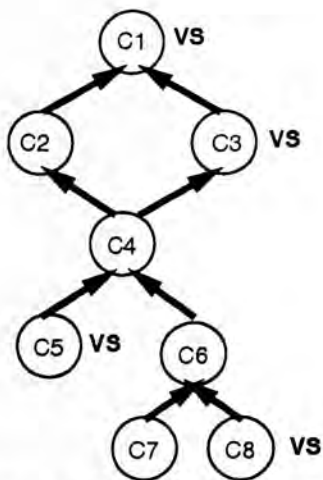
```

procedure Consistency-Status-Creation( $VS$ ) is
  Let  $VS' = (VV', VE')$  be a temporary view schema with  $VV' = VV$ .
  Let  $M_{view}$  be the incidence matrix for  $VS'$ .
  View-Schema-Creation1( $GS, VS'$ );
  for  $i, j$  from 1 to  $|VS|$  do consistency-status[ $i, j$ ] = inconsistent; end for
  for  $i, j$  from 1 to  $|VS|$  do
    if  $M_{consistent}[i, j] = 1$  then consistency-status[ $i, j$ ] = redundant;
  end for
  for  $i, j$  from 1 to  $|VS|$  do
    if  $M_{required}[i, j] = 1$  then consistency-status[ $i, j$ ] = required;
  end for
end procedure

procedure Edge-Consistency( $GS, VS, e = \langle Ci, Cj \rangle$ ) is
  case (consistency-status( $Ci, Cj$ )) is
    required:
      return(okay);
    redundant:
      return("WARNING:  $e = \langle Ci, Cj \rangle$  is a redundant is-a relationship");
    inconsistent:
      return("ERROR: Entry of inconsistent  $e = \langle Ci, Cj \rangle$  is rejected");
  end case
end procedure

```

Figure 10.1: The View-Consistency Checking Algorithm.

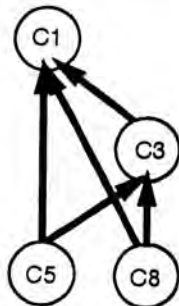


(a) Given global schema GS and VS with  $VV=\{C1,C3,C5,C8\}$ .

	C1	C3	C5	C8
C1	<i>inconsist.</i>	<i>inconsist.</i>	<i>inconsist.</i>	<i>inconsist.</i>
C3	<i>inconsist.</i>	<i>inconsist.</i>	<i>inconsist.</i>	<i>inconsist.</i>
C5	<i>inconsist.</i>	<i>inconsist.</i>	<i>inconsist.</i>	<i>inconsist.</i>
C8	<i>inconsist.</i>	<i>inconsist.</i>	<i>inconsist.</i>	<i>inconsist.</i>

(b) Initialize the Consistency-Status (CS) matrix.

	C1	C3	C5	C8
C1	0	0	0	0
C3	1	0	0	0
C5	1	1	0	0
C8	1	1	0	0

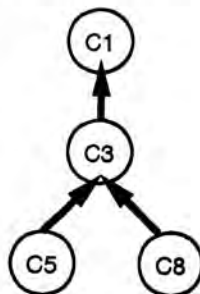


(c) Determine all required and all redundant edges of VS, i.e., generate  $M_{consistent}$

	C1	C3	C5	C8
C1	<i>inconsist.</i>	<i>inconsist.</i>	<i>inconsist.</i>	<i>inconsist.</i>
C3	<i>redundant</i>	<i>inconsist.</i>	<i>inconsist.</i>	<i>inconsist.</i>
C5	<i>redundant</i>	<i>redundant</i>	<i>inconsist.</i>	<i>inconsist.</i>
C8	<i>redundant</i>	<i>redundant</i>	<i>inconsist.</i>	<i>inconsist.</i>

(d) Update CS after determining all consistent edges.

	C1	C3	C5	C8
C1	0	0	0	0
C3	1	0	0	0
C5	0	1	0	0
C8	0	1	0	0



(e) After finding all required edges of VS, i.e., generate  $M_{required}$ .

	C1	C3	C5	C8
C1	<i>inconsist.</i>	<i>inconsist.</i>	<i>inconsist.</i>	<i>inconsist.</i>
C3	<i>required</i>	<i>inconsist.</i>	<i>inconsist.</i>	<i>inconsist.</i>
C5	<i>redundant</i>	<i>required</i>	<i>inconsist.</i>	<i>inconsist.</i>
C8	<i>redundant</i>	<i>required</i>	<i>inconsist.</i>	<i>inconsist.</i>

(f) Update CS after determining all required edges.

Figure 10.2: Preparing the Consistency-Status Matrix for Consistency Checking.

view classes  $VV$ . Figure 10.2.b shows the initialization of the consistency-status table to values equal to inconsistent. Figure 10.2.c then depicts the view schema after the generation of all required and all redundant edges. Correspondingly, the table in Figure 10.2.d is updated by storing the value redundant for each (consistent) edge of  $VS$ . Figure 10.2.e shows the final view schema with all required edges; and Figure 10.2.f demonstrates how all required edges are entered into the consistency-status table.

## 10.2 The View Closure Checker

### 10.2.1 Basic Concepts

Unlike the class generalization relationships, the property decomposition relationships are not explicitly (nor independently from the actual classes) inserted into a schema. Instead, they are implicitly determined by the specification of a derived class. Recall that the definition of a virtual class automatically determines its type, i.e., all its property relationships with other classes in the view schema. For example, I create a property decomposition arc labeled  $p$  between the classes  $C1$  and  $C2$ ,  $a = \langle C1, C2, p \rangle$ , by defining the class  $C1$  to have a property function  $p$  with the  $\text{domain}_p(C1) := C2$ . This implies that any customization of this property decomposition hierarchy by the view definer is taken care of during the class derivation phase of *MultiView*. Nonetheless, the closure criterion of a view schema can be verified only after the selection of all view classes, since it is a function of (the relationships among all classes in) the complete schema. rather than of an individual class.

As indicated in Section 3.6, instead of just checking whether a given view is closed or not, it is more useful to transform a view that is found to be not closed into a *type-closed* view schema. In this section I present an algorithm, called Closed-View Generation algorithm, that solves this problem. In particular, the algorithm automatically determines the minimal<sup>1</sup> set of classes by which the view schema  $VS$  has to be extended in order for the view to be *type-closed*. I describe this minimal set below.

**Theorem 25. (Correctness)** *Given a view schema  $VS=(VV, VE)$  defined on the global schema  $GS=(V, E)$ . Then  $MIN := (\bigcup_{C_i \in VV} (Uses^*(C_i))) - VV$  is the minimal subset of classes from  $V$  that have to be added to the view  $VS$  to make it closed.*

<sup>1</sup>I assume that all classes initially selected for the view are indeed required, i.e., none of the view classes can be dropped in order to make the view *type-closed*.

**Proof:** I prove Theorem 25 in two parts. In part I, I show the sufficiency of the set  $MIN = \bigcup_{C_i \in VV} (Uses^*(C_i)) - VV$  for closure, namely, I show that a view VS becomes *closed* if I add the set MIN to its view classes VV. In part II, I show the necessity of MIN for closure, namely, I show that MIN is the minimal set required to make VS closed. These two facts together imply the correctness of the theorem.

**Part I:** Adding the set  $MIN = (\bigcup_{C_i \in VV} (Uses^*(C_i))) - VV$  to the view VS will make the view *closed*.

**Case I.a:** Let  $VS=(VV,VE)$  be a view that is already closed. By Definition 21,  $VV = VV \cup (\bigcup_{C_i \in VV} (Uses^*(C_i)))$ . By subtracting the set VV from both sides of the equation, I derive that  $\bigcup_{C_i \in VV} (Uses^*(C_i)) - VV = \emptyset$ . This implies  $MIN = (\bigcup_{C_i \in VV} (Uses^*(C_i))) - VV = \emptyset$ . Since the view VS is assumed to be closed, no classes need to be added to the view, i.e., adding the set  $MIN = \emptyset$  trivially makes the view closed.

**Case I.b:** Let  $VS=(VV,VE)$  be a view that is not closed. Then create a new view  $VS'=(VV',VE')$  with  $VV'$  the set of classes created by adding MIN to VV, i.e.,  $VV' := VV \cup MIN$ . Then  $VV' = VV \cup MIN = VV \cup (\bigcup_{C_i \in VV} (Uses^*(C_i))) - VV = VV \cup (\bigcup_{C_i \in VV} (Uses^*(C_i)))$ . I now need to show that VS' is closed. By Definition 21, VS' is closed if and only if  $VV' = VV' \cup (\bigcup_{C_i \in VV'} (Uses^*(C_i)))$ . I prove this as follows:

$$\begin{aligned}
 & \bigcup_{C_i \in VV'} (Uses^*(C_i)) \\
 &= \bigcup_{C_i \in (VV \cup \bigcup_{C_k \in VV} (Uses^*(C_k)))} (Uses^*(C_i)) \\
 &= \bigcup_{(C_i \in VV) \vee (C_i \in \bigcup_{C_k \in VV} (Uses^*(C_k)))} (Uses^*(C_i)) \\
 &= \bigcup_{C_i \in VV} (Uses^*(C_i)) \cup \bigcup_{C_i \in (\bigcup_{C_k \in VV} (Uses^*(C_k)))} (Uses^*(C_i)) \\
 &= \bigcup_{C_i \in VV} (Uses^*(C_i)) \\
 &\subseteq VV \cup (\bigcup_{C_i \in VV} (Uses^*(C_i))) \\
 &= VV'.
 \end{aligned}$$

Finally,  $\bigcup_{C_i \in VV'} (Uses^*(C_i)) \subseteq VV'$  implies  $VV' = VV' \cup (\bigcup_{C_i \in VV'} (Uses^*(C_i)))$ . I thus have shown that the addition of the set MIN to the non-closed view VS creates the closed view VS'. ■



**Part II:** The set  $MIN = (\cup_{C_i \in VV}(Uses^*(C_i))) - VV$  is the *minimal* set of classes that has to be added to a view VS to make it closed.

**Case II.a:** Let  $VS=(VV,VE)$  be a view that is already closed. Then by part I.a,  $MIN = \emptyset$ . The empty set is obviously the equal to the smallest possible set of classes that has to be added to make the view closed.

**Case II.b:** Part II follows directly from Definition 21 for a view  $VS=(VV,VE)$  that is not closed. By Definition 21, all classes that are in the transitive closure of the  $Uses^*$  relationship of VS,  $\cup_{C_i \in VV}(Uses^*(C_i))$ , must also be part of VS in order for VS to be closed. On the other hand, classes that are already part of VS do not have to be added again. Therefore, all classes in  $\cup_{C_i \in VV}(Uses^*(C_i)) - VV$  must be added to VV. Note that  $\cup_{C_i \in VV}(Uses^*(C_i)) - VV$  is equal to MIN. ■

### 10.2.2 Closed-View Generation: Algorithm and Examples

The Closed-View Generation algorithm (CVG) is given in Figure 10.3. CVG determines whether a given view is closed or not. If the view is not closed then the algorithm automatically determines the minimal set of classes by which the view schema VS has to be extended in order for the view to be *type-closed*. This is done by recursively exploring the  $Uses$  relationships of classes. Note that the  $uses$  relationships of a class are independent from which other class of the schema I have reached the current class. Therefore, I only need to calculate the simple form of a transitive closure of the  $uses$  relationship. This observation reduces the complexity of the algorithm considerably, namely, from cube to linear complexity. Once I process a class  $C_i$  by checking its  $Uses$  relationships, it need not be processed anymore. In order to avoid unnecessary repetitive processing, the algorithm maintains a list of all classes that do not have to be checked anymore, called CVG-done. In addition, it maintains a list of all classes reached via the  $Uses$  relationship that still have to be processed, called CVG-tmp.

The algorithm proceeds as follows. While there are any classes left to be processed in the CVG-tmp set, the algorithm picks one of them, say  $C_i$ . The processing of the class  $C_i$  entails the following. If  $C_i$  is not in the view, then the view is not closed and the flag *Closed* is set to false. The algorithm also adds  $C_i$  to the CVG-done set which serves the following two purposes: first, it assures that  $C_i$  will not be processed again, and second, it collects all classes that need to be added to the view to make it closed. Next, the algorithm checks for all classes  $C_k$  in the  $Uses(C_i)$  set, whether they have to be processed for closure. They do not have to be processed for closure, if either they already have been processed (i.e., are in CVG-done) or if they

**Data Structures and Variables:**

Set of classes: CVG-tmp, CVG-done;  
 Classes:  $C_i, C_k$ ;  
 Boolean flag: Closed;

**Procedures and Functions:**

*get-and-remove-next*(set-of-classes)  $\rightarrow$  class;  
*not-element*(class,set-of-classes)  $\rightarrow$  boolean;  
*add-to-set*(class,set-of-classes);

**Input:**

Global Schema  $GS = (V, E)$  and View Schema  $VS = (VV, VE)$

**Output:**

The flag Closed indicates whether the view is closed.  
 The set of classes CVG-done contains all missing classes required for view closure.

**Algorithm CVG:** The Closed-View Generation Algorithm.

```

algorithm CVG( $GS, VS$ ) return (CVG-done: set-of-classes, Closed: boolean-flag) is
  CVG-done :=  $\emptyset$ ;
  CVG-tmp := VV;
  Closed := true;
  while ( $C_i := \text{get-and-remove-next}(\text{CVG-tmp})$ ) do
    if (not-element( $C_i, VV$ )) then
      Closed := false;
      add-to-set( $C_i, \text{CVG-done}$ );
    endif;
    for all  $C_k$  in Uses( $C_i$ ) do
      if (not-element( $C_k, \text{CVG-done}$ )
        and not-element( $C_k, \text{CVG-tmp}$ )
        and not-element( $C_k, VV$ )) then
        add-to-set( $C_k, \text{CVG-tmp}$ );
      endif;
    endfor;
  endwhile
  return (CVG-done, Closed);
end algorithm;

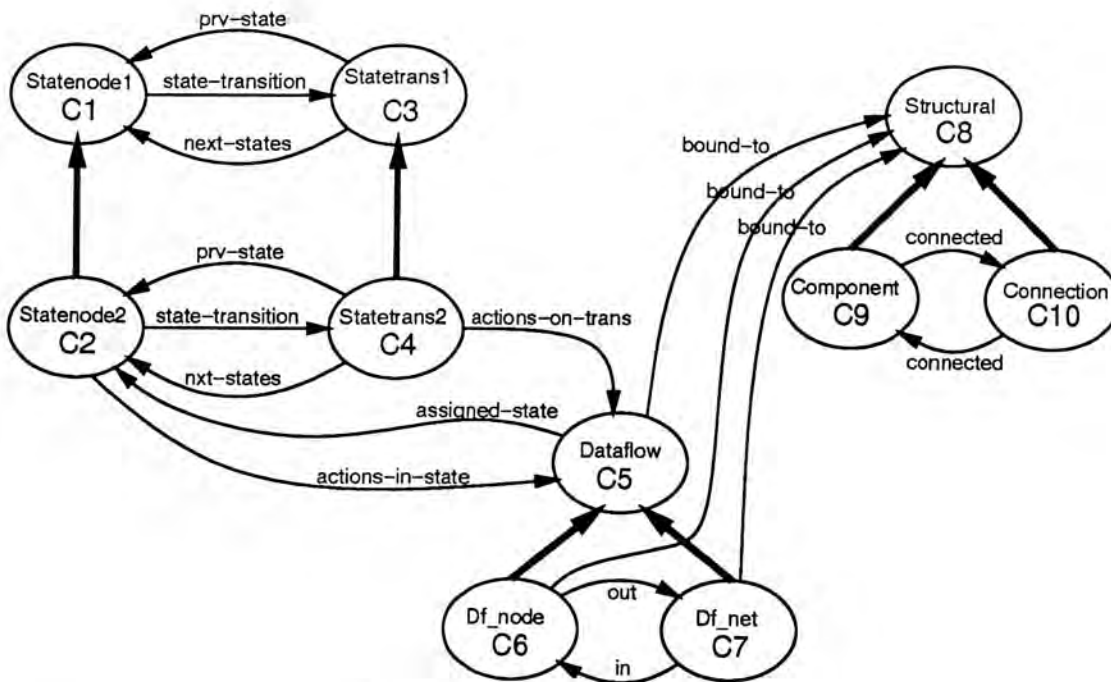
```

Figure 10.3: The Closed-View Generation Algorithm.

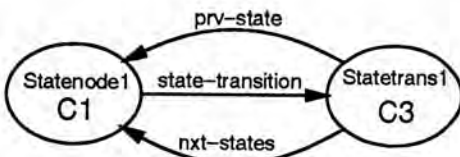
are guaranteed to be processed at some later time (i.e., are in  $VV$  or in  $CVG-tmp$ ). If they still have to be processed then they are added to  $CVG-tmp$ . The algorithm terminates when all classes reachable from the view classes of the view  $VS$  have been processed, i.e., when  $CVG-tmp$  is empty. If the view is *closed*, then the algorithm returns the flag "Closed=true" and the set "CVG-done= $\emptyset$ ". If the view is not *closed*, then the algorithm returns the flag "Closed=false" and the set "CVG-done $\neq \emptyset$ ". The latter contains all classes that have to be added to the view schema in order to make it *closed*, i.e.,  $CVG-done = MIN$  with  $MIN$  defined in Theorem 25. Below, I discuss examples of applying the CVG algorithm to the view schemata shown in Figure 10.4.

**Example 59.** *The view VS1 in Figure 10.4.b is defined on top of the global schema GS depicted in Figure 10.4.a. The CVG algorithm first initializes  $CVG-done := \emptyset$ ,  $CVG-tmp := \{C1, C3\}$ , and  $Closed := true$ . For the first iteration of the while-loop, the iteration variable  $Ci$  is equal to  $C1$ . The first if-statement evaluates to false, since  $(C1 \in VV)$ , and thus is skipped.  $Uses(C1) := \{C3\}$  due the 'state-transition' property defined for  $C1$ . Therefore, the for-loop is executed but once with the iteration variable  $Ck := C3$ . The if-statement in the body of the for-loop evaluates to false, since  $(C3 \in VV)$ . For the second iteration of the while-loop, the iteration variable  $Ci$  is set to  $C3$ . The first if-statement is again skipped.  $Uses(C3) := \{C1\}$  due to the two properties 'prv-state' and 'nxt-states' defined for  $C3$ . The for-loop is executed with  $Ck := C1$ . The if-statement in the loop body again evaluates to false, since  $(C1 \in VV)$ .  $CVG-tmp$  is now empty, and therefore the algorithm terminates. CVG returns the parameters ( $Closed=true$ ) and ( $CVG-done=\emptyset$ ). VS1 has been shown to be closed.*

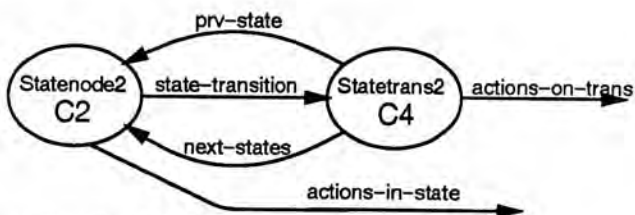
**Example 60.** *In this example, I describe how CVG is applied to the view VS2 depicted in Figure 10.4.c with VS2 defined on GS shown in Figure 10.4.a. CVG initializes the variables as follows:  $CVG-tmp := \emptyset$ ,  $CVG-tmp := \{C2, C4\}$ , and  $Closed := true$ . For the first iteration of the while-loop, the iteration variable  $Ci$  is set equal to  $C2$ . The first if-statement evaluates to false and is skipped. Since  $Uses(C2) := \{C4, C5\}$ , the for-loop has two iterations. For the iteration with  $Ck := C4$ , the if-statement is skipped. For the second iteration with  $Ck := C5$ , the if-statement evaluates to true and  $C5$  is added to  $CVG-tmp$  for further processing. For the second iteration of the while-loop with the iteration variable  $Ci:=C4$ , the first if-statement is again skipped. The for-loop has two iterations since  $Uses(C4) := \{C2, C5\}$ . For both iterations, the if-statement is skipped. For the third iteration of the while-loop with the iteration variable  $Ci:=C5$ , the first if-statement evaluates to true since  $C5 \notin VV$ . Therefore,  $C5$  is added to  $CVG-done$  and the flag  $Closed$  is set to false. Since  $Uses(C5) := \{C2, C8\}$ , the for-loop has two iterations. For the second iteration of the for-loop with  $Ck:=C8$ , the if-statement evaluates to true and  $C8$  is added to*



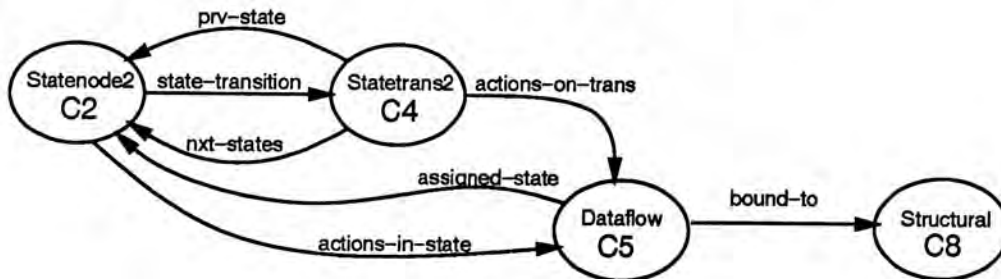
a. The global schema GS.



b. The view VS1 is closed.



c. The view VS2 is not closed.



d. The view VS2' is closed (derived from VS2 by the CVG algorithm).

Figure 10.4: Examples of the Closed-View Generation Algorithm.

*CVG-tmp*. For the fourth and last iteration of the while-loop with the iteration variable  $C_i := C_8$ , the first if-statement evaluates to true and  $C_8$  is added to *CVG-done*. Since  $Uses(C_8) := \{\}$ , the for-loop is not executed. *CVG-tmp* is now empty and the CVG algorithm terminates with ( $Closed=false$ ) and ( $CVG-done=\{C_5, C_8\}$ ). CVG thus has shown that the view  $VS_2$  is not closed. In addition, the algorithm has determined the set of classes that have to be added make a complete view out of  $VS_2$ , namely, the set *CVG-done*. The resulting augmented and thus closed view  $VS_2'$  is shown in Figure 10.4.d.

### 10.2.3 Correctness and Complexity of the Algorithm

**Theorem 26. (Correctness)** Given a view schema  $VS=(VV, VE)$  defined on the global schema  $GS=(V, E)$ , then the closed-view generation algorithm CVG in Figure 10.3 correctly generates a closed view  $VS'$ . In particular, CVG returns the value  $Closed=true$  if the view schema  $VS$  is closed, and the value  $Closed=false$ , otherwise. If the view  $VS$  is not closed, then CVG also generates the minimal set of classes that have to be added to  $VS$  to make it closed, namely,  $CVG-done = (\bigcup_{C_i \in VV}(Uses^*(C_i))) - VV$ .

**Proof:** I prove Theorem 26 in two parts. In the first part, I show that the algorithm returns the correct value for the *Closed* flag. In the second part, I show that the *CVG-done* set returned by the algorithm is equal to  $(\bigcup_{C_i \in VV}(Uses^*(C_i))) - VV$ .

**Part I:** ( $Closed=true$ )  $\iff$  ( $VS$  is closed).

In part I.a, I show " $(Closed=true) \implies (VS \text{ is closed})$ ". In part I.b, I show " $(VS \text{ is closed}) \implies (Closed=true)$ ". These together imply the desired equivalence of part I.

**Part I.a:** ( $Closed=true$ )  $\implies$  ( $VS$  is closed).

Assume that the algorithm CVG returned the flag  $Closed=true$ . This means that the first if-statement with the condition  $(C_i \notin VV)$  never evaluated to true. This implies that CVG did *not* find a single class  $C_i$  in the transitive closure of the *uses* relationship of  $VV$  (or  $C_i \in VV \cup \bigcup_{C_i \in VV}(Uses^*(C_i))$ ) for which the following holds:  $C_i \notin VV$ . I can therefore conclude that for all classes  $C_i$ ,  $C_i \in VV \cup \bigcup_{C_i \in VV}(Uses^*(C_i))$  also implies  $C_i \in VV$ . Hence,  $(\bigcup_{C_i \in VV}(Uses^*(C_i))) \cup VV \subseteq VV$ . This implies the relationship  $VV = (\bigcup_{C_i \in VV}(Uses^*(C_i))) \cup VV$ . By Definition 21, the view  $VS$  is closed. ■

**Part I.b:** (VS is closed)  $\implies$  (Closed=true).

Assume that the view VS is closed. Then by Definition 21, the relationship  $\bigcup_{C_i \in VV} (Uses^*(C_i)) \subseteq VV$  holds. This means that there is *not* a single class  $C_i$  in the transitive closure of the *uses* relationship of VV that is not also in VV, i.e.,  $(\nexists C_i \text{ in } V) ((C_i \notin VV) \wedge (C_i \in \bigcup_{C_i \in VV} (Uses^*(C_i))))$ . Therefore, for all classes  $C_i$  processed by the algorithm CVG the condition " $C_i \notin VV$ " of the first if-statement will always evaluate to false. Since the body of this first if-statement is never executed, the variable *Closed* is never modified and the initial value *Closed=true* remains. ■

**Part II:** The algorithm generates the set  $CVG\text{-done} = (\bigcup_{C_i \in VV} (Uses^*(C_i))) - VV$ .

The while-loop recursively traverses the transitive closure of the *uses* relationship of all view classes of VS. Initially it starts with all classes in VS, since CVG-tmp is initialized to VV. Later the for-loop recursively adds all classes that can be reached from a class in CVG-tmp via the *uses* relationship. Hence, over the duration of the CVG execution, the while-loop will process all classes in  $(VV \cup (\bigcup_{C_i \in VV} (Uses^*(C_i))))$  at least once. The algorithm adds all classes in the above set to the CVG-done set for which the condition " $C_i \notin VV$ " of the first if-statement evaluates to true. Therefore, CVG-done will be equal to the set  $(\bigcup_{C_i \in VV} (Uses^*(C_i))) - VV$ . By Theorem 25, the set CVG-done thus corresponds to the minimal set of classes that has to be added to the non-closed view VS to create the closed view VS'. ■

**Theorem 27. (Complexity)** *Given a view schema  $VS=(VV,VE)$  defined on the global schema  $GS=(V,E)$  with  $PGS=(V,A,L)$  the matching property decomposition hierarchy of GS as defined in Definition 10. The complexity of the closed-view generation algorithm CVG for the view VS is equal to  $O(|A|)$  with  $|A|$  the number of property decomposition arcs in PGS.*

**Proof:** I prove Theorem 27 in three parts.

**Part I:** First, I show that all functions used by CVG have constant complexity.

I assume that each class in *GS* has a unique index in the range from 1 to  $|GS|$ . Then I can implement the set-of-classes data structure by a bit-vector of length  $|GS|$  as follows: the *i*-th bit is true if the class  $C_i$  is in the set, and false, otherwise. To check whether a class  $C_i$  is or is not an element in a set using the *not-element()* function is done by checking the value of the corresponding bit. This takes constant time. To add or to remove an element using the functions *add-to-set()* and *get-and-remove-next()*, respectively, corresponds to flipping a bit. Both functions thus

take constant time. CVG-done and VV are assumed to be implemented using this scheme.

Assume the variable CVG-tmp is implemented by both a linked list and a bit-vector representation. The linked list representation is used to get the next element in the list in constant time by the *get-and-remove-next()* function, while the bit-vector representation is used by the *not-element()* function to assure that the element is not already present in the list in constant time. The *add-to-set(Ck, CVG-tmp)* function flips the corresponding bit for Ck in the vector representation to true and it also appends Ck to the matching linked list representation of CVG-tmp. Similarly, the *get-and-remove-next()* function first removes the next element  $C_i$  from the linked list representation of CVG-tmp and then updates the CVG-tmp's bit-vector representation of  $C_i$  by setting the appropriate bit to false. These functions can each be done in constant time. ■

**Part II:** Next, I show that each class of GS will be placed at most once into the CVG-tmp set.

When a class  $C_i$  is processed using the while-loop, it is first removed from CVG-tmp using the function *get-and-remove-next()*. It is then placed into CVG-done using the first if-statement (assuming that it is not an element of VV). Once a class  $C_i$  is in the CVG-done set or in VV,  $C_i$  will never be placed back into CVG-tmp for the following reason. The second if-statement is the only statement that adds elements to the CVG-tmp set, and the condition of this if-statement, which is "*(not-element(Ck, CVG-done))*" and *not-element(Ck, VV)* and ... ", assures that a class  $C_i$  with the above characteristics is not placed back into CVG-tmp. ■

**Part III:** Lastly, an analysis of the overall algorithm is conducted using part I and II.

- By part I shown above, all functions used by the CVG algorithm have constant complexity. Therefore, the two if-statements can be executed in constant time each.
- By part II shown above, I know that each node of the global schema GS is placed at most once in the CVG-tmp set. This implies that the while-loop is executed at most once for each node in GS, i.e., there are in the worst case  $|GS|$  iterations.
- For each class  $C_i$  in the CVG-tmp list (i.e., for each iteration of the while-loop), the for-loop has exactly one iteration for each class Ck in the *Uses(Ci)* set.

$|Uses(C_i)|$ , which denotes the number of distinct classes with which the class  $C_i$  in  $GS$  maintains direct property decomposition relationships, is smaller than or equal to the number of outgoing property decomposition arcs for the class  $C_i$ , denoted by  $\#arcs(C_i)$ . This is true since there may be two properties arcs labeled by the property names  $p_1$  and  $p_2$  defined for  $C_i$  with the same domain class  $C_j$ , e.g.,  $a_1 = \langle C_i, C_j, p_1 \rangle$  and  $a_2 = \langle C_i, C_j, p_2 \rangle$ . And, for each class  $C_j \in |Uses(C_i)|$ , there must be at least one arc  $a_k$  with the domain class  $C_j$ , i.e.,  $a_k = \langle C_i, C_j, p_k \rangle$  for some label  $p_k$ .

The overall complexity of the closed-view generation algorithm CVG therefore is  $complexity(CVG) \leq O(\sum_{C_i \in V} (|Uses(C_i)|)) \leq O(\sum_{C_i \in V} (\#arcs(C_i))) = O(|A|)$ . ■

**Theorem 28. (Complexity)** *Given a view schema  $VS=(VV, VE)$  defined on the global schema  $GS=(V, E)$  with  $PVS=(VV, VA, VL)$  the matching property decomposition hierarchy of  $VS$  as defined in Definition 10. If the view schema  $VS$  is closed, then the closed-view generation algorithm CVG has the complexity of  $O(|VA|)$  with  $|VA|$  the number of property decomposition arcs in  $PVS$ .*

**Proof:** Assume that the view schema  $VS$  is closed. By Definition 21, a closed view defines all classes that it uses. Therefore, the test of the second if-statement “*not-element(C, VV)*” will always evaluate to false. Therefore, no classes are added to the CVG-tmp set besides the initial set  $VV$ , and the while-loop has exactly  $|VV|$  iterations. Similarly, by Definition 21, the  $Uses$  set of a class in  $VS$  consists only of classes that are defined in  $VS$ . Therefore, the size of a  $Uses$  set for a view class of a closed view is equal to all outgoing arcs of the class; with these arcs being contained in the property decomposition hierarchy of  $VS$ . The for-loop for a class  $C_i$  has at most  $\#arcs(C_i)$  iterations. I thus have  $complexity(CVG) \leq O(\sum_{C_i \in VV} (\#arcs(C_i))) = O(|VA|)$ . ■

Since the chosen set representation used in the algorithm is a bit-vector of length  $|GS|$ , the initialization of these vectors has a complexity of  $|GS|$ . Therefore, the complexity of the CVG algorithm including initialization would be  $complexity(CVG) = O(\min(|GS|, |VA|))$ .



# Chapter 11

## The View Independence of MultiView

### 11.1 The View Independence Concept

The concept of *data independence* developed for the relational model is defined as the “immunity of applications to change in storage structure and access technique” [Date90]. This is achieved by separating the interface to the database (the conceptual data model) from the actual implementation (the physical data model). *Physical data independence* addresses the independence of users and user programs from the physical structure of the stored data, while *logical data independence* addresses the independence of users and user programs from the logical structure of the data. A system provides *physical data independence* by supporting an implementation-independent interface (a logical data schema) that the users can utilize in place of operating directly on the underlying storage structure and access paths. A system provides *logical data independence* by supporting a view definition mechanism that lets the users define their own view schema on top of the common logical schema. The concept of *data independence* thus addresses the immunity of users from changes of the underlying data model. It does, however, not protect the user from having to update the specification of possibly all existing views when the underlying data model is extended and/or reorganized.

The *MultiView* methodology is based on the actual restructuring of the global schema for generating new view schemata. Therefore, I introduce the concept of *view independence* to be the immunity of view schema definition and semantics to changes of the underlying global schema.

**Definition 28.** *A database system provides view independence if the specification and the semantics of existing view schemata are not affected by the definition of new view schemata.*

This concept of *view independence* is a necessary and important requirement for object-oriented database systems, since the underlying global schema is restructured with the definition of possibly each new view schema. A redefinition of all existing view schemata for whenever a new view schema is introduced would be an unacceptable overhead. *View independence* does not have any significance in relational databases where the definition of new views has no affect on the underlying base schema. In fact, the relational model is by default *view independent*.

For the *MultiView* methodology to be view independent would mean that the integration of new virtual classes into the global schema does not require the redefinition of the existing view schemata nor does it modify their semantics. The latter means that in spite of the restructuring of the global schema the following must hold: (1) the view classes of a view defined on the global schema do not change their type description nor their set membership and (2) the *is-a* relationships between the view classes of a given view are preserved. A more precise definition of the view independence concept for *MultiView* is given below.

**Definition 29.** Let  $G^*$  be the set of all schemata,  $C$  the set of all classes,  $O$  the set of all object instances, and  $P$  the set of all properties. Let  $GS=(V,E)$  be a global schema and  $VS=(VV,VE)$  a view schema defined on  $GS$ . Let  $VS^*$  be the set of all view schemata defined on  $GS$ . Let  $\Pi: G^* \rightarrow G^*$  be a function that applies a class derivation operator to  $GS$  and then restructures  $GS$  by integrating the resulting virtual class into  $GS^s$ . Let  $GS' = (V',E')$  be the global schema and  $VS'=(VV',VE')$  the view schema derived from  $VS$  after the integration of virtual classes into  $GS$  using the function  $\Pi$ , i.e.,  $GS' = \Pi(GS)$  and  $VS' = \Pi(VS)$ .

(a) The view classes  $VV$  of  $VS$  are **preserved** through the application of the function  $\Pi$  to  $GS$  iff the following holds:

- $\exists$  a one-to-one mapping  $m: C \rightarrow C$ , such that  $(\forall C_i \in C)((C_i \in VV) \implies (\exists !C_i' \in VV')(C_i' = m(C_i)))$ , and vice versa,  $(\forall C_i' \in C)((C_i' \in VV') \implies (\exists !C_i \in VV)(C_i = m^{-1}(C_i'))^2$ .
- $(\forall C_i \in VV) (\forall o \in O) ((o \in C_i) \text{ in } VV \iff (o \in m(C_i)) \text{ in } VV')$ .
- $(\forall p \in P)(\forall C_i \in VV) ((p \in \text{properties}(C_i) \text{ in } VS) \iff (p \in \text{properties}(m(C_i)) \text{ in } VS'))$ .

<sup>1</sup>For this report, I assume that the function  $\Pi$  corresponds to the object algebra operators and the integration algorithm presented earlier in this dissertation. Without loss of generality, other operators or integration algorithms could be substituted.

<sup>2</sup>This one-to-one mapping  $m$  is simply the equality operator on the class identifiers, since each class has a unique class identifier and  $VV \subseteq V$ .

(b) The view is-a relationships  $VE$  among the view classes  $VV$  are defined to be **preserved** through the application of the function  $\Pi$  to  $GS$  iff the following holds:

- $(\forall C_i, C_j \in VV) ((C_i \text{ is-a } * C_j) \in VE) \iff ((m(C_i) \text{ is-a } * m(C_j)) \in VE')$ .

with the mapping  $m$  as defined in (a).

(c) The view  $VS$  is **preserved** through the restructuring of  $GS$  using the function  $\Pi$  iff the type description and set membership of all classes in  $VV$  are **preserved** as defined in (a) and the view is-a relationships  $VE$  are **preserved** as defined in (b).

(d) *MultiView* is **view independent** if all view schemata in  $VS^*$  are **preserved** as defined in (c).

## 11.2 Proving *MultiView* View Independent

Below, I prove the *view independence* property of *MultiView* in two steps. First, I show that view classes (Definition 29.a) and then that the *is-a* relationships among view classes (Definition 29.b) are not affected by the restructuring of the global schema<sup>3</sup>.

### 11.2.1 Preservation of View Classes

As specified in Definition 29.a, a view class needs to preserve both its type description and its set membership through possible restructuring of the underlying global schema in order for *MultiView* to be view independent. There are two design choices for determining the type description of a view class:

- First, I can determine the type of a view class based on the type descriptions of classes visible in the view schema.

<sup>3</sup>Note that I define the **type** of a class to be the union of its defined and inherited property functions. Turning a defined property into an inherited property (as done for instance in Figure 11.1.e for property a1) is not considered to be a change of the class type. I define the set membership of a class, denoted by  $extent(C) = \{o \mid o \in C\}$ , to be the union of its direct and indirect instances; and it is this combined membership of direct and indirect members that I require to stay invariant with view creation. The *direct* membership content of a class  $C$ , defined by  $direct-extent(C) = extent(C) - \bigcup_{i=1}^k extent(C_i)$  with  $C_i$  (with  $i = 1, \dots, k$ ) the direct subclasses of  $C$ , or the *indirect* membership of  $C$ , defined by  $indirect-extent(C) = extent(C) - direct-extent(C)$ , may of course be modified by the creation of a new view. For instance, the creation of additional subclasses of a class  $C$  may diminish  $C$ 's direct membership and increase  $C$ 's indirect membership.

- Second, I can determine the type of a view class based on the complete underlying global schema, which may be partially invisible in the view.

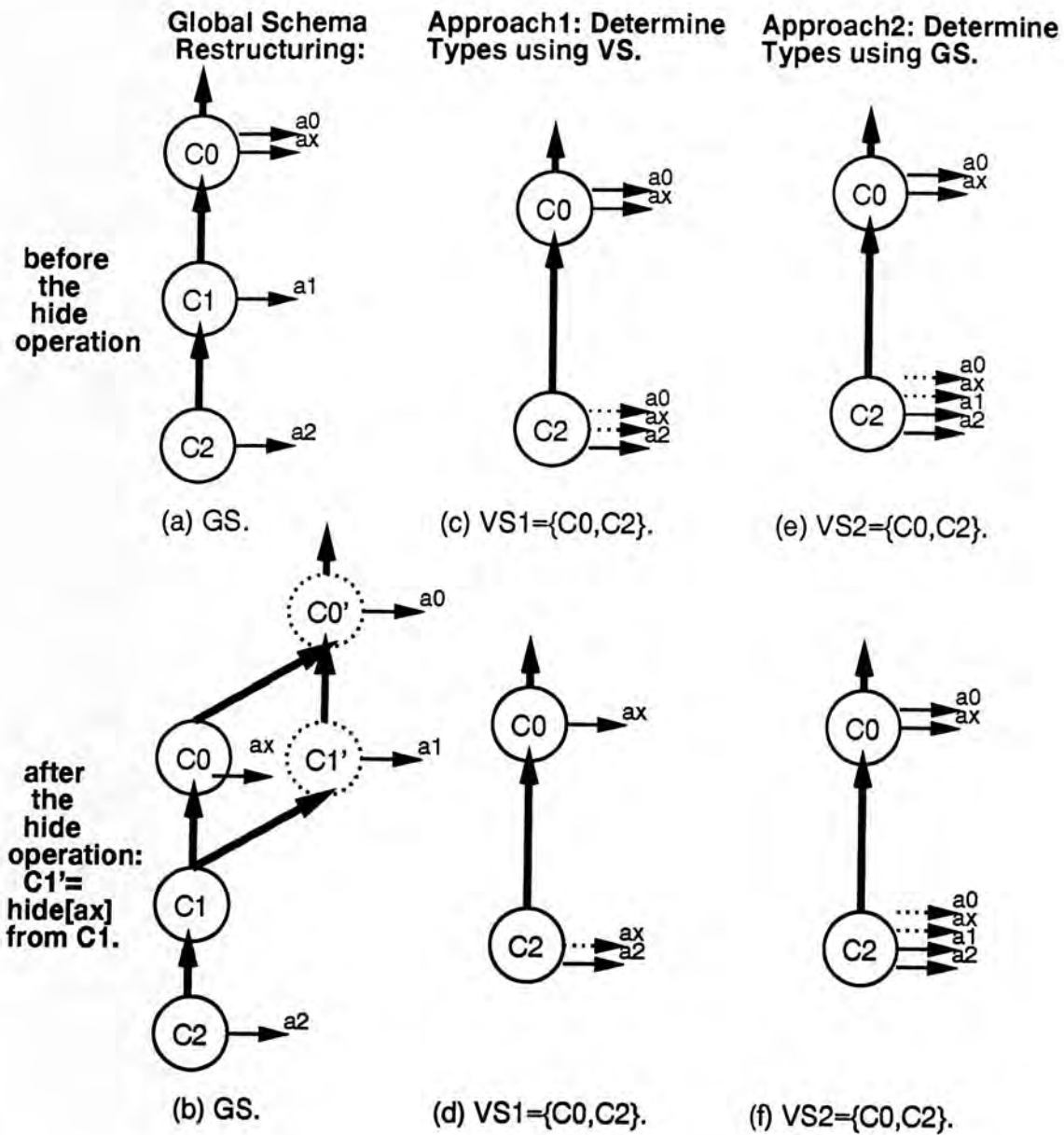


Figure 11.1: Two Approaches for Type Determination of a View Class.

The former offers the advantage that a property is only visible in a view schema, if the class that defines this property is also visible. This would thus guarantee a unique location (class) in the view hierarchy for the definition of any property

that is visible in the view schema. Unfortunately, this approach violates the view independence property as I will show below.

**Example 61.** *In this example I demonstrate the two approaches for type determination of a view class based on Figure 11.1. In this figure I use the following graphical convention: For a given class, I depict attributes that are directly defined for that class (for that view schema) by an arrow. Attributes that are inherited (for that view schema) are depicted by a dotted arrow.*

*Figures 11.1.a and 11.1.b show the global schema before and after the derivation and integration of the two virtual classes  $C0'$  and  $C1'$ . Figures 11.1.c and 11.1.d demonstrate the type determination of a view class using approach 1. Approach 1 determines types based on the classes visible in the view schema. For instance, attribute  $a1$  is defined in class  $C1$  in the global schema, and since class  $C1$  is not visible in the view schema  $VS1$ , the attribute  $a1$  is also not visible in  $VS1$ . Therefore, the class  $C2$  in Figure 11.1.c does not have the attribute  $a1$  defined. As shown in Figure 11.1.d, both classes  $C0$  and  $C2$  change their types due to the restructuring of the base schema. Class  $C0$ , for instance, has the type  $\text{properties}(C0) = \{ a0, ax \}$  before and the type  $\text{properties}(C0) = \{ a0 \}$  after the global schema restructuring. Consequently, approach 1 does not guarantee view independence.*

*Figures 11.1.e and 11.1.f demonstrate the second approach that determines types based on the class hierarchy in the global schema. Hence, the class  $C2$  in Figure 11.1.e has defined the attribute  $a1$ , even though in the global schema attribute  $a1$  is defined in a class that is not visible in the view schema  $VS2$ . Note that methods inherited from classes invisible in the view schema are displayed as defined methods for the first class that inherits them. For example, attribute  $a1$  is displayed as being defined rather than inherited for class  $C2$  in  $VS2$ . Figure 11.1.f shows that the view schema is not affected by the restructuring of the global schema, i.e., all classes maintain their original types. Approach 2 thus guarantees view independence for this example.*

**Theorem 29.** *Approach 1 for type determination of view classes does not preserve the view independence property.*

**Proof (by Counterexample):** Theorem 29 can be shown by giving one example for when the view independence property is indeed violated by approach 1. Example 61, in particular, Figures 11.1.c and 11.1.d, are such a counterexample. Namely, the integration of the virtual class  $C1'$  into GS affected the type description of the existing view classes in the view  $VS1$ . ■

I have thus shown that approach 1 may lead to type description changes of view classes when *is-a* restructuring the global schema for a new view. This violates the view independence property and thus is clearly unacceptable. This is one reason for why I have adopted the second approach in *MultiView*. Approach 2, namely, the determination of the type description of view classes based on the global schema rather than on the view schema, poses the obvious constraint on the global schema to maintain the type description of all classes during schema restructuring. I have thus reduced the problem of type preservation from the view schemata to the global schema.

**Theorem 30.** *Let  $VS^*$  be the set of all view schemata defined on GS. MultiView preserves the view classes of all view schemata in  $VS^*$  through the restructuring of GS using the function  $\Pi$  (Definition 29.a)<sup>4</sup>*

**Proof: (Intuitive)** A more detailed proof for Theorem 30 can be found in [Rund92d], while below I give the intuitive reasoning underlying the proof. As explained earlier, mapping  $m$  given in Definition 29 corresponds to the equality operator on the class identifiers. Class identifiers are unique and the set of view classes is always a subset of the set of global classes. Hence, the mapping  $m$  is a one-to-one function (Part I of Definition 29.a).

The class derivation of a virtual class creates a new class with possibly a new type description and a new content; it does obviously not modify the semantics of existing classes. Hence, I am only concerned with the integration part and not the class derivation part of the function  $\Pi$ . As discussed above, *MultiView* determines the type description and the set membership of a view class directly from the global schema. Therefore, I can reduce the problem of view class preservation from the view to the global schema. I thus need to show that all classes  $C_i$  of GS are preserved when integrating new classes into GS.

Recall that the integration algorithm of virtual classes (even if fine-tuned for particular query operators) follows the principle that the virtual class VC is inserted into GS by placing it directly below its direct superclasses, called  $\text{direct-parents}(VC)$ , and directly above its direct subclasses, called  $\text{direct-children}(VC)$ , in GS with  $(\forall C_i \in \text{direct-parents}(VC)) (VC \text{ is-a } C_i)$  and  $(\forall C_j \in \text{direct-children}(VC)) (C_j \text{ is-a } VC)$  (Chapter 7). Due to (1) VC being *is-a* related to both sets of classes and (2) the transitivity of the *is-a* relationship, I can deduce that classes in these sets were

<sup>4</sup>I define the **type** of a class to be the union of its defined and its inherited property functions. Turning a defined property into an inherited property is not considered to be a change of the class type. Similarly, I define the set membership of a class, denoted by  $\text{content}(C) = \{o \mid o \in C\}$ , to be the union of its direct and indirect instances.

*is-a* related to one another before the insertion of VC. More precisely,  $(\forall C_i \in \text{direct-parents}(\text{VC})) (\forall C_j \in \text{direct-children}(\text{VC})) (C_j \text{ is-a } * C_i)$ . Clearly, the insertion of VC does not modify the content of existing classes, i.e., part II of Definition 29.a holds. The insertion of VC also does not modify their type descriptions. All classes that are made subclasses of VC in the modified GS are also subtypes of VC; i.e., they will not inherit any new property functions and their types will be preserved. This shows part III of Definition 29.a. ■

### 11.2.2 Preservation of View *is-a* Relationships

**Theorem 31.** *Let GS be a global schema and VS\* be the set of all view schemata defined on GS. The MultiView methodology preserves the view is-a relationships among the view classes of each view in VS\* through the restructuring of GS using the function  $\Pi$  with the term preserves as defined in Definition 29.b.*

**Proof: (Intuitive)** As specified in Definition 19, *MultiView* derives the *is-a* relationships of view classes directly from their *is-a* relationships in GS, i.e.,  $(\forall C_i, C_j \in \text{VV}) ((C_i \text{ is-a } C_j \in \text{GS}) \iff (C_i \text{ is-a } C_j \in \text{VS}))$ . Consequently, if I can show that the relative *is-a* relationships are maintained for all pairs of classes in GS, then I have also shown that they are maintained for all pairs of classes in VS.

Recall that the integration algorithm of virtual classes (even if fine-tuned for the particular query operators) follows the general approach explained in Chapter 7. As shown in Theorem 30, I can thus deduce that the classes in the  $\text{direct-parents}(\text{VC})$  and the  $\text{direct-children}(\text{VC})$  set were *is-a* related before the insertion of VC. Namely,  $(\forall C_i \in \text{direct-parents}(\text{VC})) (\forall C_j \in \text{direct-children}(\text{VC})) (C_j \text{ is-a } * C_i)$ . Therefore, the insertion of VC does not add any new *is-a* relationships. Obviously, it does not remove any either. I have thus shown the preservation of *is-a* relationships in GS. ■

**Theorem 32.** *The MultiView methodology is view independent.*

**Proof:** Theorems 30 and 31 show respectively that the *MultiView* approach preserves the view classes and the view *is-a* relationships of all view schemata defined on a global schema GS through the restructuring of GS. By Definition 29 these two theorems together prove the view independence of *MultiView*. ■

## Chapter 12

# Database Support for Behavioral Synthesis: A Potpourri.

### 12.1 An Introduction to Database Support for Behavioral Synthesis

While being interested in database support for design environments in general, I'll now focus my attention to behavioral synthesis in order to apply the ideas developed in this dissertation to a concrete application example. Behavioral synthesis automates the process of mapping a behavioral specification specified in a hardware description language, like VHDL, to a structural description representing a set of interconnected components, a netlist, and finally down to layout [Gajs92]. In recent years, a great number of behavioral design tools of ever increasing sophistication have become available that automate the more difficult and time consuming parts of behavioral synthesis. I am interested in incorporating these tools into an integrated design environment, such that the full potential of these tools can be exploited. To this end, a design data management system is required that manages the diverse design objects and their complex interrelationships. In this section, I'll introduce behavioral synthesis and discuss some of the basic features of such a database system for behavioral synthesis, while in the next section I'll present *design views* for behavioral synthesis using the database technology developed in this dissertation.

As discussed in Section 1, general-purpose database management systems have been shown to be too slow and inefficient for Computer-Aided Design Applications [Kent79, Hamm81, Borg87, Afsa89]. Furthermore their data models are not expressive enough to capture the complex nature of CAD data. In addition, existing work on databases for automated design applications has been limited to lower levels of the design process, like, for instance, the structural and layout levels [Harr86, Gupt89, Chen90, Foo90].



For these reasons, I want to develop a design database for support of design processes at the behavioral synthesis level, called the Behavioral Design Database (BDDB). For the integration of all design information into one unified representation I have defined a design data model, called the *Behavioral Design Object Model* (BDOM). BDOM is composed of one central conceptual design entity graph model and numerous design representation graph models. The former corresponds to the *meta design information* that relates various chunks of design data according to semantic relationships, such as equivalent, derivation, and hierarchy. The latter are complete design representation models that represent in a unified manner all *design data* produced and consumed by different behavioral synthesis tools [Rund90a]. Therefore, database support at the behavioral synthesis level is much needed and overdue.

BDDB must support a spectrum of *design tool interactions* ranging from tightly to loosely integrated tools. Tightly integrated tools are implemented directly on top of BDOM using design views (for further details on design views for behavioral synthesis see the next chapter). Loosely integrated tools generally maintain their own local data structures for one of the following reasons: (1) they are existing design tools that are too expensive to rewrite or (2) they require special-purpose data structures for the implementation of their algorithms. For *exchanging structured design data* with these design tools, BDDB utilizes a textual specification language, called the Behavioral Design Data Exchange Format (BDEF), which is a common format for describing design data objects and their relationships.

BDDB supports design exploration by supplying the designers with a collection of *design transformations* on BDOM. These design transformations restructure the design representation in order to target the given design specification to a particular design implementation style.

In this chapter, I first discuss related work on database support for behavioral synthesis (Section 12.2). In Section 12.3, I'll then describe the behavioral design object model. The behavioral design data exchange format, BDEF, is presented in Section 12.4. Design exploration support using design transformations is outlined in Section 12.5.1. I will conclude this section with presenting typical design tasks in behavioral synthesis in Section 12.6.

## 12.2 Related Research in Object Management for Behavioral Synthesis

Related work in the CAD community focuses primarily on *design representation modeling*, i.e., developing a good internal representation. Examples are the value trace VT [McFa78, Blac88], the DSL Internal Form [Camp88], the control/data flow graph CDFG [Orai86], and the design data structure DDS [Knap85]. These models generally do not provide adequate support for signal typing, for design hierarchy and concurrency, for modeling timing constraints, and for representing state assignments. Furthermore, this research on design modeling does not discuss the integration of the proposed design representation into a database system. The issue of how the proposed models serve the needs of all design tools is generally not discussed.

Camposano, for instance, has proposed a design representation for high-level synthesis [Camp88] called DSL, which identifies clearly the need to integrate behavioral and structural information in one model. DSL uses only one flat structure for the behavioral model and provides neither levels of abstraction nor a hierarchical decomposition of the representation.

Knapp and Parker [Knap85] have proposed a design data structure (DDS) that consists of three separate models for behavior, timing, and structure. Complex binding relationships have been designed between these three separate graphs to facilitate the equivalence checking between information distributed among these three models. Unfortunately, the behavioral model is one flat data flow graph, as is the case with most current approaches [McFa78, Kowa85], instead of distinguishing between control flow and data flow layers, as is done in CDFG [Orai86].

A design representation, called CDFG [Lis89], has been proposed for the VHDL Synthesis System [Lis88]. It is an extension of the flow graph representation presented in [Orai86] in as much as it concentrates on capturing the programming language constructs typical for VHDL. The design representation model presented [Rund90b] is an extension of both as it covers constructs, such as state sequencing, timing constraints, etc.

The design automation community has also invested some effort into database support for CAD. However, much of this work is using the existing database technology, i.e., they are modifying their application in order to mold it into the existing data base models.

The Intelligent Component Database (ICDB) [Chen90] is one such example system. ICDB manages and generates low-level components as needed during the design process for the VHDL Synthesis System [Lis88]. The Intelligent Component Database (ICDB) utilizes relational database technology.

Katz [Katz85] has examined archival database system for CAD. His work also focuses on the design entity level, i.e., the construction of a generic version manager [Katz90] and of application-specific functions at the design entity level [Chiu92, Chiu92].

The OCT data manager is the foundation for the integration of many CAD tools developed at Berkeley [Harr86]. OCT is concerned with the management of low-level structural circuit information only. OCT provides for three views (alternative representations) of a cell: (1) a schematic view that shows the subcells as well as their interconnections to form the cell, (2) a symbolic view that provides additional information of rough relative placement and subcell sizes, and (3) a physical view that defines the implementation of the cell fully in terms of exact placement and specific geometry.

Cbase is an object-oriented VLSI CAD database developed at the University of Southern California [Gupt89]. Cbase models the structural (topological) descriptions of circuit elements as possibly recursive compositions of cells, ports, links, and buses. It addresses the structural, but not the behavioral domain.

EVE [Afsa89] is a data management system with support for high-level synthesis that is implemented using relational database technology. All design objects in EVE have to be translated into entries spread over several tables. They found that the relational interface, i.e., SQL, is too slow and complicated, and therefore have developed their own database interface that supports primitive operations on objects, such as delete, create and modify. EVE [Afsa89] is based on the unified design representation (DDS). EVE does not support versions, design history and other such conceptual design information. EVE is an archival database only, that is, it does not deliver working models of designs to the design tools and therefore does not actually support the design process itself.

FRED [Wolf86] is a modeling system for VLSI models constructed based on object-oriented programming techniques, in particular, inheritance. FRED provides functions for calculating attributes of modules, such as latency, delay, and area.

There are initial efforts of CAD frameworks for synthesis and of related enabling technology reported in the literature, e.g., VOV [Caso90], NELISIS [Bing90],

the data model for PLAYOUT [Siep89], and others. VOV [Caso90] describes a simple design flow manager that keeps traces of the tool invocation sequences. VOV is not concerned however with the format of the actual design data, with the translation between the different data representations, nor with design management issues in general. The NELSI CAD framework [Bing90] addresses design management related issues. The focus of this work is on the conceptual schema (what we call the Design Entity Graph) and on a graphical interface based on this schema. No attention is given to the representation of the actual design data and to the transformation of the design data such that it meets the needs of particular application tools. The data model for the PLAYOUT system is described in [Siep89]. The presented work again only addresses the conceptual graph model.

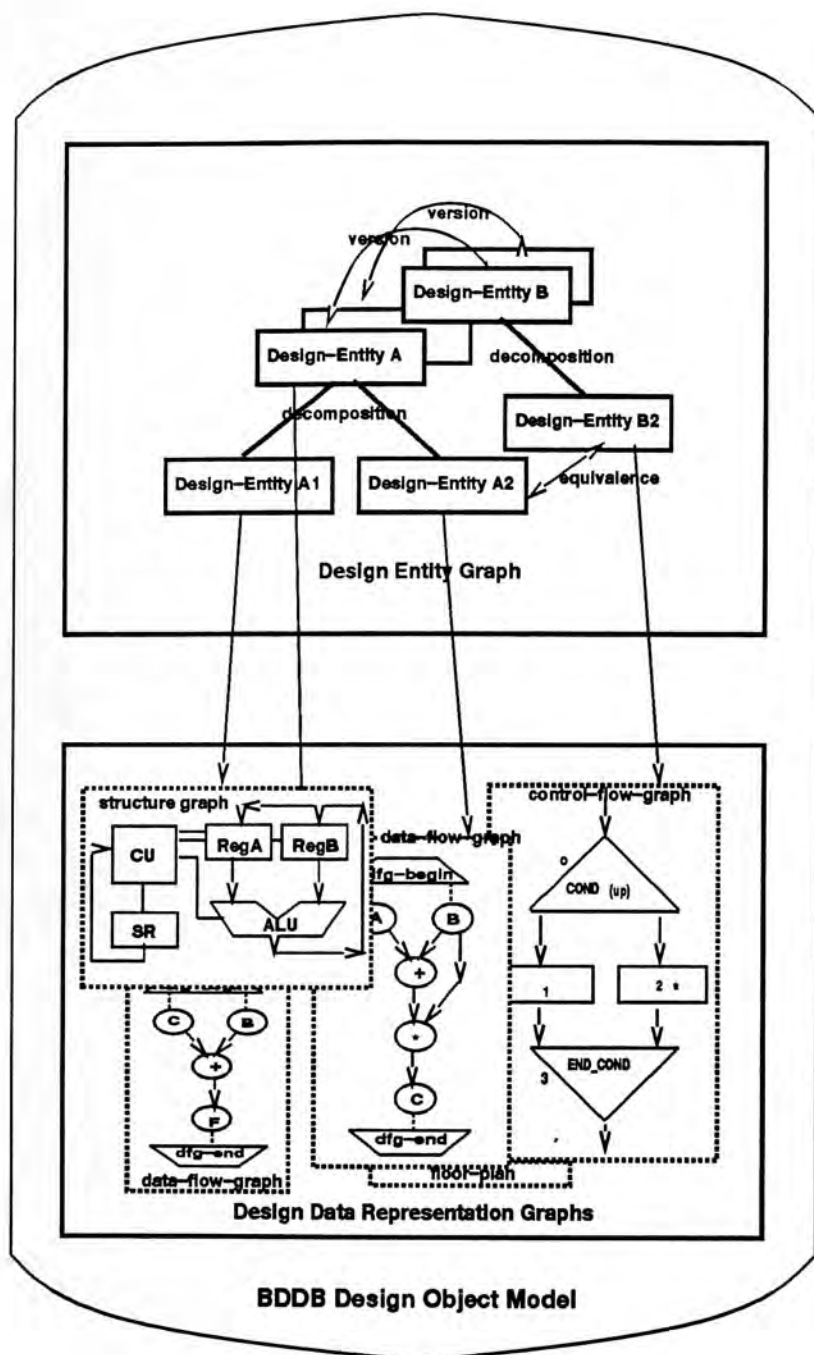
Foo and Takefuji survey existing approaches towards design management in [Foo90]. They concentrate on their frame-based data model that addresses the structural and layout information domain. It does not cope with any advanced design data management issues, like, for instance, version control.

There have been a number of attempts to apply database technologies to CAD, but to our knowledge they are largely untested for behavioral synthesis. Most research on design databases have focused on one selected subissue, such as version management [Katz82, Katz90], rather than more global system aspects. Furthermore, existing dedicated design databases deal with the lower levels of the design [Chen90], [Harr86], and [Gupt89]. We, on the other hand, want to develop a design database system which supports all important requirements of design processes. In addition, our objective is to develop a design database targeted toward the higher level of the design, in particular, behavioral synthesis.

## **12.3 A Unified Behavioral Design Object Model**

### **12.3.1 The Behavioral Design Object Model**

In this section we will give a short overview of the unified design representation model for behavioral synthesis, called the Behavioral Design Object Model (BDOM). For a more detailed presentation of BDOM the reader is referred to [Rund90b] and [Rund91a]. As can be seen in Figure 12.2, BDOM divides the design information into two levels, the design entity information and the design data information.



**Meta Data:**

- (1) version derivation
- (2) hierarchy
- (3) equivalence
- (4) tool history
- (5) transformation sequence
- (6) design decomposition
- (7) consistency information

**Design Data:**

- (1) language specification
- (2) behavioral representation.
- (3) state sequencing.
- (4) structural implementation.
- (5) floor-plan information
- (6) linkage between information domains
- (7) design constraints
- (8) user bindings

Figure 12.1: The BDD Design Object Model (BDDOM).

The first level of BDOM maintains the *design entity information*, which captures the overall organization of the design information. This *design entity information*, sometimes also called the *meta-data model*, is represented by the Behavioral Design Entity Graph Model (BDEG). It covers concepts, such as the design entity hierarchy, the version derivation tree, the levels of design abstractions, the different information domains, and configurations. The design entity construct is a concept introduced by BDDB to decompose the potentially large set of design data objects that make up a design into manageable chunks. A design entity is an abstraction for a collection of interrelated design data objects that form (part of) a design. Design entities therefore are at the granularity of database operations. For instance, the smallest unit for locking during data access is a design entity. Similarly, the smallest unit of data transfer between BDDB and the design tools is a design entity. The Behavioral Design Entity Graph Model stores organizational attributes of a design entity, such as its name, its version number, and the type of its content. Furthermore, it keeps track of semantic relationships between design entities, such as equivalence, versioning, etc. The designer is allowed to query these relationships. However, he or she is not allowed to directly manipulate them. Instead, BDDB provides a procedural interface that supports a limited set of operations on the design entity graph, such as check-in, check-out, etc. These operations have fixed update semantics associated with them, which guarantees that correct relationships are automatically being maintained by BDEG. The Behavioral Design Entity Graph Model serves as foundation for database support functions like version management and schema browsing.

The second level of BDOM maintains the *design data information*, i.e., the *actual design data* of the application domain. This *design data information* is represented by the Behavioral Design Data Representation Graph Model. It describes the design at a level at which the design tools are ultimately interested in working on. It thus includes the design specification that describes the function of the design, its mapping into an internal graph representation that can be manipulated by design tools, the synthesized state sequencing that shows the slicing of the behavior into states, the resulting structure graph that implements the design, and finally the floorplan. Since the Behavioral Design Data Representation Graph Model captures the *actual design data* of the application domain, tool access to BDDB will primarily be at the design data level. In fact, design data objects stored at this level of BDOM are generated as well as modified by design tools as well as by human designers during the design process. BDDB provides a set of primitive access routines for objects at the design data level. Example routines are the creation of such an object, the modification of an attribute value, etc. These primitive access routines can be used by tightly-coupled design tools to manipulate the design data

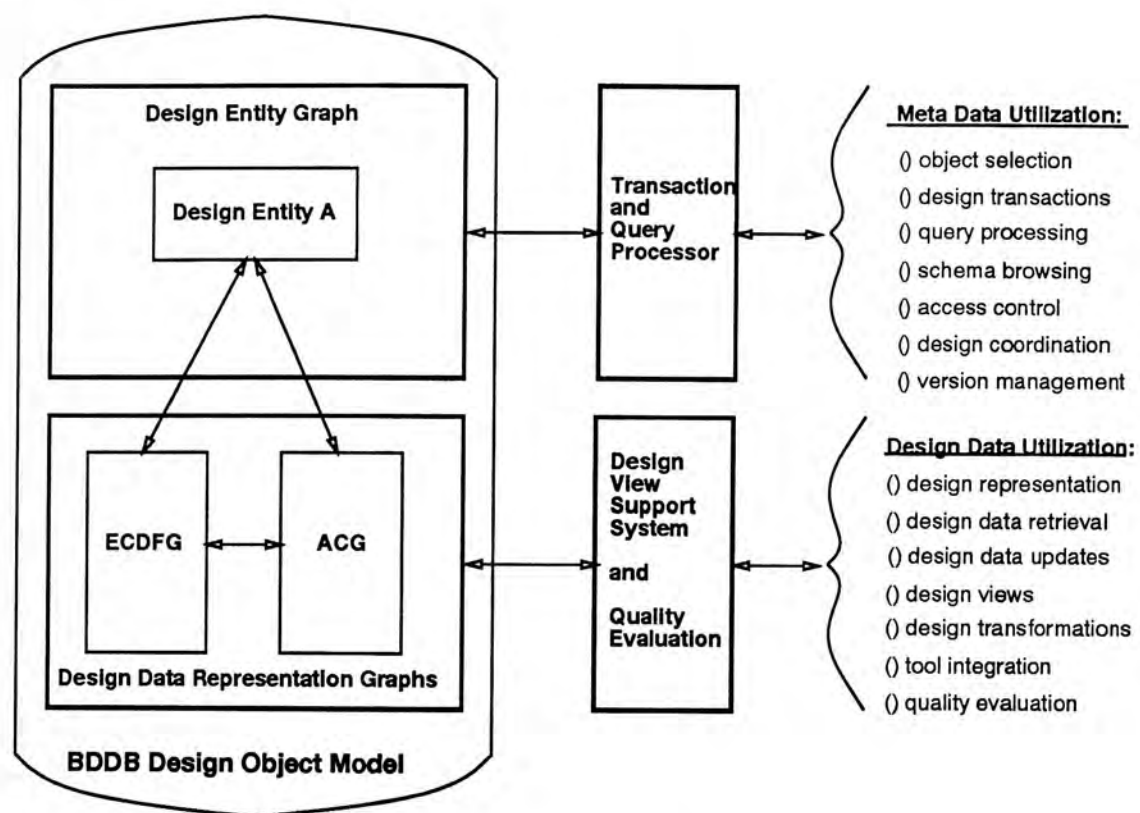


Figure 12.2: Utilization of BDOM.

objects. Furthermore, database tools, such as the consistency checker, may also use these routines for access to the design data. *Design view creation* also takes place at the design data level. Therefore, design views are constructed for the design data found in the Behavioral Design Data Representation Graphs, and not for information maintained in the Behavioral Design Entity Graph.

### 12.3.2 The Behavioral Design Data Representation Models

In this section, we discuss the lower level of BDOM, namely, the Behavioral Design Data Representation Model. The Behavioral Design Data Representation Model distinguishes between two graph models: the *behavioral graph model* and the *structural graph model*. The behavioral graph model describes the behavioral specification of the design. It corresponds to an Extended Control/Data Flow Graph Model (ECDFG) that is augmented with advanced features, such as timing constraints, memory access, events, state transition information, and structure bindings. The ECDFG model comprises the following information: (1) the VHDL input specification that describes the function of the design, (2) the flow-graph representation which captures the behavior over time, and (3) the state sequencing that shows the slicing of the behavior into states. The structural model, represented by an Annotated Component Graph, captures the hierarchical graph structure of interconnected components augmented by timing constraints. It also represents the geometric implementation of the structural implementation, called the floor-plan. A detailed description of the Behavioral Design Data Representation models and their semantics can be found in [Rund90b], and therefore is omitted here. Below we will instead summarize the most important features of ECDFG in an informal manner to give you a basic flavor of the model. We then present one example of this representation.

#### Features of the Behavioral Design Representation Model

We have developed a *complete* design representation model for behavioral synthesis, called ECDFG [Rund90b]. Most importantly, it supports the complete trajectory of behavioral synthesis by capturing user constraints in the form of state and component assignments, by supporting partial design structures, and by allowing for the incremental addition of such synthesis results to the graph model as the design evolves.



ECDFG is an attributed multi-graph model based on the familiar control/data flow graph concepts [Orai86]. This hybrid control/data flow graph model supports the explicit modeling of sequencing of the design via the control flow graph and the modeling of operations on values and variables via the data flow graph. Note that the former will be mapped to control logic and the latter to data path components and connections. This model thus allows for design style trade-offs between control and data path implementation. In addition, ECDFG is augmented by numerous features important for behavioral synthesis, such as the state transition graph, multi-cycle operation nodes, timing constraints, asynchronous events, conditions for control generation, and component allocation information.

The *state transition graph*, which sits on top of the control/data flow graph, stores information about each state as well as about the transitions between states. Each state node corresponds to one time step during the execution of the design, and hence it points to one or more control flow nodes that are executed during the given state. State transition conditions (and events in the case of asynchronous designs) are captured in the control/data flow graph.

A control flow node may be executed during one or more time steps. Therefore, each control flow node is augmented with a *state* annotation, which is an ordered list of *state references*. A data flow node can also be assigned to more than one state, then called *multi-cycle node*. To handle the representation of *multi-cycle nodes* appropriately we have augmented the data flow node structure by additional annotations, such as an ordered list of *state references* and the current cycle identifier.

For *control generation purposes*, we associate with each data flow node the condition under which the node is executed within a given state. This control condition is a function of both the conditions in the flow graph and the given state assignment.

We have also added a new control flow node type, called (*asynchronous*) *event node* which represents events in an asynchronous design. This construct is useful for capturing asynchronous designs such as those modeled by BIF [Dutt90]. Similarly, we have introduced a *condition node* construct which has no matching *end-condition node*. This construct represents conditions in an unstructured flow graph derived from an FSM-like description. Note that the conditions node pairs commonly found in the CDFG model, such as *begin-case*, *end-case*, *begin-if*, *end-if*, are useful for the representation of structured flow graphs derived from block-structured language specification, but not for the capture of FSM-style descriptions like BIF [Dutt90].

ECDFG also supports *generalized conditions* with complex conditional expressions at the control flow graph level. This allows for design transformations to

expose various design styles, for instance, by merging several conditions into one more complex condition.

Each data flow node has an associated substructure, called *component allocation information*, that describes which structural component(s) in the component graph implement the functional behavior of the node in the flow graph. This is useful for supporting *allocation constraints* in the form of operator-to-component bindings a priori suggested by the designer and for capturing component allocation bindings synthesized by a design tool. Note that in ECDFG a data flow net is represented via a separate graph node rather than an arc. It thus can carry its own attributes, such as bit-width, data type, etc. A data flow net may be implemented by a simple wire, by a multiplexor, by a bus, or by a combination of these components. Therefore, we associate this component allocation information not only with data flow nodes but also with data flow nets.

ECDFG also supports *timing constraints* between any pair of signals (events) in the data flow graph or between any two execution points in the control flow graph. For details on these features see [Rund90b].

### An Example of the ECDFG Model

Figure 12.3 shows a behavioral VHDL description of a 4 bit programmable up and down counter. For this example we assume a state assignment as shown in the state table in Figure 12.4.

The behavioral description of that counter is compiled into the ECDFG graph depicted in Figure 12.5. Note that the ECDFG model distinguishes between the control and the data flow portions of a description. The control flow explicitly models the control constructs found in the original design specification whereas the data flow models the operations on values and variables. Therefore, in Figure 12.5 the nested if-construct of the VHDL description is represented by control flow nodes (box for statement-block node and triangle for decision nodes). The data manipulations, such as assignment statements, are represented in data flow. All data flow nodes are depicted by little circles. In this figure, we draw the data flow graphs associated with each statement-block node within the control node. The state assignment (Figure 12.4) then corresponds to the state transition nodes shown on the left hand side of Figure 12.5 (depicted by large circles).

Figure 12.6 shows one possible structural implementation of the counter. The chosen counter component has four ports, called up, down, countin, and countout.

```

entity counter is
  port (countin: in BIT(3 downto 0);
        up: in BIT;
        count: in BIT;
        countout: out BIT(3 downto 0)
        )
end counter;
architecture counterbody of counter is
  signal I: BIT(3 downto 0);
begin
  -Comment: - UP==1 if count up and UP==0 if count down
  -Comment: - COUNT==1 if count and COUNT==0 if program
  process begin
    countout <= I;
    if (count = 1)
    then if (up = 1)
      then I <= I + 1;
      else I <= I - 1;
      end if;
    else I <= countin;
    end if;
    wait for 12ns;
  end process;
end counterbody;

```

Figure 12.3: VHDL Specification of a Programmable Up-and-Down Counter.

present state	condition	(value)	actions	next state
0	-	TRUE	countout <= I	1
1	count=1	TRUE	-	2
		FALSE	I <= countin	0
2	up=1	TRUE	I <= I + 1	0
		FALSE	I <= I - 1	

Figure 12.4: State Information for the Programmable Up-and-Down Counter.

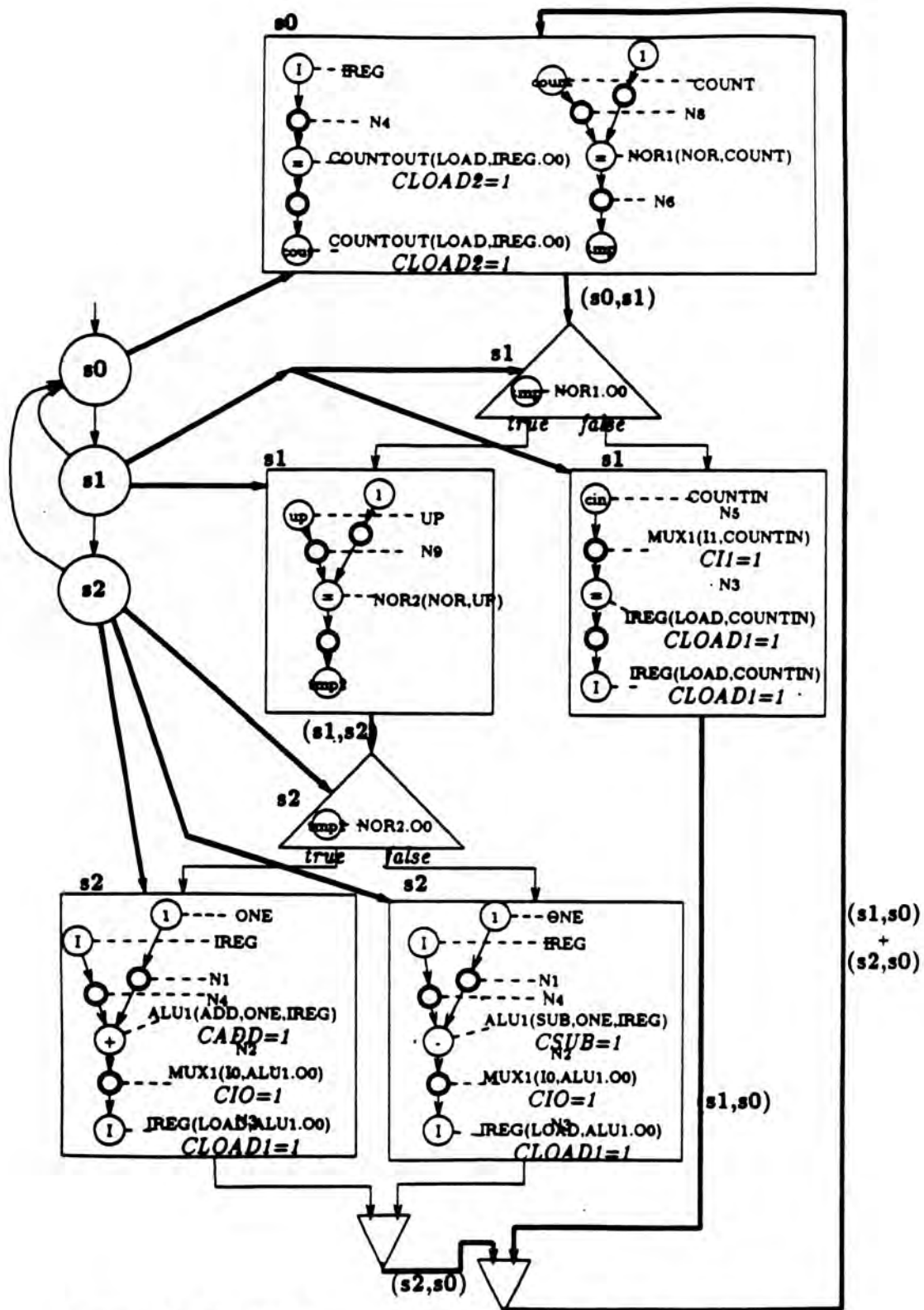


Figure 12.5: ECDFG Representation of the Up-and-Down Counter.

These ports represent the points of communication between the environment and the counter. Each component is connected to other components via nets that link the output port of one component to the input port of another component. A net is represented by a set of connection arcs (depicted by an arrow) and an interconnection node (depicted by a circle) with the corresponding net name.

Next, we discuss the structural information that is attached to ECDFG in the form of component allocation annotations (Figure 12.5). The constructs in a data flow graph are implemented by one or more constructs from the structural domain, whereas the control flow constructs have no direct structural equivalent. A control flow sequencing arc for instance does not represent a real physical connection. Rather, the control flow graph models the sequencing of the behavior over time, and thus is synthesized into control logic.

The correspondence between the data flow graph and the annotated component graph is as follows: Each operation and variable node of the behavioral description has associated the corresponding structural component that implements it. A behavioral operator or a variable access in the data flow graph gets mapped to a functional unit, register, or bus in the data path. Similarly, the data flow edges correspond to actual connections in the annotated component graph along which the values travel. Thus, data flow edges are mapped to one wire or a sequence of wires and components in the data path. This component mapping information is maintained in the form of structural annotations in the data flow graph rather than behavioral annotations to the annotated component graph. We choose this approach since the component mapping is multiplexed in time. The same hardware unit is reused multiple times. In fact, a hardware component may be bound to several data flow nodes in the same state, when scheduling across conditional branches is performed. Depending on the evaluation of the conditional branch, one of the bindings will be selected by the control unit during execution.

These component mapping annotations are represented by dashed lines in Figure 12.5. The component mapping annotation for a data flow node corresponds to the component name, the chosen function, and the set of inputs ordered from left to right. For example, the plus operation in state S2 is bound to the ALU1 component. The selected function is ADD and its inputs from left to right are ONE and IREG. In addition, a control selection variable is associated with each component mapping annotation. The control selection variable corresponds to the control line name (net of control type) and its associated value. For the previous example, the control selection variable is the tuple "CADD=1" where CADD corresponds to the control line and "1" is the value that the control unit will assign to this control line. For a control selection variable which is not used, a value of zero is assumed.

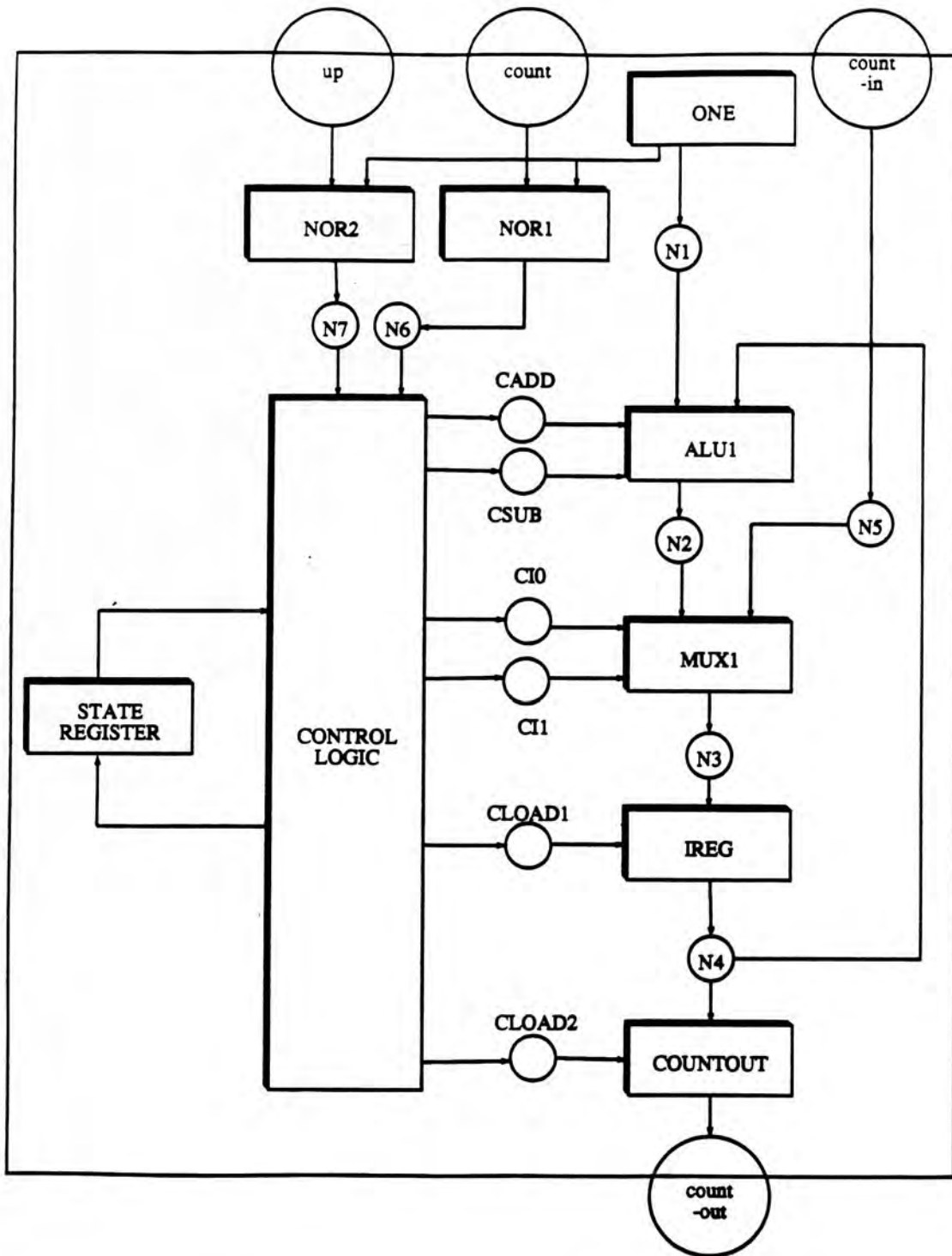


Figure 12.6: ACG Representation of the Up-and-Down Counter.

### 12.3.3 The Behavioral Design Entity Graph Model

Below, we introduce the higher level of BDOM, called the Behavioral Design Entity Graph Model (BDEG). Figure 12.7 depicted graphically the schema of the BDEG model using the Entity-Relationship diagram technique [Chen76]. A box represents an entity set, i.e., a set of objects. A diamond a relationship set between one or more entity sets. The circles describe elementary attributes. A relationship is connected to the entity sets it relates by lines. A term of the form (min,max) with min and max integer numbers and  $\text{min} \leq \text{max}$  is associated with each connection arc. It indicates the minimal and the maximal involvement of each entity of the entity set in relationship. If a relationship is connected to the entity sets by an arrow rather than by a line, then the entity set pointed to by the arrow is a dependent set (existence dependence constraint). This means that elements can only exist, if there is an associated element in other related sets. For instance, a DE Body can only exist if there is an associated DE Interface.

We will explain the Behavioral Design Entity Graph Model depicted in Figure 12.7 by first describing each of its concepts below and then by presenting an example. For the following, we use the term DE to abbreviate the words "Design Entity".

- A **DE object** is the basic unit of design that is explicitly managed by the design database. Therefore, each **DE object** has an associated collection of design data. This collection, called **design data representation graph**, or, simply, **design file**, corresponds to the domain-specific design representation. BDDDB distinguishes between two **DE information domains**, namely, the behavioral and structural domains. A **DE body** describes either the behavioral or the structural domain of the **DE object**. If a **DE body** is of the behavioral domain, then it has associated design data in the form of an ECDFG. If a **DE body** is of the structural domain, then it has associated design data in the form of an ACG.
- A **DE object** is composed of a **DE interface** and a **DE body**. A **DE interface** corresponds to the externally visible parts of a **DE object**, i.e., the input and output ports of the **DE object**. In other words, it corresponds to the **DE type** of a **DE object**. A **DE body** specifies an implementation (content) of a **DE object**. A **DE body** inherits the port information from the **DE interface**.
- There can be several implementations for a **DE object**. This means that several **DE bodies** can be associated with one **DE interface**. **DE versions** of a **DE object** then have the same interface but different **DE bodies**.

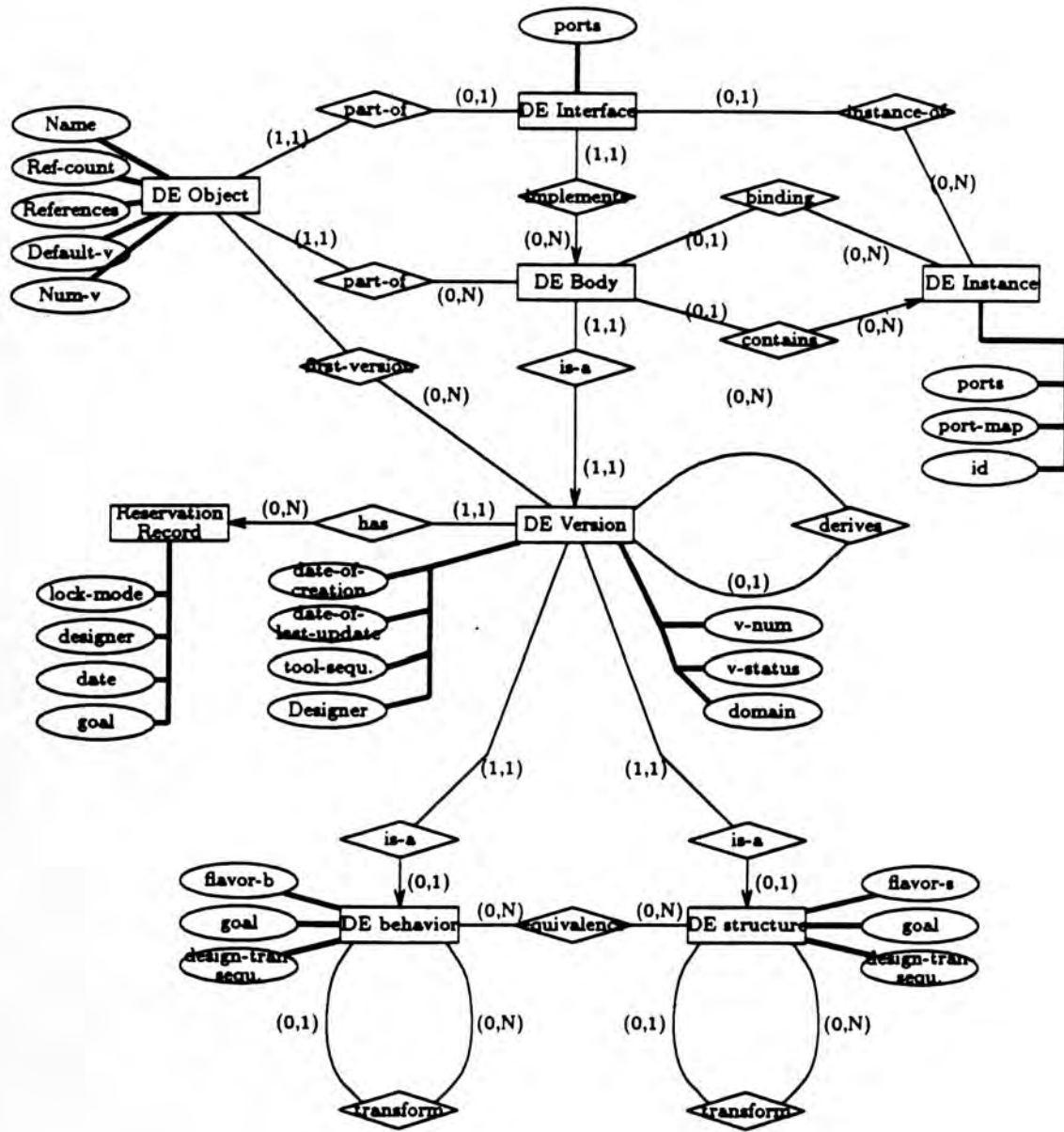


Figure 12.7: Schema for the Behavioral Design Entity Graph Model.



- The **DE version derivation graph** is a tree structure that shows the history of evolution of the **DE object** by maintaining a **is-derived-from** relationship between pairs of **DE versions**.
- A **DE object** can either be primitive or a composite object. If a **DE object** is primitive, then the corresponding **DE body** is described in terms of design data objects. If the **DE object** is composite or hierarchical, then the content is described in terms of other design entities as well as design data. We do not support the interconnection of these subcomponents as part of the schema (as suggested by other researchers). Instead, we claim that the type of interconnections depend on the underlying information domain. Therefore, we assume that these interconnections are taken care of in the associated design data representation graph. However, each **DE subcomponent** refers to the design data object that it is further decomposing.
- We distinguish between the definition of a **DE object** and of its use. A **DE instance** is a carbon-copy of a **DE-object**, in the sense that a **DE-object** corresponds to an object type definition and a **DE instance** corresponds to different instantiations of this object type. This concept allows for reuse of a **DE object** in other designs and therefore for the concept of a **DE hierarchy**.
- A **DE hierarchy** is created when a **DE object** is composite, that is, it is decomposed into lower-level **DE objects** using the **is-contained-in/is-composed** relationship. A composite **DE version** has one or more **DE subcomponents**. These **DE subcomponents** can be bound to a generic **DE object** by referencing its **DE interface** or to a specific **DE version**. If a composite **DE version** is bound to generic **DE objects**, also called a dynamic binding, then the **DE hierarchy** effectively corresponds to a collection of all possible design combinations that can be created at run-time by selecting a fixed **DE version** for the generic **DE object**.
- A **DE configuration** then corresponds to establishing bindings for the **DE subcomponents** of a composite **DE object**. Therefore, a **DE configuration** selects one fixed design combinations from a **DE hierarchy**. A **DE configuration** of a **DE body** thus leads to the creation of a new **DE version**. A **DE body** can be either a leaf body (with no **DE instances**) or a composite body. A composite body can be in one of the following status: it can be **not configured**, **partially configured**, or **completely configured**. A **DE configuration hierarchy** then corresponds to a **DE body** that is a completely configured design hierarchy.
- A **DE body** corresponds either to a behavioral or the structural **DE information domain**. The symmetric **is-behavior-of** and **is-structure-of** relationship relates equivalent **DE versions** of different domain types.

## An Example of the BDEG Model

In Figure 12.8, we depict an example of a behavioral design entity graph. The design entity object A (**DE\_object** A) consists of a **DE\_interface**, called A, and a DE version set consisting of two **DE\_bodies**, called A1 and A2. The **DE\_interface** A specifies the externally visible type of the design entity, such as the input ports A, B and C and the output port F. These are the values that are supplied to the design entity from the outside as input or that the design entity passes as output to other design entities. Both **DE\_bodies** A1 and A2 specify the implementation of the design entity object A in form of an associated design data representation graph. They are of the domain-type **behavior**, therefore the associated graphs are Extended Control/Data Flow Graph representations. **DE\_body** A1 is a simple whereas **DE\_body** A2 is a complex design entity body. That is, **DE\_object** A2 is decomposed into smaller data representation graphs using the **DE\_Hierarchy** concept. The design data object with the identifier DD-#2 of the ECDFG attached to A2 is further decomposed using the **DE\_Hierarchy** rather than expanding it in place. This decomposition is represented by the fact that **DE\_body** A2 has one subcomponent, the **DE\_instance** compl. Therefore, the ports, P1, P2, and P3 of the **DE\_instance** compl are bound to the interface of the design data object DD-#2. Furthermore, **DE\_body** A2 is completely configured, meaning, its subcomponents have a fixed binding to other **DE\_bodies**. In particular, its **DE\_instance** compl configured to the **DE\_object** B with the **is-bound-to** relationship. Hence its ports, P1, P2, and P3 are also bound to the ports of **DE\_object** B. The first set of port mappings corresponds to bindings within the **DE\_object** A, whereas the second set of bindings goes across the **DE\_hierarchy** to another **DE\_object**, namely, B.

## 12.4 A Textual Exchange Format for Design Data

### 12.4.1 Introduction

To support a flexible synthesis environment we need to be able to exchange this design information between different design tools – that are not necessarily tightly-integrated nor part of the same machine or system. Therefore, BDDDB supports two modes of tool interaction: (1) tight integration using design views as discussed earlier and (2) loose integration using design files. For this purpose, we have designed a *textual exchange format*, called BDEF, for exchanging design data modeled using the *unified design representation model* BDOM. Design tools exchange design data based on this common format. Such a design file approach is useful for design tools

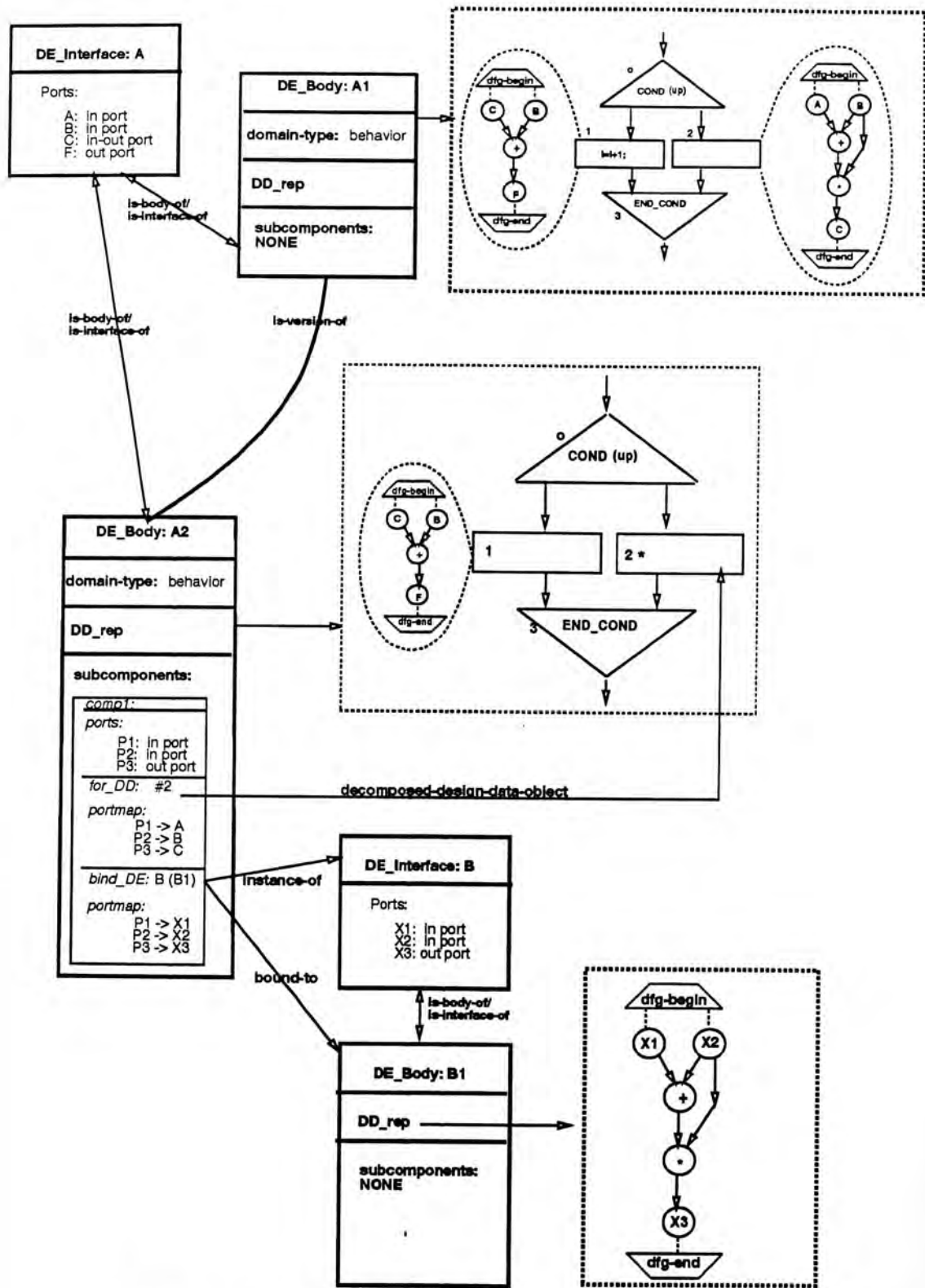


Figure 12.8: Behavioral Design Entity Graph Example.

within the same behavioral synthesis system if the individual tools are designed stand-alone modules or if foreign design tools are being imported into the system. Also, of course, such a standard format can be used to exchange design data between systems on different machines and even across sites. Note that emphasis here is on a tool interface rather than on providing a representation that is particularly suitable to humans.

#### 12.4.2 Other Work Related to Textual Exchange Formats

Most work reported in the literature on exchange formats for behavioral synthesis focuses on the capture of RTL and logic-level designs [EDIF].

TREEMOLA (tree micro operation language) [Beck90] is a language used to exchange design data between tools in the MIMOLA hardware design system. TREEMOLA is a parse-tree like structured language which can represent designs at the behavioral or at the netlist level. However, it does not capture intermediate design results, like, state assignment and structural binding.

Research most closely related to BDEF is by Eijndhoven et al [Eijn91] sponsored under the European ASCI project (ESPRIT Basic Research Action 3281). They propose a textual format for the ASCIS data flow graph [Jong91]. Their model however does not handle many of the advanced concepts that we propose, such as asynchronous events, complex condition nodes, etc. Furthermore, their data flow model and textual format are not capable of the full support of the behavioral design process, since it is a "pure data flow graph" without any extensions to handle intermediate design results produced by behavioral synthesis tools. For instance, their model does not support the binding of hardware units to the data flow graph nor the capture of state information needed to generate control logic. ECDFG, on the other hand, handles this information as integral part of the graph model.

BIF [Dutt90] is an annotated textual state-table format for behavioral synthesis. Its focus is on the user interface aspect and not on the tool aspect. Hence, it for instance describes the actions in each state using a language expression rather than representing data dependencies explicitly (as done in BDEF). BIF thus is complementary to our proposal of providing a tool-oriented exchange format.

### 12.4.3 BDEF: The Behavioral Design Data Exchange Format

Given a design representation model, like ECDFG, we now need to take the steps shown in Figure 12.9 to guarantee the usage of the model. First, we define the object type definitions for ECDFG using some general type definition language <sup>1</sup>.

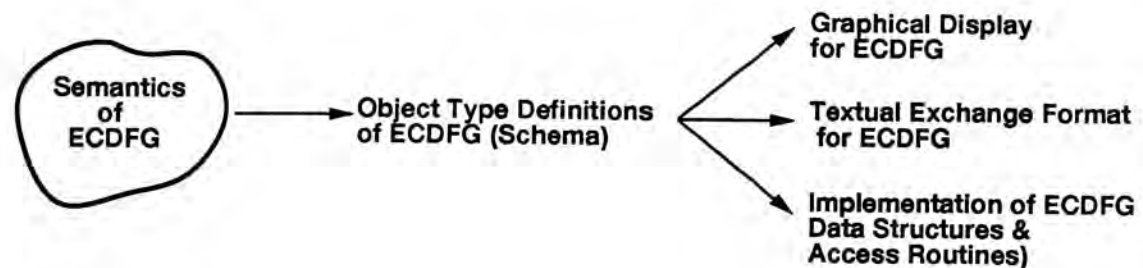


Figure 12.9: Our Approach

A textual schema description of ECDFG offers several advantages. First, it will be easier for a designer joining the team to comprehend the object model underlying the behavioral synthesis environment if its description is available in a readable format. This schema description is furthermore void of implementation details that would creep up when ECDFG were directly described in a programming language, such as, C or C++. Also, possible upgrades to ECDFG, e.g., the addition of new attributes or object types, are more straightforward when done on this high-level description level than when done at the level of a programming language. Last but not least this (computer-readable) schema description provides a foundation for developing (1) a textual exchange format for ECDFG, (2) an implementation of ECDFG, and (3) a graphical display of ECDFG. It is our long-term goal to provide meta-tools that support the automatic generation of the above mentioned software from a given schema description. Below, we take one step towards this direction by presenting generic rules of creating a textual exchange format from a given schema description.

<sup>1</sup>In traditional database terminology, a language for the definition of object types is commonly referred to as Data Definition Language. The resulting description of the database object types is called a *schema*.

## Object Type Definition of ECDFG

We have developed an Object Type Definition Language (OTDL) for the purpose of describing the design representation model. OTDL is based on parameterized type constructors, such as finite sets, lists and tuples, which support the composition of simple object types to define more complex design object types. OTDL can model *shared subobjects*, *many-to-many* possibly symmetric relationships, and even *recursively defined object types* as required to describe complex nested graph structures found in ECDFG. We have used OTDL to define the ECDFG schema. For a complete definition of the ECDFG object types the interested reader is referred to [Rund93].

### Rules for the Textual Format

In Figure 12.10 we present format rules for the textual description of design objects. Once an application domain, such as the ECDFG model, has been modeled by OTDL, then these format rules automatically determine a textual exchange format for the described model. The rules in Figure 12.10 use the following conventions:

- An arrow indicates that the object at the source (higher in the tree) is represented in terms of the objects at the destination of the arrow (lower in the tree).
- A half circle around the outgoing arrows indicates that the object is composed of *one and only one* of the objects below. Otherwise the object above is composed of *all* objects below.
- A dot at the source of an arrow indicates that the object at the source corresponds to a *set* of the objects below.
- Objects in bold print are further defined by the decomposition expressed by outgoing arrows. An underlined bold object is further defined at another location in the syntax tree. Objects in light print correspond to simple domains, like, Integer and String.

### Explaining The Format Rules Via An Example

*Design entities* are the units of data transfer between the design manager and design tools [Rund90b]. A *design entity* thus corresponds to a collection of related design data objects, i.e., it is a data flow graph, a state transition graph, etc. A *design*

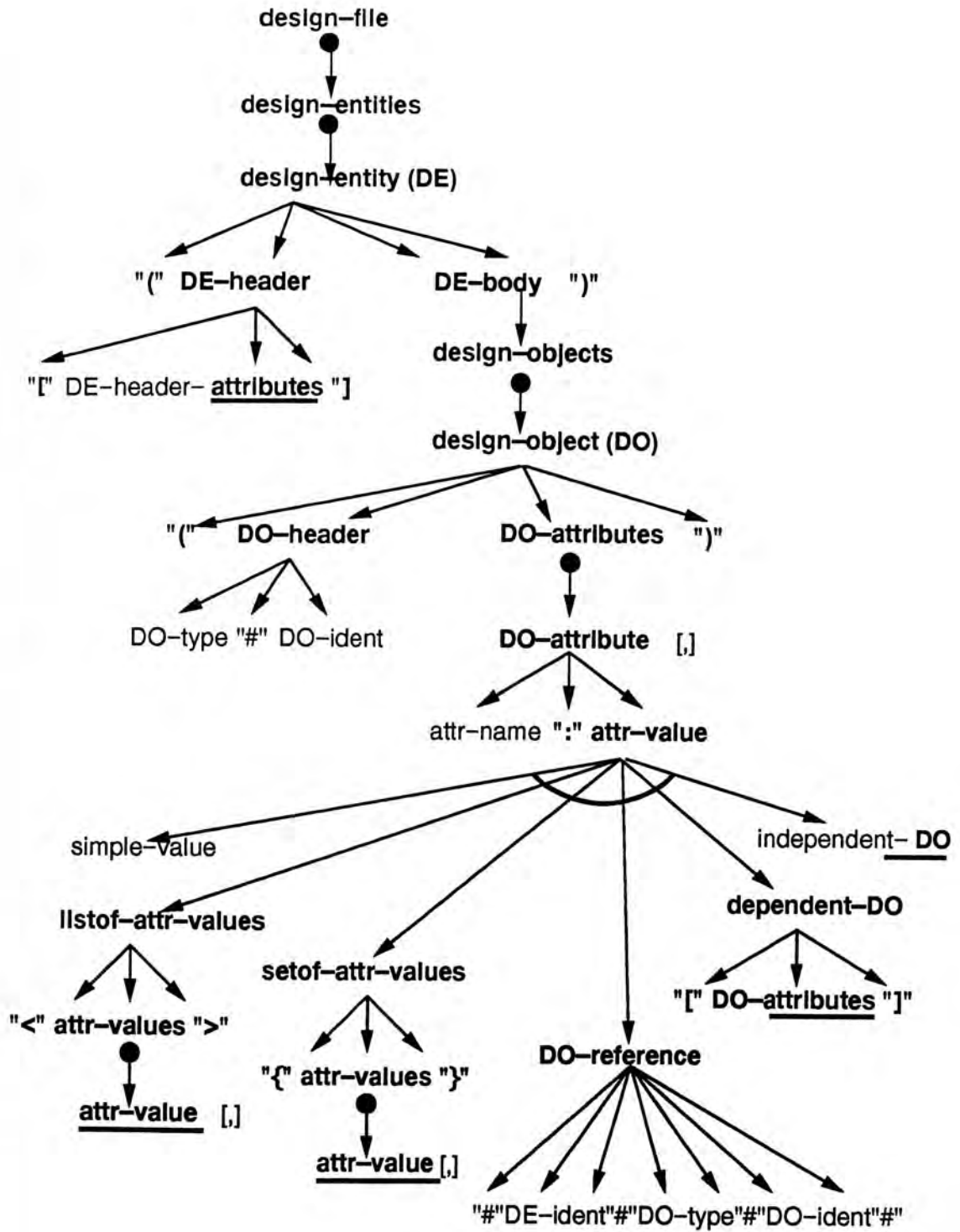


Figure 12.10: Graphical Depiction of Textual Format Syntax Rules.

*file* holds one or more such design entities. A design entity consists of a *design entity header* and a *design entity body*. The header identifies the design entity, while the body describes the composition of the design in terms of actual *design data objects*. In Figure 12.11, for instance, the *design entity header* gives the name and version number of the design entity, whereas the body holds two design data objects: a data flow and a control flow node.

Each design object, encapsulated by a set of parenthesis, is described in three parts: its object type, its identifier, and its attributes. The type and identifier of the design object uniquely identify the object within a given design entity. For instance, in Figure 12.11 a data flow node with the identifier 10 is defined by “(DF\_NODE#10 ... )”.

A design object is described by listing zero or more of its attributes. Attributes are identified by attribute names rather than by their position. This introduces redundant data as attribute names are repeated within the definition of each object. It is needed, however, since the number and type of attributes of a design object may vary with the requirements of design tools, i.e., the design view type. An attribute value can be a simple value, a list of attribute values, a set of attribute values, an independent (sub)object, a dependent (sub)object, or an object reference (Figure 12.10). An example of a simple attribute is the `node-type` attribute with the value `OPERATION`. The domain of a simple attribute is either a basic type, like `Integer`, or a predefined enumeration type, like the domain of `node-type` is { `OPERATION`, `VARIABLE-ACCESS`, ... }.

*Set-valued attributes* (*list-valued attributes*) are distinguished from simple values by enclosing the attribute value list by a pair of curly brackets ‘{’ and ‘}’ (angular brackets ‘<’ and ‘>’) with adjacent data values separated by a comma. For instance, the `input-ports` attribute with the three values `I1`, `I2`, and `I3` is a list attribute. Being a list attribute, the order of these values is significant. `I1` might correspond to the left data input, `I2` to the right data input, and `I3` to the carry input of the data flow node. The order of the attribute values for a set attribute, on the other hand, is not significant. For instance, the `operations` attribute indicates that the data flow node may either execute an `ADD` or a `SUBTRACT` operation. It says nothing about when or in which order either of these two is used.

We support the modeling of a hierarchy of design objects via attribute typing. Namely, if a design object is composed of other design objects, then these sub-objects become attributes of the super-object. An independent sub-object has its own unique identifier, whereas a dependent sub-object exists only in the context of its super-object. That is, it is a ‘structured value’ rather than an object.



```

([ /* design entity header */
  DD_NAME: CONTROL_COUNTER,
  DD_VERSION: 20,
  DD_DOMAIN: BEHAVIOR,
  DD_BEHAV_FLAVOR: BEHAV_WITH_STATES,
  DD_CHUNK_TYPE: STATE_CONTROL_DATA_FLOW
]
/* design entity body below: objects that make up the graph. */
(DF_NODE #10
  /* simple attribute */
  node-type: OPERATION,
  /* list attribute */
  input-ports: < I1, I2, I3 >,
  /* set attribute */
  operations: { ADD, SUBTRACT },
  /* list of independent subobjects attribute */
  ports: < (DF_PORT #1 ... ), (DF_PORT #2 ... ) >,
  /* dependent subobject attribute */
  condition: [[cond: C1, value: 0, op: ADD],
              [cond: C2, value: 1, op: SUBTRACT]],
  /* reference attribute */
  associated-cf-node: ##CF_NODE#5#
)
(CF_NODE #5
...
))

```

Figure 12.11: An Example Using the Textual Format Rules (BDEF).

Therefore, dependent design objects – similar to simple values – do not carry any identifiers. Dependent objects are encapsulated by a pair of angular brackets '[' and ']', while independent objects, nested or not nested, are encapsulated by a pair of round brackets '(' and ')'. For instance, the `ports` attribute corresponds to a list of independent design subobjects. Each port object has its own unique identifier and could have been represented as separate object in the textual description. We have chosen to represent ports as nested subobjects of the data flow nodes, since ports are conceptually closely related to one and only one data flow node. The `condition` attribute, which indicates the condition under which each of the operations is executed, is modeled by a set of dependent subobjects.

The concept of *references* is introduced to model relationships between design objects. A reference consists of three parts, each separated by a '#' symbol: the name the design entity in which the referenced design object is defined, the object type of the design object that is referenced, and the identifier of the design object that is referenced. The referenced design object can either be in the same design entity as the referencing design object or in a different design entity. If the first part of the reference is null, i.e., no design entity name is specified, then the reference is assumed to be referring to a design object within the current design entity. In Figure 12.11, the attribute `associated-cf-node` references the design object of type "CF\_NODE" and with the identifier "5" (also shown in the figure).

The application of these textual format rules to ECDFG results in the Behavioral Design Data Exchange Format (BDEF). In other words, BDEF determines the object types (e.g., data flow node, control flow node, timing constraint node, state node, etc.), their attributes (e.g, name and type), and their relationships. Rather than presenting the complete syntax of BDEF, we will describe several examples using BDEF in the next section. For a complete syntax see [Rund93].

## Evaluation of BDEF

BDEF is in line with other intermediate languages for design representation [Beck90, Eijn91]. Compared to them, it offers a number of advantages:

- BDEF supports the complete behavioral synthesis trajectory by representing design information ranging from the initial behavioral specification over state assignment and structural bindings down to the final component graph.

- Besides handling the capture of synthesis results, BDEF also supports the representation of constraints, such as general timing constraints, partial user-binding for both state assignment and component mapping, and an initial partial data path structure.
- BDEF allows for the incremental definition of a number of well-defined views of the unified design representation ECDFG depending on the status of the design process <sup>2</sup>.
- Lastly, BDEF supports the extraction of subsets from the complete ECDFG model as required by design tools<sup>3</sup>.

## 12.4.4 BDEF Examples

### A Simple Data Flow Graph Example

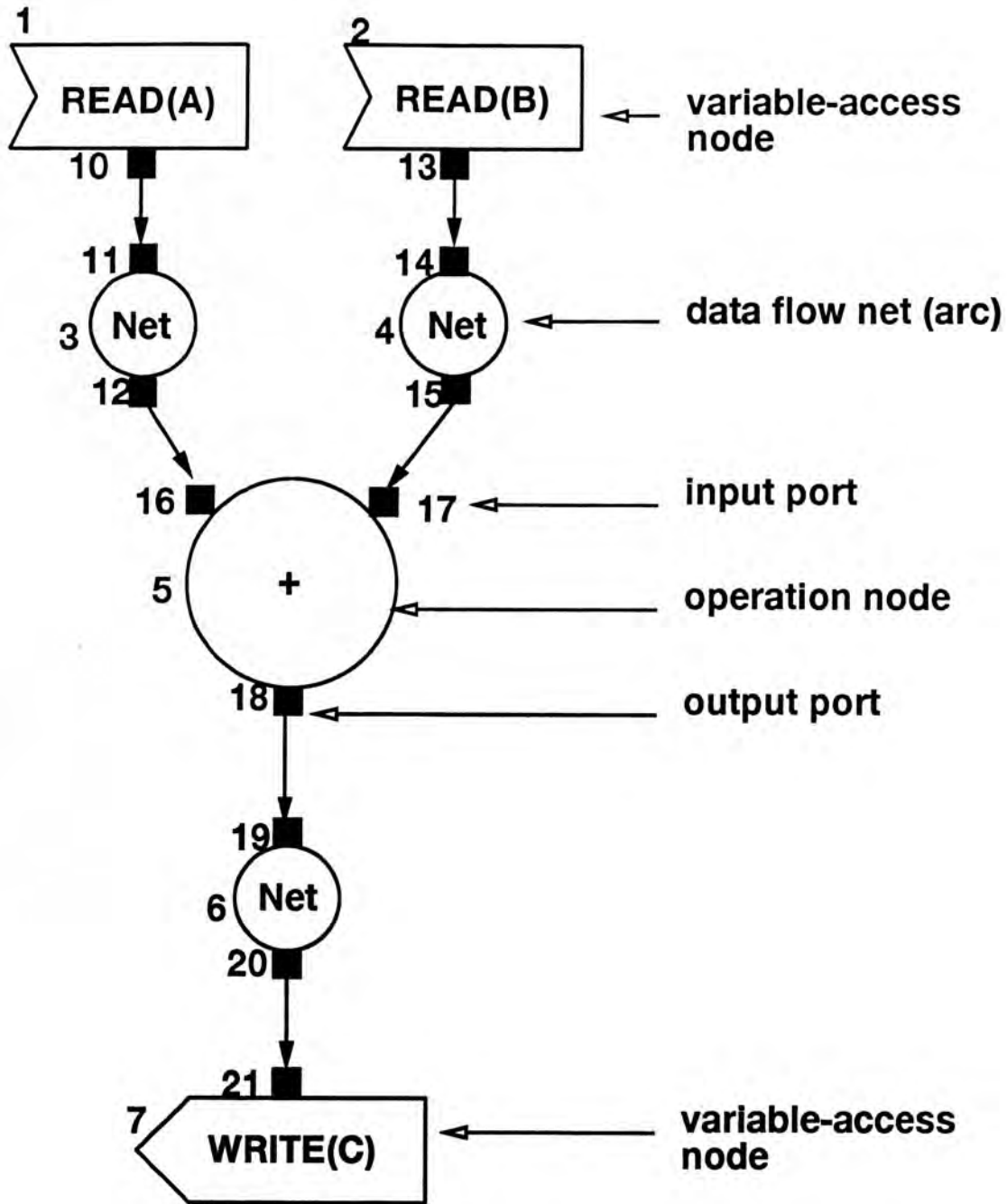
In Figure 12.13.a we show the ECDFG data flow graph for the expression “ $C = A + B$ ”. Note that nets, which represent the data values that flow between data flow nodes, are modeled as nodes themselves. For this reason, ECDFG allows for the structural binding of the data flow graph with not only functional units and registers but also interconnection units, such as wires, muxes and buses. The corresponding BDEF description is given in Figure 12.13.b.

### A Timing Constraint Example At the Data Flow Level

In ECDFG, timing constraints can be specified at both the control flow and the data flow graph level [Rund90b]. The graphical representation and matching BDEF description of a timing constraint on data flow are given in Figures 12.14 and 12.15.

<sup>2</sup>For this, BDEF labels the design entity with categorizing *design entity attributes*. For instance, the *behavioral-flavor* attribute categorizes the information content of a design entity as { *behav-pure*, *behav-states*, *behav-allocation*, *behav-binding*, *behav-control* }. It thus indicates whether it is a purely behavioral design or whether state information has already been synthesized, etc. For a BDEF example see Figure 12.11.

<sup>3</sup>For this, BDEF labels the resulting design entity with the *behavioral-design-chunk* design entity attribute, which indicates which portion of ECDFG is captured for the current design. Its domain is { *data-flow*, *control-flow*, *control-data-flow*, *state-graph*, *state-control-flow*, *state-control-data-flow* }. For instance, for control generation the design tool may only be interested in the *state-graph* itself.



**Behavioral Design Specification: "C <= A + B;"**

Figure 12.12: Graphical Representation of a Data Flow Graph.



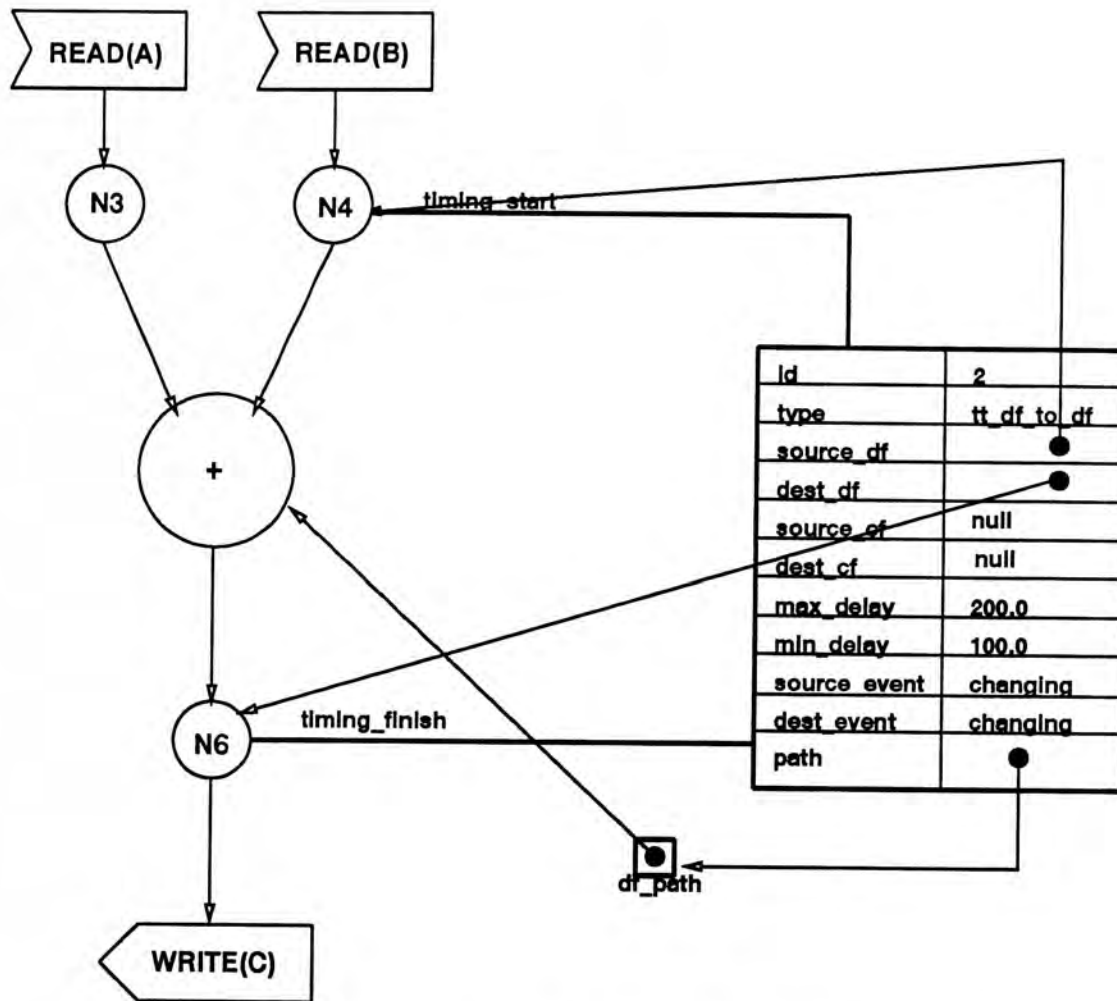


Figure 12.14: Graphical Depiction of a Timing Constraint in a DFG.

```
(DF_NET#4
TIMING_START: ##TIMING_INFO#2#
)

(DF_NET#6
TIMING_FINISH: ##TIMING_INFO#2#
)

(TIMING_INFO#2
TIMING_TYPE: TT_DF_TO_DF,
SOURCE_CF: ##DF_NET#4#,
DEST_CF: ##DF_NET#6#,
MIN_DELAY: 100.0,
MAX_DELAY: 200.0,
SOURCE_EVENT: TE_CHANGING,
DEST_EVENT: TE_CHANGING,
DF_PATHS: <
  < ##DF_NODE#5# >>
)
```

Figure 12.15: BDEF Description of the Timing Constraint in a DFG.

Timing constraints in data flow originate and end at data flow nets. The source-event and destination-event attributes thus specify events on the signal relative to which the timing constraint is being specified, such as RISING, FALLING and CHANGING. The constraint in Figure ?? places a minimum delay of 100.0 and a maximum delay of 200.0 on the execution of the operation  $A + B$  from the time that there is an event on the input B (source\_event=changing) until the result of the operation is available, i.e., there is an event on the result signal (dest\_event=changing). The timing constraints are inserted as independent design objects into the BDEF description (Figure ??b). Another important feature of our timing constraint specification method is the use of path expressions (Figure ??) which indicate if the timing constraint constrains some but not all of the threads of computation between the entry and exit point of the constraint specification. Note that BDEF allows for a more precise and fine-grained timing specification than generally supported by hardware description languages, such as VHDL.

#### 12.4.5 BDEF Implementation

In [Rund90b], we have described the *behavioral design representation model* ECDFG in detail, and in [Rund93], we describe the associated *textual exchange format*, BDEF. In [Rund93], we not only define the semantics of the model but we also develop a graphical representation for ECDFG. This will simplify the understanding and interaction of designers with this design representation.

We have implemented ECDFG data structures and access routines using the C programming language on UNIX. An object-oriented implementation is most suitable to this design representation model being centered around objects and their relationships. Hence, in the future, we plan to convert this software from C to C++.

BDEF files grow large for even reasonably sized design. Therefore we find manual inspection of BDEF files on a routine basis unacceptable. For this reason, we have implemented a simple display routine of the design representation graph using the X Window System under UNIX [Lis88]. Unfortunately, the present implementation allows viewing but not editing of the graph.

We have completed the implementation of a Parser/Generator pair from BDEF to the ECDFG data structures [Rund93] and back using C/YACC/LEX on UNIX. This BDEF Parser/Generator pair serves of course also as a template for building interfaces to other local data structures or files formats. Indeed, we have started the development of a parser/generator pair from BDEF to the textual state-table format, BIF [Dutt90], which serves as user interface in our environment. This link between



the tool-oriented (BDEF) and the user-oriented (BIF) representation will prove to be an important step towards achieving a flexible user-controllable behavioral synthesis environment.

We have successfully developed a modular behavioral synthesis system centered around the unified representation ECDFG. In this environment, design tools interact via incrementally updating ECDFG [Lis88, Rund90b, Ang92]. Design data is shipped between design tools using BDEF design files. Typical run times for the BDEF parser and BDEF generator for examples of medium complexity are under a few CPU seconds on Sun workstations. Hence, the use of a textual design file does not pose a major constraint on our synthesis system. We have found BDEF to be sufficient for capturing designs exhibiting a variety of design features.

## 12.5 Design Exploration Using Design Transformations

### 12.5.1 Design Exploration

BDDDB supports *design exploration process* by providing design transformations for the restructuring of the design representation. These design transformations restructure the design representation in order to explore various design styles. The term design style here refers to a particular architecture type characterized, for instance, by the amount of pipelining, by using buses versus point-to-point connections, etc. For example, the flow graph structure can be transformed to model a control unit with one test bit or two test bits by limiting the number of conditions tested in a decision node to one or two, respectively. In short, a transformed design specification is likely to result in a different design implementation. Therefore design transformations aid the *design exploration process*.

The application of design transformations serves several purposes besides re-targeting the design representation towards a chosen design style: (1) they eliminate the bias of a human design modeler, (2) they eliminate awkward constructs introduced due to direct translation from a textual specification of the design written in a programming language, (3) they optimize the representation, and (4) they filter the design representation to provide a suitable view of the data to the different tools.

Transformations change the structural composition of the design representation while preserving its algorithmic behavior. The *computational equivalence* of the

original representation and the new representation created by applying a sequence of transformations needs to be preserved. Note that these transformations only restructure the design but they do not add additional information. Design tools, on the other hand, alter and refine the design by adding additional information. Thus, they are making actual synthesis decisions.

At first, the designer requests whether and if so which design transformation should be applied to the design representation. Once the interaction of design transformations is better understood, we may attempt to automate this process. In the next section, we will describe several example design transformations.

## 12.5.2 The Collection of Design Transformations

In this section, we give a listing of the BDDB design transformations. Design transformations can be applied to various levels of the design representation, for instance, to the process, the control flow, and the data flow graph level. Therefore, the transformations are organized according the levels of design representation to which they can be applied. For a more detailed explanation of some of them see Section ??.

### Transformations at the process level

#### 1. Grouping/Flattening:

- (a) Group a set of related assignment statements into one block.
- (b) Flatten a hierarchy of nested groups to remove block nestings.
- (c) Combine two blocks at the same level of the hierarchy into one block.

#### 2. Expansion/Encapsulation:

- (a) Expansion: Inline expand a procedure call hierarchy by replacing the procedure call node by a copy of the procedure's body.
- (b) Expansion: Inline expand a design entity instance in place by the body of the design entity to which the instance is bound.
- (c) Encapsulation: Encapsulate a piece of code into a separate design entity and use a design entity instance reference to instantiate this new design entity in place of the actual code.

- (d) Encapsulation: Encapsulate a piece of code into a separate procedure. Then, replace the code by a call node to this procedure.

### 3. Sequential vs Concurrent Style:

- (a) Map concurrent to sequential code.
- (b) Map sequential to concurrent code.

(1) The **grouping** and **flattening** transformations create or eliminate hierarchy *in place*. Consequently, no separate encapsulated piece of code, like, for instance, a procedure, is created that could be reused in more than one place of the design representation. One goal of these transformations is to organize the design representation in a manner that is more comprehensive to the human designer. Or, vice versa, to remove grouping constructs that were originally introduced by the human for readability reasons but that are no longer relevant for efficient synthesis. Another goal is the fact that groupings of operators may be used to indicate to the synthesis tools the relationships of statements. For instance, several VHDL statements may be used to describe the behavior of a single component [Rund90b].

(2) The **expansion** and **encapsulation** transformations are used if a piece of code is reused more than once in the design. This sharing of a piece of the design would mean that in the final structural design implementation one piece of hardware is synthesized separately and then utilized by different pieces of the design.

**Inline expansion** of procedure and function calls is done by replacing the CALL operator with copies of the actual procedure or function body, respectively. Inline expansion of design entity instances is done by replacing the design entity instance with a copy of the body of the referenced design entity. These transformations allow you to avoid micro-subroutine jumps since they create one flat flow graph structure. **Procedure elimination** can be performed when a procedure is no longer called from the current flow graph. The **procedure formation** transformation encapsulates a specified set of operators into a separate flow graph which then is called from the original flow graph via a procedure CALL node. This allows the reuse of the code in other designs.

(3) The last set of transformations is concerned with mapping concurrent descriptions to sequential descriptions, and vice versa. These transformations are useful if a design tool has assumption about the type of target hardware to be used for each given description style. VSS [Lis88], for instance, maps VHDL sequential process constructs into a control-unit/data-path design model and VHDL concurrent block constructs into a functional unit data path model. In this case, these transformations can be used to select among the target hardware style.

## Transformations at the control flow graph level

Transformations at the control flow graph level reshape the overall flow graph structure by mapping data flow portions into control flow blocks, and vice versa. The creation of larger data flow graphs out of control flow helps to uncover potential data parallelism, which can be exploited for the data path design.

1. Condition nesting:
  - (a) Combine nested conditional statements into one more complex condition.
  - (b) Decompose a complex condition into a set of nested conditional statements.
2. Condition manipulation:
  - (a) Move the condition evaluation data flow graph attached to a decision node into the preceding data flow graph.
  - (b) Or vice versa, attach the data flow graph that evaluates the condition directly to the decision node.
3. Control flow to Data flow translation:
  - (a) Transform a conditional statement expressed in control flow to a conditional statement expressed in data flow.
  - (b) Transform a conditional statement from data flow to a control flow representation.
4. Loop Treatment:
  - (a) Loop types: transform between while and repeat loops.
  - (b) Loop unwinding.
  - (c) Map a control flow loop into a data flow graph without unrolling.
5. Code motion:
  - (a) into: move a data flow graph into a conditional statement by moving it from below into bottom of each branch or from above into top of each branch.
  - (b) out of: move a data flow graph out of a conditional statement by moving it from the bottom of all branches of the conditional statement below the condition, or by moving it from the top of all branches of the conditional statement above the condition.

## 6. Clean-up:

- (a) Combine two consecutive data flow blocks into one data flow block.
- (b) Split a straight-line data flow block into two data flow blocks.
- (c) Clean-up: Reduce a constant conditional expressed in control flow.

(1) The first group of transformations deals with **nested conditional statements**. The first transformation **combines nested conditional statements**, such as nested CASE or IF statements, into a complex conditional statement. The second transformation does the reverse, i.e, it **decomposes a conditional statement** into a several nested conditions. An example application of this is shown in the Figures 12.16 and 12.17. These transformations trade off the size of conditional tests for the number of these tests. This has direct influence on the control unit design, since it determines the width of the condition word. If several nested conditional statements are combined into one more complex condition, then the resulting design will probably have a wider control word. If a complex condition is decomposed into a set of nested conditional statements, then there is a need for a condition multiplexor. However, the control word of the design will be smaller.

The **condition-combine** transformation combines nested conditional statements, such as nested case- or if-statements, into one complex conditional statement. An example application of this transformation is shown in the Figures 12.16 and 12.17. Figure 12.16 shows the VHDL description and Figure 12.17 the flow graph representation both of the initial and the modified design. This transformation requires a move of the statement block with the statement "flag=1" into the second conditional statement. Now, the two decision nodes based on the conditions A and B are adjacent and can be combined into one by concatenating the conditions A and B into the complex condition A&B and their corresponding test values. The **condition-decompose** transformation does the reverse, i.e, it decomposes a conditional statement into a several nested conditions as shown in Figures 12.16 and 12.17.

(2) The **condition manipulation** transformation deals with the type of conditions, i.e., whether a condition is a simple comparison of bits or whether it corresponds to the evaluation of an expression. They thus have a direct effect on the design style of the resulting hardware.

The conditions of the first type result if there is only a simple comparison associated with a decision node in the control flow graph. For this transformation moves the data flow graph associated with the decision node to describe the condition evaluation into the preceding data flow graph. For this design style, the synthesized

```

case A is
  when 0 => flag:=0;
  when 1 => flag:=1;
  case B is
    when 0 => X:=X+1;
    when 1 => X:=X-1;
  end case;
end case;

```

```

case A&B is
  when 00 => flag:=0;
  when 01 => flag:=0;
  when 10 => flag:=1; X:=X+1;
  when 11 => flag:=1; X:=X-1;
end case;

```

a. Nested CASE Statement.

b. CASE Stmt with Complex Test.

Figure 12.16: VHDL Description of Conditional Statements.

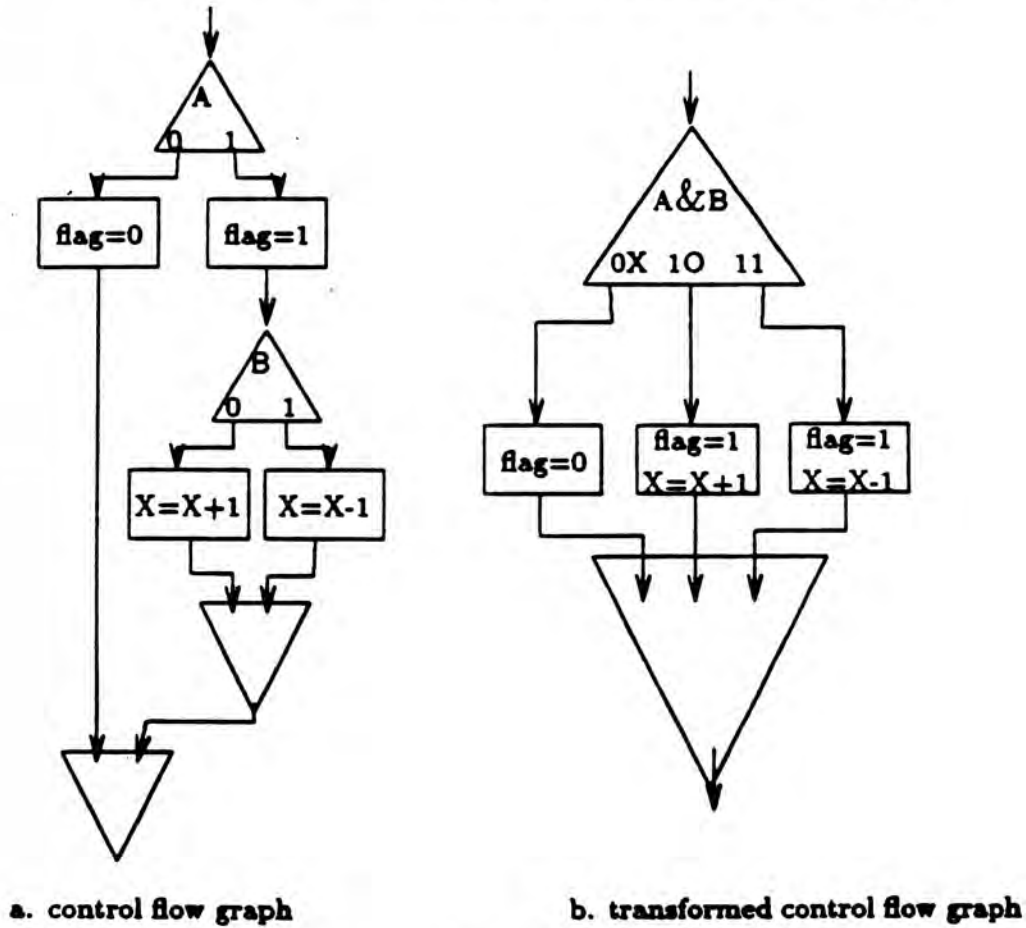


Figure 12.17: The Condition-Decompose Design Transformation.

control unit only has to check condition flip-flops at the beginning of a clock cycle to determine the result of a condition evaluated in the previous state. Vice versa, if the data flow graph that evaluates the condition is directly associated with the decision node, then no condition flip-flops are needed in the structural implementation. This however would increase the clock cycle, since the condition has to be evaluated at the beginning of the clock cycle and thereafter actions are being done based on the result of the condition evaluation.

(3) The third set of transformations deals with the mapping of a conditional statement expressed in **control flow** to a conditional statement expressed in **data flow**, and vice versa. A VHDL description and a flow graph representation of each are shown in Figures 12.18 and 12.19. The figures show that the data flow blocks of the different conditional branches have to be merged into a single larger block by the insertion of choose-value nodes (depicted by a triangle node in Figure 12.19.b). For each variable updated within at least one branch of the conditional statement a choose-value node is created. These choose-values data flow nodes model the condition and assure that the correct values are passed on. In the example of Figure 12.19.a, there are two statement blocks,  $S1 = \{X := X + 1\}$  and  $S2 = \{X := X - 1\}$ . Since they both update the same variable,  $X$ , one choose-value node has to be introduced in the corresponding data flow graph in Figure 12.19.b. The condition that guards this choose value node is gotten from the decision node of the control flow graph, namely, the test of the variable  $A$  and  $B$ . Similarly, there are two statement blocks  $S3 = \{flag := 1\}$  and  $S4 = \{flag := 0\}$ , which both update the same variable,  $flag$ . Therefore, a second choose-value node has to be introduced in the corresponding data flow graph in Figure 12.19.b for the variable  $flag$ . It is guarded by the variable  $A$ .

(4) The next set of transformations is concerned with **loop constructs**. First, one can bring the condition of a loop from the beginning to the end of the data flow block by copying it. This corresponds to replacing a while-loop by an if-statement followed by an until-loop. The reverse can also be done, namely, to move a condition of a loop from the bottom of the loop to the top of it.

A loop is unrolled by one iteration by duplicating the body of the loop once and by reducing the condition count of the control flow loop by one. This transformation can be repeatedly executed to unroll a loop completely by replacing the control flow information by one large data flow graph assuming that the loop has a fixed bound. Such an unrolled loop may produce more efficient schedules since potential parallelism can be exposed in the resulting data flow graph. However, this may result in an explosion of micro-code, and hence a waste of controller area.

```

case A is
  when 0 => flag:=0;
  when 1 => flag:=1;
  case B is
    when 0 => X:=X+1;
    when 1 => X:=X-1;
  end case;
end case;

```

```

with A&B select
X <=
  X+1 when 10,
  X-1 when 11,
  X    when others;
with A select
flag <=
  0 when 0,
  1 when 1;

```

a. Behavioral Description Style

b. Dataflow Description Style

Figure 12.18: Conditional VHDL Descriptions.

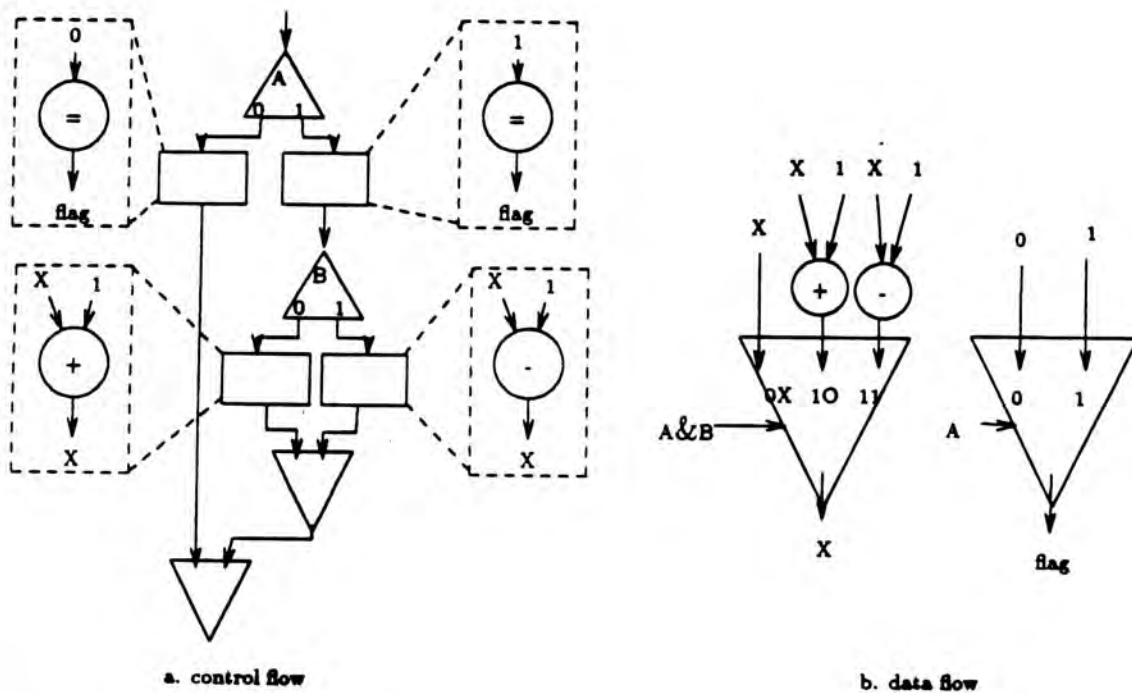


Figure 12.19: The Control-To-Data-Flow Design Transformation.



(5) **Code motion** transformations address the movement of code into or out of conditional statements that are expressed in data flow. They generally are used in order to prepare the graph for additional transformations. An operator can be moved up from below a conditional statement into the bottom of each branch of the statement or from above a conditional statement into the top of each branch of the statement. The other two options of moving code out of a conditional statement are also possible if the code is in all branches of the statement.

(6) The last set of data flow transformations is useful to **clean up** the graph after the application of other transformations. The first two transformations work on blocks of data flow graphs. They are likely to be used to prepare the flow graph for any of the other transformations or to clean up after other transformations have created consecutive data flow graphs.

The transformation 6.a combines two consecutive data flow blocks into one data flow block. For this, it is required that the outputs of the first block are connected with the inputs of the second one. This transformation will be used for the **condition-manipulation** and **condition-nesting** transformations.

The transformation 6.b is the reverse of the first, i.e., it splits a straight-line data flow block into two data flow blocks. This may be useful for scheduling since the values that need to be saved between the two data flow blocks are being isolated. This transformation will be used together with the **condition-manipulation** and **control-flow to data-flow** transformations.

The transformation 6.c removes a condition which is based on a constant. Such a condition can be statically evaluated and therefore is redundant. An example of a constant conditional is "if (1) then X". This statement is equivalent to "X" and therefore can be reduced accordingly.

### **Transformations at the data flow graph level**

The following transformations are applicable within a data flow graph. Thus they generally do not change the overall structure of the control flow graph. They are similar to optimizations found in optimizing compilers.

1. Expression/Constant manipulation:

- (a) Merge common subexpressions or eliminate redundant subexpressions.

- (b) Reorganize an expression tree based on the commutativity and associativity law to isolate common subexpressions and for expression height reduction.
  - (c) Constant manipulation: Constant folding/propagation and the reduction of constant conditionals.
  - (d) Dead-code elimination: Eliminate unused or dead operators or variable references that do not affect the output of the routine.
2. Condition nesting:
- (a) Combine nested conditional statements into a complex condition.
  - (b) Decompose a complex condition into a set of nested conditional statements.
3. Code motion:
- (a) into: move from underneath a choose value node to all branches above it.
  - (b) out of: if a piece of code appears in all branches above a choose value node then move it below the node.
4. Hierarchy Manipulation:
- (a) Flatten the hierarchy by function inline expansion.
  - (b) Flatten the hierarchy by replacing a decoder node by a data flow graph that represents the associated truth table captured by the decoder node
  - (c) Create an extra hierarchy level by substituting an expression data flow graph by a higher-level function node that encapsulates the data flow graph.
  - (d) Create an extra hierarchy level by encapsulating a subgraph into a function body and replace this subgraph by a call node to this function.
5. Customization for Synthesis:
- (a) Data type conversion from integer and enumeration types into a binary representation.
  - (b) bit width consistency achieved by padding/selection of bits.
  - (c) Replace a complex or irregular choose value node. by a simple choose value node (selector) and an associated decoder node
  - (d) Timing constraints: distribute a timing constraint for a group of operations into timing constraints for each individual node.

(1) The first set of transformations is similar to optimizations found in optimizing compilers. Most of them restructure the data flow graph in order to remove or reduce redundant information. Expression height reduction transformations are performed to increase the number of operations that can be executed in parallel rather than serially. Switching the inputs of operators, if commutative, or rewriting an expression based on the associativity law may, for instance, be useful during resource allocation. Exchanging inputs of an operator may result in reuse of already existing wires without insertion of a new multiplexer. Then, it saves additional connectivity costs.

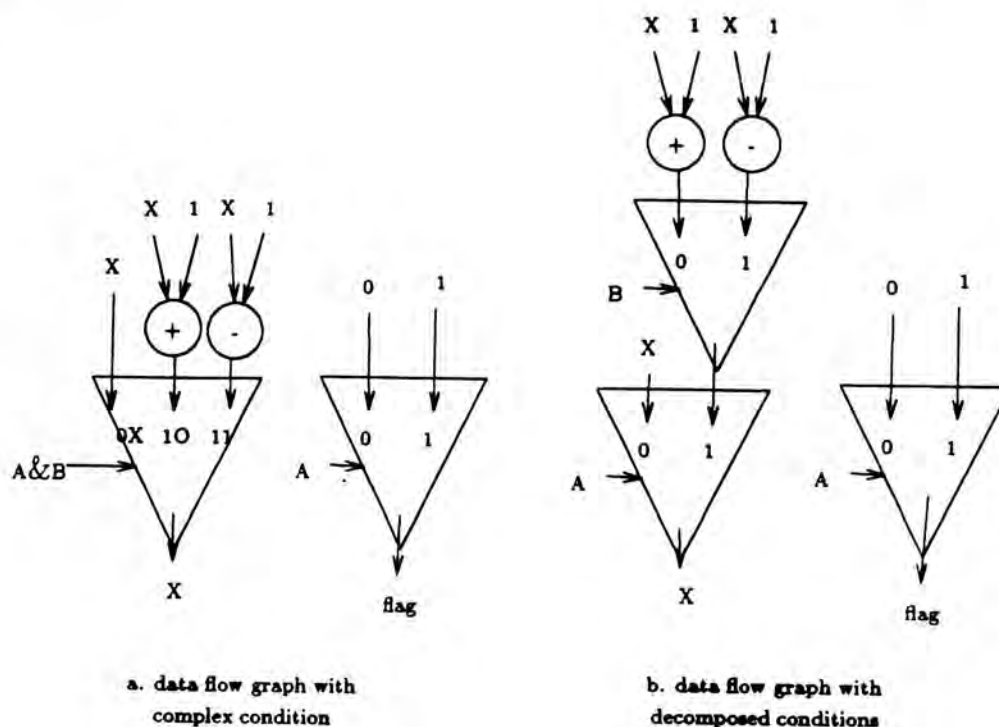


Figure 12.20: The Condition-Nest Design Transformation.

(2) The second set of transformations deals with **condition-nesting** at the data flow graph level. It either combines nested conditional statements into a complex condition, or it decomposes a complex condition into a set of simpler nested conditional statements. (represented by a sequence of choose value nodes). An example of **condition-nesting** type of transformation is given in Figure 12.19. Figure 12.19.a shows a complex condition expressed by a choose value node with a concatenation of two condition guards. After the **condition-nest** transformation is applied, the data flow graph in Figure 12.19.b is generated. The complex condition

is decomposed into two simpler conditions represented by a sequence of two choose value nodes each with one condition guard only.

(3) **Code motion** transformations can take on many forms. One can reorder data flow operators by for instance moving them into or out of a conditional statement. An operator can be moved up from below a conditional statement into the bottom of each branch of the statement. An operator can be moved out of a conditional statement if the same piece of code appears in all branches above a choose value node.

(4) **Hierarchy manipulation** at the data flow graph level can be done by function creation and elimination and by decoder creation and elimination. A function is a mechanism for encapsulating several constructs into one place. It allows for the reuse of this piece of code. The decoder nodes are special-purpose functions that capture truth-table information, i.e, boolean equations that control the execution of an operator node or a choose value node (see [Rund90b]). These decoder data flow nodes can directly be synthesized into a random logic component. Therefore there is often no need to expand the content of a decoder node into the data flow graph using numerous "AND" and "OR" operator nodes.

**Flattening of a hierarchy** can be done by **function inline expansion**. If the function body is a data flow graph then the function call node can be directly replaced by the function body. However, if the function body is a control flow graph then we first have to map the function body from control flow to data flow. Thereafter we can replace the function call node by this data flow graph. Flattening of a hierarchy can also be done by replacing a decoder node by a data flow graph that describes the information captured by the decoder node.

An extra level of hierarchy can be created by substituting an expression data flow graph in place by a higher-level function node that encapsulates the data flow graph. Similarly, we can create an extra hierarchy level by encapsulating a subgraph into a function body and by replacing this subgraph by a call node to this function. The first is an example of grouping in place and the second of encapsulation for reuse (See Section 12.5.2).

(5) The last set of transformations are particular to synthesis. They prepare the design representation of the behavioral information such that a mapping into a "good" implementation structure can be achieved. Examples are the conversion of all data types into bit strings or the padding of bits to variable accesses to assure that the inputs of an operator are of identical width.

### 12.5.3 A Comprehensive Example of Design Transformations

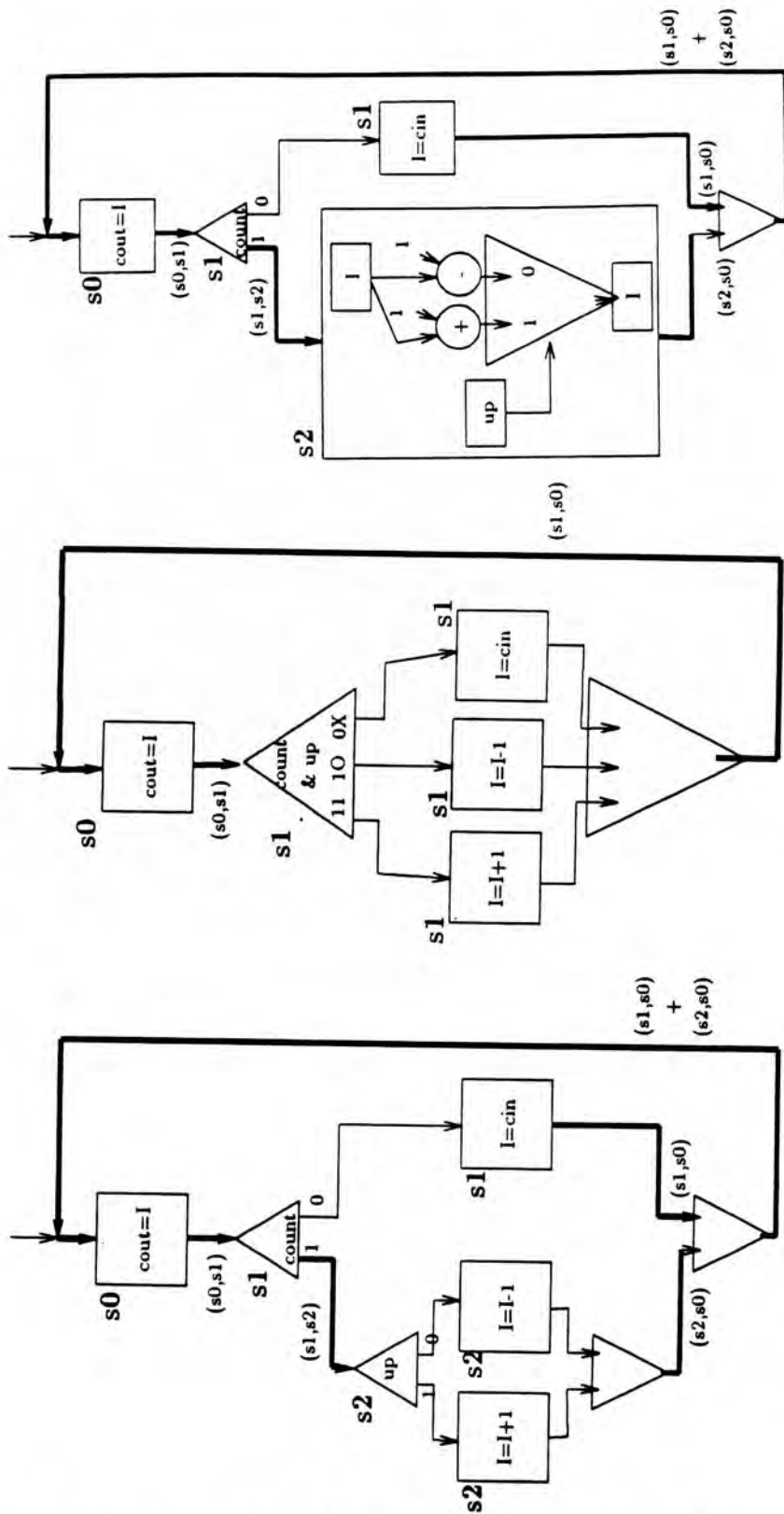
In this section, we give a comprehensive example to show how a sequence of design transformations can be used to restructure a control/data flow graph description of a design. The example used throughout this section is the behavioral VHDL specification of the programmable up-and-down counter example given in Figure 12.3. The ECDFG design representation of this VHDL specification is depicted in Figure 12.21.a. In this section, we show how different design representations can be created for this specification by applying design transformations.

First, we describe how the two control flow graphs shown in Figure 12.21.b and 12.21.c are created from the one in Figure 12.21.a. The flow graph in Figure 12.21.b has been generated from Figure 12.21.a by merging the two decision nodes into one more complex decision node. This corresponds to the transformation “**condition-combine**” transformation that is introduced in Section 12.5.1 and is depicted in Figure 12.17.

The flow graph in Figure 12.21.c has been generated from the one in Figure 12.21.a by moving the second decision node from control flow to data flow. This entails the merging of the two statement blocks that contain a data flow graph for “ $I:=I+1$ ” and “ $I:=I-1$ ”, respectively, into one statement block with a combined data flow graph. This corresponds to the **control-to-data-flow** transformation, which is described in Section 12.5.1 and depicted in Figure 12.19.

The shape of the control flow graphs in Figure 12.21 differs on the number of tests per decision node. The control flow graphs in Figure 12.21.a and 12.21.c both have one test variable per decision node, whereas the control flow graph in Figure 12.21.b has two test variables per decision node. The number of tests to be done concurrently determines the width of the control status lines (the control register). Therefore, one representation may be preferred over the others depending on the desired design style.

Once a flow graph has been updated with state assignment, then a BIF table can be generated from its representation [Dutt90]. The ECDFG graphs shown in Figures 12.21.a, 12.21.b, and 12.21.c are annotated with state information, i.e., each control node is associated with a state. Thus, sufficient information is available to generate the corresponding BIF tables. The result of this extraction for the three designs in Figure 12.21 is shown in the Figure 12.22.



a. control flow graph

b. transformed control flow graph

c. transformed control flow graph

Figure 12.21: ECDFG Graphs for the Counter Example.

present state	condition	(value)	actions	next state
0	-	TRUE	countout = 1	1
1	count=1	TRUE	-	2
		FALSE	I = countin	0
2	up=1	TRUE	I = I + 1	0
		FALSE	I = I - 1	

a. Operation-Based State Table For the Flowgraph in Figure 12.21a.

present state	condition	(value)	actions	next state
0	-	TRUE	countout = 1	1
1	count&up	10	I = I - 1	0
	count&up	11	I = I + 1	
	count&up	0X	I = countin	

b. Operation-Based State Table For the Flowgraph in Figure 12.21b.

present state	condition	(value)	actions	next state
0	-	TRUE	countout = 1	1
1	count=1	TRUE	-	2
		FALSE	I = countin	0
2		TRUE	if (up=1) I = I + 1 else I = I - 1	0

c. Operation-Based State Table For the Flowgraph in Figure 12.21c.

Figure 12.22: Operation-Based BIF Tables for Counter ECDFG Graphs.

Some design tools require data flow graph representations rather than the hybrid control/data flow graph representation. Since they work only on a design that is completely expressed by data flow constructs. To accomplish this for the given example design, the designer would have to apply design transformations that map control to data flow. Hence, the **control-to-data-flow** transformation (defined in Section 12.5.1 and depicted Figure 12.19) is applied to the graph shown in Figure 12.21.c. The result of this design transformation, one single data flow graph, is described in Figure 12.23.

Again a BIF table can be generated from this restructured design and is shown in Figure 12.24. The data flow graph contains two levels of choose value nodes. (A choose value node is a decision node in data flow depicted by a triangle). Thus, this design representation corresponds to a BIF table with conditions in the action column as can be seen in Figure 12.24. They demonstrate the conditional actions that are associated with the state.

If a designer is interested in a BIF table of the counter design that does not contain nested conditional actions, then further design transformations have to be applied. In this case, the transformational subsystem would transform the data flow graph shown in Figure 12.23 by merging the choose values nodes into one choose value node. The latter is guarded by a more complex condition as shown in Figure 12.25. The design transformation used for this restructuring, called **condition-unnest** transformation, is explained in Section 12.5.1 and also graphically displayed in Figure 12.20. Given the modified data flow graph in Figure 12.25, it is now possible to generate a BIF table without any nested conditional actions. The resulting BIF table is presented in Figure 12.26.

## 12.6 Design Tasks in a Behavioral Synthesis Environment

Figure 12.27 details the example VHDL synthesis system. The design tools are depicted on the left-hand and the Design Data Representation Graphs maintained by BDDDB on the right-hand side of Figure 12.27. In the middle of Figure 12.27 we describe the type of design data exchanged between the design tools and the Behavioral Design Object Model. Each tool expects certain predefined design view for the different design data chunks it consumes and produces.



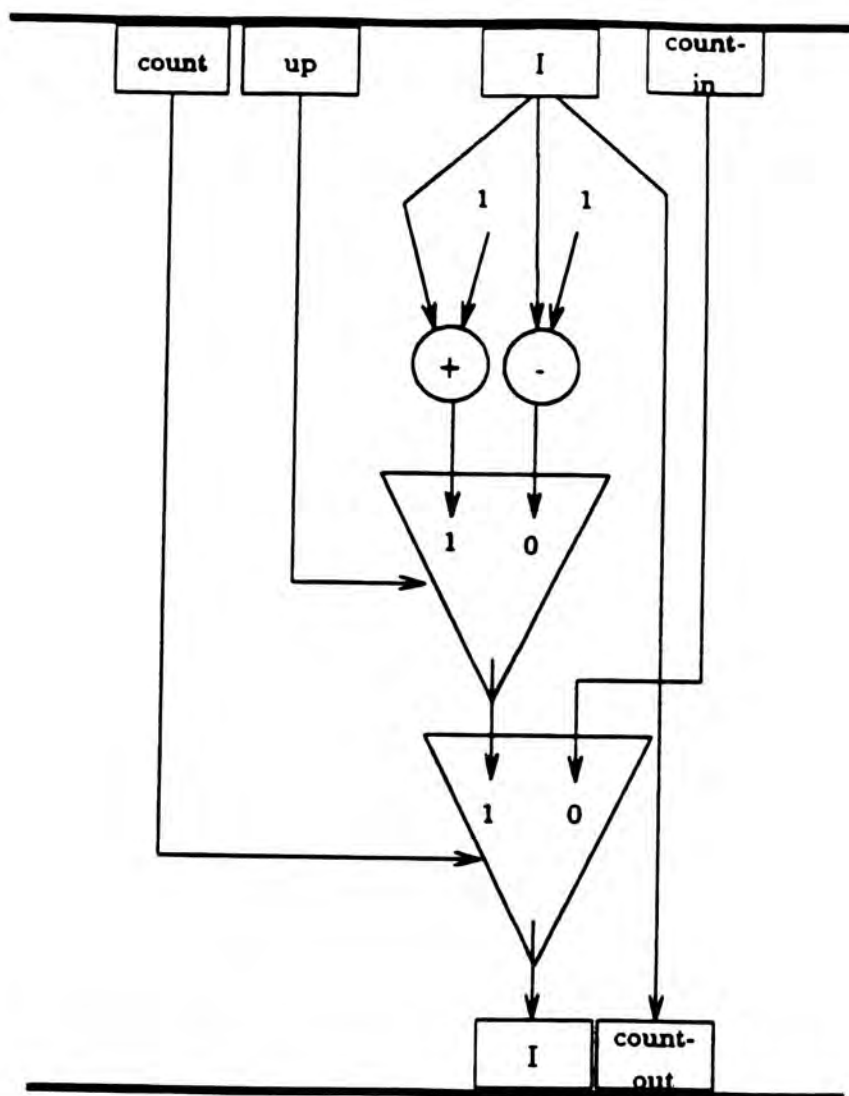


Figure 12.23: DFG with Nested Conditions for the Counter Example.

present state	condition	(value)	actions	next state
1	-	TRUE	countout=1 if (count=1) then if (up=1) then I = I + 1 else I = I - 1 endif else I = countin; endif	j

Figure 12.24: A BIF Table for the DFG with Nested Conditions.

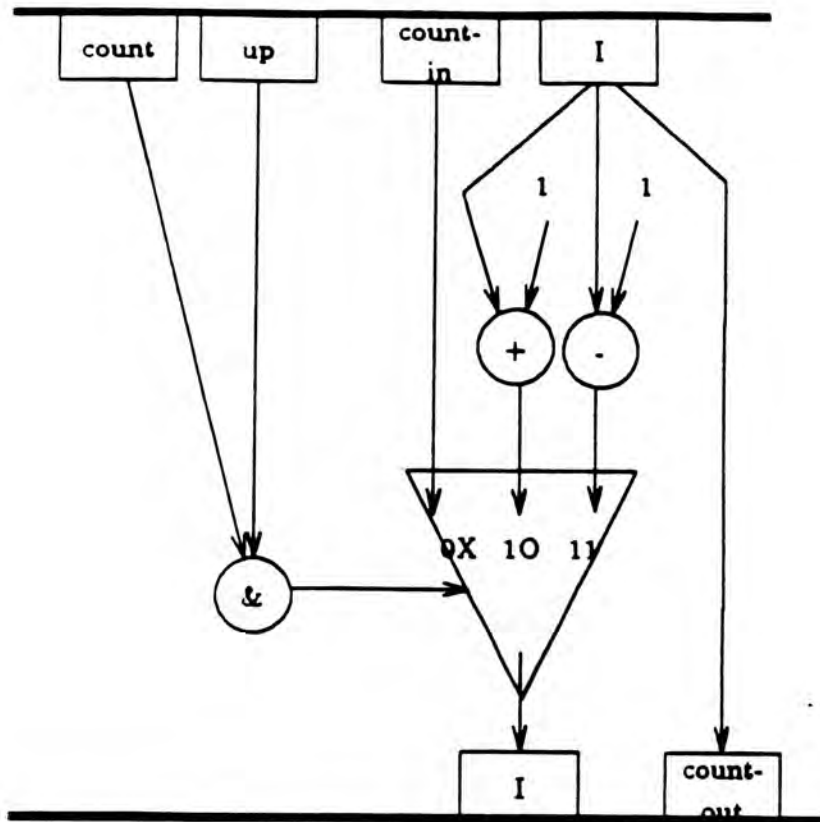


Figure 12.25: DFG with One Complex Condition for the Counter Example.

ps	condition	(value)	actions	ns
i		TRUE	if count&up=10 { countout=I I = I - 1 } else if count&up=11 { countout=I I = I + 1 } else if count&up=0X { countout=I I = countin }	j

Figure 12.26: A BIF Table for the DFG with One Complex Condition.

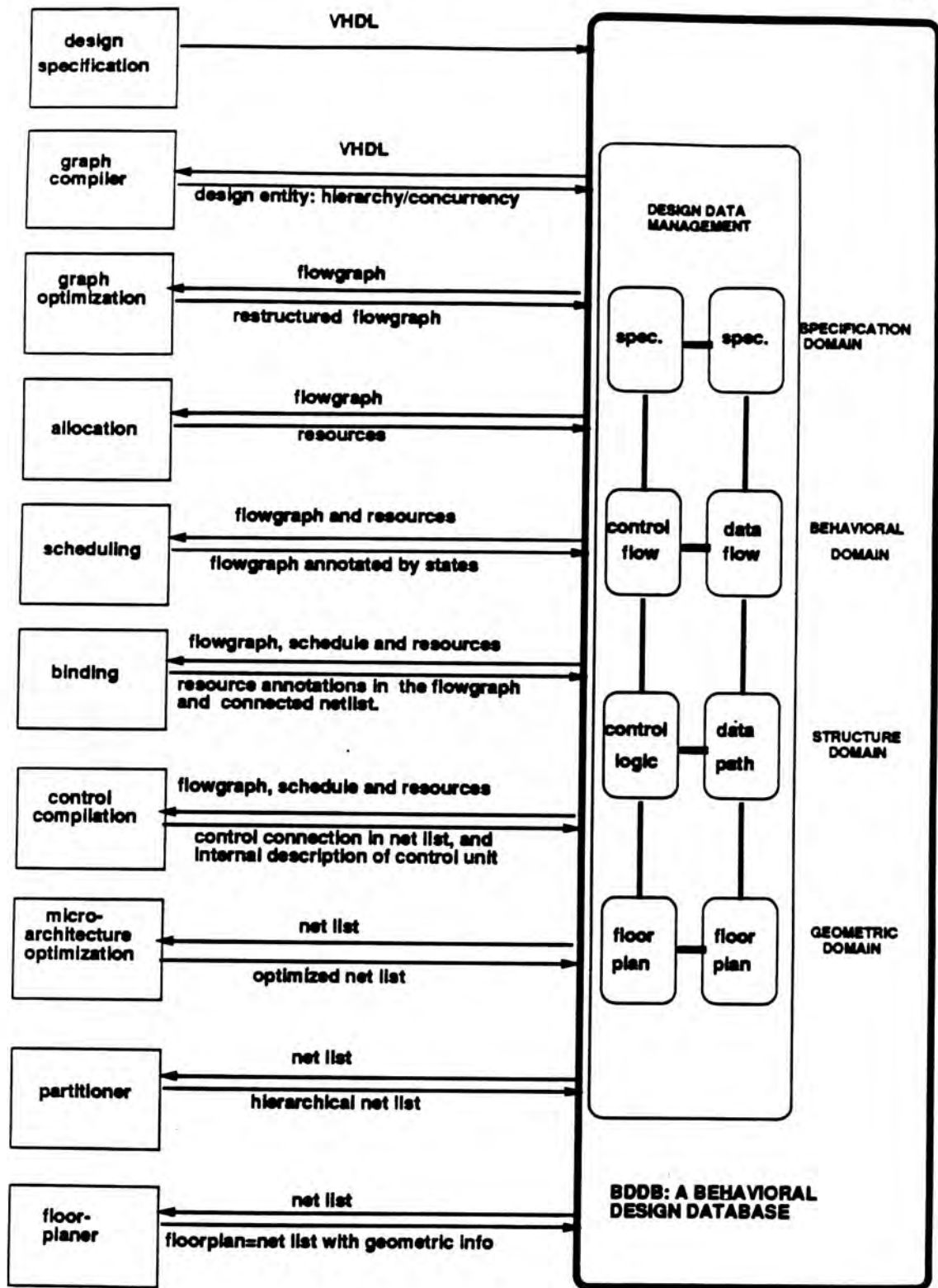


Figure 12.27: A VHDL Design Synthesis Environment

The VHDL Graph Compiler, for instance, expects a design specification written in VHDL and generates one or more design entities in the form of Extended Control/Data Flow Graphs (ECDFG).

The Graph Optimizer checks out one ECDFG, optimizes it using graph transformation routines, and finally places the reorganized ECDFG back into the database.

The Allocator accepts an ECDFG, allocates a set of resources required for the design, and returns the allocated set of resources to BDDDB in the form of an Annotated Component Graph (ACG) [Rund90c, Rund92e]. This new ACG design entity is incomplete, since it contains components but no registers or connections between components.

The Scheduling Tool accepts an ECDFG and the initially allocated ACG as input and produces an ECDFG augmented by a State Transition Graph as data output.

The Binder uses the Scheduled Flow Graph produced by the Scheduler and the initial set of resources allocated by the Allocator. The result of the binding design process is both a modified flow graph as well as a new component graph. The former is annotated by component allocation information, and the later is extended by connection objects, such as buses, multiplexers and wires.

The Control Compiler uses the information generated by the Binder to generate the control logic for the control unit component, i.e., a set of equations that control the execution of all components in the ACG for each state in the ECDFG.

The Micro-Architectural Optimizer regroupes the ACG by decomposing a complex component into simpler components, or vice versa, by merging a group of units into a more complex unit based on a component library. The result of this optimization is a newly structured ACG.

The Partitioner then prepares the ACG for floor-planning by grouping components that should be laid out together into groups. For instance, it may group all gates and un-sliceable units into a random logic partition and all bit-sliceable units, such as registers, ALUs and counters, into the stack group. The resulting ACG is partitioned into different groups of components, i.e., it is a hierarchical ACG.

Lastly, the Floor-Planner places the groups of components onto a chip. For this, the components in the ACG will be assigned geometric information, such as their relative position on the chip, their approximate area, and the position of all pins.

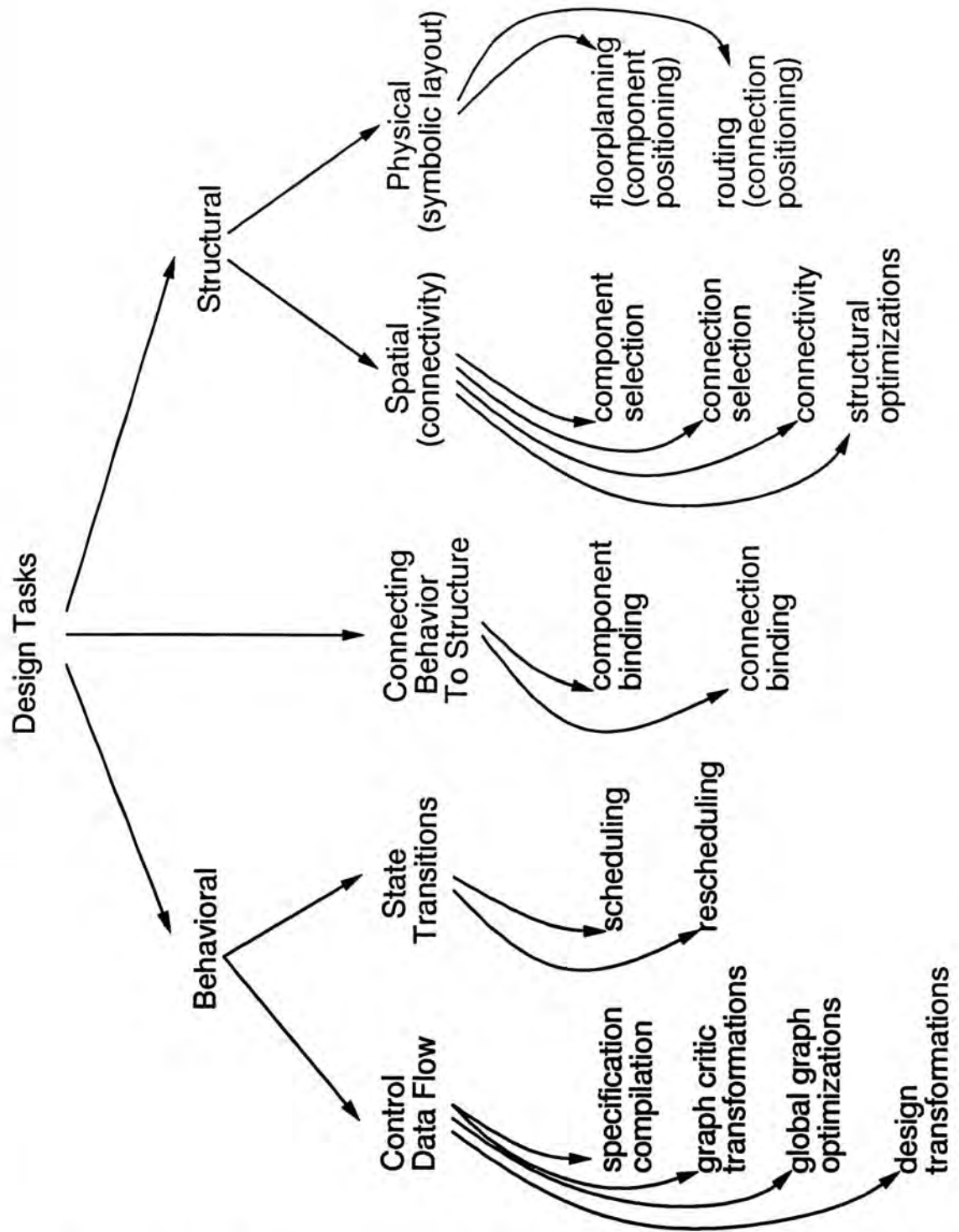
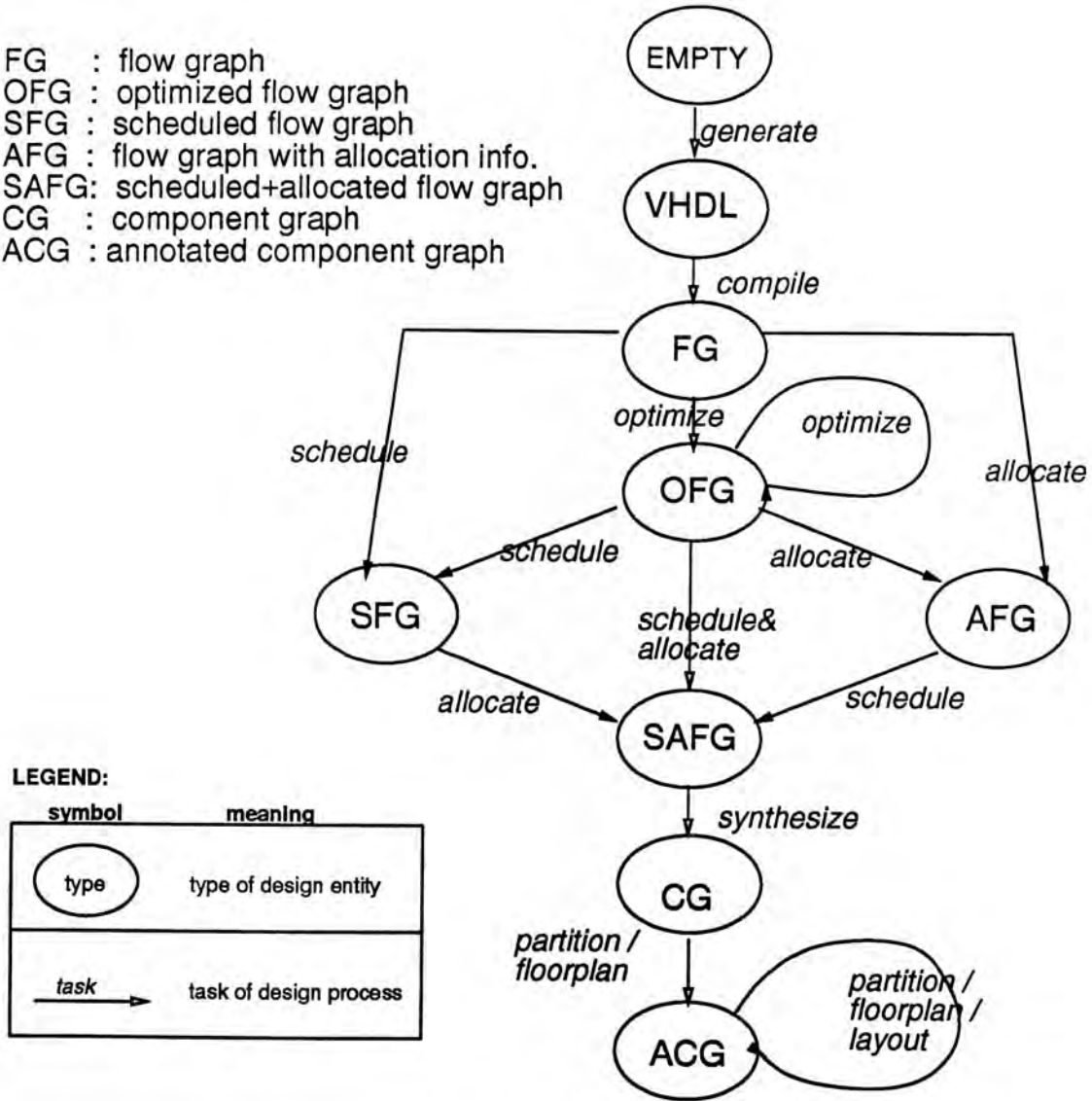


Figure 12.28: A Classification of Behavioral Synthesis Design Tasks.

FG : flow graph  
 OFG : optimized flow graph  
 SFG : scheduled flow graph  
 AFG : flow graph with allocation info.  
 SAFG: scheduled+allocated flow graph  
 CG : component graph  
 ACG : annotated component graph



**LEGEND:**


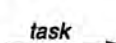
symbol	meaning
	type of design entity
	task of design process

Figure 12.29: Design Flow Information.

We have identified the design flow information for this example application, namely, the flow of design *tasks* and related design data *types* and formats depicted in Figure 12.29. Implicitly, this flow diagram shows some of the *semantic* relationships that are created between different design entities in the Design Entity Graph Model. It can therefore be used to determine which relationship to place between the design entity checked-out by a particular design tool and the corresponding design entity checked-in by the same tool.

Next, we describe an example of tool interaction with the design database using Figure 12.30. The ovals in the middle of Figure 12.30 correspond to *tool interfaces* in the form of check-in and check-out requests.

In this example scenario, we assume that each tool invocation successfully generates a new design entity version. The first design transaction creates a new design entity in the database with the name "X1" using the database command "create\_DD". BDDB will therefore prepare the Design Entity Graph to include a new design entity. This design entity is at first empty. The VHDL Generator (i.e., a designer) then will generate a VHDL specification, which is checked into the database as a first version of this new design entity "X1". The type of this design entity is "VHDL". Thereafter, the second design transaction checks out the previously created design entity and invokes the Graph Compiler on the corresponding design data. The Graph Compiler generates the design representation for the VHDL specification, i.e., an Extended Control/Data Flow Graph. The result of this design process is checked back into the design database as a new version of type "ECDFG" using the command "checkin\_DD".

BDDB checks the consistency of these design transactions using the design flow information in Figure 12.29. For instance, it will check that a design tool of the *task* group "compile" that accepts as input a *data* of type "VHDL" must produce as output a *data* of type "FG". Once the input and output formats of the transaction are confirmed, then the newly checked-in design entity is related to the previously created design entity by means of a *version-of* relationship. This explains how the design entity graph information maintained by BDDB is gradually modified with each tool invocation (see right-hand side of Figure 12.30.) This process of checking out the most up-to-date version of the design "X1" and of creating a more detailed version is repeated by most design tools in this example.

Finally, the Netlist Generator checks out a design entity that consists of a scheduled and allocated flow graph design representation. It returns a design implementation of the design in terms of an Annotated Component Graph. This newly checked-in design entity belongs to the structural domain. Hence, an *equivalence*

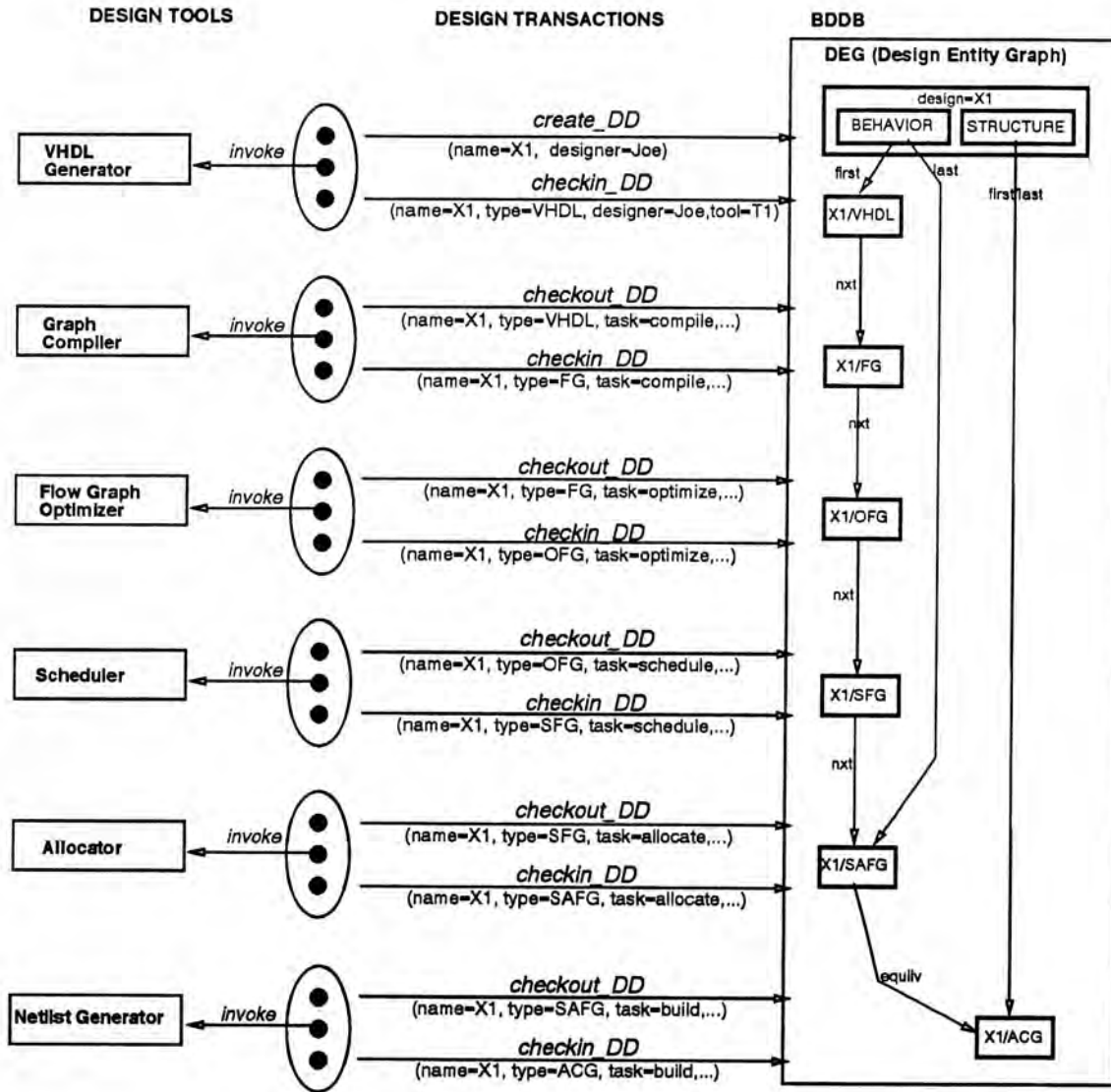


Figure 12.30: Design Tool Interactions with BDDB.



rather than a *version-of* relationship is inserted between it and the previous design entity. The result of the set of design transactions shown in Figure 12.30 is the design entity graph for design "X1" on the right-hand side of Figure 12.30.

# Chapter 13

## Specifying Behavioral Design Views Using MultiView

### 13.1 An Introduction to Design Views

In this chapter, I present a number of design view examples for behavioral design tools constructed using the *MultiView* methodology. The goal of this work is (1) to demonstrate the usefulness and power of the view paradigm, and (2) to present a solution to the tool integration problem existing in most behavioral synthesis systems. All of the design views specified in this section is defined on top of the uniform behavioral design object model presented in Section 12. Each design view is targeted to the requirements of one of the design tasks found in most behavioral synthesis environments. A detailed description of these behavioral design tasks can be found in Section 12, and therefore is omitted in this chapter.

The remainder of this chapter is structured as follows. Section 13.2 presents a design view for the floorplanning design task, while Section 13.3 describes the construction of a design view for the (operator-) binding design task. The next two sections then describe how *MultiView* can be utilized for information hiding purposes. In particular, Section 13.5 demonstrates how the data flow graph representation can be (virtually) simplified by hiding and redefining object classes of the graph structure. Section 13.4 discusses how different details of timing constraints can be hidden from the Control/Data Flow Graph representation.

## 13.2 Design Views for the Floorplanning Design Task

In this section, I first describe the floorplanning design task. In particular, I discuss the information requirements of the design task i.e., what types of objects need to be accessed and what type of operators need to be supported. Then I present incremental steps for refining the global object model such as to suit the requirements of the *floorplanning* design task.

### 13.2.1 The Floorplanning Design Task

*Floorplanning* positions the components in the component graph onto a given grid, called symbolic layout. The goal is to minimize the overall area of the symbolic layout while avoiding overlap of components and of minimizing the delay through the chip.

As can be seen in Figure 12.28, the *floorplanning* design task does not need design information from the behavioral graph, the state transition graph, or even the binding between behavior and structure. Therefore, the parts of the global schema modeling the behavioral domain and the linkage across domains must be hidden from this view. In the behavioral design object model, the linkage from the components to the data flow nodes is uni-directionly, namely, from the data-flow nodes to the components, but not back. Then there are no references in the component graph to objects in the behavioral graph. Therefore, the manipulation of the component graph for removing such references to isolate it from the behavioral graph is not needed. Therefore, this part of the view construction can be accomplished simply by not adding these behavioral object classes to the floorplanning view in the view specification script. Therefore, the remainder of this discussion can now focus on the component graph portion of the global schema depicted in Figure 13.1.

Even though the *floorplanning* tool works exclusively in the structural domain, i.e., the structural subgraph of the global schema, it is not allowed to modify characteristics of existing components and their connections (besides floorplan-related ones) nor the connectivity among them. Therefore, the design view for the floorplanning design task must hide all such structure-related operations from the component graph. Examples of illegal operations (that is, illegal in the context of floorplanning) are the insertion or deletion of components, the establishing or removing of connections between components, and the change of component types or characteristics.

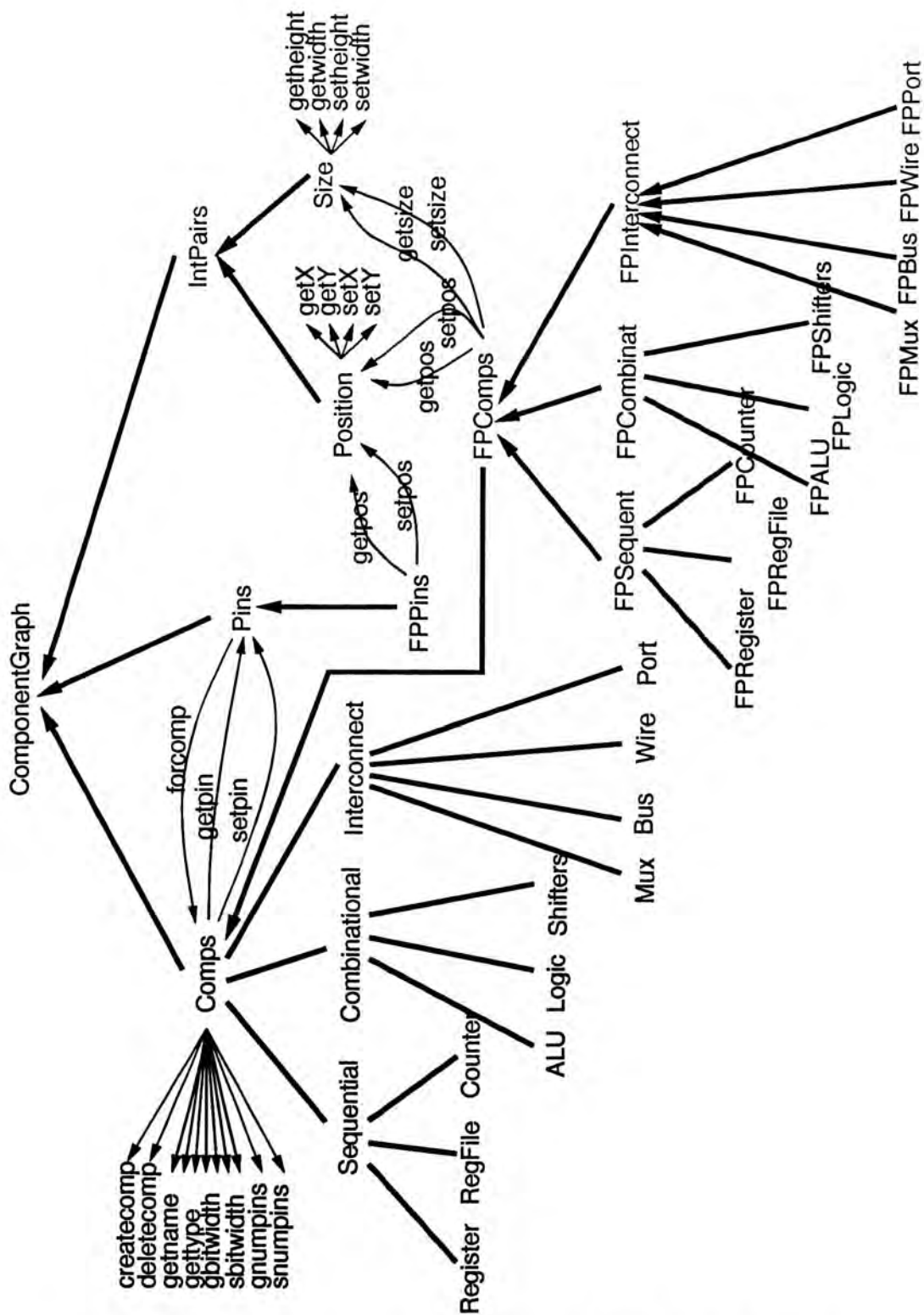


Figure 13.1: The Component Graph Portion of the Global Model.

Examples of legal operations, on the other hand, are the repositioning of components and their connections.

In our design model, the positioning of components is taken care of by modifying floorplan-related attributes of components, such as the X-coordinate and the Y-coordinate attributes. The *floorplanning* design view must thus allow for the manipulation of these attributes, while protecting all other objects in the design from (incorrect) manipulation. For simplicity, I make the assumption that the orientation of components can be ignored.

### 13.2.2 Grouping of Objects

Floorplanning is concerned with the placement of 'regular' register-transfer level components, but not with the placement of wires and interconnection units. Assume that the floorplanning tool, for which I am constructing this design view, is able to determine the placement of both 'regular' RTL components as well as multiplexors. In this case, one may be interested in designing an object class that holds all 'placeable' target objects. In *MultiView*, this can be accomplished with the **union** operator as shown below:

```
class FP1 := union(FPSequential,FPCombinational,FPMux);
```

As defined in Section 5.2, the **union** operation generates a virtual class named **FP1** that collects all object instances that belong to the classes **FPSequential**, **FPCombinational**, and **FPMuxes**. The type description of this new class is equal to the intersection of the functions in the type descriptions of the three source classes. In this example, this is equal to all functions supported by the **FPComp** class.

Next, *MultiView* integrates the virtual class **FP1** into the global schema. Since no new type was generated, the simple positioning of the **FP1** class using the class placement algorithm described in Section 7.4 is sufficient. By definition, **FP1** must be placed above the classes **FPSequent**, **FPCombinat**, and **FPMux**. Since **FPComp** is a superclass of all three classes, **FPComp** will also be a superclass of **FP1**. Lastly, **FP1** cannot be a superclass of **FPInterconnect**, since **FP1** contains only a subset of the object instances that belong to the **FPInterconnect** class. The *MultiView* class integration algorithm therefore places **FP1** directly below **FPComp** and directly above **FPSequent**, **FPCombinat**, and **FPMux**. The result of this class integration into the global schema is shown in Figure 13.2.

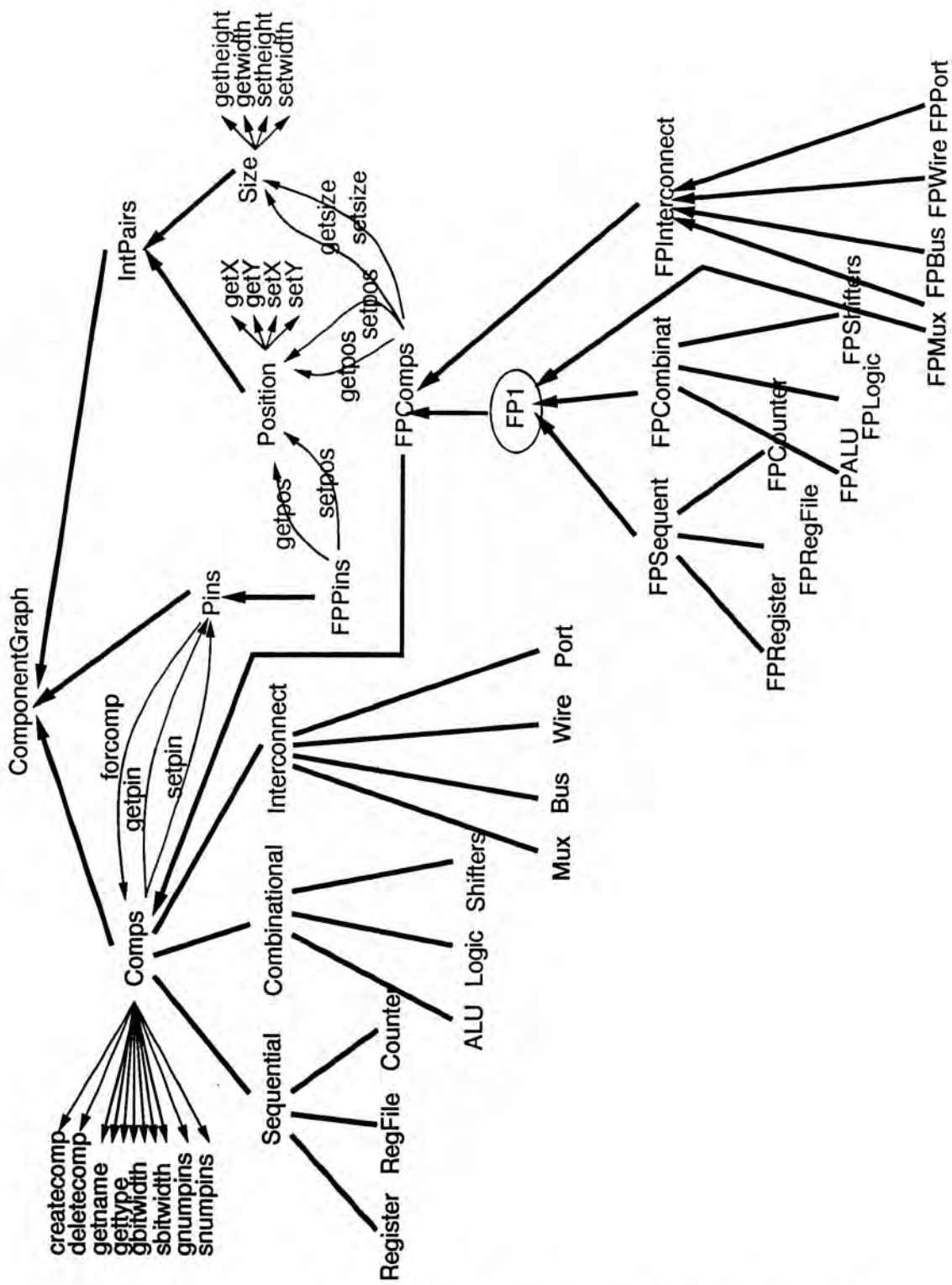


Figure 13.2: The Global Schema after Integration of the union class **FP1**.

### 13.2.3 Removing of Structure Manipulation Operators

In the design object model presented in Figure 13.2, the component graph classes with floorplan-related attributes are modeled as a specialization of the (unplaced) component graph classes.

For instance, the **FPSequential** class has all basic methods required for the manipulation of the floorplan-related attributes of sequential components. However, being a subclass of the **Comp** class, it also has all general methods of the component graph, such as the operators for adding or deleting components from the structure graph. As stated earlier, the floorplanner is not allowed to modify the topology of the design, therefore these operators should be removed from the target class in the floorplanning view. In *MultiView*, the view definer can use the **hide** operator to remove the operators from the target class **FP1**.

```
class FP2 := hide [createcomp, deletcomp, sbitwidth, snumpins, setpin]
from FP1;
```

The resulting class **FP2** has the same collection of object instances as the virtual class **FP1**, i.e.,  $\text{contents}(\mathbf{FP2}) = \text{contents}(\mathbf{FP1})$ . The type description of **FP2** is equal to the one of **FP1** except for all functions listed in the **hide** clause, i.e.,  $\text{type}(\mathbf{FP2}) = [\text{getname}, \text{gettype}, \text{gbitwidth}, \text{gnumpins}, \text{getpos}, \text{setpos}, \text{getsize}, \text{setsize}, \text{getpin}]$ . The **FP2** class is depicted in the lower left-hand corner of Figure 13.3. Information related to the structure graph, such as, the number of pins and the bit width of a component, can still be accessed through **FP2**. It can however no longer be updated.

Next, *MultiView* places the new class **FP2** into the global schema using the class integration algorithm (note that I have to use the complex class integration algorithm described in Section 7.3 since the **hide** operator has generated a new type). The class integration algorithm finds that there are three equivalence groups with respect to **FP2** (See Figure 13.3). For an explanation of the class integration algorithm and the definition and explanation of these equivalence groups see Section 7.3. The first equivalence group  $\text{Equiv}_1$ , representing all classes  $C_i$  with  $C_i \sqcap \mathbf{FP2} = []$ , contains all classes of the global schema graph that are not part of the component graph, i.e., that are not shown in Figure 13.1. These classes do not model components and therefore have a non-overlapping set of functions in their respective type descriptions<sup>1</sup>.  $\text{Equiv}_1$  contains the **ComponentGraph** class and all classes in the

<sup>1</sup>This is a very important observation, since it forms the basis of our argument that the size of global schema is not likely to grow exponentially with the creation of new views for realistic application examples.

subgraph rooted at the **IntPair** class (Figure 13.1). The second equivalence group **Equiv2** contains the **Components** class and all its structure-related subclasses (all subclasses but the **FPComp** subclass in Figure 13.3). The **Components** class is the representative class of **Equiv2**. Classes in **Equiv2** model the structure-related characteristics of components (but not the floorplan-related ones). Therefore  $\text{type}(\text{Equiv2}) = \text{type}(\text{Components}) \sqcap \text{type}(\text{FP2}) = [\text{getname}, \text{gettype}, \text{gbitwidth}, \text{gnumpins}, \dots]$ . These are all structure-related functions that are also part of the **FP2** type description. Lastly, the third equivalence group **Equiv3** contains all classes that are in the subgraph rooted at the **FPComp** class.  $\text{Type}(\text{Equiv3}) = \text{type}(\text{FPComp}) \sqcap \text{type}(\text{FP2}) = \text{type}(\text{FP2})$ . **FPComp** is the representative class of **Equiv3**.

Next the class integration algorithm generates three intermediate classes  $I_i$ , one for each equivalence group (See Figure 13.4). The first intermediate class  $I_1$  is equal with the root of the first equivalence group **Equiv1**, and therefore is merged with the **ComponentGraph** class. The intermediate classes  $I_2$  and  $I_3$  are interconnected with one another and with the representatives of their respective equivalence groups **Equiv2** and **Equiv3**. For a discussion of this algorithm see Section 7.3. Finally, the virtual class **FP2** is integrated into the now prepared class generalization hierarchy using the class placement algorithm presented in Section 7.4. The global schema graph after insertion of the virtual class **FP2** is depicted in Figure 13.4. This completes the class integration of **FP2**, the second step of *MultiView*.

During the third step of *MultiView* the view definer needs to specify the classes from the global schema to be included into the view. For the sake of this example, I assume that the classes surrounded by the dotted line in Figure 13.5 have been selected for the floorplanning view. They are the three classes **FP2**, **Size**, and **Position**.

*MultiView* now applies the automatic view generation algorithm to interconnect the view classes so as to generate a view schema. In this example, none of the view classes are *is-a* related with one another. Hence, the algorithm generates a flat generalization hierarchy for the view schema. The result of this view generation is shown in Figure 13.6.

Next, I run the view consistency checker on the Floorplanning1 view in Figure 13.6 to determine whether the view is closed. See Chapter 10 for a description of the view consistency checker algorithm. The consistency checker determines that the **FP2** class has the `getpin()` function with **Pin** as domain class. The **Pin** class is however not part of the view. The view consistency checker therefore informs the view definer that the specified view is incomplete. It furthermore suggests the addition of the **Pin** class to the view in order to make it closed. The view definer



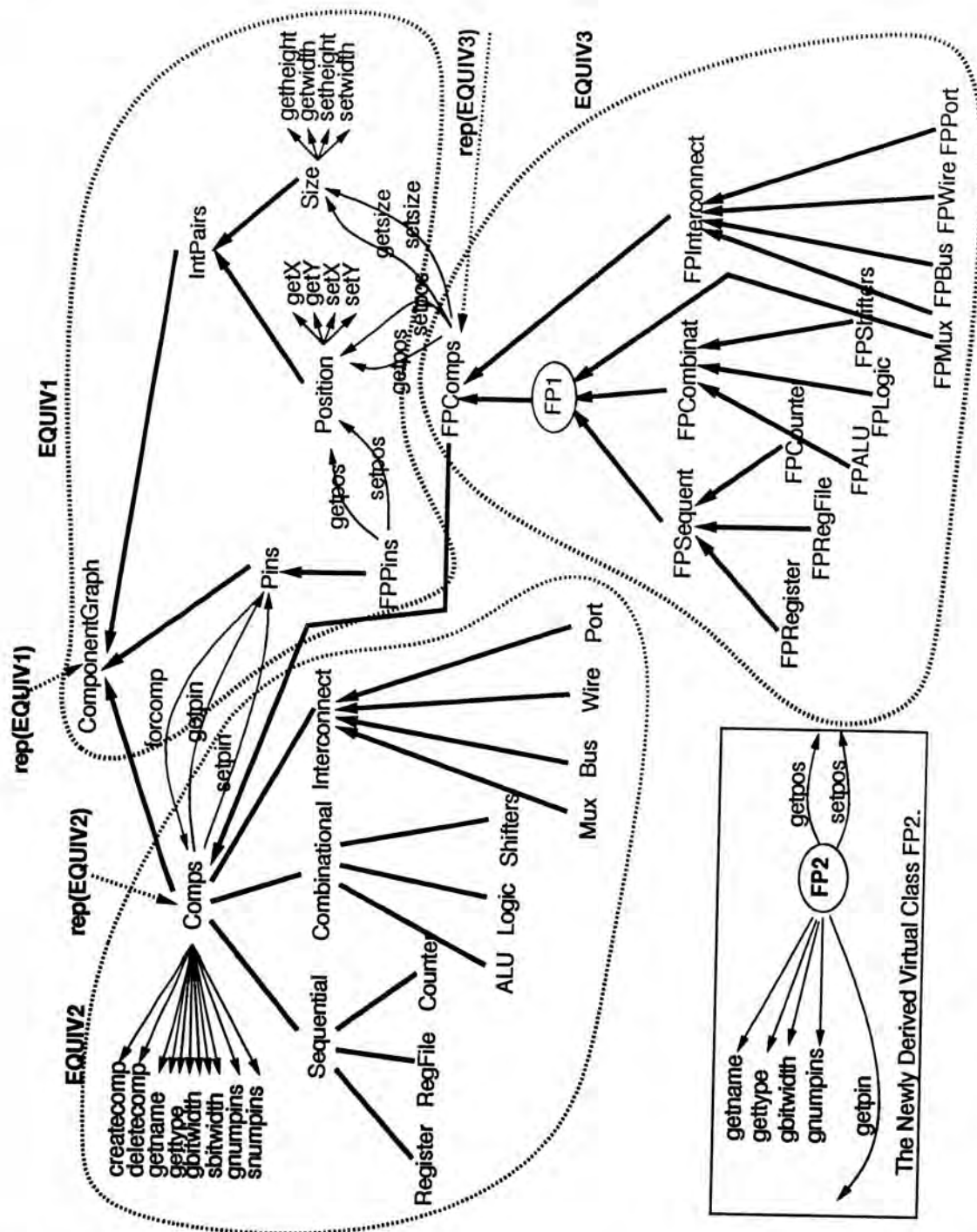


Figure 13.3: Partitioning the Global Schema into Equivalence Groups Using the hide class FP2.

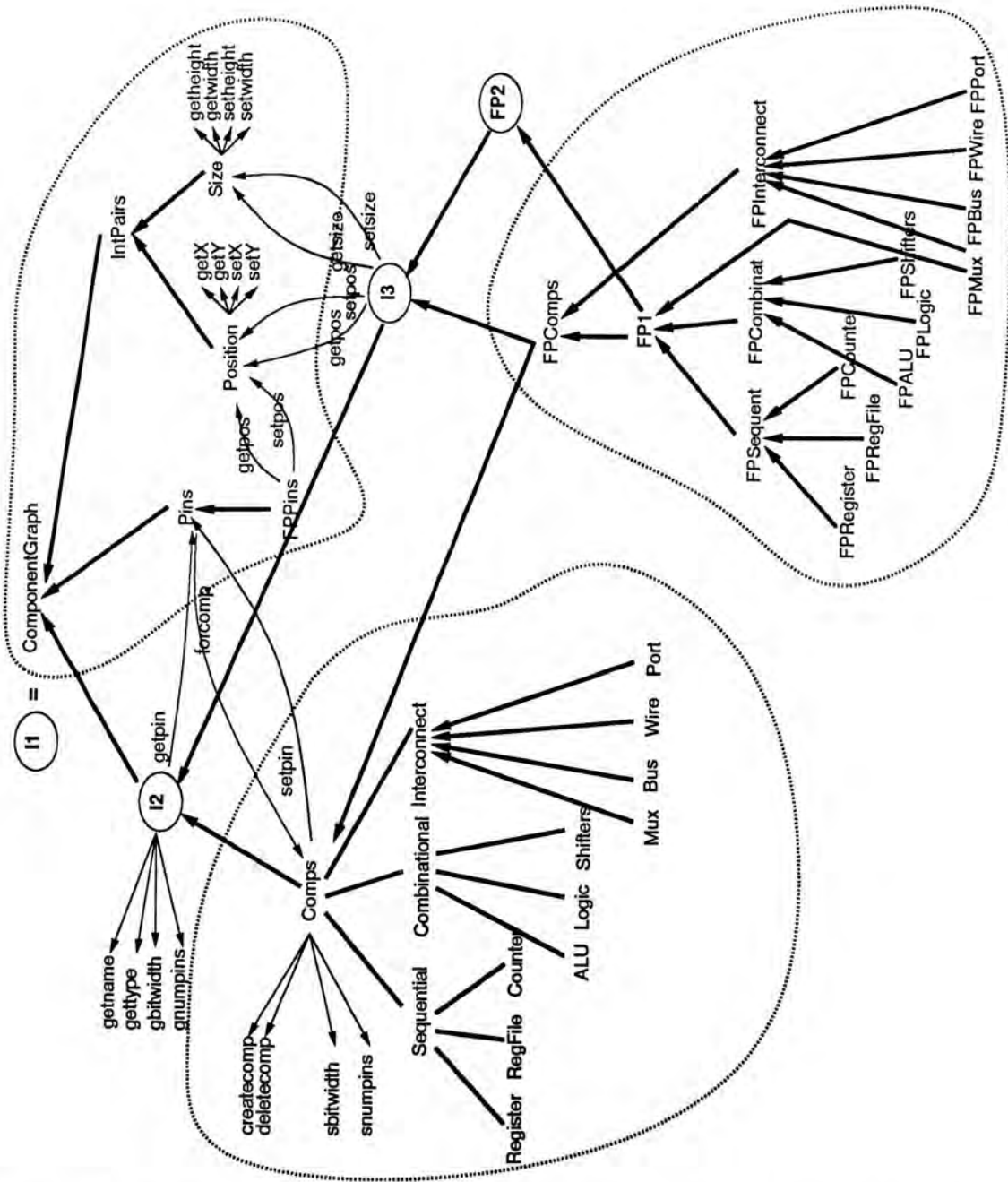


Figure 13.4: Integrating **FP2** into the Global Schema: Generation and Integration of Intermediate Classes and Placement of **FP2**.

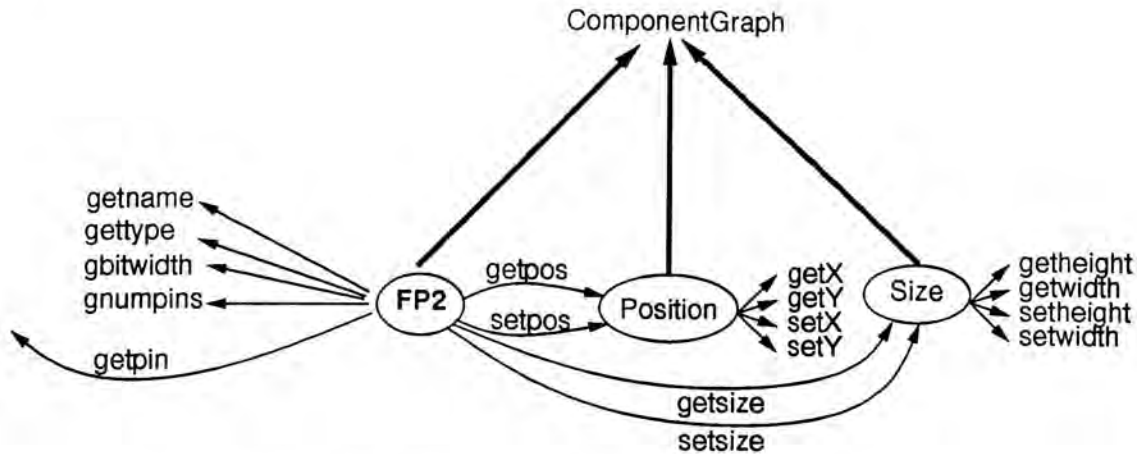


Figure 13.6: Generation of the Floorplanning1 View.

now has two options: (1) either to add the **Pin** class to the view or (2) to respecify the view specification by removing the `getpin()` function from the **FP2** class. In this example, I assume that the view definer decides to do the first. The resulting, now closed, view schema is shown in Figure 13.7.

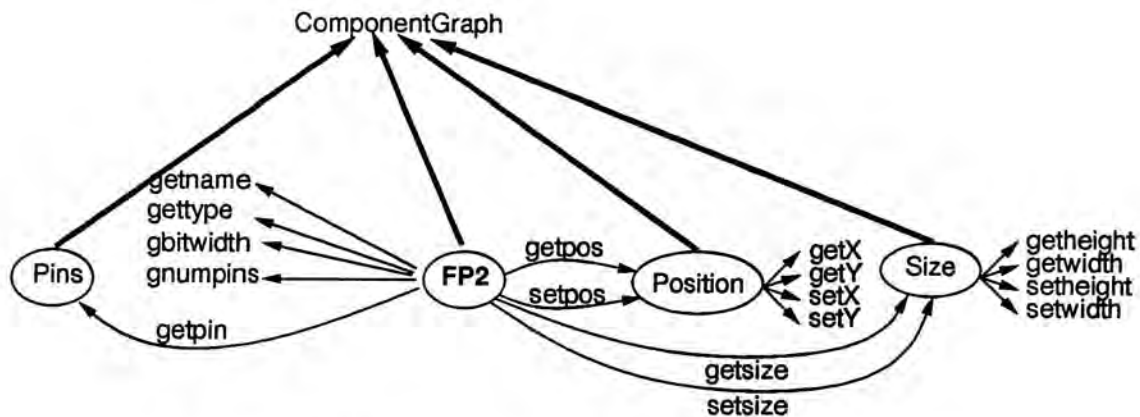


Figure 13.7: Correcting Floorplanning1 View for Closure.

Above I have detailed the incremental steps involved in generating the Floorplanning1 view in Figure 13.6. Most of the discussion has focussed on the modification of the global schema, which though a necessary step for assuring the consistency of the view support system and the synchronization of the views with the global schema, is of course hidden from the view definer. Now, I want to describe the above discussed view specification from the user's point of view. For this purpose, the view definer would simply have to create a view specification script describing the floorplanning view. An example of the view specification script is depicted in Figure 13.7.

### View Creation Script for the Floorplanning1 View:

```

DEFINE-VIEW Floorplanning1
  class FP1 := union(FPSequent,FPCombinat,FPMux);
  class FP2 := hide [createcomp,deletcomp,sbitwidth,
    snumpins,setpin] from FP1;
  ADD-CLASS (FP2);
  ADD-CLASS (Position);
  ADD-CLASS (Size);
  ADD-CLASS (Pin);
  SAVE-VIEW;
END-VIEW

```

### 13.2.4 Customizing the View with Application-Specific Functions

Notice that the global schema provides only very elementary functions for manipulating the symbolic layout, namely, the two functions `set-position(x,y)` and `get-position(x,y)`. If the floorplanner is using other functions in the application problem, then the view definer may want to further customize the design view by associating these application-specific functions directly with the view (rather than requiring the tool developer to define their own functions). This has several advantages. First, the view gives a semantic model of the design and of the required actions that is close to the information model of the application domain. All necessary application-specific functions being available through the view, the tool can now work directly off the database instead of having to translate the data into local data structures. Secondly, the view definer can assure the consistency of these functions, since the function implementation is kept with the view description (rather than with different application programs). Thirdly, other floorplanning tools developed at a later time can also utilize this functionality (rather than having to reinvent it).

Assume that the view definer may want to add functions for moving a component in a horizontal or in a vertical direction. This refinement of the placement functions can be accomplished in *MultiView* using the `refine` operator. An example command using the `refine` operator is given next.

```

class FP3 := refine FP2 with
  move-horizontal(horiz) :=
    { x := self.getpos.getX() + horiz;
      y := self.getpos.getY();
      self.setpos(x,y); };
  move-vertical(vert) :=
    { x := self.getpos.getX();
      y := self.getpos.getY() + vert;
      self.setpos(x,y); };
];

```

The view definer now adds the refinement of the placement functions described above to the floorplanning view using a view specification script. In this example, I assume that the view class **FP2** is kept in the view, though, it could have easily been removed.

#### View Specification Script for the Floorplanning1 view:

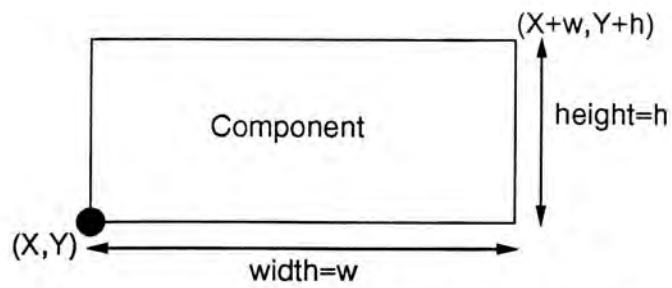
```

MODIFY-VIEW Floorplanning1
  class FP3 := refine FP2 with [move-horizontal(horiz),
    move-vertical(vert)];
  ADD-CLASS (FP3);
  SAVE-VIEW;
END-VIEW

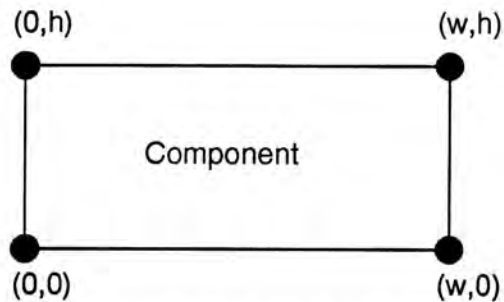
```

Given this view specification script, *MultiView* first generates the virtual class **FP3**. **FP3** has the same object instance membership as **FP2**, i.e.,  $\text{contents}(\mathbf{FP3}) = \text{contents}(\mathbf{FP2})$ . The type description of **FP3** is equal to the type description of **FP2** extended by the two functions `move-horizontal(horiz)` and `move-vertical(vert)`. Next, *MultiView* integrates **FP3** into the global schema. Class integration of a refined virtual class is straightforward, namely, the virtual class becomes a direct subclass of its source class. In this example, **FP3** thus becomes a direct subclass of **FP2**. Since the class integration is so simple, I do not present a diagram of the modified global schema. The view specification script given above modifies the Floorplanning1 view by adding the class **FP3**. *MultiView* thus adds the **FP3** class to the view schema and then recalculates the generalization hierarchy of the view schema. The resulting view is shown in Figure 13.8. *MultiView* then runs the view consistency checker to make sure that the modified view is closed. The checker finds that the view is indeed closed.

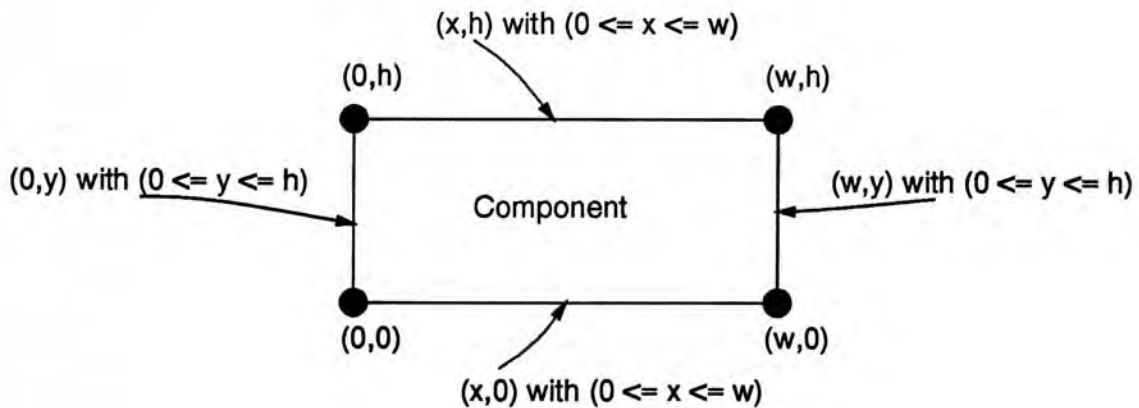




(a) A component with  $(\text{width}, \text{height}) = (w, h)$  and position in grid equal to  $(X, Y)$ .



(b) Use the component as reference point for the positioning of its pins, and ignore the grid position.



(c) All legal positions of pins relative to their component.

Figure 13.9: Pin Positioning Relative to Height and Width of Components.

```

function legalpinpos (h, w, x, y) return boolean;
// h, w: height and width of component
// x, y: x and y coordinate position of pin
{
  if (x=0 and 0 <= y <= h)
    or (x=w and 0 <= y <= h)
    or (0 <= x <= w and y = 0)
    or (0 <= x <= w and y = h)
  then
    return(true);
  else
    return(false);
  endif;
};

```

This condition can now easily be incorporated into a function that moves a pin only if the new pin position is a legal one. The function `movepin(horiz,vert)` is given below.

```

class Pin2 := refine FPPin with
  movepin(horiz,vert) :=
    { if pinassigned() and
      legalpinpos(self.forcomp.getpos.getwidth,
                  self.forcomp.getpos.getheight,
                  self.getpos.getX() + horiz,
                  self.getpos.getY() + vert)
    then
      setpos(self.getpos.getX() + horiz, self.getpos.getY() + vert);
    else
      fprintf(errorfile, "illegal move rejected!");
    endif; };
];

```

The `movepin(horiz,vert)` function is added to the **Pin2** class by using a **refine** operator. The `pinassigned()` function used in the `movepin()` function checks whether the pin does indeed have an associated component. It also checks if the pin has some initial position. If it does not, then a default position of (0,0) is assumed.



### 13.2.6 Customizing Using Derived Attributes

As discussed above, the position of a pin is specified in relative terms. For some applications, it may be important to know the absolute position of a pin on the given grid. This information is needed for instance when verifying the correctness of the pin position in relationship to the position of the connected wire, or when generating a diagram of the design. In *MultiView*, I can add an attribute to represent the absolute pin position. This is done by adding a function that computes the desired value from the relative pin position stored in the database. The value of the absolute pin position attribute is not stored but rather recomputed whenever accessed. This type of an attribute is generally referred to as *derived attribute* in the database literature. Note that to the user of the view this will be transparent, i.e., he or she won't realize that `getpos()` is a stored value while `getabsolutepos()` is a derived value.

For the following, I assume that the position of a pin is relative to its component with the left lower corner of the component the coordinate of (0,0) as shown in in Figure 13.9.c. Similarly, the position of a component is also indicated by the left lower corner. Then the absolute position can be calculated as shown below.

```
class Pin2 := refine FPPin with
  getactualpos() :=
    { tmp pos p;
      createpos( p );
      p.setpos( self.getpos.getX() + self.forcomp.getpos.getX(),
                self.getpos.getY() + self.forcomp.getpos.getY() );
      return ( p ); }
];
```

It is now easy to create a second interface for the second floorplanning tool, since the new view can share many object classes with the original floorplanning view. An example of the specification of a second floorplanning view is given next.

View Creation Script for the Floorplanning2 view:

```

DEFINE-VIEW Floorplanning2
  ADD-SCHEMA Floorplanning1;
  class Pin2:=hide [setpos] from FPPin;
  class Pin3:=refine FPPin with [movepin(horiz,vert),getactualpos()];
  ADD-CLASS (Pin3);
  DELETE-CLASS (Pin);
  DELETE-CLASS (FP2);
  SAVE-VIEW;
END-VIEW

```

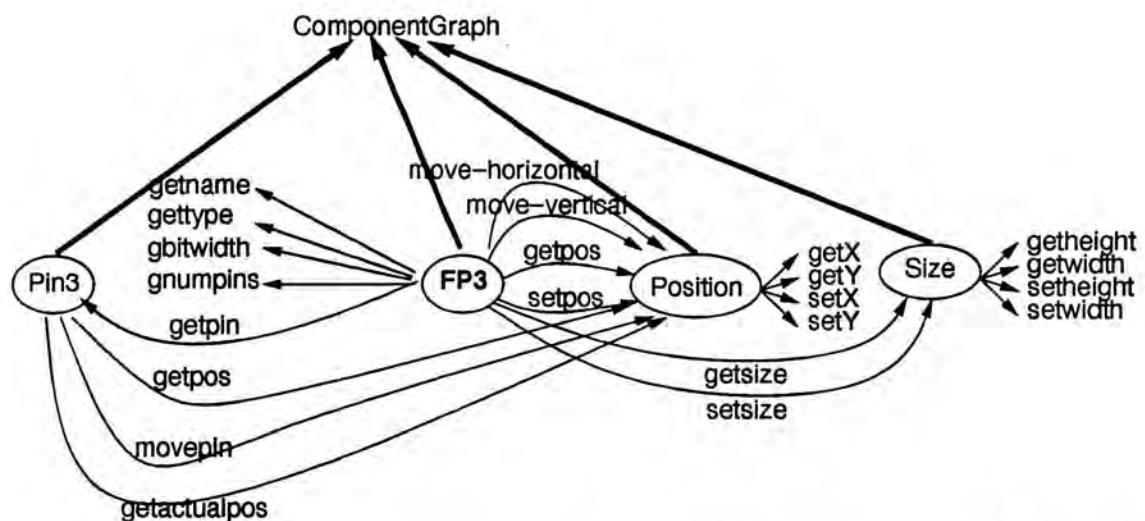
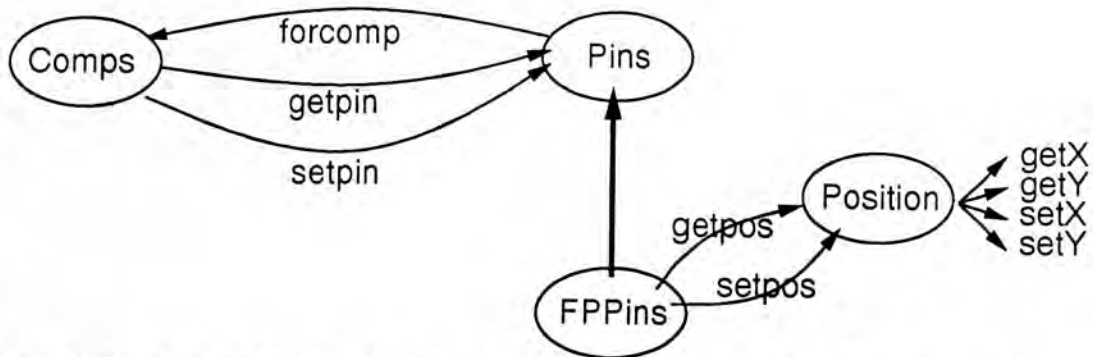


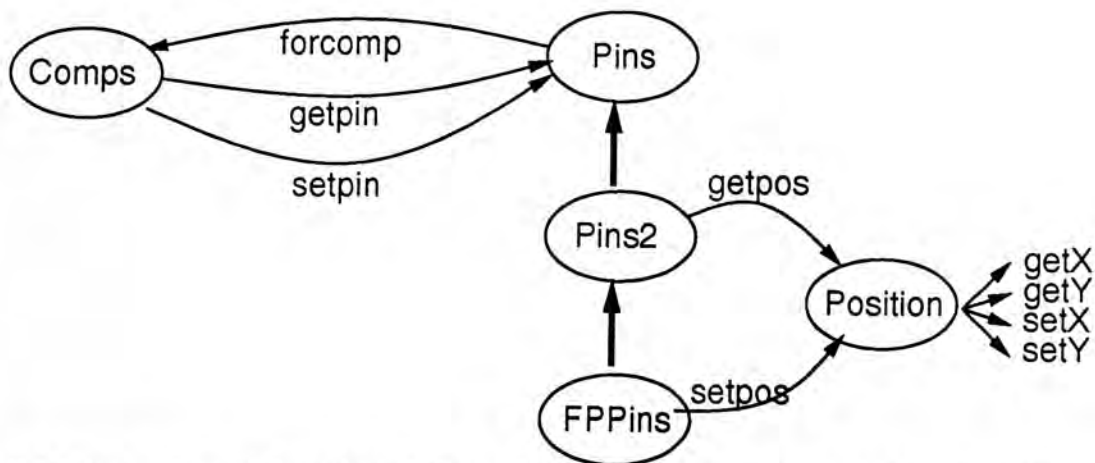
Figure 13.10: Floorplanning2 View Derived by Adding Consistency in Pin Movement to Floorplanning1 View.

The resulting view schema Floorplanning2 is given in Figure 13.10. It does not differ significantly from the Floorplanning1 view. However, it has the added flexibility of easily moving pin positions without having to worry about generating inconsistent designs. The original Floorplanning1 view is still intact and exists concurrently with this view. Therefore, two floorplanners can interface with the database, each using their (view) model of the shared design information.

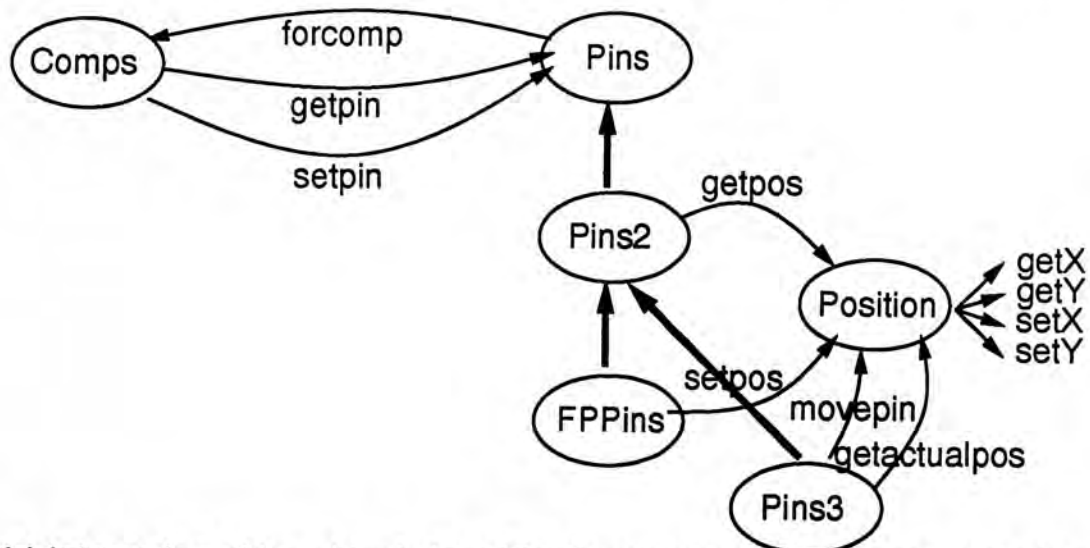
The modification of the global schema to handle the creation of the second view schema, called the Floorplanning2 view, is shown in Figure 13.11. Figure 13.11.a shows the global schema before the generation of the view. Figure 13.11.b shows how the virtual class **Pin2** is added to the global schema. Figure 13.11.c depicts the integration of the virtual class **Pin3** into the global schema.



(a) Portion of the Global Schema Modeling the Pins of Components.



(b) Integration of the virtual class Pins2 generated by the hide operator.



(c) Integration of the virtual class Pins3 generated by the refine operator.

Figure 13.11: Adjusting the Global Model for the Floorplanning2 View.

Finally, Figure 13.12 depicts the complete global schema incorporating all adjustments for both the Floorplanning1 and Floorplanning2 view. This example shows that tools that work on the same design task generally use the design information in a similar manner. Hence, the two floorplanning design views have many overlaps.

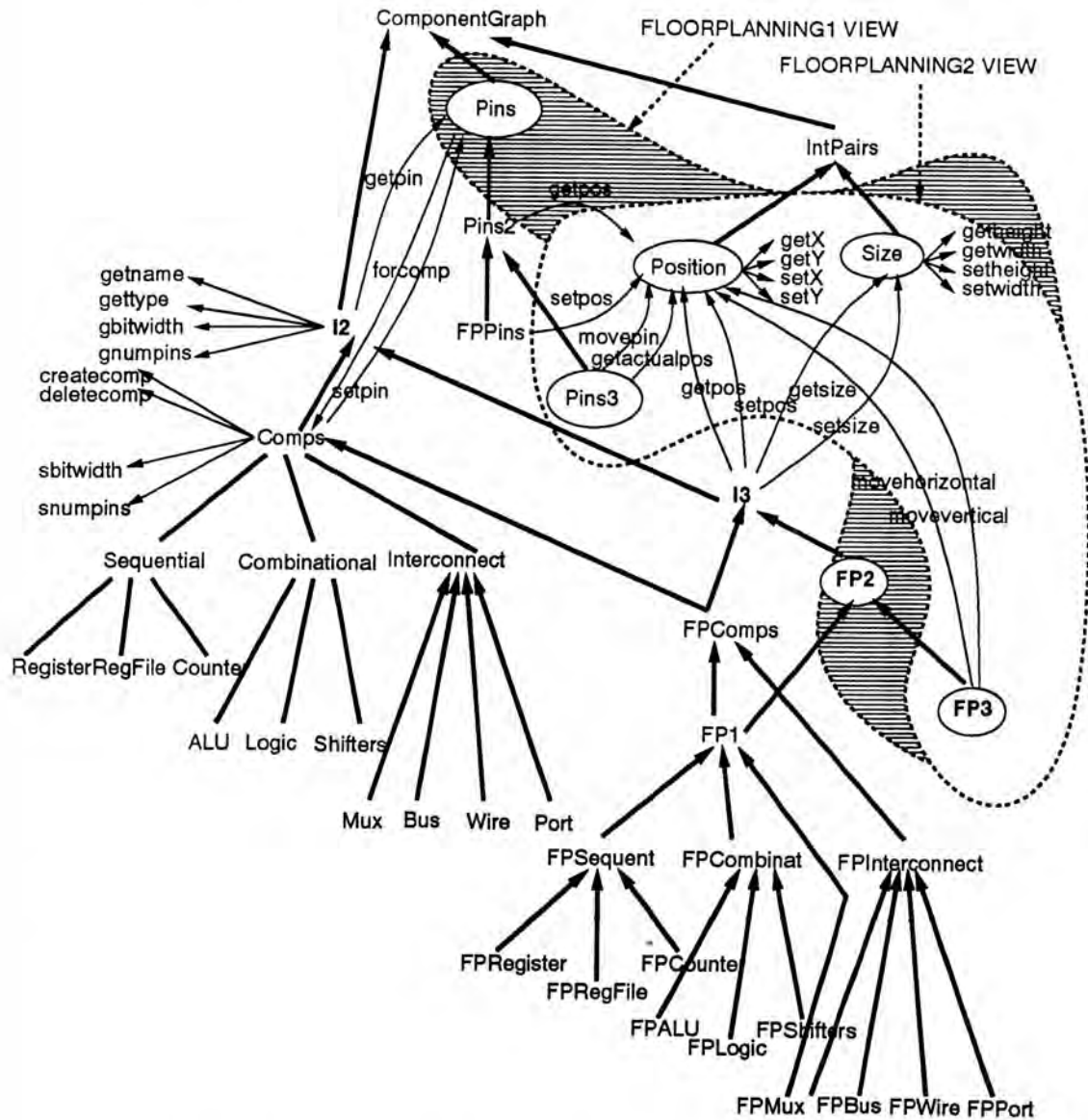


Figure 13.12: The Global Model after the Creation of Two Floorplanning Views.

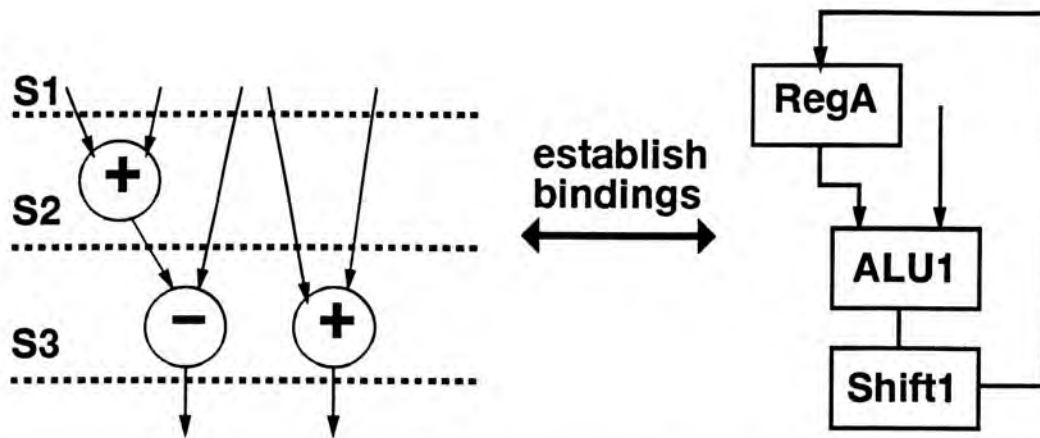
### 13.3 Design Views for the Binding Design Task

In this section, I discuss the construction of a design view for the (operator) *binding* design task. *Binding* establishes a mapping between the operators in the data flow graph and the hardware units that are to implement the operator (Figure 13.13.a). Binding has the following constraints: (1) every operator in the data flow graph should be bound to exactly one hardware unit, (2) two operators can be bound only to the same hardware unit if they are mutually exclusive by for instance being in different states. The binding tool needs information about the operators in the data flow graph, their assigned state, and components available for a given state.

The global design object does of course contain this type of information. It is however spread over several graph structures (namely, the state graph, the data flow graph, and the component graph). It would be more convenient if instead this information were grouped into a table-like structure, such as the binding table shown in Figure 13.13.b or the list of bindings shown in Figure 13.13.c. In addition, the binding tool should not modify any of these graph structures, i.e., it should not change the state assignment, the shape of the data flow graph, or the set of allocated components. The only legal operations for the binding design task are to establish a binding (i.e., to add a row into the table in Figure 13.13.c) or to undo a binding (i.e., to remove a row from the table in Figure 13.13.c).

In the relational model, this sort of a table collected from different parts of the database would be modeled using the JOIN operator. However, the object-oriented algebra proposed in Section 5 does not support join operators. Furthermore, the binding design tool needs the capability to update this binding information rather than just to retrieve it (recall that many relational join views are not updatable; and they thus would not be useful for this application). Therefore, in *MultiView* I will model this situation by adding appropriate functions for representing the bindings as explained below. Note that each data flow node is unique in the binding table, while both states and components appear multiple times. This is because there is a unique mapping for each data flow operator to the state in which the data flow node is executed (disregarding multicycle operators for the sake of simplicity). Similarly, there is a unique mapping for each data flow operator to the component which will implement the data flow node in the structure graph. Therefore, I simulate the join operator in *MultiView* by adding functions – representing these unique mappings – to the data flow operator class.

The initial global schema is shown in Figure 13.14. Recall that the binding tool should not be allowed to modify the set of allocated components, i.e., the number



(a) The Binding Design Task.

*components*

	C1	C2	C3	C4
S1	op1		op2	
S2	op3	op4		op5
S3		op6		op7
S3				

*states*

(b) A Binding Table.

state	op	comp
S1	op1	C1
S1	op2	C3
S2	op3	C1
S2	op4	C2
S2	op5	C4
S3	op6	C2
S3	op7	C4

*operator op5  
bound to  
component C4  
in state S2*

*operator op5  
bound to  
component C4  
in state S2*

(c) A Listing of Binding Triplets.

Figure 13.13: The Binding Design Task.

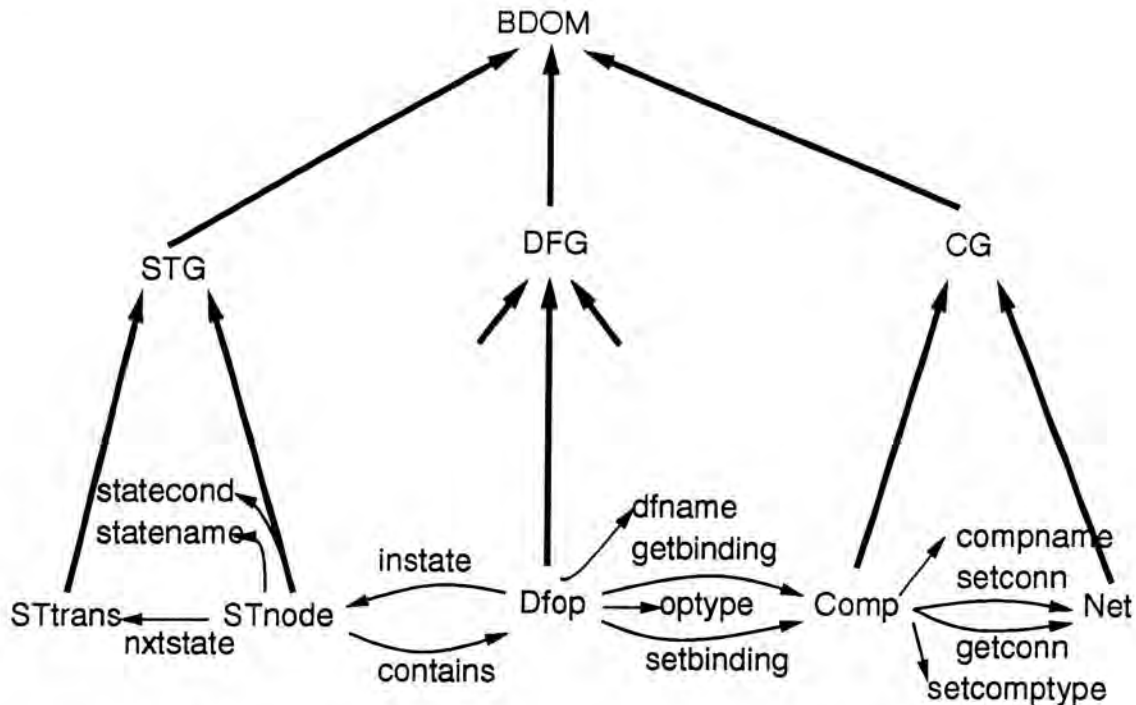


Figure 13.14: The Global Schema Before Constructing the Binding View.

or type of components, and their connectivity. Therefore, the design view must protect the **Comp** class. This is accomplished by hiding all update functions from the **Comp** class using the **hide** operator.

```

class CompB :=
  hide [setcomptype, setconn, getconn, ... ] from Comp;
  
```

The **hide** command generates the virtual class **CompB**. **CompB** has the same object instance set as the **Comp** class, but a restricted type description. The **CompB** class thus limits access to the components in a design. The binding tool can for instance scan the class of available components using the **CompB** class, but it cannot modify individual components. The integration of the **CompB** class into the global schema does not result in the creation of an intermediate class (For the class integration algorithm see Section 7). The modified global schema graph is depicted in Figure 13.15.

Next, note that the binding tool should not change the state assignment (a task accomplished by a scheduling tool). Therefore, I now define a new function, called **boundstate()**, that gives me the name of the state associated with a data flow node. This function is added to the schema using the **refine** operator as shown below.

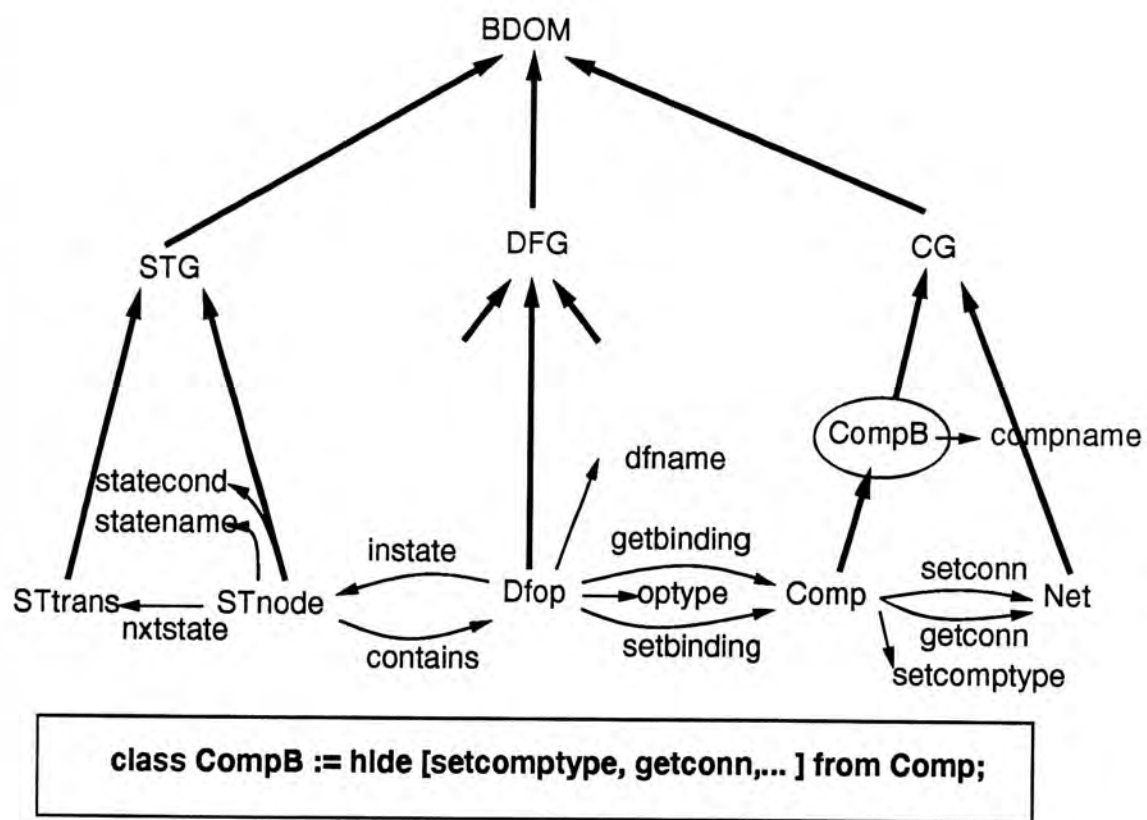
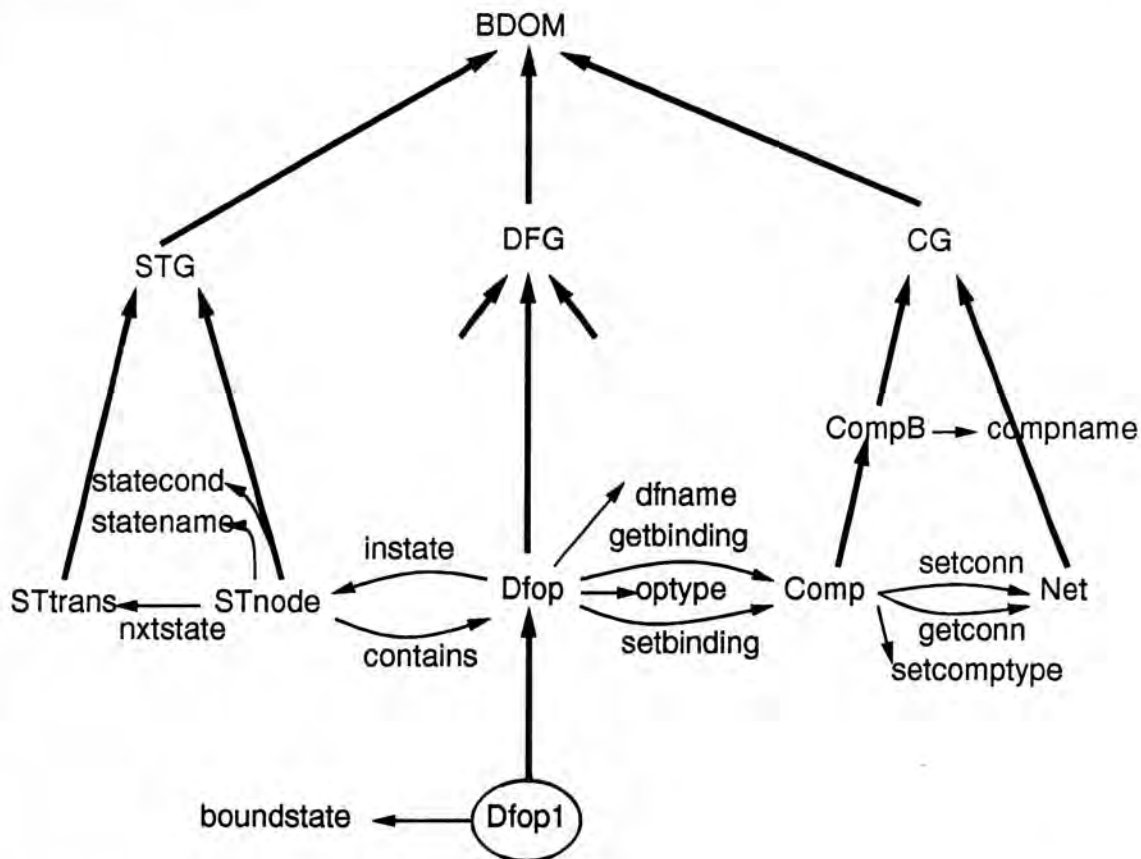


Figure 13.15: Inserting the **CompB** Class into the Global Schema.



```
class Dfop1 := refine Dfop with
  [boundstate() := { return(self.instate.statename) }];
```

The virtual class **Dfop1** is integrated into the global schema shown in Figure 13.14. The result of this integration is depicted in Figure 13.16. Using the `boundstate()` function in place of the `instate()` function will assure that the design tool cannot manipulate the state graph. The `boundstate()` function allows the user to retrieve the name of the state associated with a given data flow node, but not the actual state object.



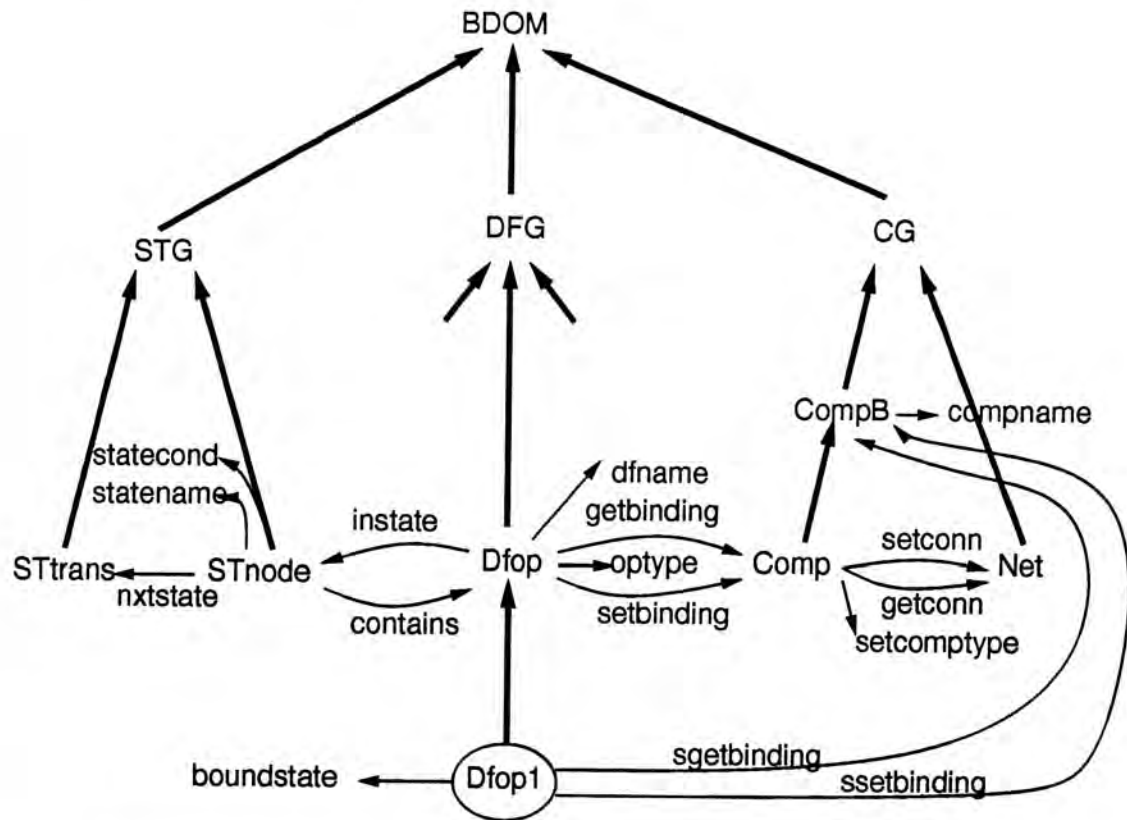
```
class Dfop1 := refine Dfop with
  [ boundstate(String) := return(self.Instate.statename); ];
```

Figure 13.16: Inserting the **Dfop1** Class into the Global Schema.

Note that the `setbinding()` function allows for the binding of an arbitrary data flow node to any component. If, instead, the view definer wants to assure that only legal bindings are being generated, he or she may want to augment the binding design

view with application-specific binding functions that include appropriate consistency checks. In this example, this can be achieved by using the `refine` operator as follows:

```
class Dfop1 := refine Dfop with
  [ssetbinding(c:CompB) :=
    {if compatible(self,c) then setbinding(c)};
  sgetbinding() := { ... }];
```



```
class Dfop1 := refine Dfop with
  [ boundstate(String) :=
    {return(self.instate.statename)};
    ssetbinding(c:CompB) :=
    {if compatible(self,c) then setbinding(c)};
    sgetbinding(c:CompB) := ...
  ... ]
```

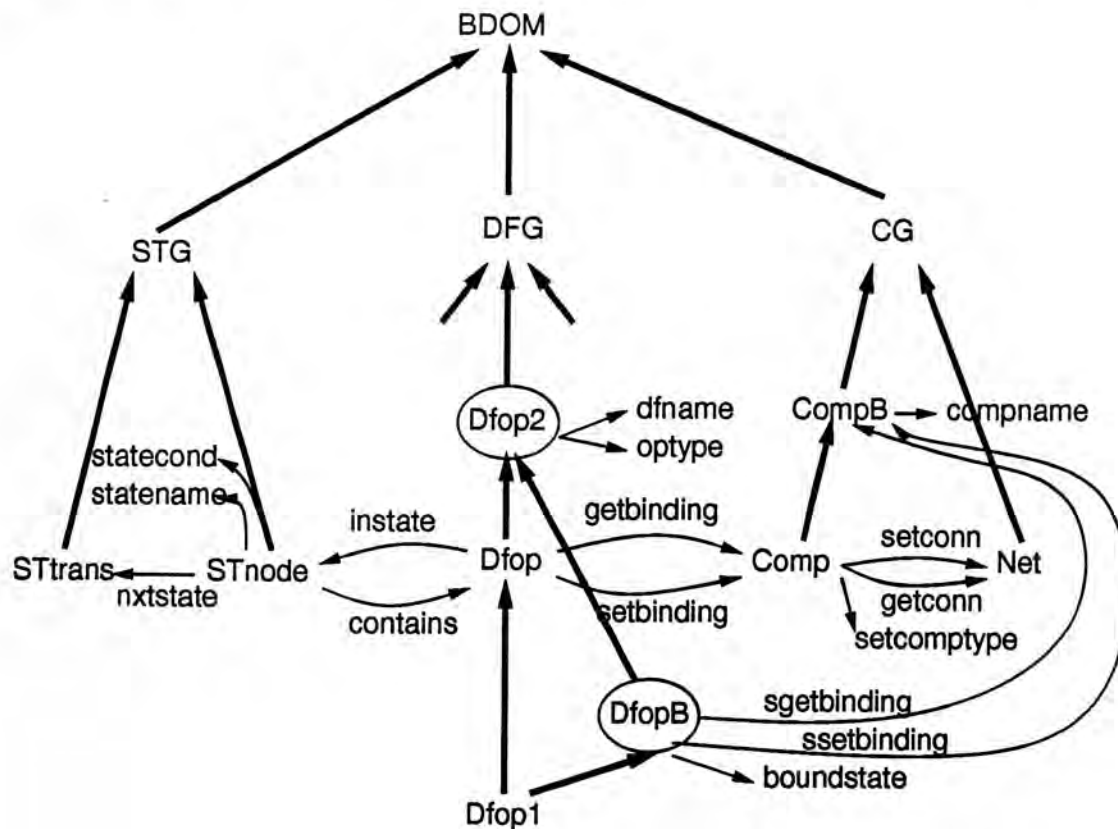
Figure 13.17: Inserting the modified **Dfop1** Class into the Global Schema.

The result of integrating the revised virtual class **Dfop1** into the global schema is shown in Figure 13.17.

Using the `boundstate()` function in place of the `instate()` function will assure that the design tool cannot manipulate the state graph. Therefore, the `instate()` function should be removed from the design view. In *MultiView*, this is done using the `hide` operator.

```
class DfopB := hide [instate, setbinding, ... ] from Dfop1;
```

This command generates a virtual class called **DfopB** which has the new `boundstate()` function but not the `instate()` function in its type interface. The integration of this new class into the global schema depicted in Figure 13.17 results in the creation of an intermediate class, say **Dfop2**. An explanation of this class integration algorithm in general and the reason for creating these intermediate classes in particular can be found in Section 7. The modified global schema graph is depicted in Figure 13.18.



```
class DfopB := hide [instate, setbinding, ... ] from Dfop1;
```

Figure 13.18: Inserting the **DfopB** Class into the Global Schema.

Next, the appropriate classes must be selected from the global schema to be included in the design view. This is accomplished using the view specification script listed below.

### View Creation Script for the Binding View:

```

DEFINE-VIEW Binding
  class CompB := hide [setcomptype, setconn, getconn, ... ]
    from Comp;
  class Dfop1 := refine Dfop with
    [boundstate(), ssetbinding(), sgetbinding()];
  class DfopB := hide [instate, setbinding(), ...] from Dfop1;
  ADD-CLASS (DfopB);
  ADD-CLASS (CompB);
  SAVE-VIEW;
END-VIEW

```

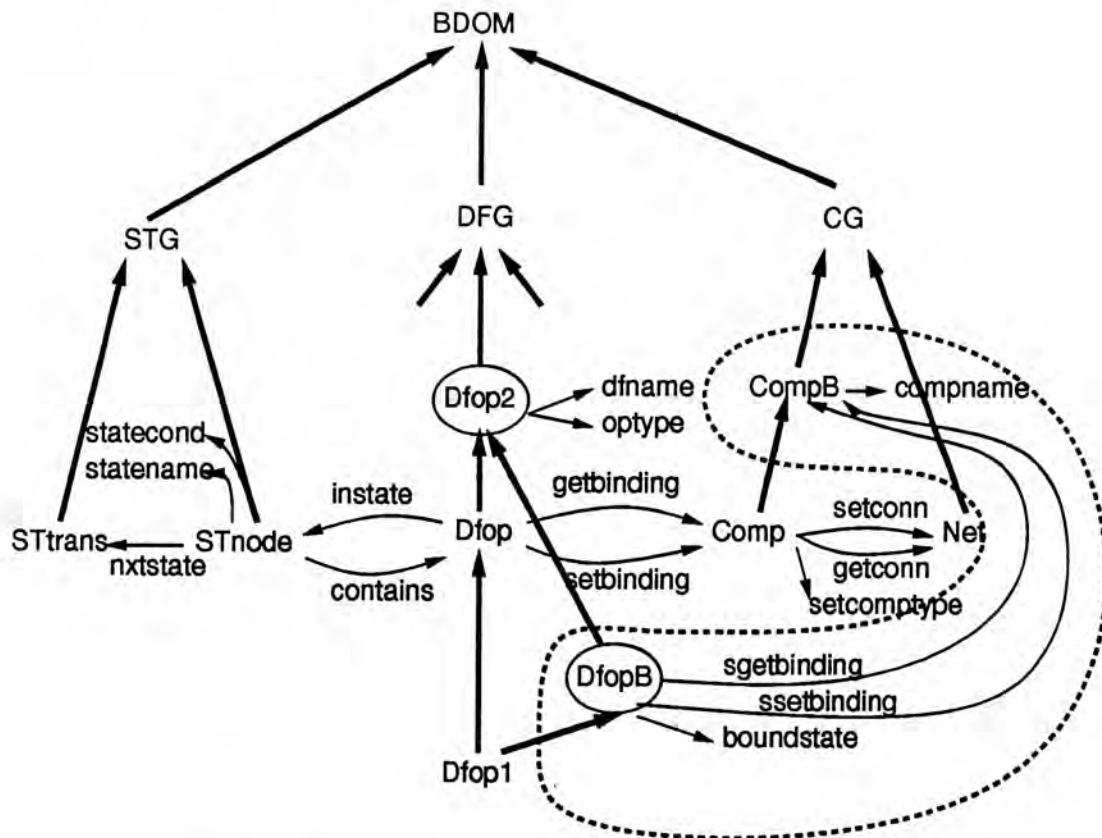


Figure 13.19: Selecting View Classes for the Binding View.

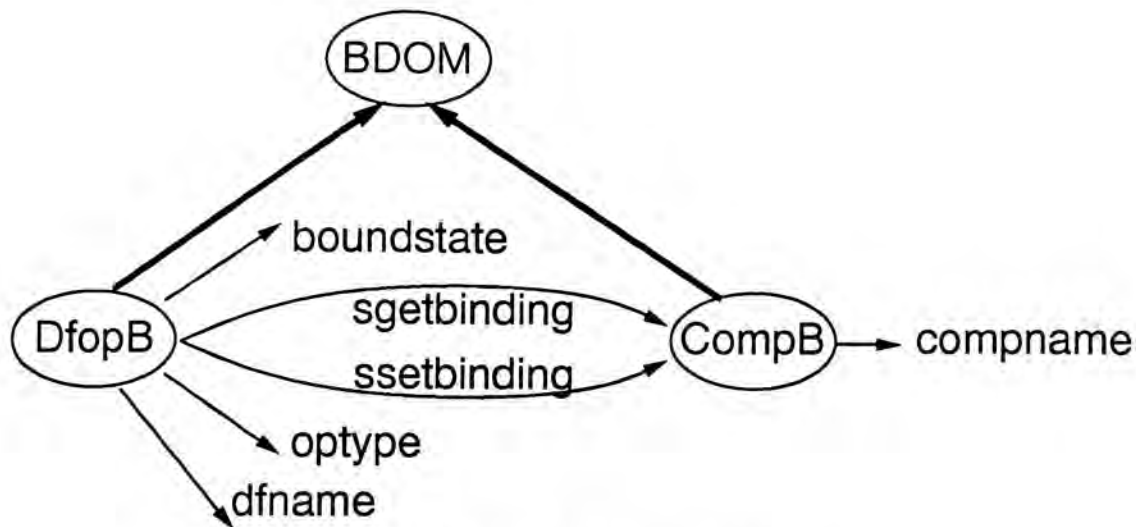


Figure 13.20: The Binding View Schema.

The selected view classes are indicated in Figure 13.19 by encircling them by a dotted line. The view generation algorithm introduced in Section 9 extracts the view classes from the global schema and interconnects them into a view schema. The resulting view schema for the Binding view is depicted in Figure 13.20. The Binding view has all features required for the binding design task (and no others). First, the binding triplets can be retrieved and updated using the `sgetbinding()` and `ssetbinding()` functions. Second, the list of components can be scanned to determine what components are available, but the allocation of existing components cannot be modified. Third, since each data flow node is assigned to one state, we can retrieve the state associated with a data flow node using the `boundstate()` function. We can however not modify this state assignment. Clearly, the Binding view protects the design information from being changed in any illegal manner during the binding design task.

## 13.4 Design Views for the Timing Constraints in a Control Flow Graph

In this section, I will demonstrate that design views can be utilized not only to control tool access (i.e., to hide the operators that tools use to manipulate the design data), but also to let the user of the design view perceive the existing design data in a different format or at a different level of detail. I will explain this idea based on the timing constraints that are embedded into the control data flow graph

model. A portion of the global schema on which the following discussion is based is shown in Figure 13.21. The global schema in Figure 13.21 describes the control flow graph object types and the related timing constraints (TCG stands for Timing Constraint Graph).

### 13.4.1 The Graph Compiler View

The graph compiler constructs the initial flow graph given a textual specification of a design (See Section 12.6). The graph compiler therefore needs to have access to all update functions of the control flow graph (including the timing constraint classes) in order to create and interconnect the initial graph objects. This tool is not concerned with scheduling of the graph nor with allocating hardware components in the component graph. Therefore, the graph compiler needs to access only a subset of the classes in the global schema. For the sake of this example, I assume that Figure 13.21 depicts the portion of the global schema that is required and sufficient for graph compilation. Hence, the design view for graph compilation, say the Timing1 view, is constructed by selecting a subset of classes from the global schema to be included in the view.

#### View Creation Script for the Timing1 View:

```
DEFINE-VIEW Timing1
  ADD-SCHEMA (CFG);
  ADD-SCHEMA (TCG);
  SAVE-VIEW;
END-VIEW
```

Next, I introduce one design example that I will use throughout this section to demonstrate how design views may actually modify the user's perception of the design data. This is based on the example design specification of the Design1 design shown in Figure 13.22. The specification includes a timing constraint that is encoded in a VHDL comment statement. This timing constraint T1 specifies a minimum delay of 10ns and a maximum delay of 250ns for the execution of the if-statement.

The design specification of Design1 is compiled into a ECDFG design representation with a VHDL Graph Compiler [Lis89]. The resulting graphical representation of the control/data flow graph is given in Figure 13.23. The depiction of the design representation is, of course, simplified. It does for instance not detail the interrelationships and attributes of the design objects at the data flow graph level.

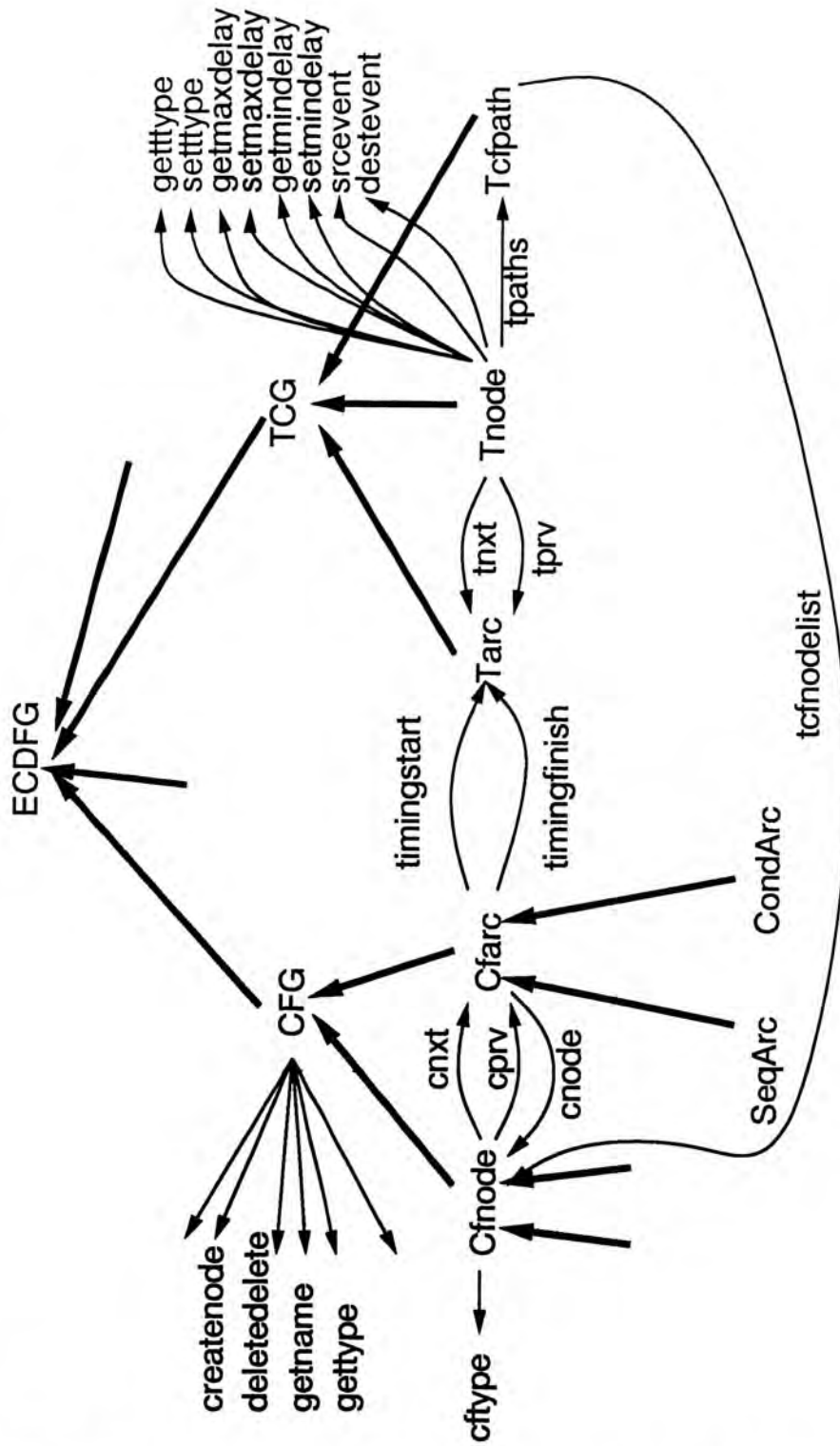


Figure 13.21: A Global Schema with Control Flow and Timing Constraint Classes.

```
entity Design1 is
  port (
    FULL_sig: out BIT
  );
end Design1;

architecture BEHAVIOR of Design1 is
begin
  process
    variable SP : BIT_VECTOR(2 downto 0);
  begin

    ## TIMING T1 250.0 10.0

    if (SP = B"100") then
      FULL_sig <= 0;
    else
      FULL_sig <= 1;
      SP := SP + 1;
    end if;

    ## TIMING T1

  end process;
end BEHAVIOR;
```

Figure 13.22: VHDL Design Specification of Design1.



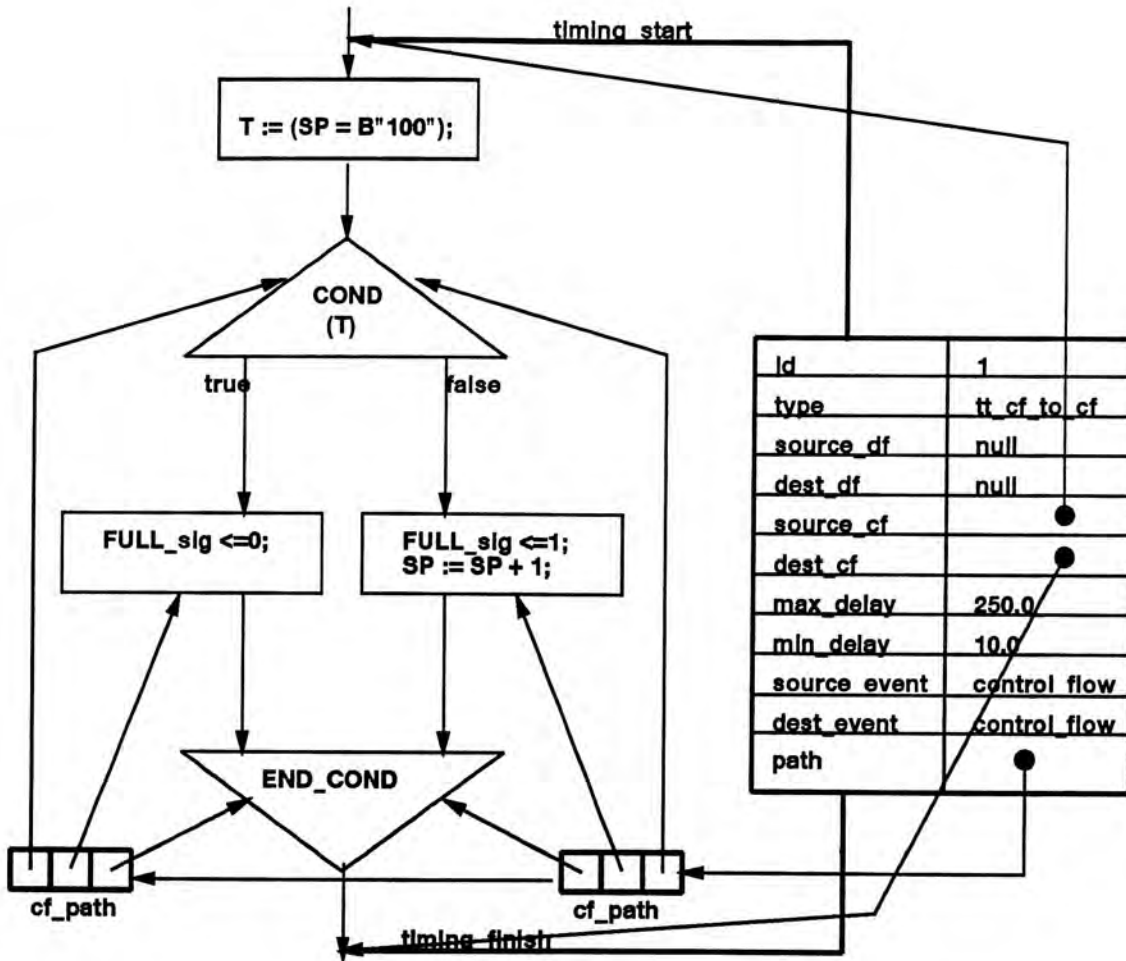


Figure 13.23: The Design1 Example Using the Timing1 View.

In Figure 13.23, the timing constraint T1 is depicted by the box on the right-hand side. The timing constraint originates and ends at a control flow graph connection, a sequencing arc. In Figure 13.23, this is displayed by the bold timing arcs labeled `timing_start` and `timing_finish` that connect control flow sequencing arcs with the timing specification node. The timing specification is a constraint on the execution of all behavior specified between these two points in the control flow graph. Since the timing constraint specifies delay values for the execution of both the true and the false branches, a path expression indicating these two paths is given. This path expression is implemented by an ordered list of object references to the control flow nodes that make up the path, for instance, the control flow nodes `CF_NODE#6`, `CF_NODE#8`, and `CF_NODE#7`. Another timing attribute is the timing type, which in this case corresponds to the value `TT_CF_TO_CF`. It indicates the object type of the source and the destination of the timing constraint. This is necessary since the same timing constraint construct is used to model timing constraints at both the control flow and the data flow level.

### 13.4.2 Simplifying The Timing Information in the Control Flow Graph

Once the flow graph has been constructed by the graph compiler, then other design tools will work on the graph to incrementally refine it. Assume that it is known that Design1 contains only point-to-point timing constraints but no path-specific timing constraints. Then I may want to remove all information about the timing paths associated with the timing constraints, which, while not adding any additional information substance, clutters the design data. In *MultiView*, this can be accomplished using the following steps: first, I remove the `Tcfpath` class from the view and second, I modify the `Tnode` class such as to hide the `tpaths()` reference to the hidden `Tcfpath` class.

```
DELETE-CLASS (Tcfpath);

and

class Ttmp := hide [tpaths] from Tnode;
```

These two commands then allow you to ignore the path information associated with timing constraints.

Secondly, some design tools, say a scheduler, may not want to manipulate the timing constraint information, rather it only consults the timing constraints to constrain its choice in creating a schedule. Hence, the design view could be further refined by removing all manipulating operators, such as the `settype()` and `setmaxdelay()` functions, from the timing class `Tnode`. Assuming further that the tool is only interested in basic timing information, such as the minimum and maximum delay, then this may be accomplished by the `hide` operator using the following command:

```
class Tnodesimple :=
  hide [settype,setmaxdelay,setmindelay,srcevent,destevent]
  from Tnode;
```

In summary, the desired design view can be generated using the following view specification:

#### View Creation Script for the Timing2 View:

```
DEFINE-VIEW Timing2
  ADD-SCHEMA (Timing1);
  class Tnodesimple := hide [settype,setmaxdelay,setmindelay,
    srcevent,destevent,tpaths] from Tnode;
  ADD-CLASS (Tnodesimple);
  DELETE-CLASS (Tnode);
  DELETE-CLASS (Tcfpath);
  SAVE-VIEW;
END-VIEW
```

The view creation script for the Timing2 view first generates a new class called `Tnodesimple`. `Tnodesimple` is a superclass of `Tnode`. Therefore it is integrated into the global schema by placing it above `Tnode` (See Figure 13.24). This completes the modification of the global schema for this design view. Next, the Timing2 view is constructed by selecting all its view classes from the global schema. In this case, the selected view classes are indicated in Figure 13.25 by encircling them by a dotted line. The selected view classes then are interconnected into one view schema using the view generation algorithm explained in Section 9. The resulting view schema Timing2 is shown in Figure 13.26.

Next I'll show how the Design1 example shown in Figure 13.23 is perceived through the Timing2 view. The same graph, when viewed through the Timing2 view, is depicted in Figure 13.27. There are two visible changes in the example

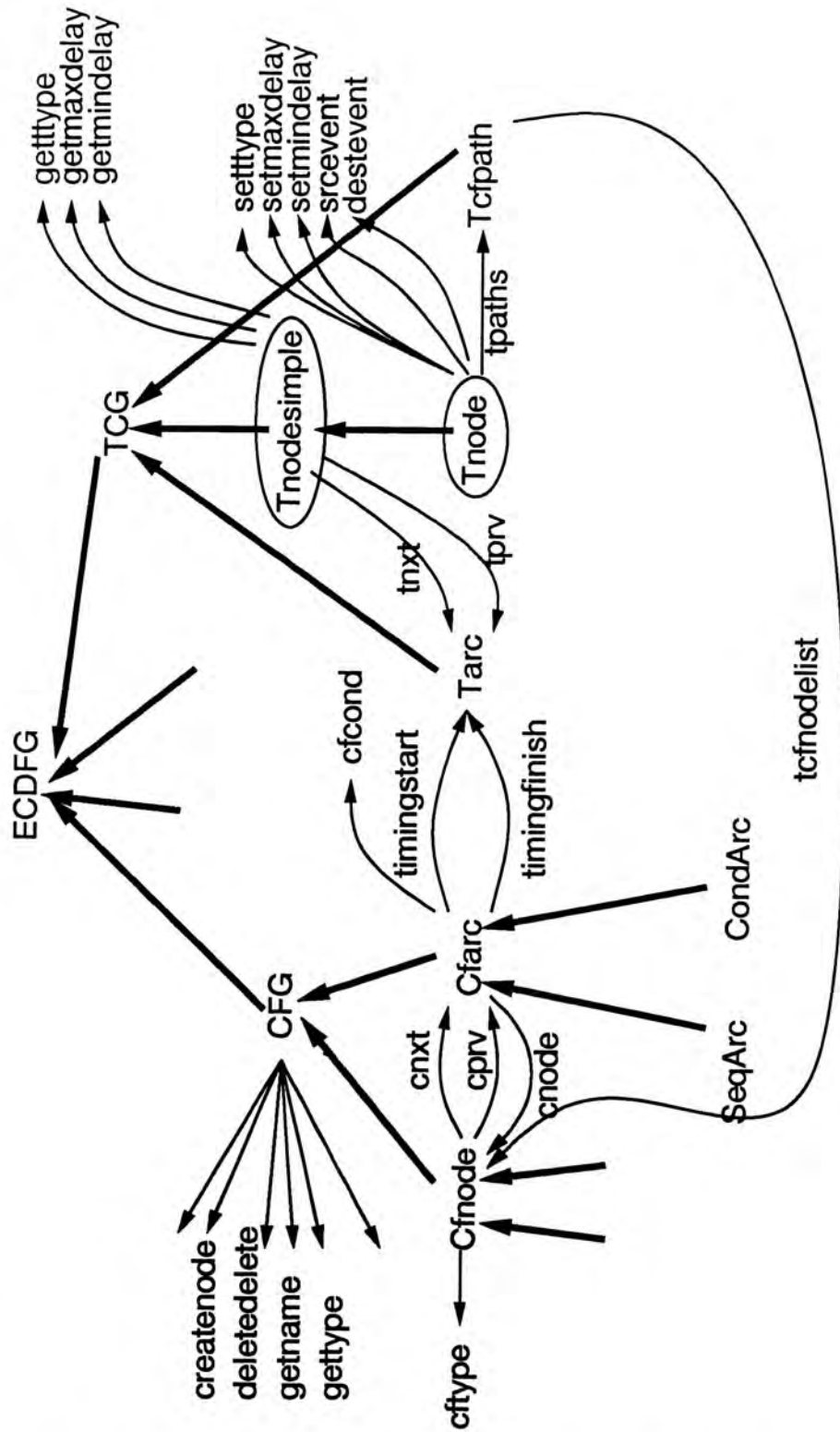


Figure 13.24: Refining the Timing Constraint Classes.

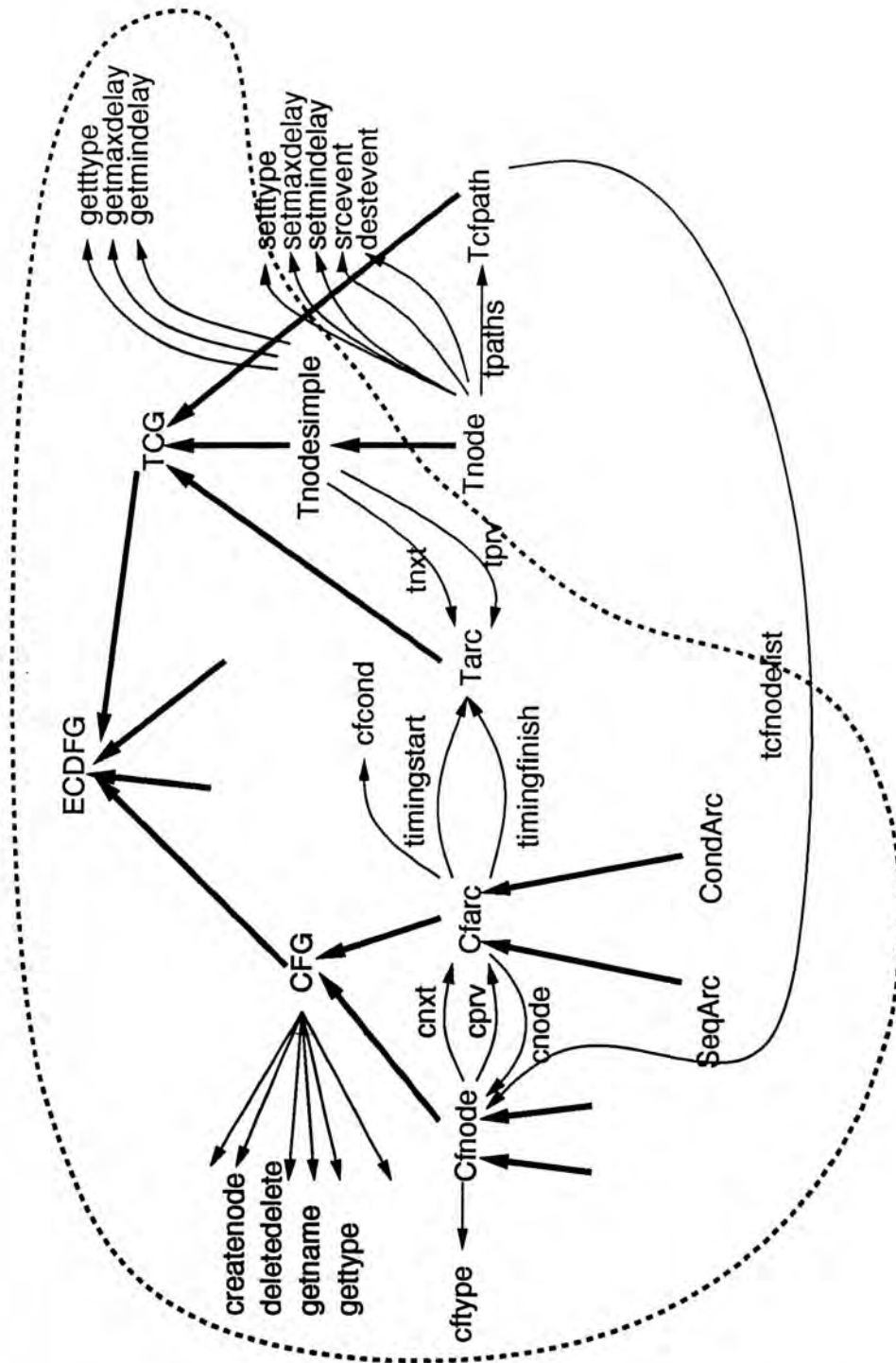


Figure 13.25: Selecting View Classes for the Timing2 View.

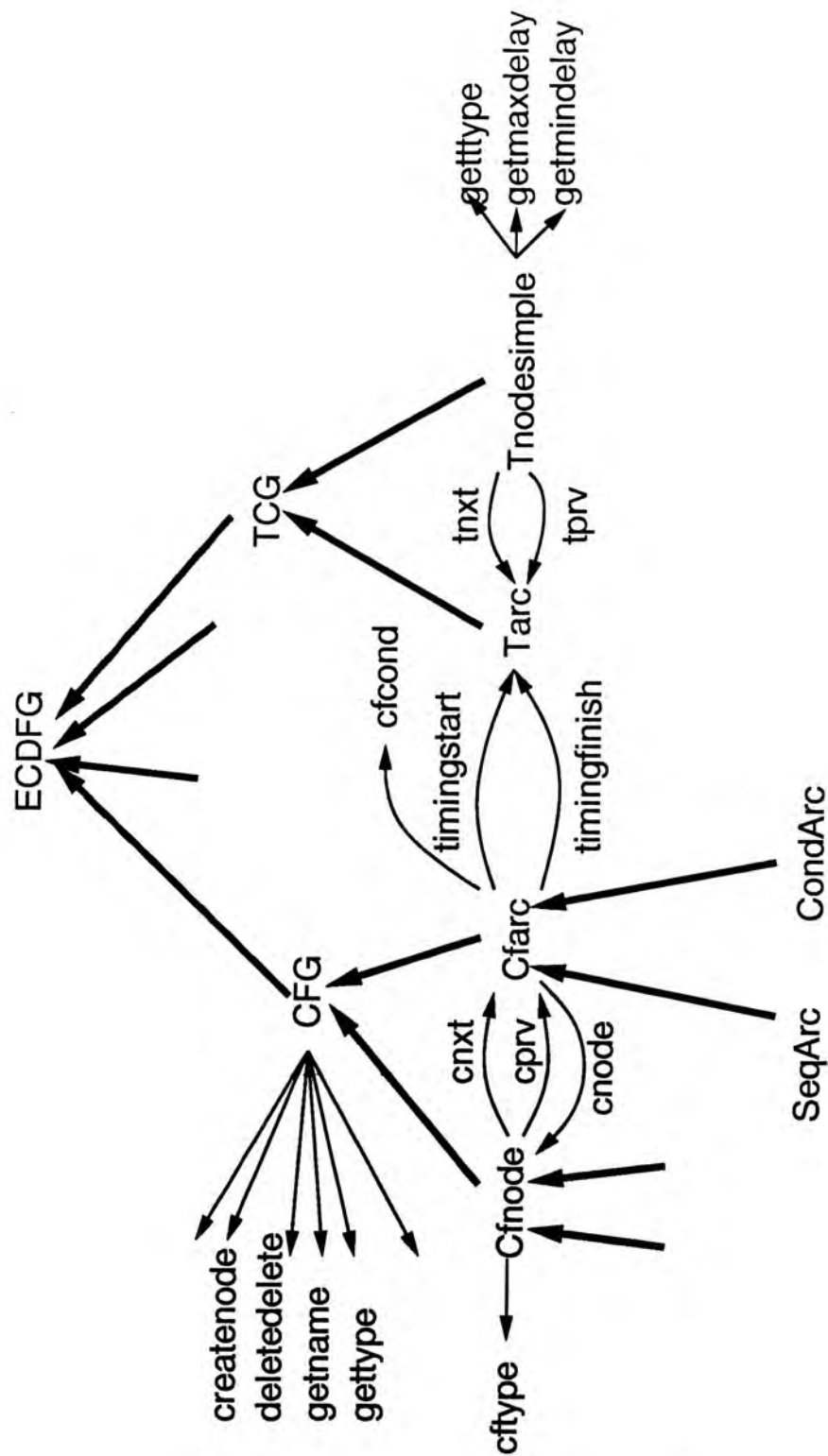


Figure 13.26: The Timing2 View Schema.

graph: first, the timing node is simpler than in the original view, and two, the timing path information is no longer visible through this view.

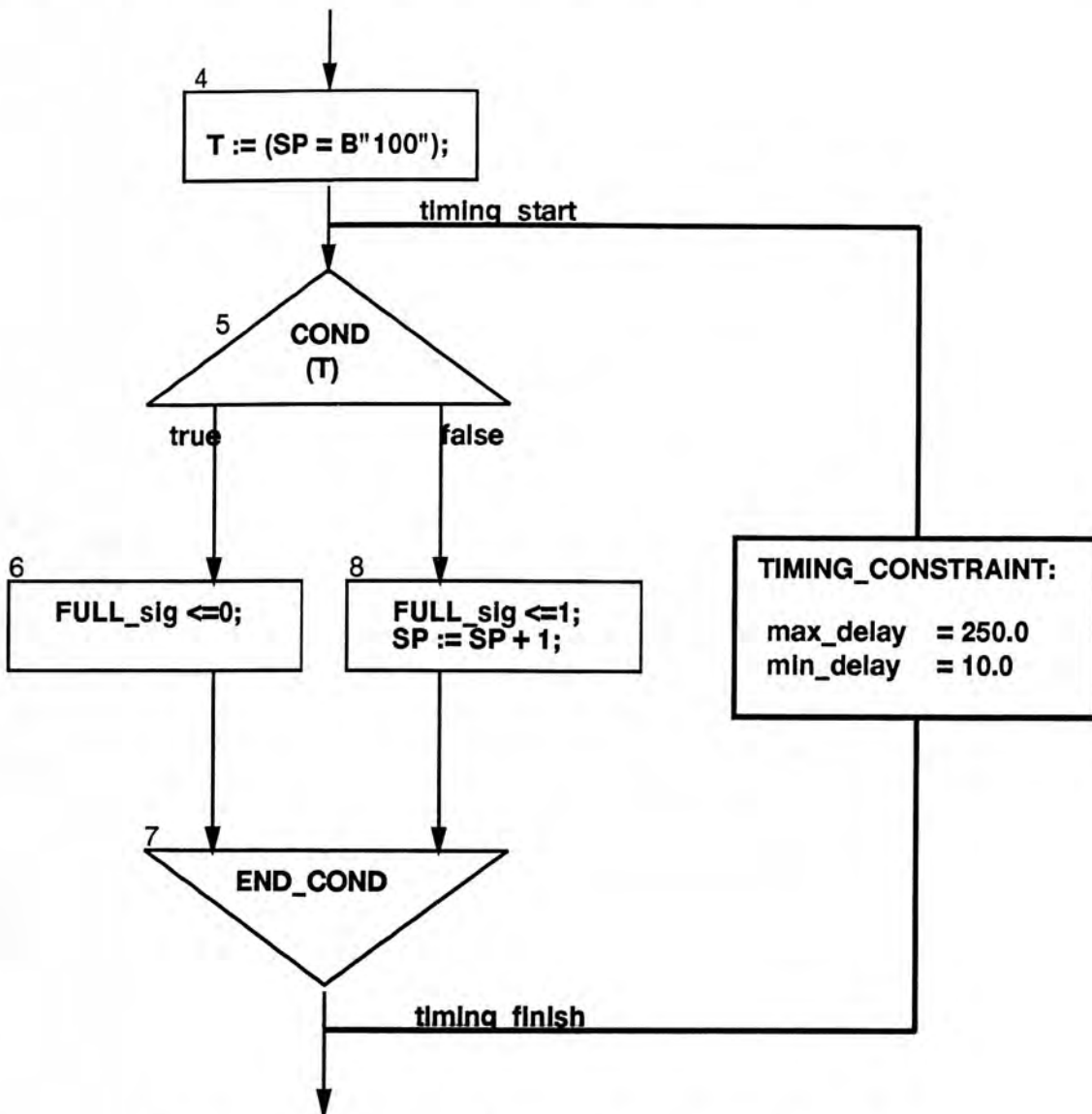


Figure 13.27: The Design1 Example Using the Timing2 View.

### 13.4.3 Removing the Timing Information from the Control Flow Graph

Now I want to construct a design view for a scheduling tool that is not able to utilize timing constraints. Therefore, the design view should remove all timing-related information from the control flow graph. This removal has to be done virtually of course, since there may be other tools accessing the database that can indeed utilize the timing information. I construct this view by hiding all timing-related classes from the view. Secondly, I need to remove references from all object classes that have timing-related domain classes. In the global schema shown in Figure 13.21, the **Cfarc** class and all its subclasses have such references. Therefore, I'll use the **hide** macro-operator that **hides** these functions from all classes in the schema rather than just from one class at a time.

```
class Tcfarc* := hide* [timingstart, timingfinish] from Cfarc*;
```

This command creates three classes, **CfarcT**, **SeqarcT** and **CondarcT**, one for each of the classes in the subschema graph rooted at the **Cfarc** class. For an explanation of the macro-operator the reader is referred to Section 5.3. The three classes then are integrated into the global schema using the class integration algorithm described in Section 7. The resulting global schema, now containing virtual classes generated for all three example design views, is shown in Figure 13.28.

The complete view specification of the **Timing3** design view is given below, while the resulting view schema is shown in Figure 13.29.



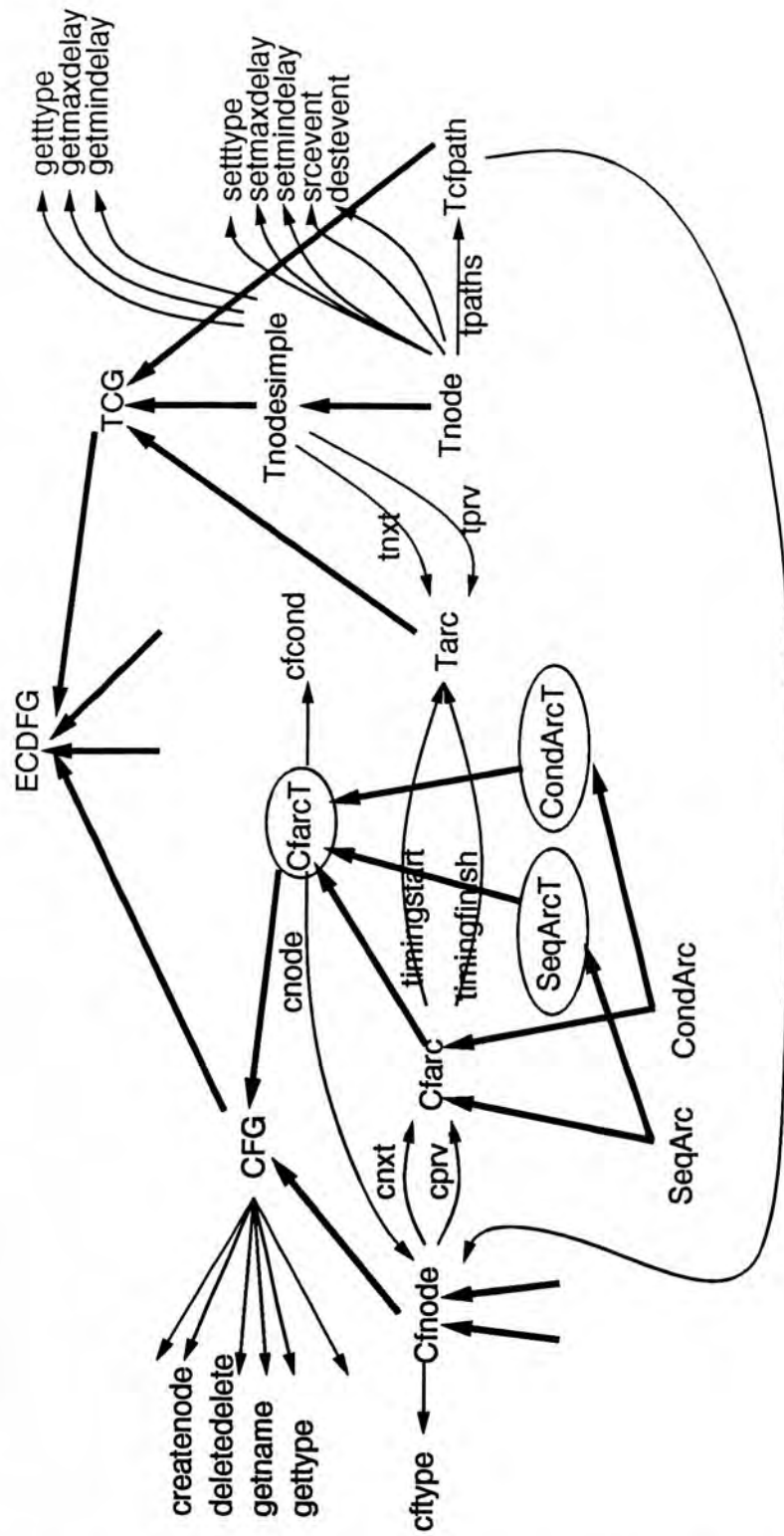


Figure 13.28: Further Refining the Timing Constraint Classes.

## View Creation Script for the Timing3 View:

```

DEFINE-VIEW Timing3
  ADD-SCHEMA (Timing1);
  DELETE-SCHEMA (TCG);
  class Cfarct* := hide* [timingstart, timingfinish] from Cfarct*;
  ADD-CLASS (Cfarct);
  ADD-CLASS (SeqarcT);
  ADD-CLASS (Condarct);
  DELETE-CLASS (Cfarct);
  DELETE-CLASS (Seqarc);
  DELETE-CLASS (Condarct);
  SAVE-VIEW;
END-VIEW

```

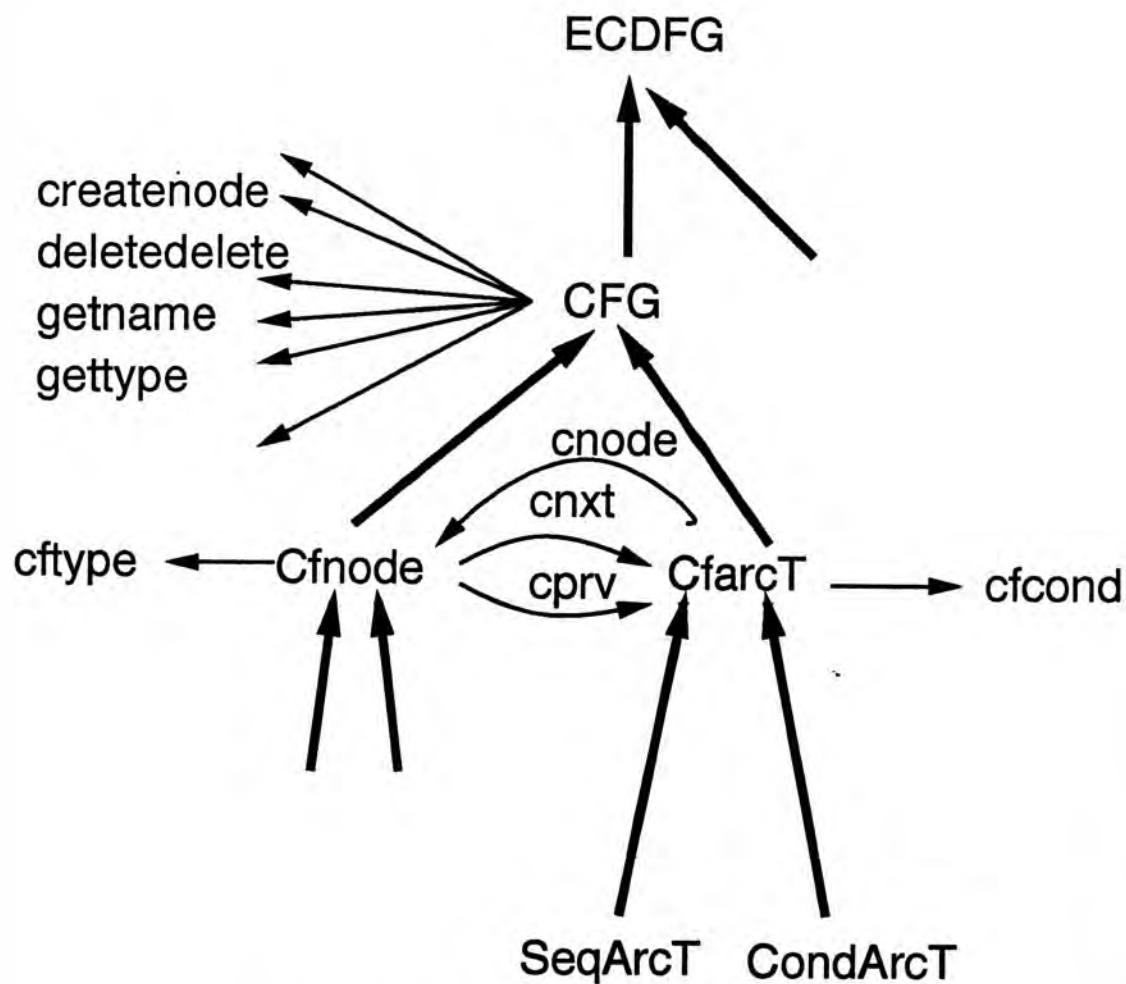


Figure 13.29: The Timing3 View Schema.

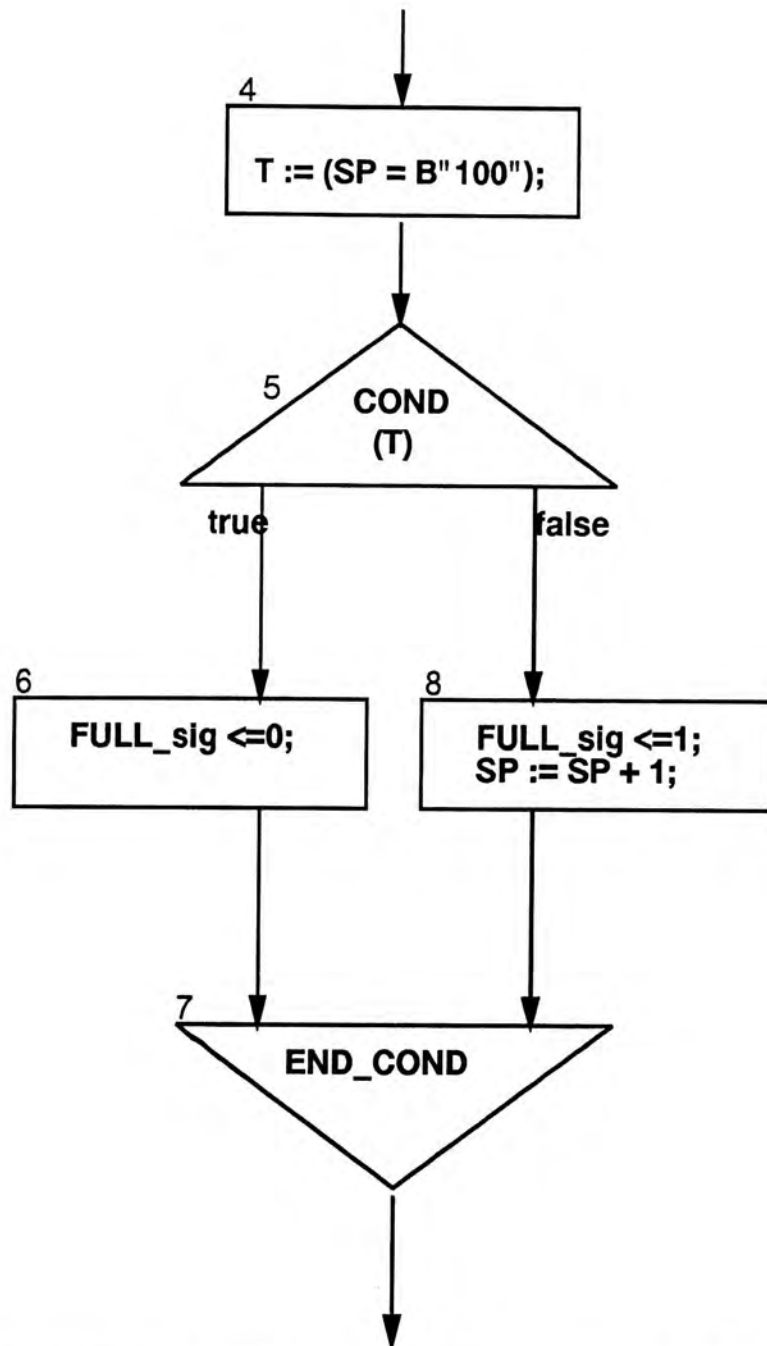


Figure 13.30: The Design1 Example Using the Timing3 View.

Next I'll demonstrate how the control flow graph representation of Design1 shown in Figure 13.23 is perceived through the Timing3 view. The graph, when viewed through the Timing3 view, is depicted in Figure 13.30. Note that there are no timing constraints present in the control data flow graph, when viewed through the Timing3 view. The design data thus is no longer cluttered with information irrelevant to the simple scheduling tool.

#### 13.4.4 Three Design Views of a Control Flow Graph

To summarize, I will demonstrate how diverse the effect of using different design views on the same piece of design can be. On top of Figure 13.31 you can see the design data actually stored in the global database to model the specification of Design1. On the bottom of the figure, there are three views of this design data based on the design views developed in this section. Design tools operating through these views have different perceptions about what type of design data they are dealing with, while the underlying *MultiView* methodology assures the consistent integration of these views into one global data model.

### 13.5 Views For Simplification of the Data Flow Graph Model

The graph compiler, the graph critic as well as the design transformation tools may arbitrarily restructure the control/data flow graph. These tools execute before any of the synthesis tools, such as allocation or scheduling, are performed. Therefore, an appropriate design view for these tools would simply be the complete set of all CDFG design object classes (as for instance shown Figure 13.33).

The allocation tool on the other hand traverses the data flow graph to determine the number and type of operators. It does not change the structure of the data flow graph, and hence should not have access to the graph-manipulation operators. In addition, the allocation tool may be less interested in the details of how the data flow graph structure is implemented. It may for instance not want to see the ports of data flow nodes.

I'll explain the latter based on the example data flow graphs shown in Figure 13.32. The data flow graph in Figure 13.32.a represents the internal design representation of the behavioral design specification " $C = A + B$ ". The graph corresponds



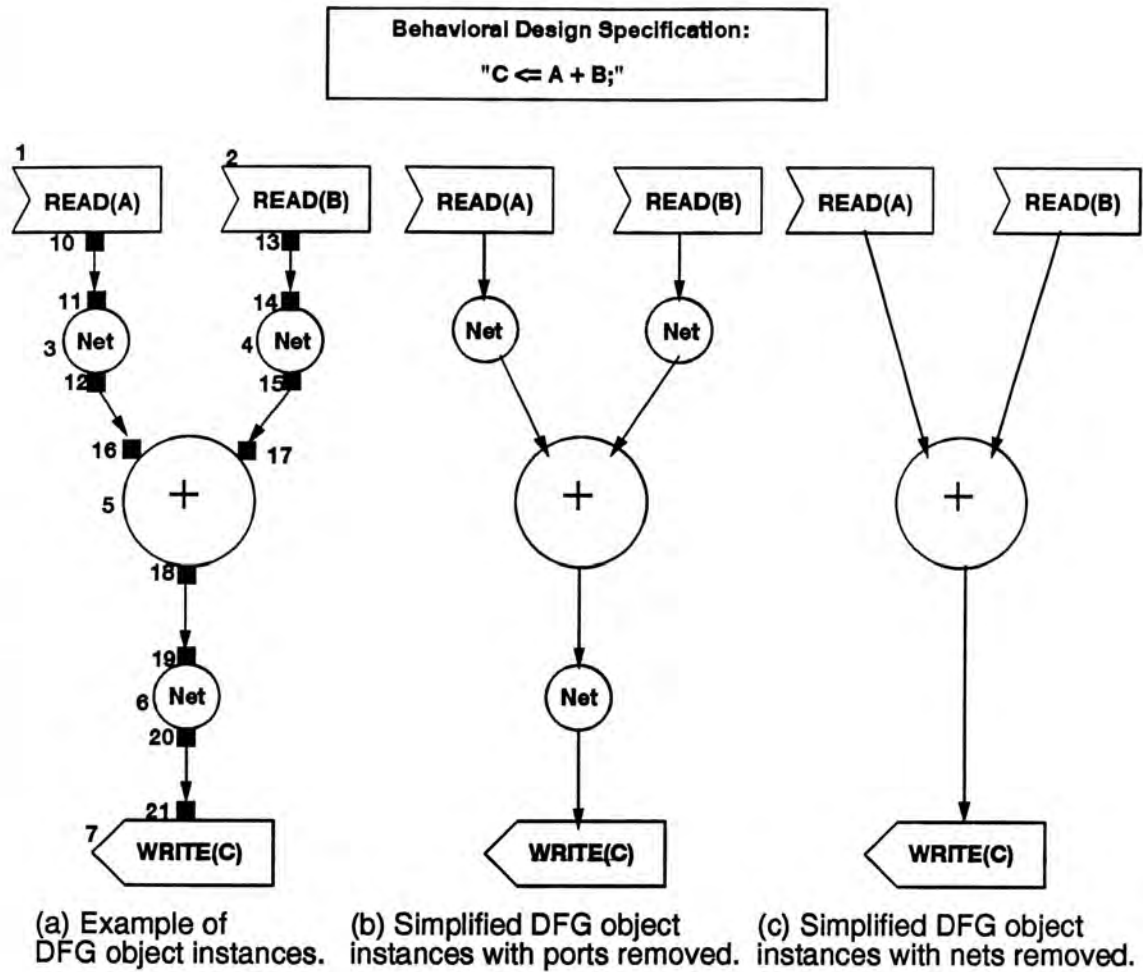


Figure 13.32: An Example of Three Design Views for the Data Flow Graph Model.

to a very detailed representation of the information model. It depicts for instance the net objects and the port objects. The net objects represent the data values that flow between data flow node objects. The port objects are (independent) subobjects of data flow nodes and data flow nets that model the interconnection points of these nodes. The other two data flow graphs shown in the middle and on the right-hand side of the figure demonstrate how the same data flow graph may be represented using less detail. The data flow graph in the middle, for instance, no longer represents the ports of the data flow nodes and nets. In the data flow graph on the left-hand side even the data flow nets are removed. The goal of this section is to develop three design views that reflect the three different perceptions depicted in the examples in Figure 13.32.

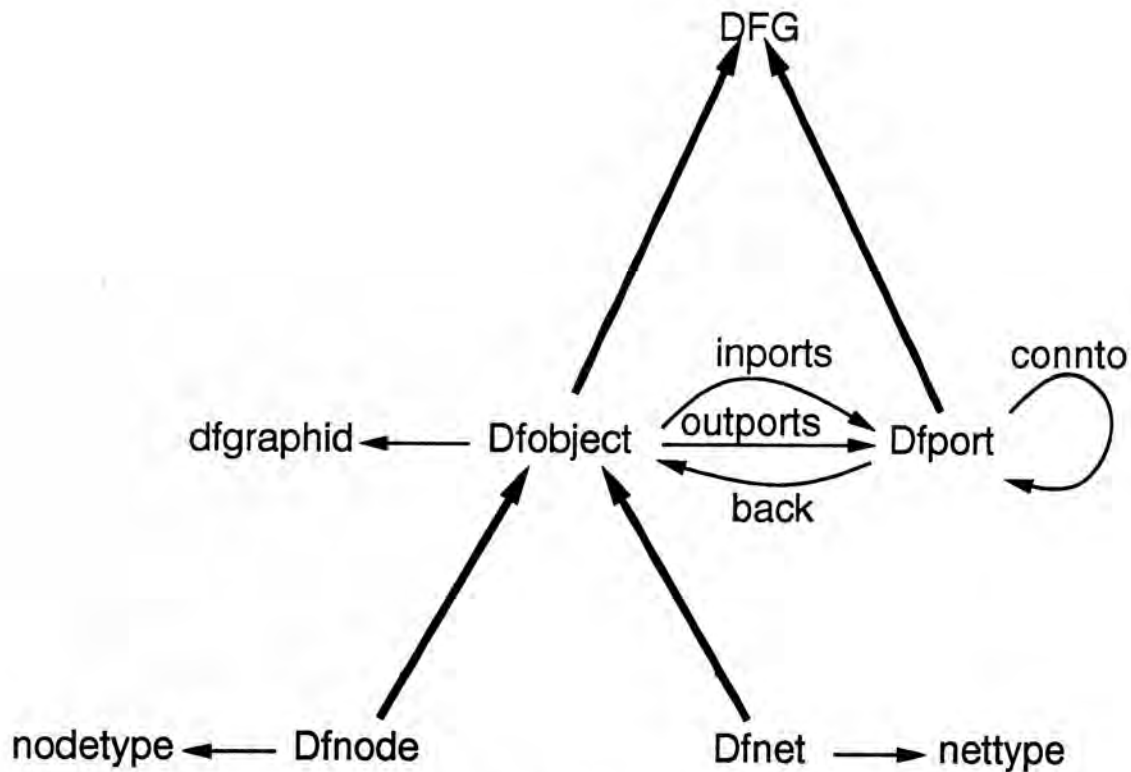


Figure 13.33: The Global Schema for the DFG Model (equal to the DFG1 View).

As basis of this design view construction I assume the global schema graph depicted in Figure 13.33. First, I will explain how to construct a design view, say DFG1, that presents the design data at the level of detail depicted in Figure 13.32.a. Note that this is equivalent to the information model captured by the global schema graph. Hence, the design view DFG1 is simply a subset of the global schema containing all classes related to the data flow graph structure.

**View Creation Script for the DFG1 View:**

```

DEFINE-VIEW DFG1
  ADD-SCHEMA (DFG);
SAVE-VIEW;
END-VIEW

```

The result of the view specification script for the DFG1 view is represented by Figure 13.33.

Next, I want to construct a design view that presents the design data at the level of detail depicted in Figure 13.32.b. Since the example data flow graph in Figure 13.32.b does not model the ports, I must remove the **Dfport** class from the design view DFG2. This also includes the hiding of all references to the **Dfport** class. Since the **Dfobject** class has references to the **Dfport** class, its subclasses **Dfnode** and **Dfnet** also inherit these references. Therefore, the two functions `inports()` and `outports()` have to be removed from all three classes. In *MultiView*, this can be achieved by a macro-operator that works on a complete subgraph of the schema rather than on an individual class. For this purpose, I use the **hide** macro-operator as follows:

```
class Dfobject2* := hide* [inports(),outports()] from Dfobject*;
```

This query generates three virtual classes **Dfobject2**, **Dfnode2**, and **Dfnet2**, namely, one for each of the classes in the subschema graph rooted at the **Dfobject** class. The result of integrating these three classes into the global schema is shown in Figure 13.34.

By removing the port information from the view schema, we would also remove the information necessary to retrieve the connectivity between data flow nodes and data flow nets. Therefore, this connectivity between data flow nodes and data flow nets must be entered into the view in a manner so as not to utilize the port objects. This can be accomplished by a function that composes the `outports()` and `inports()` functions so as to calculate the desired connectivity information.

```

class Dfnode3 := refine Dfnode2 with
  [nxt(Dfnet2) := { return(self.outports.connto.back); };
  prv(Dfnet2) := { return(self.inports.connto.back); }; ];

```



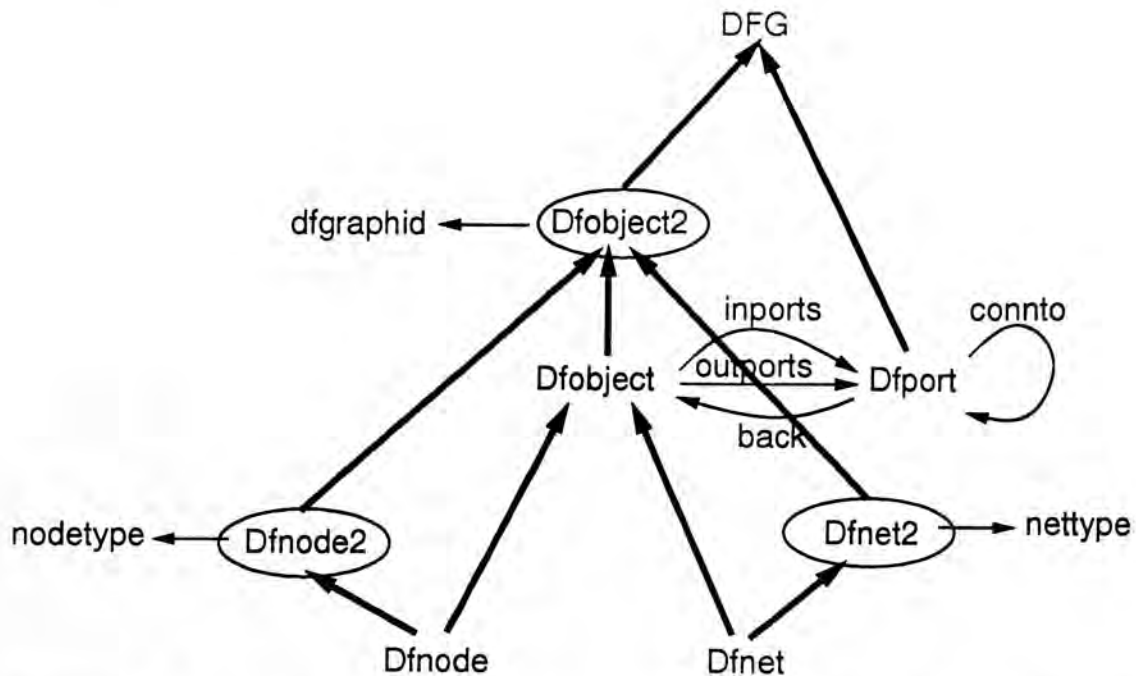


Figure 13.34: Integration of the Virtual Classes **Dfobject2**, **Dfnode2**, and **Dfnet2** into the Global Schema.

This **refine** query constructs a virtual class **Dfnode3**, which has the two additional functions **nxt()** and **prv()** in addition to the type description of its source class **Dfnode2**. **Dfnode3** is integrated into the global schema by placing it directly below its source class as shown in Figure 13.35.

The final view schema for **DFG2** can now be constructed by selecting view classes using the following view specification script.

#### View Creation Script for the **DFG2** View:

```

DEFINE-VIEW DFG2
  class Dfobject2* := hide* [inports(),outports()] from Dfobject*;
  class Dfnode3 := refine Dfnode2 with [nxt(Dfnet2),prv(Dfnet2)];
  ADD-CLASS (Dfnode3);
  ADD-CLASS (Dfnet2);
SAVE-VIEW;
END-VIEW
  
```

The classes **Dfnode3** and **Dfnet2** are selected to be included in the **DFG2** view by the view specification script. These view classes are indicated in Figure 13.36 by

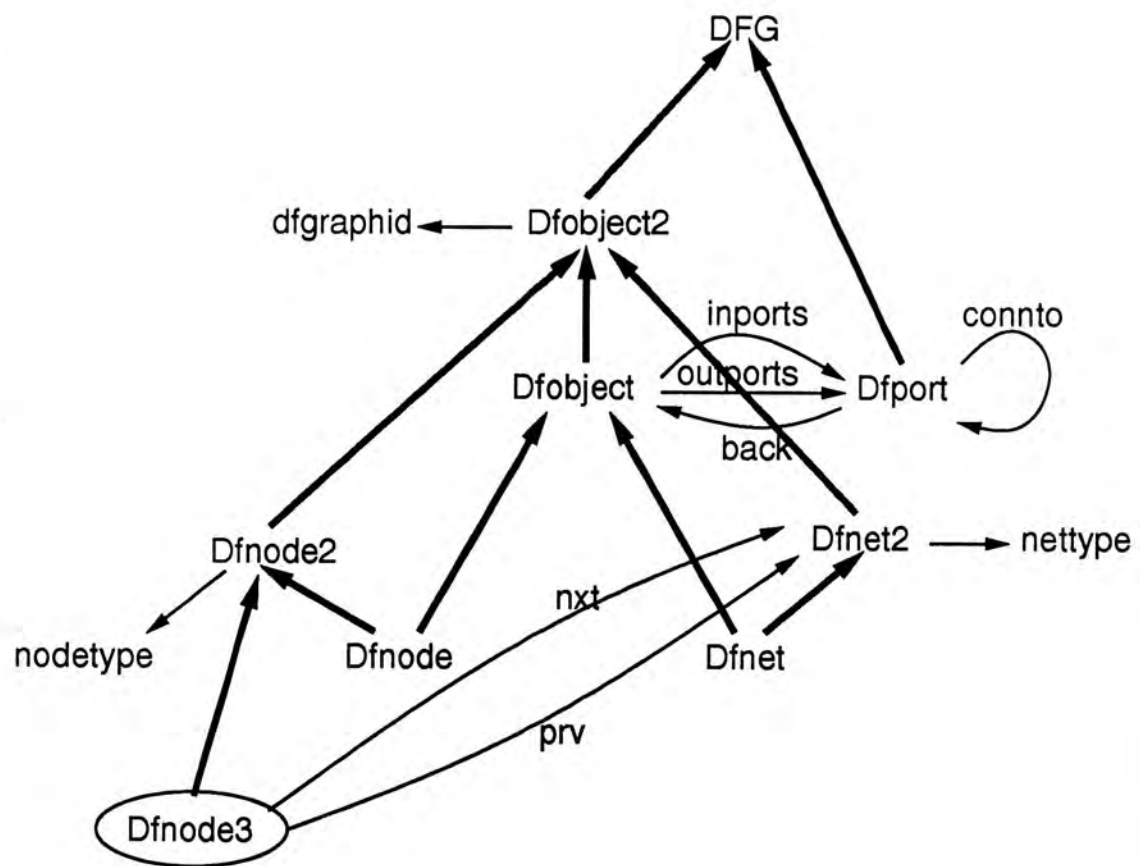


Figure 13.35: Integration of the Virtual Class **Dfnode3** into the Global Schema.

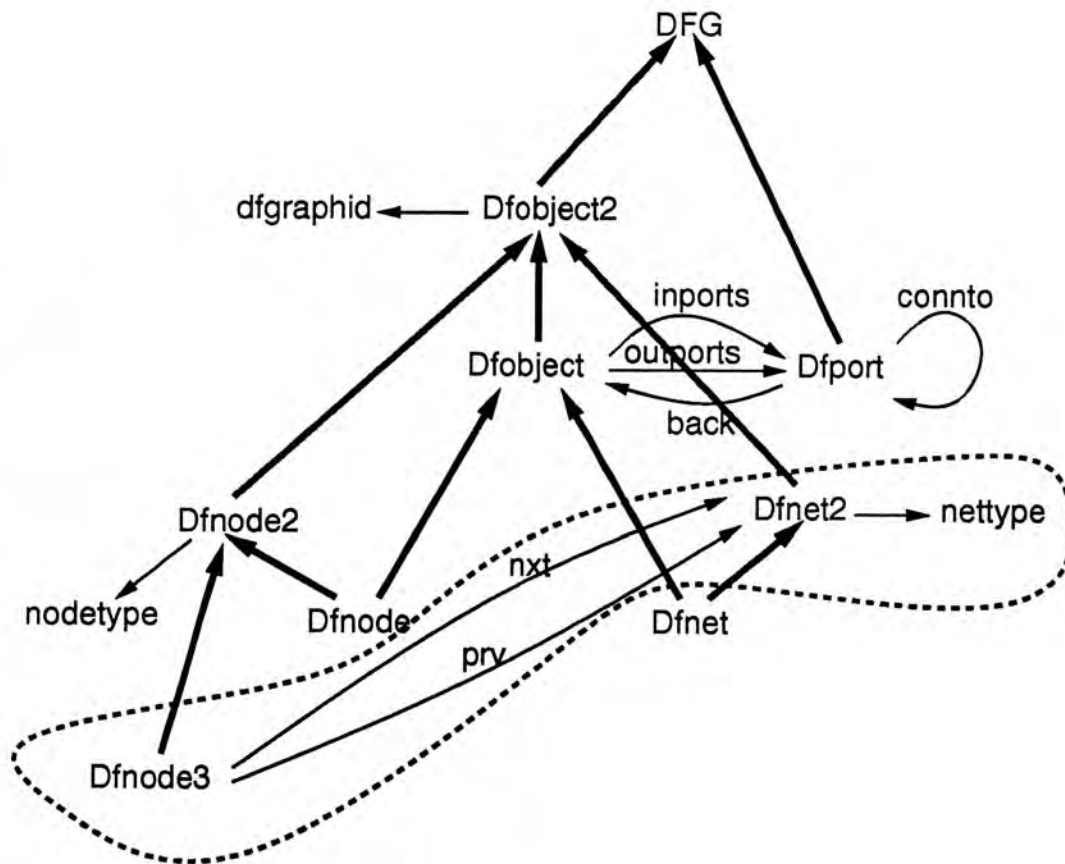


Figure 13.36: Selecting View Classes for the DFG2 View.

encircling them with a dotted line. Finally, *MultiView* extracts these view classes from the global schema and interconnects them using generalization relationships. The resulting view schema DFG2 is depicted in Figure 13.37.

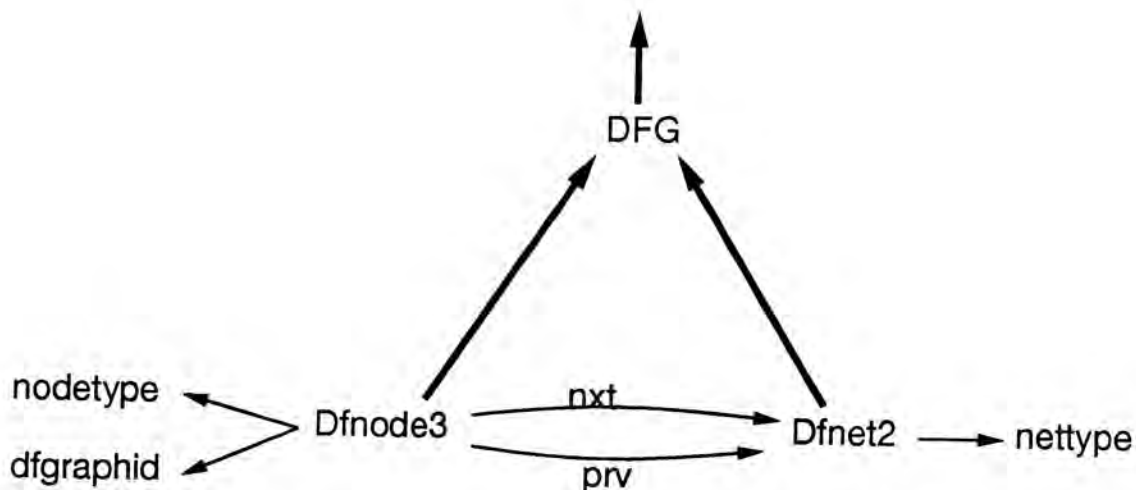


Figure 13.37: The DFG2 View Schema.

Next, I want to construct a design view that presents the design data at the level of detail depicted in Figure 13.32.c. Since the example data flow graph in Figure 13.32.b does not model port nor net objects, I must remove information related to these classes from the design view. This includes the removal of the **Dfnet** classes from the DFG2 view. Note however that the net objects serve as interconnection points between two data flow nodes. Hence, the connectivity information represented by these net nodes must be incorporated into appropriate retrieval functions. This can be accomplished as follows:

```

class Dfnode5 := refine Dfnode3 with
  [nxtdfnodes(Dfnode5) := {return ((cast Dfnode5)
    self.outports.connto.back.outports.connto.back)}];
  prvdfnodes(Dfnode5) := {return ((cast Dfnode5)
    self.inports.connto.back.inports.connto.back)}];
];
  
```

The virtual class **Dfnode5** generated by the **refine** operator would still have access to all functions in the type description of **Dfnode3**, in particular, to the **nxt()** and **prv()** functions that reference the **Dfnet** class. Therefore, I'll take a slightly different approach for creating the desired design view as described below.

## View Creation Script for the DFG3 View:

```

DEFINE-VIEW DFG3
  class Dfnode4 := select from Dfobject2
    where (self in Dfnode);
  class Dfnode5 := refine Dfnode4 with
    [nxtdfnodes(Dfnode5),prvdfnodes(Dfnode5)];
  ADD-CLASS (Dfnode5);
  SAVE-VIEW;
END-VIEW

```

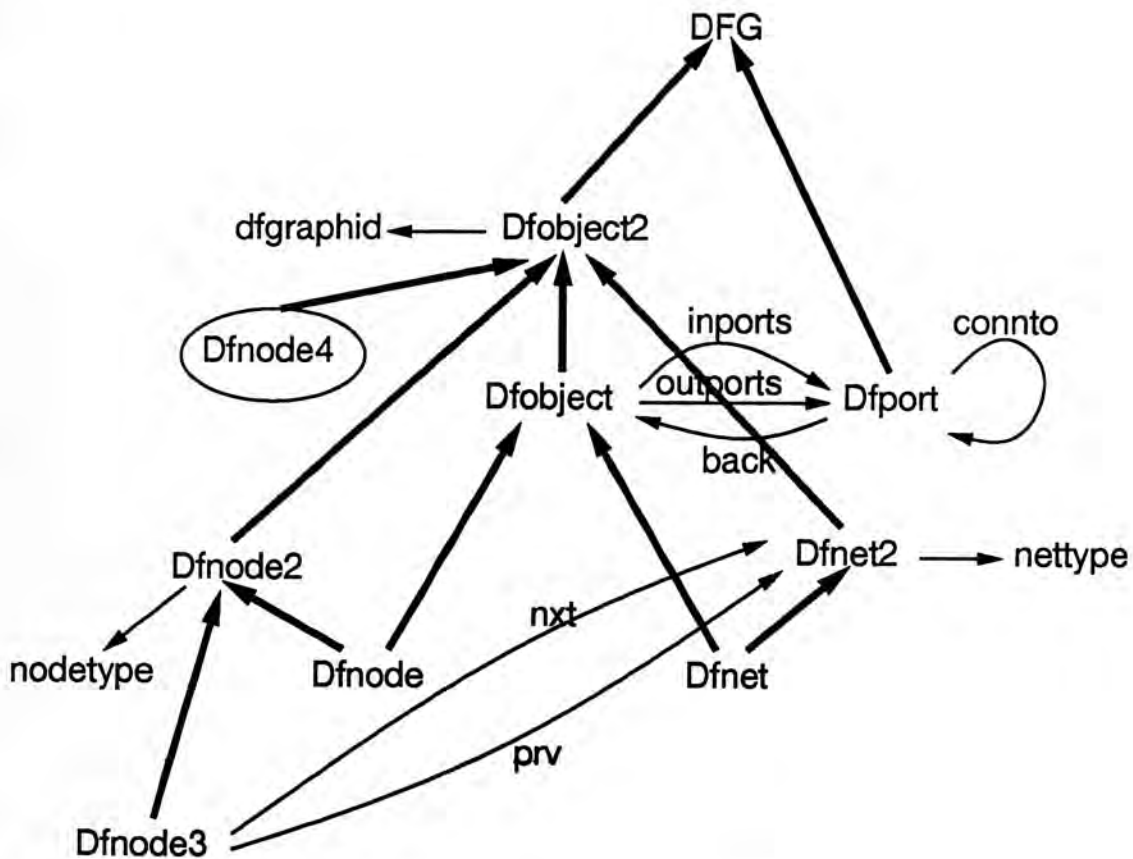


Figure 13.38: Integration of the Virtual Class **Dfnode4** into the Global Schema.

The view specification script for design view DFG3 first generates the virtual class **Dfnode4**, which contains all object instances that belong to the **Dfnode** class. The **Dfnode4** class does however have a limited type description, not allowing access to most functions defined for the original **Dfnode** class. The integration of **Dfnode4** into the global schema is demonstrated in Figure 13.38. Next, the view specification script generates the virtual class **Dfnode5**. **Dfnode5** has the desired functions that indicate the connectivity information between two data flow nodes (while hiding the

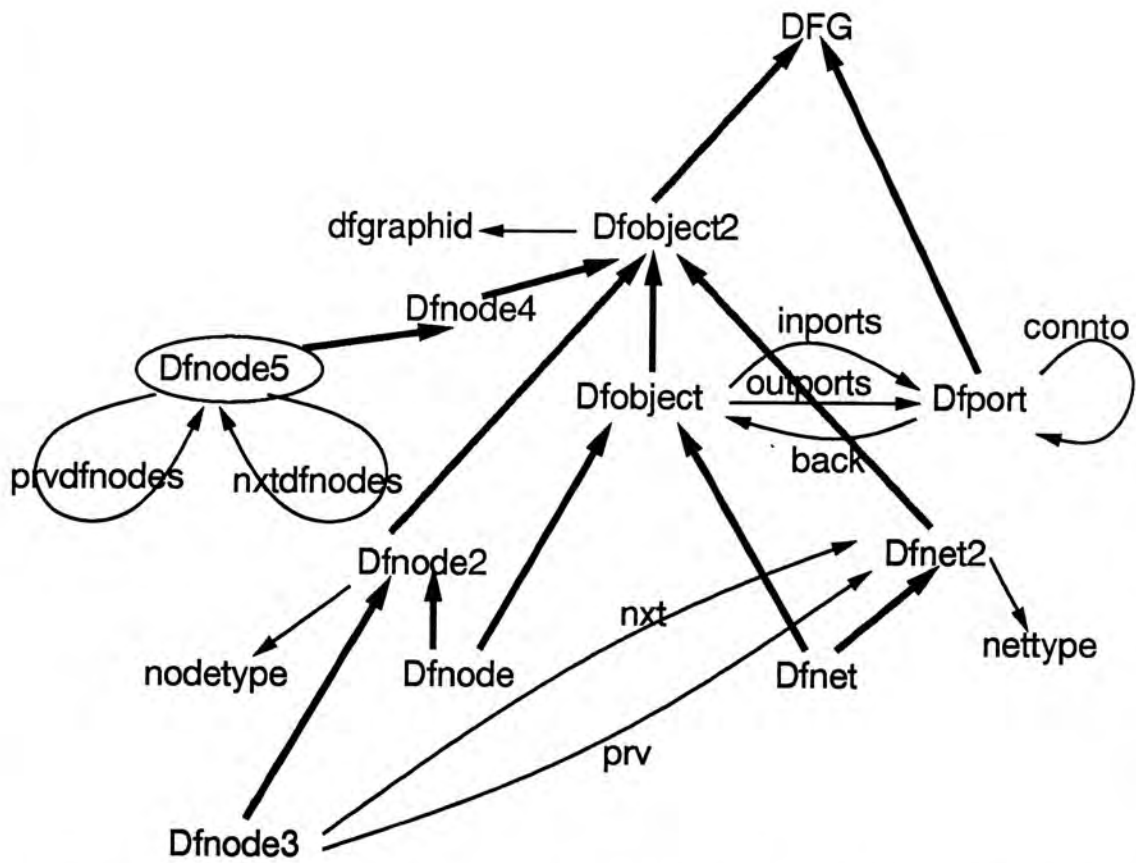


Figure 13.39: Integration of the Virtual Class **Dfnode5** into the Global Schema.

intermediate net and port objects). *MultiView* integrates **Dfnode5** into the global schema by making it a direct subclass of its source class **Dfnode4**. The resulting global schema can be seen in Figure 13.39.

Lastly, the view specification script adds the virtual class **Dfnode5** to the design view DFG3. This selection of view classes is graphically indicated in Figure 13.40 using the dotted line. Lastly, Figure 13.41 depicts the third view schema DFG3 that is generated by *MultiView*.

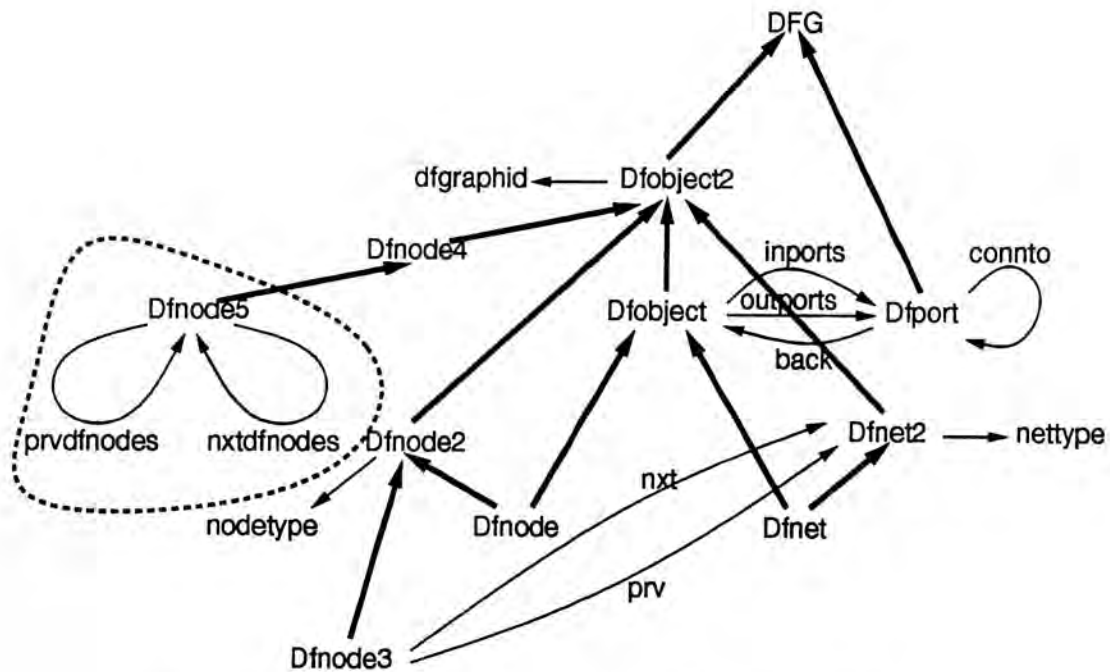


Figure 13.40: Selecting View Classes for the DFG3 View.

In summary, I have demonstrated the creation of three design views DFG1, DFG2 and DFG3 of the global schema modeling data flow classes. These three design views hide different levels of detail from the complex data flow graph model. In particular, for the example graph presented in Figure 13.32, the design views DFG1, DFG2 and DFG3 represent the design data using the data flow graph model on the left-hand side, in the middle, on the right-hand side of the figure, respectively.

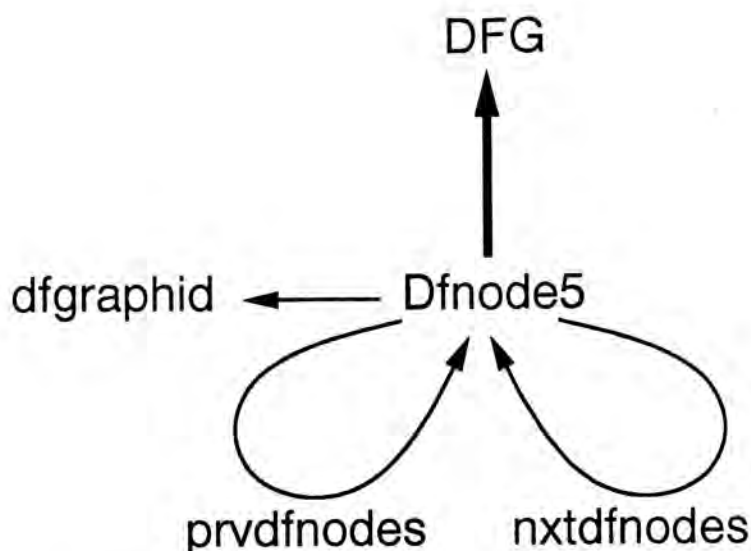


Figure 13.41: The DFG3 View Schema.

## 13.6 Concluding Remarks

In this section, I have presented a number of design view examples constructed using the *MultiView* methodology. These examples demonstrate some of the advantages of a tool integration scheme using design views over directly interfacing with the central database. I'll give a summary of some of these illustrated advantages next:

- design views can filter different levels of detail of the otherwise complex design information (e.g., in the timing constraints view example);
- design views can virtually restructure the design representation graphs so as to narrow the gap between the tool's model and the global model (such as the complexity of the data flow nets and ports);
- design views can hide irrelevant information domains (such as the control/data flow graph classes from the floorplanning design view);
- design views can simplify application-specific processing by augmenting the schema with customized update functions (such as the move-horizontal and move-vertical functions from the floorplanning design view);
- design views increase the level of data consistency by incorporating consistency checks directly into the set of legal access and update functions of the view (such as the update functions for the pins of components in the floorplanning design view);



- design views can be augmented with derived attributes that precompile information that is frequently needed by the users of the design views (such as the `absolute-position()` attribute of pins in the floorplanning design view);
- design views assure the correct update by preparing appropriate update functions and associating them with the view, while hiding all illegal operators from the view.

A number of observations arose from this experience of defining design views for the different behavioral synthesis tasks. I'll list the key observations below.

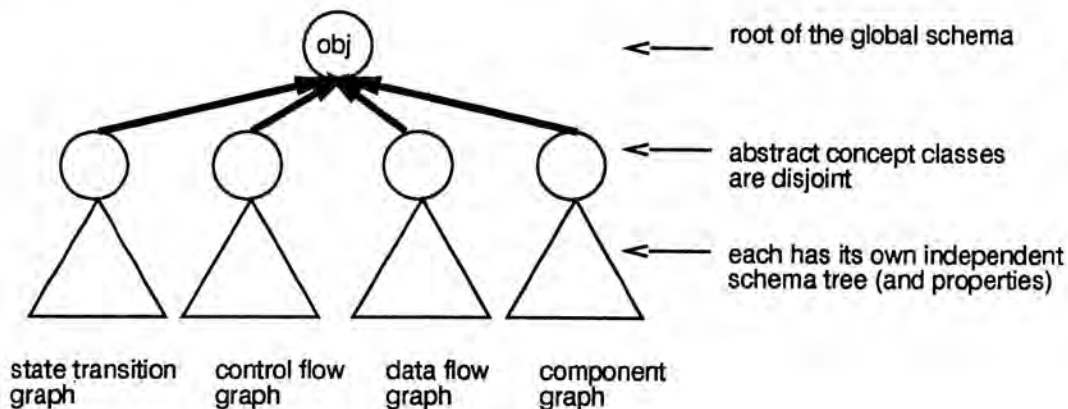


Figure 13.42: Growth of the Global Schema.

- I found that the specification of design views is relatively simple when using the view definition language. It requires of course an understanding of (a) the global model of the design information and (b) the information needs of the particular design task.
- The generation of design views is much less labor-intensive compared to manually having to construct a tool interface and/or data file translators.
- I found that *MultiView* was sufficiently expressive to handle the specification of the design views for all behavioral synthesis tasks that we explored.
- The type-manipulating object algebra operators, such as **hide** and **refine**, were more frequently used than the set-manipulating ones, such as **select** and **union**. A reason for this may be that the base schema already represents an appropriate classification of the design objects into classes. It would be interesting to study other application domains with this question in mind.
- The construction of these design views gave us a good handle on the question of whether the global schema is likely to explode in size with the addition of each new design view. In the worst case, if there are  $P$  different properties

defined in the schema, and if each design view wants a different subset combination of these  $P$  properties, then this could lead up to  $2^P$  different classes. In the example application we have studied we found that class explosion was not a problem. The reason for this may be the following. First, there is generally only a fixed and limited number of different view schemata (design views). Secondly, different view schemata often use subparts of the global schema in the same manner. For instance, both the allocation and binding design tools use the part of the schema referring to the behavioral design representation in the same manner, i.e., for read-only purposes. Similarly, the structural design representation parts are used only to insert and/or delete components from the structure, i.e., all properties referring to the floorplan information are not utilized. In general, we may say that properties are naturally broken into partitions depending on their information domain (See Figure 13.42). Combinations of properties across these 'graphs of independent base classes' are not likely. Second, properties for a given information domain are not arbitrarily grouped into combinations. Instead, it is most likely that subgroups of properties are used together for a given task, are shared across various design views. This of course cuts down the size of the schema considerably.

# Chapter 14

## Conclusions and Future Work

There are two parts to this dissertation with one being in the database and the other in the Computer-Aided Design field, namely,

- the development of a view methodology for object-oriented databases, and
- the application of this view mechanism as tool integration approach for CAD.

While the first represents the major contribution of this dissertation in terms of database research, the second represents the major contribution of this dissertation in terms of CAD research. Therefore, I will divide this section into two parts. When discussing contributions, I will first summarize the contributions of the dissertation in terms of database theory and then in terms of CAD technology. Similarly, when discussing future work, I will first describe how our work on the object-oriented view methodology can be extended and then I will outline the future work that will evolve from using a view-based tool integration scheme in CAD.

### 14.1 Contributions

#### 14.1.1 Database Contributions

In this dissertation, I have presented a novel approach for supporting multiple views of an object-oriented schema, called *MultiView*. A view is defined to be a *virtual, possibly restructured, subschema graph* of the global schema. This is the first systematic approach towards view specification in object-oriented databases that handles complete view schema graphs rather than just individual *virtual classes* as otherwise done in the literature. This approach is simple yet powerful. It, for instance, allows for the customization of a view schema by virtually restructuring both the generalization and the property decomposition hierarchies of the underlying global schema. *MultiView* fulfills all requirements for a view specification mechanism identified in Chapter 4:

1. An object-oriented view in *MultiView* looks and “feels” like a regular object schema. In fact, in *MultiView*, the base schema is just one special (predefined and non-modifiable) type of a view schema.
2. Second, view specification using *MultiView* is (relatively) simple. Non-database experts may be able to use the view system to define their application-specific views. A graphical user interface that supports the interaction of the view definer with *MultiView* would of course be an even more desirable option for computer novices.
3. Third, *MultiView* relieves the view definers from tedious tasks, whenever possible. This is done by automating the two more involved tasks of view specification, namely, class integration and view generalization hierarchy generation. For the other two tasks, namely, for class derivation and view class selection, *MultiView* provides languages that can be used by the view definer to specify the desired information. It can easily be seen that the latter two tasks need user involvement and thus cannot be automated.
4. Fourth, the last requirement of enforcing view consistency is also met by *MultiView*. Namely, *MultiView* is equipped with two view consistency checkers, one for verifying the closure of the property decomposition hierarchy and the other for assuring the minimality and the consistency of the class generalization hierarchy. Both algorithms not only verify the correctness of the view, but if found incorrect, they take the appropriate actions necessary to remedy the problem.

*MultiView* breaks view specification into the following four tasks: (1) customization and derivation of virtual classes using object algebra, (2) integration of virtual classes into *one* consistent global schema graph, (3) extraction of both virtual and base classes from the global schema as required by the view, and (4) generation of a view class hierarchy for these selected view classes. *MultiView*'s division of view specification into a number of well-defined subtasks, some of which have been successfully automated, makes it a powerful tool for supporting the specification of views by non-database experts while enforcing view consistency. In this dissertation, I have presented solutions to each of the subtasks related to the proposed view paradigm.

For the first task (the customization of classes), I have formally defined a set of object algebra operators that can be used to customize the type structure and object membership of virtual classes. I study in particular the class relationships between the virtual and the source classes, since this is required for solving *MultiView*'s second task.

The maintenance of one global schema graph that contains all base and virtual classes is the key feature of *MultiView* that ultimately supports the formation of arbitrarily complex view schema graphs composed of both base and virtual classes. In other words, the integrated global schema graph corresponds to the backbone of our view support system based on which all view schemata are being designed. *MultiView* therefore needs to support the integration of all virtual classes into this one global schema graph. Class integration, the second task of *MultiView*, tackles the problem of how a virtual class relates to, and can be integrated with, the remaining classes in the global schema. A class in an object schema is interrelated with other classes via an is-a hierarchy (for property inheritance) and via a property decomposition hierarchy (for forming complex objects). Class integration needs to guarantee the consistency of these class relationships when adding new classes.

For the second task, *MultiView* supports the automatic integration of classes rather than requiring manual placement of classes in the schema graph (and then checking the entered information for consistency). Automatic classification does not only prevent the introduction of inconsistencies into the schema, but it also considerably simplifies the task of the view definer. This decreases the view creation time, and, more importantly, it allows a non-database expert to specify an application-specific view on his or her own without having to be concerned with the tedious class integration task.

For these reasons, I present in this dissertation an algorithm for the automatic classification of virtual classes into a global schema graph. I have identified two class integration problems, called the type inheritance mismatch and the *is-a* incompatible subset/subtype hierarchies problem. Our classification algorithm solves both problems. Both solutions are based on type lattice classification proposed in the literature, the essence of which is the creation of additional intermediate classes that restructure the schema graph. I present proofs of correctness and a complexity analysis for the classification algorithm.

Furthermore, I characterize classification requirements of virtual classes derived by different object algebra operators. This characterization helps us to reduce the complexity of the classification task for most cases. For instance, I am able to reduce classification from quadratic to linear complexity for classes derived by the Select, the Union, and the Difference operators and to constant complexity for those derived by the Refine operator.

For the third task (the extraction view classes), I have designed a view definition language that can be used by the view definer to select the view classes required

for a particular user view from the global schema. This includes both base and virtual classes. I present examples that illustrate the ease of use of the view definition language.

For the last task (the view class hierarchy generation), I have developed two algorithms for the automatic generation of the view generalization hierarchy, one for a global schema graph with single and one for a graph with multiple inheritance. Both algorithms are guaranteed to generate a *complete* and *minimal* and *consistent* view schema. Automatic view generation is preferable over its manual specification, since the proposed view generation algorithms not only simplify the task of the view definer by automating the more tedious tasks of view specification, but they also enforce the consistency of the resulting view. In the dissertation, I present proofs of correctness, complexity analysis, and numerous illustrative examples for the proposed algorithms.

*MultiView* provides support for schema design in terms of automating some parts of the specification process (for instance tasks two and four mentioned above) and in terms of enforcing the consistency of a view schema. For instance, in this dissertation I have presented an algorithm that only verifies the closure of the property decomposition hierarchy of a view schema but, if the view is found to be incomplete, transforms the view schema into a minimal yet closed view. Similarly, I present an algorithm for checking the *completeness* and *consistency* of the class generalization hierarchy of the view. Again, the consistency checker will support appropriate actions necessary to remediate the inconsistency of the view generalization hierarchy. In this dissertation, I have also introduced the concept of *view independence*, which I argue to be a fundamental requirement for any view mechanism developed for object-oriented databases. I prove *MultiView* to be *view independent*.

To summarize, the contribution of this work in the database field is three-fold: (1) the idea of generating one global schema graph incorporating both virtual and base classes as foundation of view support, (2) the development of a general class integration algorithm for object-oriented data models that supports the generation of such a graph, and (3) the development of view consistency checkers.

A major contribution of the proposed approach lies in its simplicity compared to alternative proposals [Shil89, Gilb90], and hence the potential ease in implementing it with existing OODB technology. Note that the paradigm is not specific to a

particular OODB model. This generality allows the *MultiView* approach to be incorporated into most existing OODBs. *MultiView* would enrich these systems by allowing them to support a more powerful notion of views. Our paradigm builds on existing work in as much as it is independent of the class derivation operators chosen from the set of proposed operators in the literature [Heil90, Kim89, Scho91, Rund92b].

### 14.1.2 Computer-Aided Design Related Contributions

In this dissertation, I have presented a characterization of design information in terms of a three-layered design object model (Section 1.4). The primary advantage of the proposed design object model is its comprehensiveness – whereas other object models proposed in the CAD literature generally capture only some subset of our model. The design object model represents a vehicle based on which features of existing object models can be compared in a systematic manner.

High-level synthesis is based on the interaction of various design tools working at different levels of abstraction and in different information domains, and thus the need for integrating these tools into a cooperative framework has become apparent. A design database forms the foundation for such a framework. In the dissertation, I have outlined the main issues of design database support for high-level synthesis and I have presented some solutions. In particular, I reported on the unified design representation model for behavioral synthesis (BDOM) that captures of design object types required for behavioral synthesis.

More importantly, I have proposed a new mechanism for tool integration in design environments. Design views simplify the design data access for tools with diverse representation needs. Our approach solves the problem of point-to-point data translations between all pairs of design tools that exchange information by providing customized interfaces for these tools to the central database. To my knowledge, this is the first time that the concepts of database views has been applied to CAD applications.

There is some work in the CAD literature that uses the terms 'views', however, the associated meaning and with it the usage of the view construct is different and very simplistic [Lann91, Knap85]. Sometimes the different information domains of an application, such as the behavioral domain and the structural domain in behavioral synthesis are referred to as *views* of the design [Knap85]. Note, however, that these two domains are two different parts of the global schema, each captured by different data structures. The behavioral-graph and the structural-graph thus are trivial views in *MultiView*; they both correspond to simple subsets of the complete information

model. In fact, note that the structural view (domain) of the design is generated from the behavioral view (domain) using design tools, i.e., there are actual design decisions involved in generating the structural view (domain). For this reason, this does not correspond to concept of design views presented in this dissertation.

The view-based object server approach offers all the advantages of the object-server approach, such as one explicit design information model, controlled access to shared data, integrity control, the development of one interface per tool only, the possibility for incremental update, and a flexible object model that is powerful enough to capture the complex design information. Furthermore, the approach offers additional advantages, such as ease through customization, robustness, flexibility, extensibility, and consistency.

Views *customize* the global object model so that it suits the needs of particular design tools, i.e., they may hide irrelevant information or restructure the information according to the user's perception. Having narrowed the gap between the database and the tool's local model of the design information simplifies the task interfacing with this central object server. It is also likely to reduce errors and misunderstandings caused by differences in the global database schema and the local tool models of the design information.

*Robustness* of the CAD system is achieved by shielding tools from changes in the data model. As long as the view on which a particular tool operates is maintained the tool need not worry about any changes to the global data model, such as extensions of the conceptual model or the reimplementation of the model using different data structures. The database will take care of these mappings between the view and the global schema automatically.

This view-based organization provides *flexibility* since new customized views, i.e., tool interfaces, can be created rapidly. This supports the extensibility of the system since new tools can be easily added to the system by simply defining a new view (or possibly using an existing one). Existing tools can work with new ones without having to develop a new interface to these tools. This is likely to decrease the development time of new tools, since they can exploit existing access functions of the global object model. It will also increase the productivity of design tool developers, who generally spend a major portion of their time designing and implementing tool interfaces.

The approach also guarantees the *extensibility* of the data model since new information can be added to the design data without disturbing existing views. One unified object model as suggested in the literature represent a bottleneck to the overall system for the following reason. If a change of the global object model is



required due to the introduction of a new tool into the system, then all existing tools may have to be revised.

An added advantage of the design view approach is the increased level of consistency since each design tool have only a restricted access privilege to the global design data. In fact, views can be used to control the access privileges of design tools at any stage during the design process. For instance, the floorplan view of the register-transfer design provides operations to change the positioning of components of the design, such as the `set-position()` and `modify-position()` operators, but not for the modification of the actual components themselves. The database system can thus guarantee that - as long as a tool operates during this view - the semantics of the design have not been modified or corrupted, meaning, the positioning of components may have changed but that the type of components and their interconnectivity have not been changed. Note that this could be guaranteed without actually having to verify it. This is a very important contribution since the verification of the equivalence of two design versions is generally a costly and in some circumstances impossible task.

## 14.2 Future Work

### 14.2.1 Future Database Work

While the correctness of the algorithms presented in this dissertation has been tested in isolation, a prototype implementation of the complete *MultiView* system on top of some existing object-oriented database system represents a necessary next step for further evaluating our approach. For this purpose, we need to evaluate available OODBs for their suitability. This evaluation includes the requirement for the OODB to support some basic features, such as that an object instance can be a member of many classes simultaneously and thus can take on different types [Scho91]. Several implementation issues immediately arise from this requirement, such as the development of efficient strategies for method resolution.

Other important database issues, such as query processing techniques on views, materialized views, and view updates, which have been extensively studied in the context of the relational model, must now be reexamined in the context of object-oriented data models.

In this dissertation, I have restricted the set of object algebra operators to be object-preserving [Scho91]. I believe strongly that this is sufficient for many

application domains, in particular, since the join operator can be simulated using the refine operator. However, it would be interesting to investigate whether, and if so how, *MultiView* could be extended to also handle object-generating algebra operators [Kim89, Heil90].

As indicated in Chapter 7.1, the classification problem for object-oriented models is not decidable since it may involve the comparison of arbitrary functions and predicates. Hence, the development of a realistic *subsumes()* function for some of the 'standard' object models available on the market needs to be investigated. The goal of such a project would be not to restrict the expressive power of the model nor the constructs used for deriving new classes, while guaranteeing that the *subsumes()* function stays computable.

A closely related problem is the recursiveness of type descriptions, i.e., the existence of loops in the property decomposition. This requires the development of a *subsumes()* function that can handle recursive type descriptions.

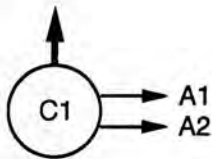
In this dissertation, I took the extreme position of requiring complete classification, i.e., of automatically forcing the placement of a virtual class in its (syntactically) most appropriate position. A more detailed discussion on this issue can be found in Chapter 7.1. If one were interested in dropping this stringent restriction and allow the user to place new classes in non-optimal but nonetheless syntactically correct positions, then our work on the classification algorithm would still be a useful component of classification. In particular, it could serve as core for a consistency checker that verifies the correctness of a manual class placement, once specified (See Section 10.1). The automatic classifier could also be utilized to guide the user during an interactive view specification phase.

Furthermore, the design of a graphical interface for incremental view definition would be a useful feature for application domains. It would open the avenue for non-database experts to utilize *MultiView* to define their desired application-specific views. Indeed, the development of *MultiView* has been driven by our need to provide multiple design views for CAD tools working on a central database, and therefore this would be a next logical step in the addressing this problem.

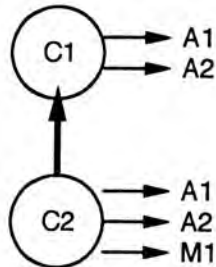
Lastly, I propose view optimization as a new open problem that arises from our research on object-oriented views. View optimization is concerned with restructuring the view specification such as to minimize the number of intermediate classes that are created. View optimization attempts to minimize the size of the resulting schema graph in terms of the number of (virtual) classes, whereas query processing on views is concerned with finding the best execution plan for a given view schema. View optimization is permanent since it influences the final schema structure whereas

1. C2 = refine C1 with [ M1 ];  
 2. C3 = hide [ A1 ] from C2;  
 desired view classes: C3

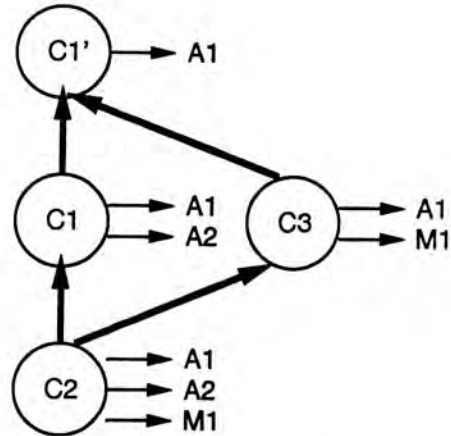
(1a) Example View Specification.



(1b) Initial schema.



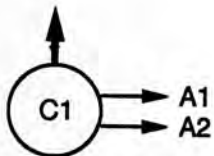
(1c) After refine operator.



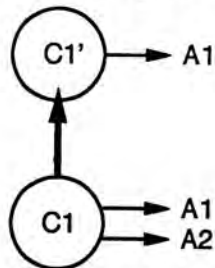
(1d) After hide operator.

1. C1' = hide [ A1 ] from C1;  
 2. C3 = refine C1' with [ M1 ];  
 desired view classes: C3

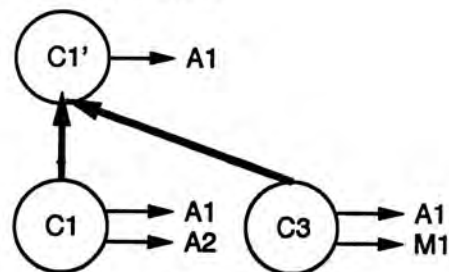
(2a) Rewriting the Example View Specification.



(2b) Initial schema.



(2c) After hide operator.



(2c) After refine operator.

C3 = refine ( hide [ A1 ] from C1 ) with [ M1 ];  
 C3 = hide [ A1 ] from ( refine C1 with [ M1 ] );

(3) Composing Algebra Operators into Complex Queries.

Figure 14.1: An Example of the View Construction Optimization Problem.

view query optimization is temporary since it is concerned with the performance of executing an individual query. As a matter of course, view optimization may have an influence on the effectiveness of query optimization, and thus should take the potential impact of its decisions on query processing into account.

### 14.2.2 Future Work Related to Computer-Aided Design

I have outlined the types of design information that need to be captured by a CAD database using a three-layered model in Section 1.4. It is necessary of course to refine the proposed design object model for a particular application. I have, for instance, done this for the behavioral synthesis domain in this dissertation. In general, one would have to add domain-specific attributes to the design entity objects, and one would have to create a particular design representation modeling the objects at the design data level.

Even though I discussed the different functionalities of design object management in terms of this one design object model, this does not necessarily mean that the object model is to be 'implemented' as one system. It is in fact more likely that it will be spread over a distributed database system providing different support technology for the different layers. For instance, the design process layer tends to be mainly read but scarcely updated compared to the actual design data layer, which is constantly updated. Also, there is obviously a larger quantity of data stored at the design data than at the design process level. Therefore, different database technology supporting these distinct requirements should be chosen for each of the three levels of the design object model. The identification of a proper architecture for the design object model and its development is an important research issue for future CAD frameworks.

Design views provide us with a mechanism to control the manipulation of the design data during each stage of the design process. This is an important tool for increasing the data consistency. More importantly, I could exploit this knowledge at the next higher level of the design object model, namely, for design entity management (See middle of Figure 14.2). Knowing through which view a piece of design has been changed may give us several advantages: (1) it may allow us to determine which of the domain-specific attributes may be out-of-date and thus should be recalculated and which ones are still unchanged, and (2) it may help us to determine how two pieces of design relate to one another. An example of the first case would be that all attributes relating to the structural design, such as the component count, the types-of-components, and the architectural style of the design, do not have to be verified after accessing a design entity through the floorplanning view.

On the other hand, attributes related to the floorplan, in particular, the size-of-the-floorplan, should be flagged as out-of-date. This may result in significant saving in performance, since the calculation of these quality measures is often costly or not even computable.

Similarly, I could attempt to exploit the use of design views at the design data level for design process management (See top of Figure 14.2). Design process management deals, among other issues, with the sequencing of tools in order to accomplish certain design tasks. The design process manager often knows which tool needs to be invoked next for a given piece of design depending on the design goal and the state of the design. The design process manager can therefore prepare a particular design view for the current design, before it has actually been requested. This is so because there generally is a fixed design view for a given design task. This may increase the speed of the data exchange, in particular, if a larger quantity of data is to be exchanged.

While I have proposed the use of design views at the design data level, it remains an open question whether design views can be applied to the next higher level of design object management. Primary users of the information at the design entity level and thus of views at this level are human designers and project managers. Design views may thus be useful if designers want to see the information organized according to their project goals, i.e., they may be interested in only the derivation history of a design in the behavioral domain. Then a design view in which all information related to the structural domain is hidden should be created for this user group. The view mechanism could of course also be used to limit access privileges at the design entity level. Lastly, equally unexplored is the exploitation of the view-based approach to the design process level.

The design views approach for tool integration has one drawback, namely, it requires the development of a unified representation model of the application domain and that all tools interface with this unified representation - using views. Consequently, the approach cannot be used to salvage an already existing CAD system or set of design tools, rather it would require the rewrite of these tools (or at least their interfaces). The design views approach is very useful however when developing a new CAD system. It would be interesting to determine how our approach functions in a mixed environment where sets of related tools (that work on the same level of abstraction or on the same information domain) use the view-based approach, while the overall CAD system uses a different (possibly the file-based approach) among tool groups. Such an architecture would then offer the advantages that both existing and new tools can be combined into one CAD system that gradually will move from being file-based to being view-based. Extensive performance

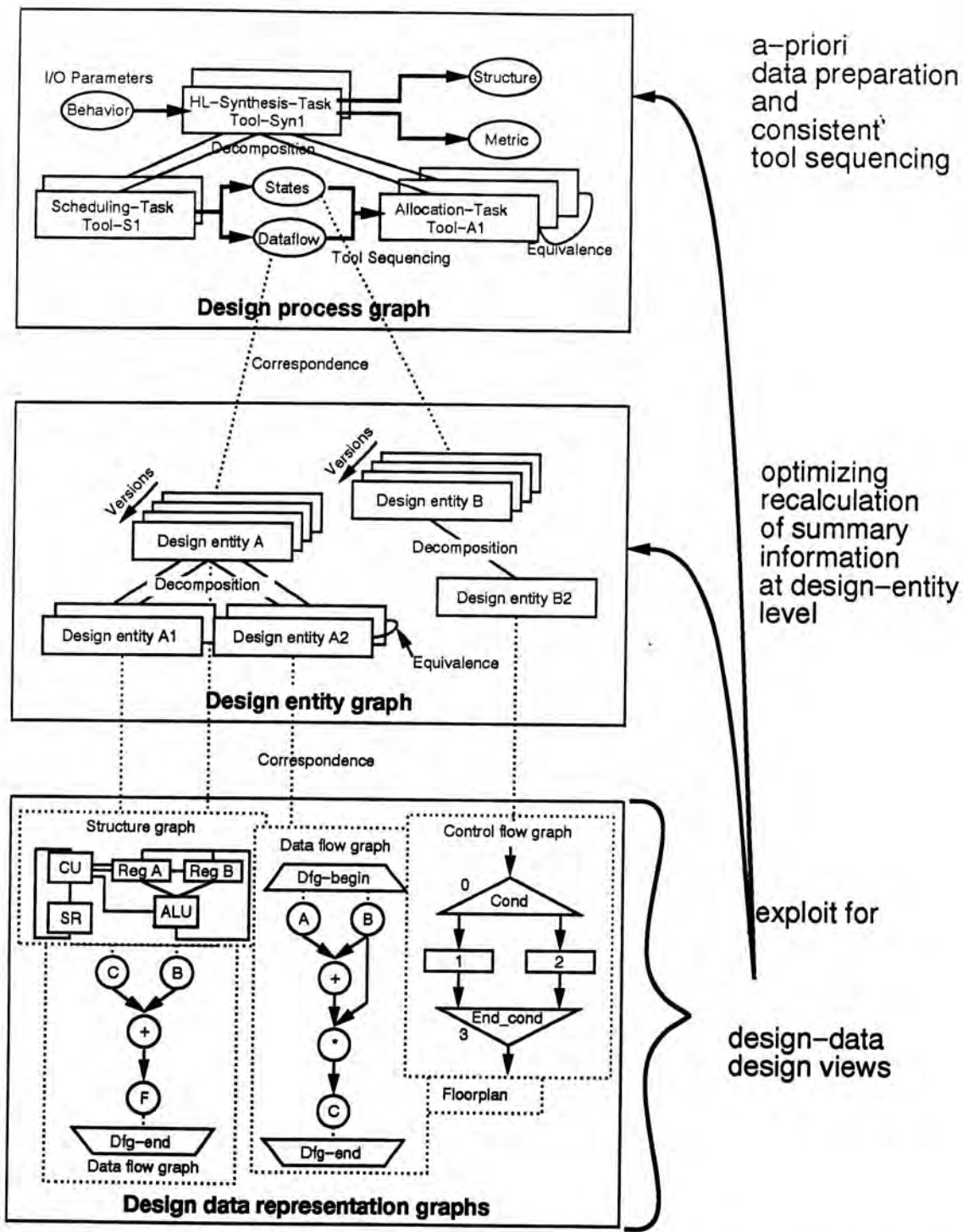


Figure 14.2: Exploiting Design Views at Higher Levels of Design Management.

studies have to be executed to evaluate the advantages and disadvantages of these types of configurations.

A comparison study of different tool integration schemes would be a worthwhile exercise. It would not only further validate the *design view* approach but, if the results are encouraging, it may also increase the likelihood of it being accepted by industry. An additional hope would be that this evaluation may discover a classification scheme that determines which tool integration approach works best for which application domains.

Lastly, *MultiView* is not specific to behavioral synthesis applications. Therefore, the application of *MultiView* to other design environments and possibly other application domains, such as software engineering environments, should be explored.

# Bibliography

- [Abit87] Abiteboul, S. and Hull, R., IFO: A Formal Semantic Database Model, in *ACM Trans. on Database Systems*, vol. 12, issue 4, pp. 525 - 565, Dec. 1987.
- [Abit91] Abiteboul, S., and Bonner, A., Objects and Views, in *Proc. SIGMOD*, pp. 238 - 247, May 1991.
- [Afsa89] Afsarmanesh, H., Brotoatmodjo, E., Ryeon, K. J., Parker, A. C., The EVE VLSI Information Management Environment, *IEEE Int. Conf. on CAD*, pp. 384 - 387, 1989.
- [Aho72] Aho, A. V., Garey, M.R., and Ullman, J.D., The Transitive Reduction of a Directed Graph, *SIAM J. Computing*, Vol. 1, No. 2, June 1972.
- [Aho74] Aho, A. V., Hopcroft, J. E., and Jeffrey, D. U., *The Design and Analysis of Computer Algorithms*, Addison-Wesley Pub. Company, 1974.
- [Andr87] Andrews, T., and Harris, C., Combining Language and Database Advances in an Object-Oriented Development Environment. *Proc. of the Conf. on Object-Oriented Programming Systems, Languages and Applications*, Orlando, FL., Oct. 1987.
- [Ang92] Ang, R., and Dutt, N. D., "Equivalent Design Representations and Transformations for Interactive Scheduling," *Proc. ICCAD'92*, Nov. 1992.
- [Atki87] Atkinson, M. P. and Buneman, O. P., Types and Persistence In Database Programming Languages, in *ACM Computing Surveys*, vol. 19, no. 2, pp. 105 - 190, June 1987.
- [Bach73] Bachman, C. W., The Programmer as Navigator, *CACM* 16, no. 1., Nov. 1973.
- [Ball88] Ballou, N., Chou, H.T., Garza, J.F., Kim, W., Petrie, C., Russinoff, D., Steiner, D., and Woelk, D., Coupling an Expert System Shell with an Object-Oriented Database System, *Journal of Object-Oriented Programming*, vol. 1, no. 1, pp 12 - 21, April 1988.
- [Banc88] Bancilhon, F., et al., The Design and Implementation of O2 an Object-Oriented Database System, *Rapport Technique Altair 20-88*, April 1988.
- [Banc81] Bancilhon, F., and Spyrtos, N. Update Semantics of Relational Views, *ACM Trans. on Database Systems*, vol. 6, iss. 4, pp. 557 - 575, Dec. 1981.



- [Bane87a] Banerjee, J., Kim, W., Kim, H. J., and Korth, F., Semantics and Implementation of Schema Evolution in Object-Oriented Databases, *Proc. of SIMOD*, pp. 311 - 322, May 1987.
- [Bane87b] Banerjee, J., Chou, H.T., Garza, J.F., and Kim, W., Woelk, D., Ballou, N., and Kim, H.J., Data Model Issues for Object-Oriented Applications, *Transactions on Office Information Systems*, vol. 5, no. 1, pp. 3 - 26, Jan. 1987.
- [Baro81] Baroody, J. and DeWitt, D., An Object-Oriented Approach to Database System Implementation, *Transaction on Database Systems*, vol. 6, no. 4, pp. 576 - 601, Dec. 1981.
- [Bato85] Batory, D. S., and Kim, W., Modeling Concepts for VLSI CAD Objects, *ACM Tran. on Database Systems*, vol. 10, no. 3, pp. 322 - 346, Sep. 1985.
- [Beck90] Beckmann, R., Schenk, W., Pusch, D., and R. Joehnk, The TREEMOLA Language, Reference Manual, Version 4.0, Univ. of Dortmund, Germany, Report No. 364, 1990.
- [Beec87] Beech, D., Groundwork for an Object Database Model, *Research Directions in Object-Oriented Programming*, (eds. Bruce Shriver, Peter Wegner), pp. 317 - 354, 1987.
- [Bing90] Bingley, P., and P. Van der Wolf, A Design Platform for the NELSIS CAD Framework, *DAC'90*, 146 - 149.
- [Blac88] Blackburn, R. L., Thomas, D. E., and Koenig, P.M., Linking the Behavioral and Structural Domains of Representation for Digital System Design, *IEEE Trans. on CAD*, vol. CAD-6, No. 1, Jan. 1987.
- [Borg87] Borgida, A., Conceptual Modeling of Information Systems, in *On Knowledge Base Management Systems*, Springer-Verlag. Brodie, M.L. and Mylopoulos, J. (eds), 1987.
- [Bouz84] Bouzeghoub, M., MORSE: A functional query language and its semantic data model, *Proc. of 84 Trends and Applications of Databases*, IEEE-NBS, Gaithersburg, 1984.
- [Brach83] Brachman, R. J., What IS-A is and isn't: An Analysis of Taxonomic Links in Semantic Networks, in *IEEE Computer*, pp. 30 - 36, Oct. 83.
- [Brac85] Brachman, R. J., and Schmolze, J. G., An Overview of the KL-ONE Knowledge Representation System, *Cognitive Science*, 9, 1985.
- [Bret90] Bretschneider, F. Kopf, C., Lager, H., Hsu, A., and Wi, E., Knowledge Based Design Flow Management, *Proc. IEEE Internat. Conf. on Computer-Aided Design*, pp. 350 - 353, 1990.
- [Brod87] Brodie, M. L. and Mylopoulos, J. (eds), *On Knowledge Base Management Systems*, Springer-Verlag, 1987.
- [Brod84] Brodie, M. L., Mylopoulos, J., and Schmidt, J. W. (Eds), *On Conceptual Modelling*, Springer Verlag, 1984.

- [Bush89] Bushnell, M. and Director, S., Automated Design Tool Execution in the Ulysses Design Environment, *IEEE Trans. on Computer-Aided Design*, vol. 8, no. 3, pp. 279 - 287, 1989.
- [Camp89] Camposano, R. and W. Rosenstiel, Synthesizing Circuits from Behavioural Descriptions, *IEEE Trans. on CAD*, Vol. 8., No. 2, Feb. 1989.
- [Camp88] Camposano, R. and R. M. Tabet, Design Representation for the Synthesis of Behavioral VHDL Models, Research Report, IBM General Technology Division, Burlington, VT, RC 14282, Dec. 1988.
- [Care88] Carey, M. J., DeWitt, D. J., Richardson, J. E., and Shekita, E. J., Storage Management for Objects in EXODUS, *Object-Oriented Concepts, Applications, and Databases*, Addison-Wesley, 1988.
- [Caso90] Casotto, A., A. R. Newton, and A. Sangiovanni-Vincentelli, Design Management Based on Design Traces, *DAC'90*, 136 - 141.
- [CFI92] CAD Framework Initiative, Panel Discussion, *29th ACM/IEEE Design Automation Conf. (DAC'92)*, Anaheim, California, June 1992.
- [Cham75] Chamberlin, D.D., Gray, J.N., and Traiger, I.L. View Authorization and Locking in a Relational Database System, in *Proc. Nat'l. Computer Conf.*, AFIPS Press, vol. 44, pp. 425 - 430, 1975.
- [Chen76] Chen, P. P., The Entity-Relationship Model - - Toward a Unified View of Data, in *ACM Trans. on Database Systems*, vol. 1, issue 1, pp. 9 - 36, Mar. 1976.
- [Chen90] Chen, G.D. and D. Gajski, An Intelligent Component Database for Behavioral Synthesis, *DAC'90*, 150 - 155.
- [Chil68] Childs, D. L., Feasibility of a Set-Theoretic Data Structure - A General Structure Based on a Reconstituted Definition of Relation, in *Proc. of the IFIP Congress, Information Processing 1968*, vol. 1, pp. 420 - 430, 1969.
- [Chiu92] Chiueh, T.-C., and Katz, R.H., Incremental Meta-Data Construction for VLSI Design Databases, *EDAC'92*, pp. 399 - 403, Feb. 1992.
- [Chiu92] Chiueh, T.-C., and Katz, R.H., Intelligent VLSI Design Object Management, *EDAC'92*, pp. 410 - 414, Feb. 1992.
- [Codd70] Codd, E. F., A Relational Model of Data for Large Shared Data Banks, in *Comm. ACM*, vol. 13, issue 6, pp. 377 - 387, June 1970.
- [Codd79] Codd, E. F., Extending the Database Relational Model to Capture More Meaning, in *ACM Trans. on Database Systems*, vol. 4, issue 4, pp. 397 - 434, Dec. 1979.
- [Cope84] Copeland, G. and Maier, D., Making Smalltalk a database system, *SIGMOD '84*, Boston, MA, pp. 316 - 325, June 1984.
- [Cosm83] Cosmadakis, S. and Papadimitriou, Updates of Relational Views, *Proc. 2nd ACM SIGACT-SIGMOD Symp. on Principle of Database Systems*, Mar 1983.

- [Dani99] Daniell, J. and Director, S., An Object Oriented Approach to CAD Tool Control, *Proc. 26th Design Automation Conference*, pp. 197 - 202, 1989.
- [Date90] Date, C. J., *An Introduction to Database Systems*, Vol. I, Fifth Edition, Addison-Wesley Publishing Company, Inc., 1990.
- [Daya82] Dayal U. and Bernstein, P.A., On Correct Translation of Update Operations on Relational Views, *ACM Trans. on Database Systems*, vol. 7, iss. 3, pp. 381 - 416, Sept. 1982.
- [Ditt86] Dittrich, K. R., Object-oriented database systems: the Notion and the Issues, *1986 International Workshop on Object-Oriented Database Systems* Pacific Grove, California, pp. 2 - 4, Sep. 1986.
- [Dutt90] Dutt, N., T. Hadley, and D., Gajski, An Intermediate Representation for Behavioral Synthesis, *DAC'90*, pp. 14 - 19, Jun 1990.
- [EDIF] EDIF Electronic Design Interchange Format, Version 200, Recommended Standard EIA-548, (ed. by P. Stanford and P. Mancuso), 1989.
- [Eijn91] Van Eijndhoven, J. T.J., G.G. De Jong, and L. Stok, The ASCIS Data Flow Graph: Semantics and Textual Format, EUT Report 91-E-251, 1991.
- [Fish87] Fishman, D., et al., Iris: An Object-Oriented Database Management System, in *ACM Trans. on Office Information Systems*, vol. 5, no. 1, Jan. 1987.
- [Foo90] Foo, S. Y, and Takefuji, Y., Databases and Cell-Selection Algorithms for VLSI Cell Libraries, *Computer*, Vol. 23, No. 2, pp. 18 - 30, Feb. 1990.
- [Gajs92] Gajski, D. D., Dutt, D. N., Wu, A. C.-H., and Lin, S. Y.-L., *High-Level Synthesis: Introduction to Chip and System Design*, Kluwer Academic Press, 1992.
- [Garl87] Garlan, D., Views for Tools in Integrated Environments, Carnegie-Mellon University, Tech. Report CMU-CS-87-147, Ph.D. dissertation, May 1987.
- [Gilb90] Gilbert, J. P., Supporting User Views, *OODB Task Group Workshop Proceedings*, Ottawa, Canada, Oct. 1990.
- [Gilb90b] Gilbert, J. P., PolyView: An Object-Oriented Data Model for Supporting Multiple User Views, Univ. of Cali, Irvine, Tech. Rep. #90-05, Ph.D. dissertation, 1990.
- [Gupt89] Gupta, R., Cheng, W. H., Gupta R., Hardonag, I. and Breuer, M. A.. An Object-Oriented VLSI CAD Framework, *IEEE Computer*, vol. 22, no. 5, 28 - 37, May 1989.
- [Hame90] Hamer, P. and Treffers, M., A Data Flow Based Architecture for CAD Frameworks, *Proc. IEEE Internat. Conf. on Computer-Aided Design*, pp. 482 - 485, 1990.
- [Hamm81] Hammer, M. and McLeod, D. J., Database Description with SDM: A Semantic Data Model, in *ACM Trans. on Database Systems*, vol. 6, no. 3, pp. 351 - 386, Sept. 1981.

- [Harr86] Harrison, D. S., Moore, P., Spickelmier, R. L., and Newton, A. R., Data Management and Graphics Editing in the Berkeley Design Environment, pp. 24 - 27, *ICCAD'86*.
- [Huds86] Hudson, S.E. and King, R., CACTIS: A database system for specifying functionally-defined data., *Proc. of the Workshop on Object-Oriented Databases*, 1986.
- [Hull87] Hull, R. and King, R., Semantic Database Modeling: Survey, Applications and Research Issues, in *ACM Computing Surveys*, vol. 19, no. 3, pp. 201 - 260, Sept. 1987.
- [Heil90] Heiler, S., and Zdonik, S. B., Object views: Extending the vision, In *Proc. IEEE Data Engineering Conf.*, Los Angeles, pp. 86 - 93, Feb. 1990.
- [Jong91] De Jong, G.G., Data Flow Graphs: System Specification with the most unrestricted semantics, *Proc. European Conf. on Design Automation*, Amsterdam, pp. 401 - 405, Feb. 1991.
- [Kash92] Kashai, Y., Flow - A Concurrent Methodology Manager, *EDAC'92*, pp. 20 - 24, Feb. 1992.
- [Katz90] Katz, R. H., Towards a Unified Framework for Version Modeling in Engineering Databases, *ACM Computing Surveys*, Vol. 22, No. 4, pp. 375 - 408, Dec. 1990.
- [Katz85] Katz, R. H., Information Management for Engineering Design, *Surveys in Computer Science*, 1985.
- [Katz82] Katz, R. H., A Data Base Approach for Managing VLSI Design Data, *Proc. of 19th DAC*, pp. 274 - 282, 1982.
- [Kaul90] Kaul, M., Drost, K., and Neuhold, E.J., ViewSystem: Integrating Heterogeneous Information Bases by Object-Oriented Views, In *Proc. IEEE Data Engineering Conf.*, pp. 2 - 10, Feb. 1990.
- [Kent79] Kent, W., Limitations of record-based information models, *Comm. ACM*, vol. 4, issue 1, pp. 107 - 131, Mar. 1979.
- [Kent78] Kent, W., *Data and Reality: Basic Assumptions in Data Processing Reconsidered*, North Holland, 1978.
- [Khos86] Khoshafian, S. N., and Copeland, G. P., Object Identity, in *Proc. OOPSLA '86*, ACM, pp. 406 - 416, Sep. 1986.
- [Kim87] Kim, W., Woelk, D., Garza, J., Chou, H.T., Banerjee, J., and Ballou, N., Enhancing the object-oriented concepts for database support, *IEEE proceeding of Data Engineering 1987*, pp. 291 - 292, 1987.
- [Kim89] Kim, W., A model of queries in object-oriented databases, In *Proc. Int. Conf. on Very Large Databases*, Aug. 1989, pp. 423 - 432.
- [Knap85] Knapp, D. W., and A. C. Parker, A unified representation for design information, In *Proc. CHDL-85*, Elsevier, 1985.

- [Kowa85] Kowalski, T. J., and Thomas, D. E., The VLSI Design Automation Assistant: What's in a Knowledge Base, *22<sup>nd</sup> Design Automation Conference*, pp. 252 - 258, June 1985.
- [Lann91] Lanneer, D., Goossens, G., Catthoor, F., Panwels, M., and H. De Man, An Object-oriented framework supporting the full high-level synthesis trajectory, *CHDL'91*, pp. 281 - 300, 1991.
- [Lis89] Lis, J. S., and D. D. Gajski, VHDL Design Representation in the VHDL Synthesis System (VSS), Tech. Rep. #89-15, Uni. of California, Irvine, 1989.
- [Lis88] Lis, J. S., and D. D. Gajski, Synthesis from VHDL, *Proc. of ICCD*, Oct. 1988.
- [Maie87] Maier, D., and Stein, J., Development and Implementation of an Object-Oriented DBMS, *Research Directions in Object-Oriented Programming*, B. Shriver and P. Wegner, (Eds.), MIT Press, 1987.
- [Maie86] Maier, D., Stein, J., Otis, A., and Purdy, A., Development of an Object-Oriented DBMS, in *Proc. OOPSLA '86*, pp. 472 - 482. Sep. 1986.
- [McFa78] McFarland, M., The Value Trace: A Database for Automated Digital Design, Tech. Report DRC-01-04-80, Carnegie-Mellon University, Dec. 1978.
- [McFa88] McFarland, M. C., Parker, A. C., and Camposano, R., Tutorial on High Level Synthesis, *DAC*, 1988.
- [Missi89] Missikoff, M., and Scholl, M., An Algorithm for Insertion into a Lattice: Application to Type Classification, *Foundations of Data Organization and Algorithms*, 3<sup>rd</sup> Int. Conf., FODD'89, France, pp. 64 - 82, June 1989.
- [Missi87] Missikoff, M., MOKA: A User-friendly Front-End for Knowledge Acquisition, *Int'l Workshop on Database Machines and Artificial Intelligence*, Minowbrook, N.Y., July 1987.
- [Mylo80] Mylopoulos, J., Bernstein, P. A., and Wong, H.K.T., A Language Facility for Designing Database-Intensive Applications, in *ACM Trans. on Database Systems*, vol. 5, issue 2, pp. 185 - 207, June 1980.
- [Neum83] Neumann, T., On representing design information in a common database, *Proc. of the Engineering Design Applications at the Data Base Week*, San Jose, pp. 81 - 87, 1983.
- [Orai86] Orailoglu, A. and D. D. Gajski, Flow graph representation, *DAC'86*, pp. 503 - 509, June 1986.
- [Pate90] Patel, M. R. K., A Design Representation for High Level Synthesis, *Proc. European Conf. on Design Automation (EDAC)*, pp. 63 - 73, 1990.
- [Peck88] Peckham, J. and Maryanski, F., Semantic Data Models, *ACM Computing Surveys*, vol. 20, issue 3, pp. 153 - 189, Sept. 1988.

- [Ramm90] Rammig, F., (editor), IFIP WG 10.2, Workshop on Electronic Design Automation Frameworks, Nov. 1990.
- [Reiss90] Reiss, S. P., Connecting tools using message passing in the Field environment, *IEEE Software*, 7, 4, pp. 57 - 66, July 1990.
- [Reiss84] Reiss, S. P., Graphical Program Development with PECAN Program Development Systems, *Proc. of ACM SIGSOFT/SIGPLAN Software Eng. Symposium on Practical Software Development Environments*, April 1984
- [Rowe79] Rowe, L. A. and Shoens, K. A., Data Abstraction, Views and Updates in RIGEL, In *Proc. of SIGMOD 1979 Conf.*, pp. 71 - 81, 1979.
- [Rund89] Rundensteiner, E. A., Hawkes, L. W., and Bandler, W., On Nearness Measures in Fuzzy Relational Data Models, in *International Journal of Approximate Reasoning*, vol. 3, no. 3, pp. 267 - 298, July 1989.
- [Rund90a] Rundensteiner, E. A., and Bic, L., Set Operations in a Data Model Supporting Complex Objects, in *Int. Conf. on Extending Data Base Technology (EDBT)*, March 1990. (published in *Lecture Notes in Computer Science*, vol. 416, pp. 286 - 300, 1990.)
- [Rund90b] Rundensteiner, E. A., and Gajski, D. D., A Design Representation for High-Level Synthesis, Info. and Computer Science Dept., UCI, Tech. Rep. 90-27, Sep. 1990.
- [Rund90c] Rundensteiner, E. A., Gajski, D. D., and Bic, L., Component Synthesis Algorithm: Technology Mapping for Register Transfer Descriptions, *IEEE Int. Conf. on Computer-Aided Design*, pp. 208 - 211, Nov. 1990.
- [Rund91a] Rundensteiner, E. A., Bic, L., Gilbert, J., and Yin, M., A Semantic Integrity Framework: Set Restrictions for Semantic Groupings, in *IEEE Int. Conf. on Data Engineering (ICDE-7)*, April 1991.
- [Rund91b] Rundensteiner, E. A., and Gajski, D. D., A Design Data Base for Behavioral Synthesis, *High Level Synthesis Workshop*, 1991.
- [Rund91c] Rundensteiner, E. A., and Gajski, D. D., BDEF: The Behavioral Design Data Exchange Format Info. and Computer Science Dept., UCI, Tech. Rep. 91-34.
- [Rund92d] Rundensteiner, E. A., *MultiView: A Methodology for Supporting Multiple View Schemata in Object-Oriented Databases*, *18th Int. Conference on Very Large Data Bases (VLDB'92)*, Vancouver, Canada, Aug. 1992. (extended version is Univ. of Cal., Irvine, Technical Report #92-07, Jan. 1992.)
- [Rund92b] Rundensteiner, E. A., and Bic, L., Set Operations in Object-Based Data Models, in *IEEE Transaction on Data and Knowledge Engineering*, to appear in Volume 4, Issue 3, June 1992.

- [Rund92a] Rundensteiner, E. A. and Bic, L., Automatic View Schema Generation in Object-Oriented Databases, Dept. of Information and Computer Science, Univ. of Cal., Irvine, Tech. Rep. 92-15, Jan. 1992.
- [Rund92d] Rundensteiner, E. A., A Class Integration Algorithm and its Application For Supporting Consistent Object Views, Univ. of Cal., Irvine, Tech. Rep. #92-50, May 1992.
- [Rund92e] Rundensteiner, E. A., and Gajski, D. D., Functional Synthesis Using Area and Delay Optimization, *29th ACM/IEEE Design Automation Conf. (DAC'92)*, Anaheim, California, June 1992.
- [Rund93] Rundensteiner, E. A., Bic, L., Gilbert, J., and Yin, M.-Y., Set-Restricted Semantic Groupings, in *IEEE Trans. on Data and Knowledge Eng.*, to appear in April 1993.
- [Rund91d] Rundensteiner, E. A., and Gajski, D. D., BDEF: The Behavioral Design Data Exchange Format Info. and Computer Science Dept., UCI, Tech. Rep. 91-34, April 1991.
- [Sche91] Scheichenzuber, J., Hardware Specification and Representation Using a Dataflow Notation, *High Level Synthesis Workshop*, pp. 61 - 68, March 1991.
- [Schm83] Schmolze, J. G., and Lipkis, T. A., Classification in the KL-ONE Knowledge Representation System, *The Eighth Int. Joint Conf. on Artificial Intelligence, (IJCAI'83)*, vol.1, pp. 330 - 332, Aug. 1983.
- [Scho91] Scholl, M. H., Laasch, C. and Tresch, M., Updatable Views in Object-Oriented Databases, *Proc. 2nd DOOD Conf.*, Muenich, Dec. 1991.
- [Shil89] Shilling, J. J., and Sweeney, P. F., Three Steps to Views: Extending the Object-Oriented Paradigm, in *Proc. of the Int. Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'89)*, New Orleans, pp. 353 - 361, Sep. 1989.
- [Ship81] Shipman, D. W., The Functional Data Model and the Data Language DAPLEX, in *ACM Trans. on Database Systems*, vol. 6, issue 1, pp. 140 - 173, Mar. 1981.
- [Siep89] Siepmann, E. and Zimmermann, G., An Object-Oriented Data Model for the VLSI Design System PLAYOUT, *DAC'89*, pp. 814 - 817, 1989.
- [Siep91] Siepmann, E., Entwurfstheorie und Entwurfsdatenmodellierung fuer CAD-Frameworks, Universitaet Kaiserslautern, Germany, Ph.D. dissertation, September 1991.
- [Smit77] Smith, J. M. and Smith, D.C.P., Database Abstractions: Aggregation and Generalization, in *ACM Trans. on Database Systems*, vol. 2, no. 2, pp. 105 - 133, June 1977.

- [Snod86] Snodgrass, R., and Shannon, K., Supporting Flexible and Efficient Tool Integration, *Lecture Notes in Computer Science, Vol. 244, Advanced Programming Environments: Proceedings of an International Workshop*, Springer-Verlag, Trondheim, Norway, 1986.
- [Ston86] Stonebraker, M. and Rowe, L., The Design of POSTGRES, In *Proc. of SIGMOD 1986 Conf.*, 1986, pp. 340 - 355.
- [Su86] Su, Y. W. S., Modeling Integrated Manufacturing Data with SAM\*, in *IEEE Computer*, vol. 19, issue 1, pp. 34 - 49, 1986.
- [Tana88] Tanaka, K., Yoshikawa, M., and Ishihara, K., Schema Virtualization in Object-Oriented Databases, In *Proc. IEEE Data Engineering Conf.*, pp. 23 - 30, Feb. 1988.
- [Thom92] Thomas, I. and Nejme, B. A., Definitions of Tool Integration for Environments, *IEEE Software*, 9, 2, pp. 29 - 35, March 1992.
- [Ullm88] Ullman, J. D., *Principles of Database and Knowledge-Base Systems*, Vol. 1, Computer Science Press, 1988.
- [VHDL88] VHDL Language Reference Manual, Addison Wesley, 1988.
- [Walk85] Walker, R. A., and Thomas, D. E., A Model of Design Representation and Synthesis, *DAC'85*, pp. 453 - 459, 1985.
- [Walk87] Walker, R. A., and Thomas, D. E., Design Representation and Transformations in the System's Architect's Workbench, *ICCAD'87*, pp. 166 - 169, 1987.
- [Wolf86] Wolf, W., An Object-Oriented Procedural Database for VLSI Chip Planning, *Proc. of the 27th Design Automation Conf.*, pp. 142 - 145, 1990.
- [Wolf90] Van der Wolf, P., Sloof, G. W., Bingley, P., and Dewilde, P., Meta Data Management in the NELSI CAD Framework, *DAC'90*, pp. 142 - 145, 1990.
- [Yu91] Yu and Osborn, An Evaluation Framework for Algebraic Object-Oriented Query Models, in *Proc. IEEE Data Eng. Conf.*, Feb. 1991.
- [Zane92] Zanella, M., Principles of Design Methodology Management for Electronic CAD Frameworks, *EDAC'92*, pp. 25 - 29, Feb. 1992.